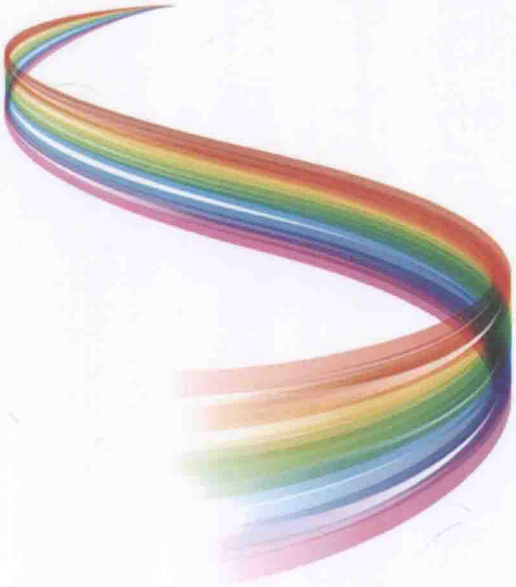
HZ BOOKS
华章IT

阿里巴巴集团专家鼎力推荐，阿里巴巴资深Java开发和大数据专家撰写

结合大量图和示例，对Spark的架构、部署模式和工作模块的设计理念、实现源码与使用技巧进行了深入的剖析与解读

大数据
技术丛书



Spark Internals
Core Design and Source Code Analysis

深入理解Spark

核心思想与源码分析

耿嘉安◎著

机械工业出版社
China Machine Press

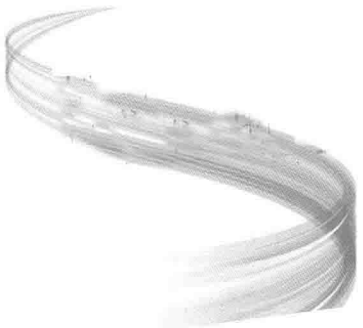
大数据技术丛书

Spark Internals
Core Design and Source Code Analysis

深入理解Spark

核心思想与源码分析

耿嘉安◎著



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

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：三河市宏图印务有限公司　　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

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

Contents 目　录

扩 展 篇

准 备 篇

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。

```
[      ja@v218142118 ~]$ java -version
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
OpenJDK (      ) 64-Bit Server VM (build 24.45-b08-internal, mixed mode)
```

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。

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。

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。

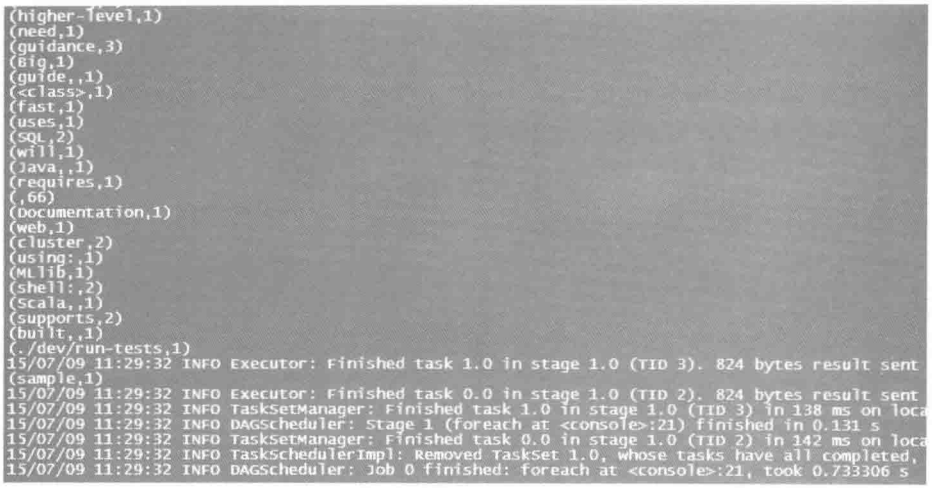


图 1-10 步骤 5 输出结果

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。

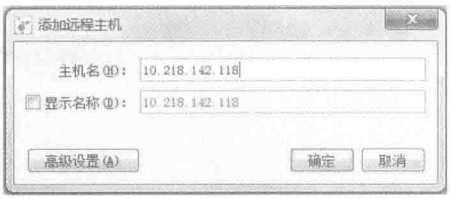


图 1-11 添加远程主机

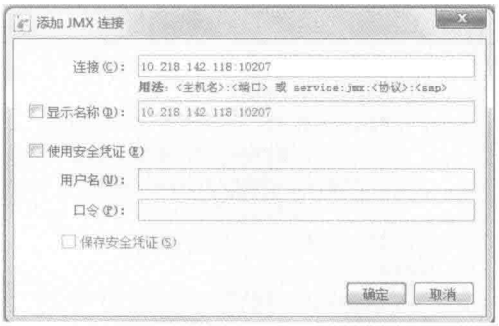


图 1-12 添加 JMX 连接

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。

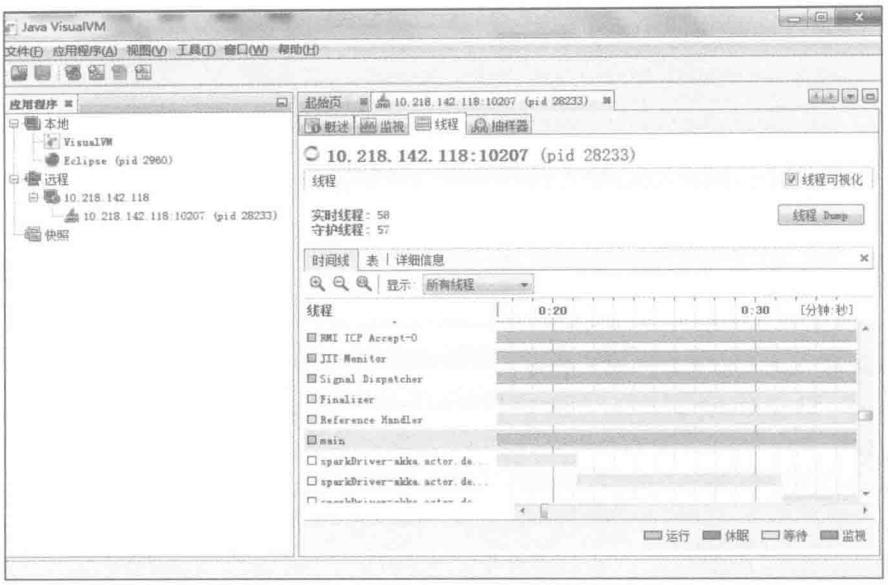


图 1-13 查看 Spark 线程

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。

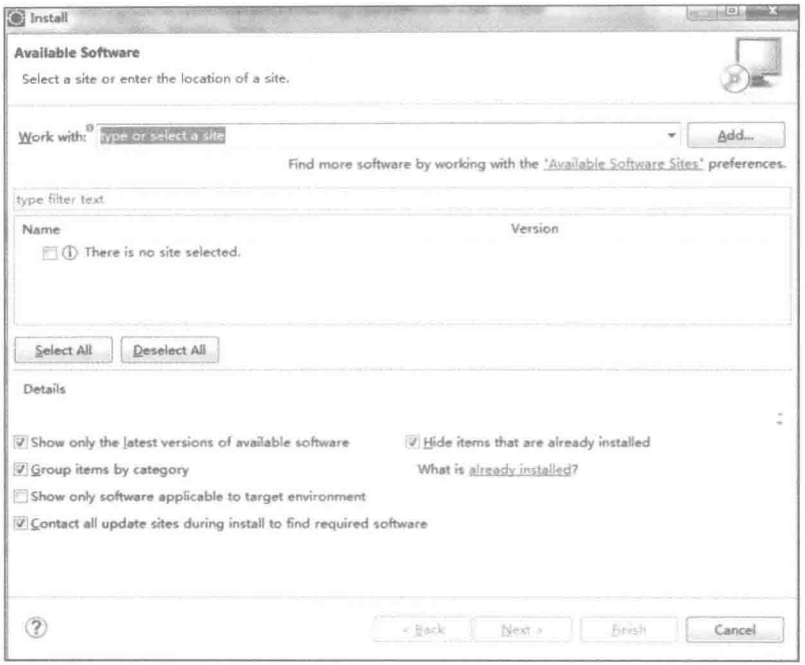


图 1-15 Install 对话框

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。

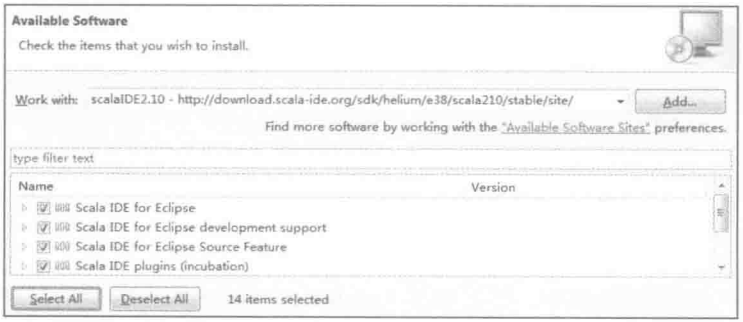


图 1-17 安装 Scala IDE 插件

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。

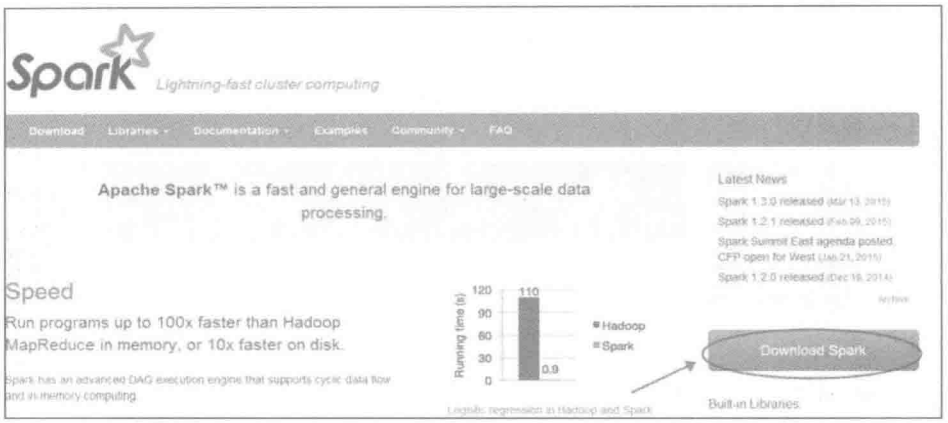


图 1-18　Spark 官网

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。

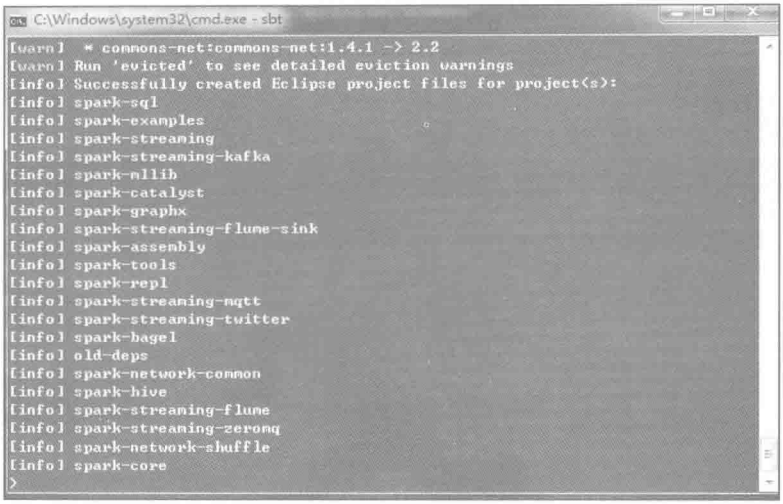


图 1-21 sbt 编译过程

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。

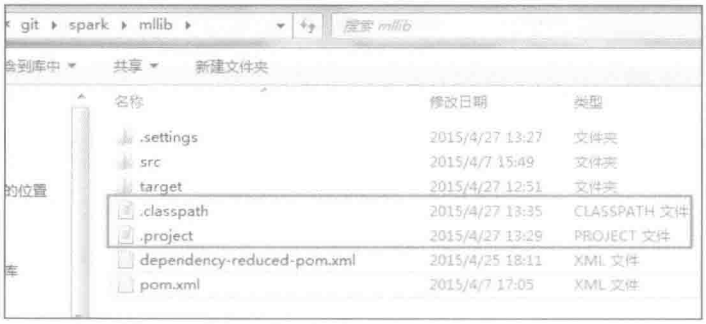


图 1-22 sbt 生成的项目文件

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。

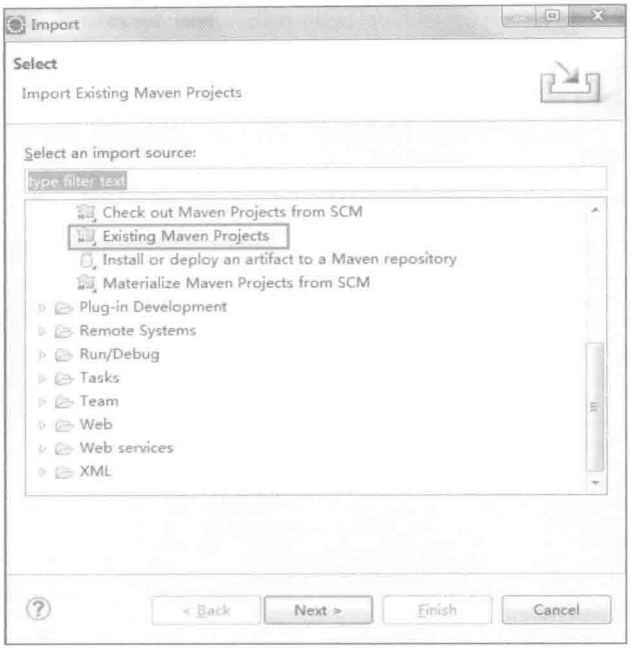


图 1-23 导入 Maven 项目

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

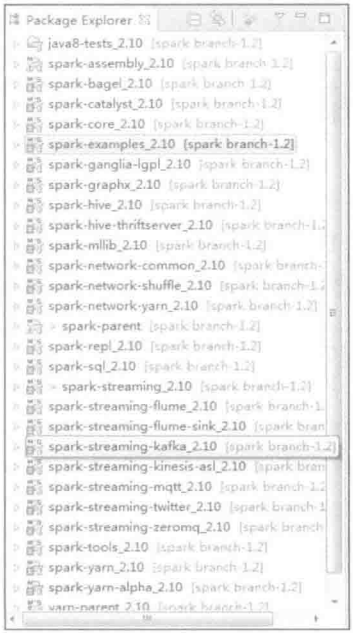


图 1-25 导入完成的项目

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。

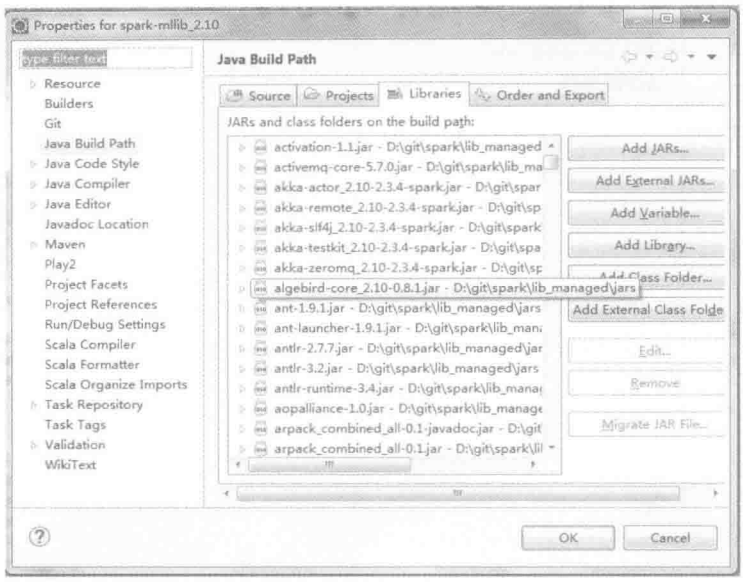


图 1-26 Java 编译目录

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。

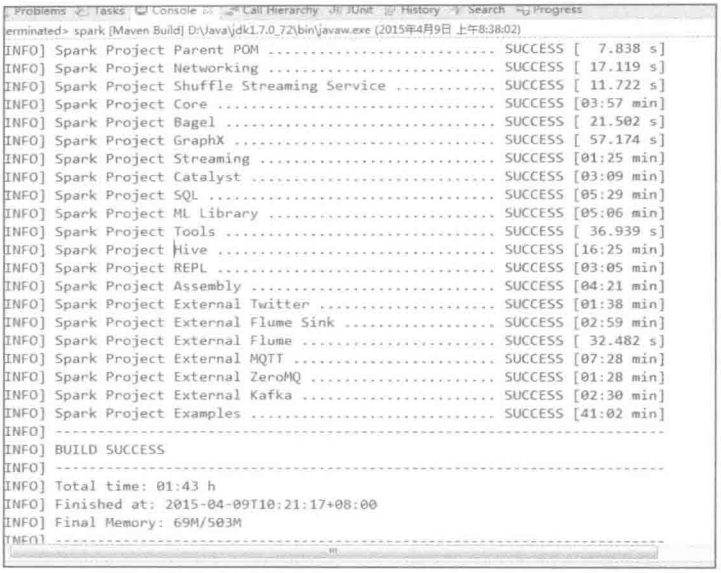


图 1-27 编译成功

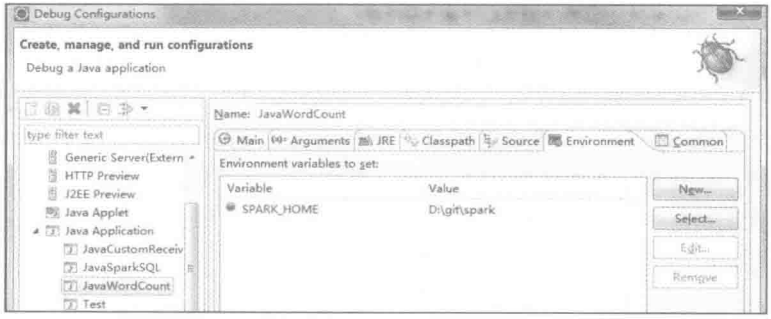


图 1-28 源码调试

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。

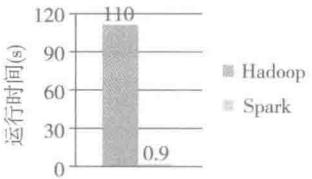


图 2-3 Hadoop 与 Spark 执行逻辑回归时间比较

- 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。

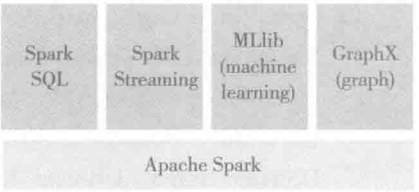


图 2-4　Spark 各模块依赖关系

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。

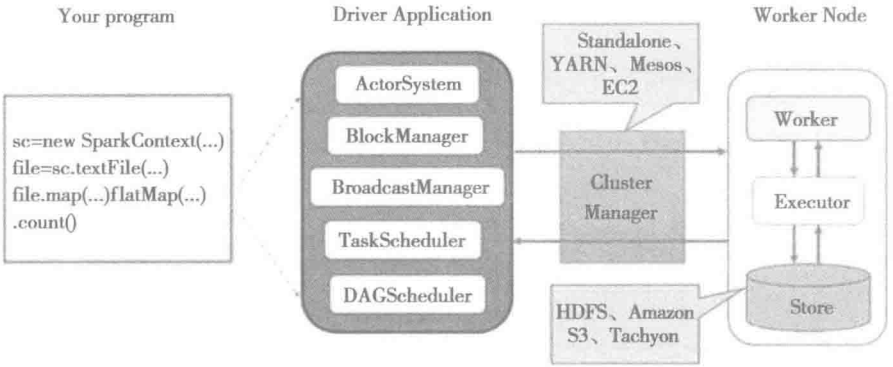


图 2-5　代码执行过程

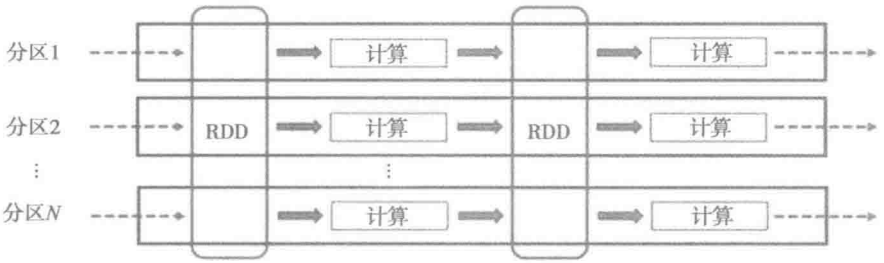


图 2-6　RDD 计算模型

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。

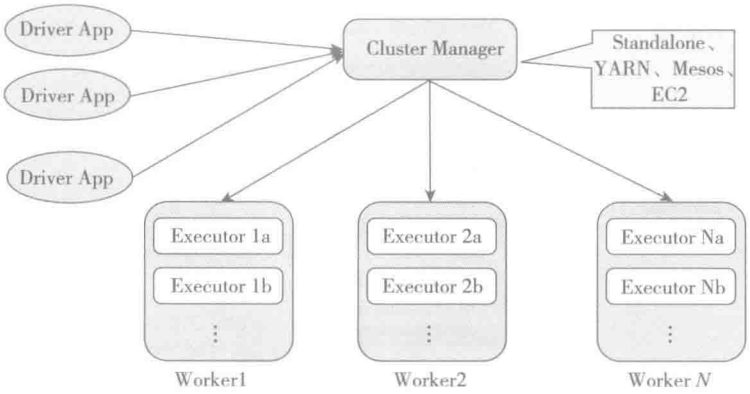


图 2-7 Spark 基本架构图

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

- 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。

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 BlockManagerMasterActor 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)
```

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。

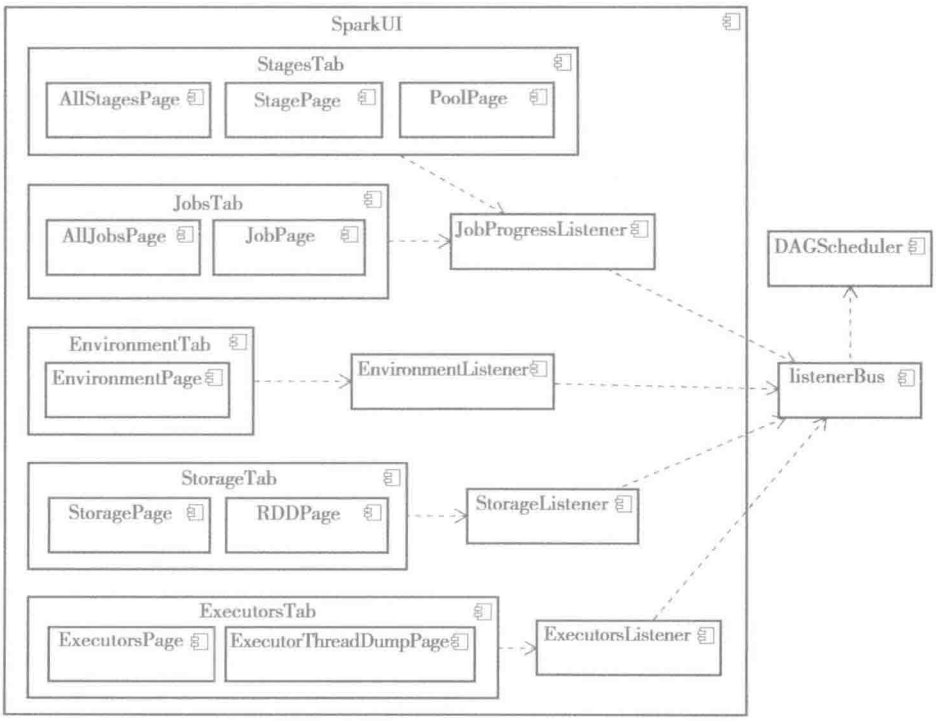


图 3-1 SparkUI 架构

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。

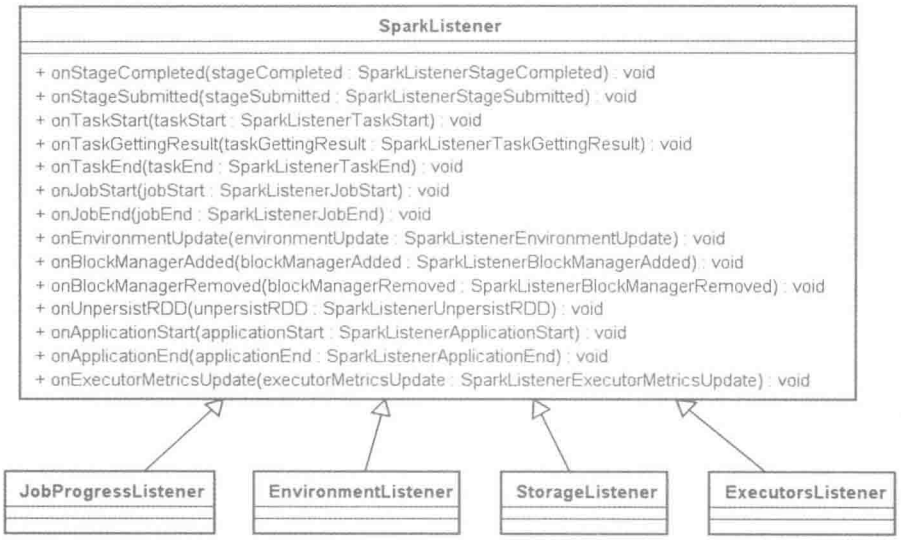


图 3-2　SparkListener 的类继承体系

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。

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 DAGScheduler-EventProcessActor，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TaskSetManager。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。

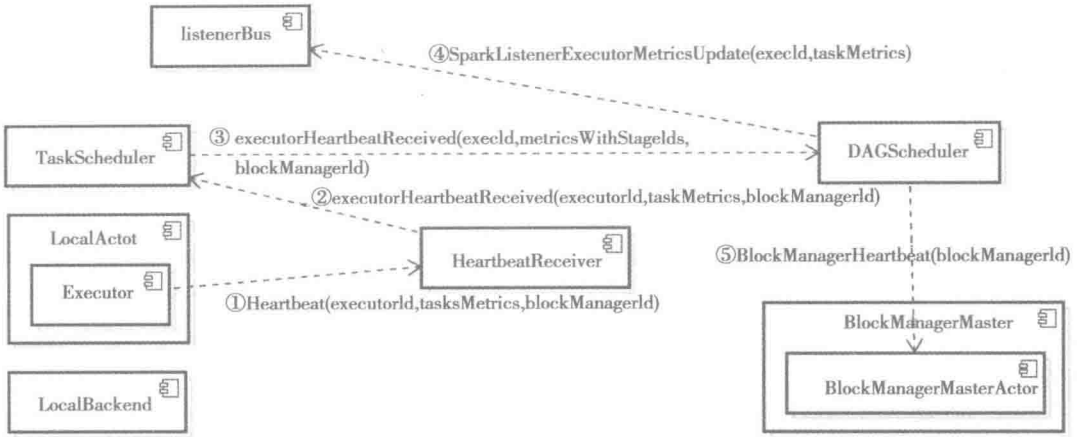


图 3-3　Executor 的心跳通信过程

```
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 Concurrent-Map<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。

第 4 章 Chapter 4

存储体系

天行健，君子以自强不息；地势坤，君子以厚德载物。

——《易经》

本章导读

从本章开始，我们一起来学习 Spark 的核心知识，掌握好这些内容是阅读全书的关键。无论是 Spark 的初始化阶段还是任务提交、执行阶段，始终离不开存储体系。笔者将所有涉及存储的内容都集中在本章，以便于对存储体系有好的抽象，也便于读者对其内容有更宏观的认识。

Spark 为了避免 Hadoop 读写磁盘的 I/O 操作成为性能瓶颈，优先将配置信息、计算结果等数据存入内存，这极大地提升了系统的执行效率。正是因为这一关键决策，才让 Spark 能在大数据应用中表现出优秀的计算能力。

4.1 存储体系概述

4.1.1 块管理器 BlockManager 的实现

块管理器 BlockManager 是 Spark 存储体系中的核心组件，因此本章内容主要围绕 BlockManager 展开。Driver Application 和 Executor 都会创建 BlockManager，BlockManager 的实现见代码清单 4-1。

代码清单4-1 BlockManager的实现

```
val diskBlockManager = new DiskBlockManager(this, conf)

private val blockInfo = new TimeStampedHashMap[BlockId, BlockInfo]

// Actual storage of where blocks are kept
private var tachyonInitialized = false
private[spark] val memoryStore = new MemoryStore(this, maxMemory)
private[spark] val diskStore = new DiskStore(this, diskBlockManager)
private[spark] lazy val tachyonStore: TachyonStore = {
    val storeDir = conf.get("spark.tachyonStore.baseDir", "/tmp_spark_tachyon")
    val appFolderName = conf.get("spark.tachyonStore.folderName")
    val tachyonStorePath = s"$storeDir/$appFolderName/${this.executorId}"
    val tachyonMaster = conf.get("spark.tachyonStore.url", "tachyon://
       localhost:19998")
    val tachyonBlockManager =
        new TachyonBlockManager(this, tachyonStorePath, tachyonMaster)
    tachyonInitialized = true
    new TachyonStore(this, tachyonBlockManager)
}

private[spark] val shuffleClient = if (externalShuffleServiceEnabled) {
    val transConf = SparkTransportConf.fromSparkConf(conf, numUsableCores)
    new ExternalShuffleClient(transConf, securityManager, securityManager.
        isAuthenticationEnabled())
} else {
    blockTransferService
}

private val slaveActor = actorSystem.actorOf(
    Props(new BlockManagerSlaveActor(this, mapOutputTracker)),
    name = "BlockManagerActor" + BlockManager.ID_GENERATOR.next)

private val metadataCleaner = new MetadataCleaner(
    MetadataCleanerType.BLOCK_MANAGER, this.dropOldNonBroadcastBlocks, conf)
private val broadcastCleaner = new MetadataCleaner(
    MetadataCleanerType.BROADCAST_VARS, this.dropOldBroadcastBlocks, conf)

private lazy val compressionCodec: CompressionCodec = CompressionCodec.createCodec(conf)
```

上面代码中声明的 BlockInfo：TimeStampedHashMap[BlockId, BlockInfo]，用于 BlockManager 缓存 BlockId 及对应的 BlockInfo。从代码清单 4-1 看到，BlockManager 主要由以下部分组成：

- shuffle 客户端 ShuffleClient；
- BlockManagerMaster（对存在于所有 Executor 上的 BlockManager 统一管理）；
- 磁盘块管理器 DiskBlockManager；
- 内存存储 MemoryStore；
- 磁盘存储 DiskStore；

- Tachyon 存储 TachyonStore；
- 非广播 Block 清理器 metadataCleaner 和广播 Block 清理器 broadcastCleaner；
- 压缩算法实现 CompressionCodec。

BlockManager 要生效，必须要初始化，它的初始化方法见代码清单 4-2。BlockManager 的初始化步骤如下：

1）BlockTransferService 的初始化和 ShuffleClient 的初始化（具体参见 4.2 节）。ShuffleClient 默认是 BlockTransferService，当有外部的 ShuffleService 时，调用外部 ShuffleService 的初始化方法。

2）BlockManagerId 和 ShuffleServerId 的创建。当有外部的 ShuffleService 时，创建新的 BlockManagerId，否则 ShuffleServerId 默认使用当前 BlockManager 的 BlockManagerId。

3）向 BlockManagerMaster 注册 BlockManagerId，具体实现见 4.3.3 节（当有外部的 ShuffleService 时，还需要向 BlockManagerMaster 注册 ShuffleServerId）。

代码清单4-2 BlockManager的初始化

```
def initialize(appId: String): Unit = {
    blockTransferService.init(this)
    shuffleClient.init(appId)

    blockManagerId = BlockManagerId(
        executorId, blockTransferService.hostName, blockTransferService.port)

    shuffleServerId = if (externalShuffleServiceEnabled) {
        BlockManagerId(executorId, blockTransferService.hostName,
            externalShuffleServicePort)
    } else {
        blockManagerId
    }

    master.registerBlockManager(blockManagerId, maxMemory, slaveActor)
    // Register Executors' configuration with the local shuffle service, if one
        should exist.
    if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
        registerWithExternalShuffleServer()
    }
}
```

4.1.2 Spark 存储体系架构

在详细介绍存储体系之前，我们先用图 4-1 说明 Spark 存储体系的架构。

这里对图 4-1 中的调用关系做个说明：

- 记号①表示 Executor 的 BlockManager 与 Driver 的 BlockManager 进行消息通信，例如，注册 BlockManager、更新 Block 信息、获取 Block 所在的 BlockManager、删除

Executor 等；

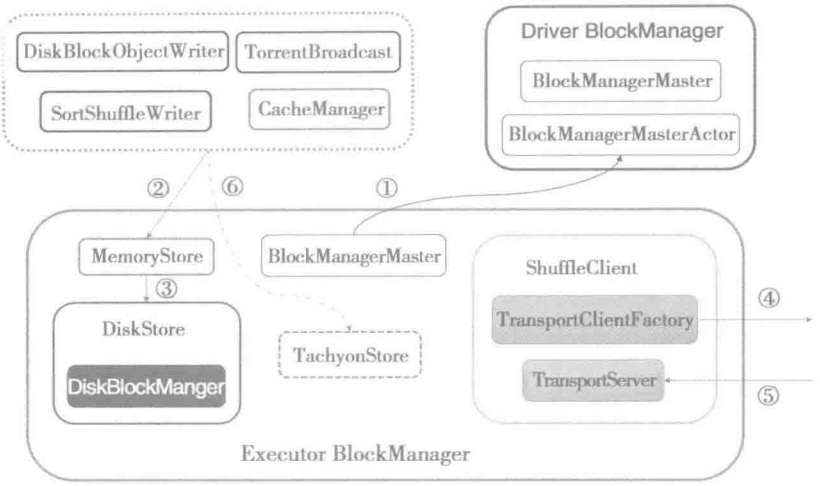


图 4-1 Spark 存储体系架构

- 记号②表示对 BlockManager 的读操作（例如 get、doGetLocal 以及 BlockManager 内部进行的 MemoryStore、DiskStore、TachyonStore 的 getBytes、getValues 等操作）和写操作（例如 doPut、putSingle、putBytes 以及 BlockManager 内部进行的 MemoryStore、DiskStore、TachyonStore 的 putBytes、putArray、putIterator 等操作）；
- 记号③表示当 MemoryStore 的内存不足时，写入 DiskStore，而 DiskStore 实际依赖于 DiskBlockManager；
- 记号④表示通过访问远端节点的 Executor 的 BlockManager 中的 TransportServer 提供的 RPC 服务下载或者上传 Block；
- 记号⑤表示远端节点的 Executor 的 BlockManager 访问本地 Executor 的 BlockManager 中的 TransportServer 提供的 RPC 服务下载或者上传 Block；
- 记号⑥表示当存储体系选择 Tachyon 作为存储时，对于 BlockManager 的读写操作实际调用了 TachyonStore 的 putBytes、putArray、putIterator、getBytes、getValues 等。

Spark 目前支持 HDFS、Amazon S3 两种主流分布式存储系统，还使用也诞生于 UCBerkeley 的 AMP 实验室的 Tachyon 这种高效的分布式文件系统作为缓存。

Spark 定义了抽象类 BlockStore，用于制定所有存储类型的规范。目前 BlockStore 的具体实现包括 MemoryStore、DiskStore 和 TachyonStore。BlockStore 的继承体系如图 4-2 所示。

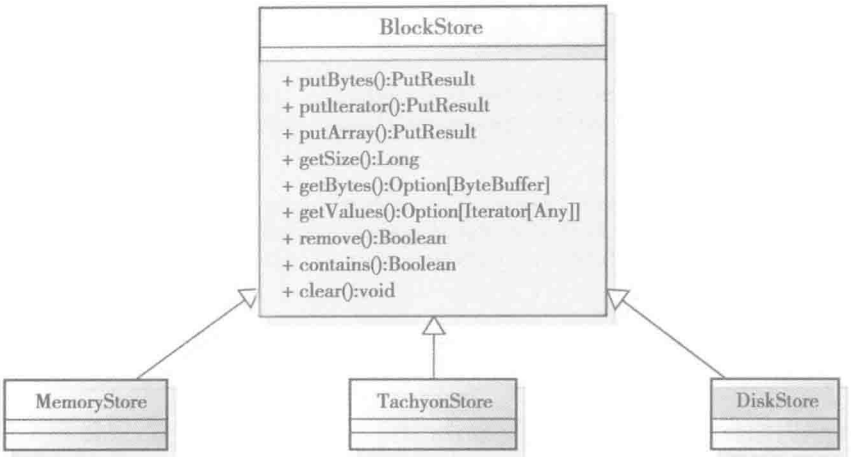


图 4-2 BlockStore 继承体系

4.2 shuffle 服务与客户端

读者可能奇怪：为什么需要把由 Netty 实现的网络服务组件也放到存储体系里面？这是由于 Spark 是分布式部署的，每个 Task 最终都运行在不同的机器节点上。map 任务的输出结果直接存储到 map 任务所在机器的存储体系中，reduce 任务极有可能不在同一机器上运行，所以需要远程下载 map 任务的中间输出。因此将 ShuffleClient 放到存储体系是最合适的。

ShuffleClient 并不像它的名字一样，是 shuffle 的客户端，它不光是将 shuffle 文件上传到其他 Executor 或者下载到本地的客户端，也提供了可以被其他 Executor 访问的 shuffle 服务。读到这里，熟悉 Hadoop YARN 的读者可能已经发现 Spark 与 Hadoop 一样，都采用 Netty 作为 shuffle server。从代码清单 4-1 可知，当有外部的 ShuffleClient 时，新建 ExternalShuffleClient，否则默认为 BlockTransferService。BlockTransferService 只有在其 init 方法被调用，即被初始化后才提供服务。以默认的 NettyBlockTransferService 的 init 方法为例，见代码清单 4-3。NettyBlockTransferService 的初始化步骤如下：

1）创建 RpcServer；

2）构造 TransportContext；

3）创建 RPC 客户端工厂 TransportClientFactory；

4）创建 Netty 服务器 TransportServer，可以修改属性 spark.blockManager.port（默认为 0，表示随机选择）改变 TransportServer 的端口。

代码清单4-3 NettyBlockTransferService的初始化

```
override def init(blockDataManager: BlockDataManager): Unit = {
    val (rpcHandler: RpcHandler, bootstrap: Option[TransportClientBootstrap]) = {
        val nettyRpcHandler = new NettyBlockRpcServer(serializer, blockDataManager)
```

```
            if (!authEnabled) {
                (nettyRpcHandler, None)
            } else {
                (new SaslRpcHandler(nettyRpcHandler, securityManager),
                    Some(new SaslClientBootstrap(transportConf, conf.getAppId, securityManager)))
            }
        }
        transportContext = new TransportContext(transportConf, rpcHandler)
        clientFactory = transportContext.createClientFactory(bootstrap.toList)
        server = transportContext.createServer(conf.getInt("spark.blockManager.port",
            0))
        appId = conf.getAppId
        logInfo("Server created on " + server.getPort)
    }
```

接下来我们逐步讲解 Block 的 RPC 服务，构造 TransportContext，创建 RPC 客户端工厂 TransportClientFactory，创建 Netty 服务器 TransportServer 的实现。此外还会介绍 reduce 任务是如何拉取 map 任务中间结果的（即 shuffle 过程的数据传输）。

4.2.1 Block 的 RPC 服务

当 map 任务与 reduce 任务处于不同节点时，reduce 任务需要从远端节点下载 map 任务的中间输出，因此 NettyBlockRpcServer 提供打开，即下载 Block 文件的功能；一些情况下，为了容错，需要将 Block 的数据备份到其他节点上，所以 NettyBlockRpcServer 还提供了上传 Block 文件的 RPC 服务，NettyBlockRpcServer 的实现见代码清单 4-4。

代码清单4-4 NettyBlockRpcServer的实现

```
class NettyBlockRpcServer(
        serializer: Serializer,
        blockManager: BlockDataManager)
    extends RpcHandler with Logging {

    private val streamManager = new OneForOneStreamManager()

    override def receive(
            client: TransportClient,
            messageBytes: Array[Byte],
            responseContext: RpcResponseCallback): Unit = {
        val message = BlockTransferMessage.Decoder.fromByteArray(messageBytes)
        logTrace(s"Received request: $message")

    message match {
        case openBlocks: OpenBlocks =>
            val blocks: Seq[ManagedBuffer] =
                openBlocks.blockIds.map(BlockId.apply).map(blockManager.getBlockData)
            val streamId = streamManager.registerStream(blocks.iterator)
            logTrace(s"Registered streamId $streamId with ${blocks.size} buffers")
            responseContext.onSuccess(new StreamHandle(streamId, blocks.size).toByteArray)
```

```
            case uploadBlock: UploadBlock =>
                val level: StorageLevel =
                    serializer.newInstance().deserialize(ByteBuffer.wrap(uploadBlock.metadata))
                val data = new NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData))
                blockManager.putBlockData(BlockId(uploadBlock.blockId), data, level)
                responseContext.onSuccess(new Array[Byte](0))
            }
        }

        override def getStreamManager(): StreamManager = streamManager
    }
```

4.2.2 构造传输上下文 TransportContext

TransportContext 用于维护传输上下文，它的构造器如下。

```
public TransportContext(TransportConf conf, RpcHandler rpcHandler) {
    this.conf = conf;
    this.rpcHandler = rpcHandler;
    this.encoder = new MessageEncoder();
    this.decoder = new MessageDecoder();
}
```

TransportContext 既可以创建 Netty 服务，也可以创建 Netty 访问客户端。TransportContext 的组成如下：

- TransportConf：主要控制 Netty 框架提供的 shuffle 的 I/O 交互的客户端和服务端线程数量；
- RpcHandler：负责 shuffle 的 I/O 服务端在接收到客户端的 RPC 请求后，提供打开 Block 或者上传 Block 的 RPC 处理，此处即为 NettyBlockRpcServer；
- decoder：在 shuffle 的 I/O 服务端对客户端传来的 ByteBuf 进行解析，防止丢包和解析错误；
- encoder：在 shuffle 的 I/O 客户端对消息内容进行编码，防止服务端丢包和解析错误。

为什么需要 MessageEncoder 和 MessageDecoder？因为在基于流的传输里（比如 TCP/IP），接收到的数据首先会被存储到一个 socket 接收缓冲里。不幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使发送了 2 个独立的数据包，操作系统也不会作为 2 个消息处理，而仅仅认为是一连串的字节。因此不能保证远程写入的数据会被准确地读取。举个例子，假设操作系统的 TCP/TP 协议栈已经接收了 3 个数据包：ABC、DEF、GHI。由于基于流传输的协议的这种统一的性质，在应用程序读取数据时很可能性被分成下面的片段：AB、CDEFG、H、I。因此，接收方不管是客户端还是服务端，都应该把接收到的数据整理成一个或者多个更有意义并且让程序的逻辑更好理解的数据。

4.2.3 RPC 客户端工厂 TransportClientFactory

TransportClientFactory 是创建 Netty 客户端 TransportClient 的工厂类，TransportClient 用于向 Netty 服务端发送 RPC 请求。TransportContext 的 createClientFactory 方法用于创建 TransportClientFactory，实现如下。

```
public TransportClientFactory createClientFactory(List<TransportClientBootstrap>
    bootstraps) {
    return new TransportClientFactory(this, bootstraps);
}
```

从代码清单 4-5 可以看到，TransportClientFactory 由以下部分组成：

- clientBootstraps：用于缓存客户端列表；
- connectionPool：用于缓存客户端连接；
- numConnectionsPerPeer：节点之间取数据的连接数，可以使用属性 spark.shuffle.io.numConnectionsPerPeer 来配置，默认为 1；
- socketChannelClass：客户端 channel 被创建时使用的类，可以使用属性 spark.shuffle.io.mode 来配置，默认为 NioSocketChannel；
- workerGroup：根据 Netty 的规范，客户端只有 work 组，所以此处创建 workerGroup，实际是 NioEventLoopGroup；
- pooledAllocator：汇集 ByteBuf 但对本地线程缓存禁用的分配器。

TransportClientFactory 里大量使用了 NettyUtils，关于 NettyUtils 的具体实现，请看附录 G。

代码清单4-5 TransportClientFactory的实现

```
public TransportClientFactory(
        TransportContext context,
        List<TransportClientBootstrap> clientBootstraps) {
    this.context = Preconditions.checkNotNull(context);
    this.conf = context.getConf();
    this.clientBootstraps = Lists.newArrayList(Preconditions.checkNotNull(client
        Bootstraps));
    this.connectionPool = new ConcurrentHashMap<SocketAddress, ClientPool>();
    this.numConnectionsPerPeer = conf.numConnectionsPerPeer();
    this.rand = new Random();

    IOMode ioMode = IOMode.valueOf(conf.ioMode());
    this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);
    this.workerGroup = NettyUtils.createEventLoop(ioMode, conf.clientThreads(),
        "shuffle-client");
    this.pooledAllocator = NettyUtils.createPooledByteBufAllocator(
        conf.preferDirectBufs(), false /* allowCache */, conf.clientThreads());
}
```

提示 NIO 是指 Java 中 New IO 的简称，其特点包括：为所有的原始类型提供 (Buffer) 缓存支持；字符集编码解码解决方案；提供一个新的原始 I/O 抽象 Channel，支持锁和内存映射文件的文件访问接口；提供多路非阻塞式 (non-bloking) 的高伸缩性网络 I/O 。其具体使用属于 Java 语言的范畴，本文不过多介绍。

4.2.4 Netty 服务器 TransportServer

TransportServer 提供了 Netty 实现的服务器端，用于提供 RPC 服务（比如上传、下载等）。创建 TransportServer 的代码如下。

```
public TransportServer createServer(int port) {
    return new TransportServer(this, port);
}
```

TransportServer 的构造器实现如下。

```
public TransportServer(TransportContext context, int portToBind) {
    this.context = context;
    this.conf = context.getConf();

    init(portToBind);
}
```

上面代码中的 init 方法用于对 TransportServer 初始化，通过使用 Netty 框架的 EventLoopGroup、ServerBootstrap 等 API 创建 shuffle 的 I/O 交互的服务端，init 的主要代码见代码清单 4-6。

代码清单4-6 init的主要代码

```
private void init(int portToBind) {

    IOMode ioMode = IOMode.valueOf(conf.ioMode());
    EventLoopGroup bossGroup =
        NettyUtils.createEventLoop(ioMode, conf.serverThreads(), "shuffle-server");
    EventLoopGroup workerGroup = bossGroup;

    PooledByteBufAllocator allocator = NettyUtils.createPooledByteBufAllocator(
        conf.preferDirectBufs(), true /* allowCache */, conf.serverThreads());

    bootstrap = new ServerBootstrap()
        .group(bossGroup, workerGroup)
        .channel(NettyUtils.getServerChannelClass(ioMode))
        .option(ChannelOption.ALLOCATOR, allocator)
        .childOption(ChannelOption.ALLOCATOR, allocator);

    bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            context.initializePipeline(ch);
```

```
        }
    });

    channelFuture = bootstrap.bind(new InetSocketAddress(portToBind));
    channelFuture.syncUninterruptibly();

    port = ((InetSocketAddress) channelFuture.channel().localAddress()).
        getPort();
}
```

ServerBootstrap 的 childHandler 方法调用了 TransportContext 的 initializePipeline。initializePipeline 中创建了 TransportChannelHandler，并将它绑定到 SocketChannel 的 pipeline 的 handler 中，见代码清单 4-7。

代码清单4-7 initializePipeline方法的实现

```
public TransportChannelHandler initializePipeline(SocketChannel channel) {
    try {
        TransportChannelHandler channelHandler = createChannelHandler(channel);
        channel.pipeline()
            .addLast("encoder", encoder)
            .addLast("frameDecoder", NettyUtils.createFrameDecoder())
            .addLast("decoder", decoder)
            .addLast("handler", channelHandler);
        return channelHandler;
    } catch (RuntimeException e) {
        logger.error("Error while initializing Netty pipeline", e);
        throw e;
    }
}

    private TransportChannelHandler createChannelHandler(Channel channel) {
        TransportResponseHandler responseHandler = new TransportResponseHandler(channel);
        TransportClient client = new TransportClient(channel, responseHandler);
        TransportRequestHandler requestHandler = new TransportRequestHandler(cha
            nnel, client,rpcHandler);
    return new TransportChannelHandler(client, responseHandler, requestHandler);
}
```


注意 本节很多代码都是通过使用 Netty API 来实现的，有兴趣的读者可以去 http://netty.io/ 查阅 API 的使用。一些读者可能注意到 Spark 在使用 Netty 时，都是用 Java 作为编程语言，实际上也可以使用 Scala 作为编程语言的。这个问题，笔者不了解其发生的背景，如果有读者知道，希望能及时通知笔者。

4.2.5 获取远程 shuffle 文件

NettyBlockTransferService 的 fetchBlocks 方法用于获取远程 shuffle 文件，实际是利用 NettyBlockTransferService 中创建的 Netty 服务，见代码清单 4-8。

代码清单4-8 获取远端节点上的shuffle文件

```
override def fetchBlocks(
    host: String,
    port: Int,
    execId: String,
    blockIds: Array[String],
    listener: BlockFetchingListener): Unit = {
    val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter {
        override def createAndStart(blockIds: Array[String], listener:
            BlockFetchingListener) {
            val client = clientFactory.createClient(host, port)
            new OneForOneBlockFetcher(client, appId, execId, blockIds.toArray,
                listener).start()
        }
    }

    val maxRetries = transportConf.maxIORetries()
    if (maxRetries > 0) {
        new RetryingBlockFetcher(transportConf, blockFetchStarter, blockIds,
            listener).start()
    } else {
        blockFetchStarter.createAndStart(blockIds, listener)
    }
}
```

4.2.6 上传 shuffle 文件

NettyBlockTransferService 的 uploadBlock 方法用于上传 shuffle 文件到远程 Executor，实际也是利用 NettyBlockTransferService 中创建的 Netty 服务，见代码清单 4-9。NettyBlockTransferService 上传 Block 的步骤如下：

1）创建 Netty 服务的客户端，客户端连接的 hostname 和 port 正是我们随机选择的 BlockManager 的 hostname 和 port。

2）将 Block 的存储级别 StorageLevel 序列化。

3）将 Block 的 ByteBuffer 转化为数组，便于序列化。

4）将 appId、execId、blockId、序列化的 StorageLevel、转换为数组的 Block 封装为 UploadBlock，并将 UploadBlock 序列化为字节数组。

5）最终调用 Netty 客户端的 sendRpc 方法将字节数组上传，回调函数 RpcResponseCallback 根据 RPC 的结果更改上传状态。

代码清单4-9 上传Block到远端节点

```
override def uploadBlock(
    hostname: String,
    port: Int,
    execId: String,
    blockId: BlockId,
```

```
        blockData: ManagedBuffer,
        level: StorageLevel): Future[Unit] = {
    val result = Promise[Unit]()
    val client = clientFactory.createClient(hostname, port)

    // StorageLevel is serialized as bytes using our JavaSerializer. Everything else
        is encoded
    // using our binary protocol.
    val levelBytes = serializer.newInstance().serialize(level).array()

    // Convert or copy nio buffer into array in order to serialize it.
    val nioBuffer = blockData.nioByteBuffer()
    val array = if (nioBuffer.hasArray) {
        nioBuffer.array()
    } else {
        val data = new Array[Byte](nioBuffer.remaining())
        nioBuffer.get(data)
        data
    }

    client.sendRpc(new UploadBlock(appId, execId, blockId.toString, levelBytes,
        array).toByteArray,
        new RpcResponseCallback {
            override def onSuccess(response: Array[Byte]): Unit = {
                logTrace(s"Successfully uploaded block $blockId")
                result.success()
            }
            override def onFailure(e: Throwable): Unit = {
                logError(s"Error while uploading block $blockId", e)
                result.failure(e)
            }
            })

        result.future
    }
```

4.3 BlockManagerMaster 对 BlockManager 的管理

Driver 上的 BlockManagerMaster 对存在于 Executor 上的 BlockManager 统一管理，比如 Executor 需要向 Driver 发送注册 BlockManager、更新 Executor 上 Block 的最新信息、询问所需要 Block 目前所在的位置以及当 Executor 运行结束需要将此 Executor 移除等。但是 Driver 与 Executor 却位于不同机器中，该怎么实现呢？ Driver 上的 BlockManagerMaster 会持有 BlockManagerMasterActor，所有 Executor 也会从 ActorSystem 中获取 BlockManagerMasterActor 的引用，所有 Executor 与 Driver 关于 BlockManager 的交互都依赖于它。

4.3.1 BlockManagerMasterActor

BlockManagerMasterActor 只存在于 Driver 上。Executor 从 ActorSystem 获取 BlockManager-

MasterActor的引用，然后给BlockManagerMasterActor发送消息，实现和Driver交互。BlockManagerMasterActor的实现见代码清单4-10。

代码清单4-10 BlockManagerMasterActor的实现

```
private val blockManagerInfo = new mutable.HashMap[BlockManagerId, BlockManagerInfo]
private val blockManagerIdByExecutor = new mutable.HashMap[String, BlockManagerId]
private val blockLocations = new JHashMap[BlockId, mutable.HashSet[BlockManagerId]]

private val akkaTimeout = AkkaUtils.askTimeout(conf)
val slaveTimeout = conf.getLong("spark.storage.blockManagerSlaveTimeoutMs",
    math.max(conf.getInt("spark.executor.heartbeatInterval", 10000) * 3, 45000))
val checkTimeoutInterval = conf.getLong("spark.storage.blockManagerTimeoutIntervalMs",
     60000)
var timeoutCheckingTask: Cancellable = null

override def preStart() {
    import context.dispatcher
    timeoutCheckingTask = context.system.scheduler.schedule(0.seconds,
        checkTimeoutInterval.milliseconds, self, ExpireDeadHosts)
    super.preStart()
}

override def receiveWithLogging = {
    case RegisterBlockManager(blockManagerId, maxMemSize, slaveActor) =>
        register(blockManagerId, maxMemSize, slaveActor)
        sender ! true

    case UpdateBlockInfo(
        blockManagerId, blockId, storageLevel, deserializedSize, size,
            tachyonSize) =>
        updateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize,
            size, tachyonSize)

    case GetLocations(blockId) =>
        sender ! getLocations(blockId)

    case GetLocationsMultipleBlockIds(blockIds) =>
        sender ! getLocationsMultipleBlockIds(blockIds)

   case GetActorSystemHostPortForExecutor(executorId) =>
        sender ! getActorSystemHostPortForExecutor(executorId)

    case GetMemoryStatus =>
        sender ! memoryStatus

    case GetStorageStatus =>
        sender ! storageStatus

    case GetBlockStatus(blockId, askSlaves) =>
        sender ! blockStatus(blockId, askSlaves)

    case GetMatchingBlockIds(filter, askSlaves) =>
```

```
        sender ! getMatchingBlockIds(filter, askSlaves)

    case RemoveRdd(rddId) =>
        sender ! removeRdd(rddId)

    case RemoveShuffle(shuffleId) =>
        sender ! removeShuffle(shuffleId)

    case RemoveBroadcast(broadcastId, removeFromDriver) =>
        sender ! removeBroadcast(broadcastId, removeFromDriver)

    case RemoveBlock(blockId) =>
        removeBlockFromWorkers(blockId)
        sender ! true

    case RemoveExecutor(execId) =>
        removeExecutor(execId)
        sender ! true

    case StopBlockManagerMaster =>
        sender ! true
        if (timeoutCheckingTask != null) {timeoutCheckingTask.cancel()}
        context.stop(self)

    case ExpireDeadHosts =>
        expireDeadHosts()

    case BlockManagerHeartbeat(blockManagerId) =>
        sender ! heartbeatReceived(blockManagerId)

    case other =>
        logWarning("Got unknown message: " + other)
```

上面代码展示了 BlockManagerMasterActor 维护的很多缓存数据结构：

- blockManagerInfo：缓存所有的 BlockManagerId 及其 BlockManager 的信息；
- blockManagerIdByExecutor：缓存 executorId 与其拥有的 BlockManagerId 之间的映射关系；
- blockLocations：缓存 Block 与 BlockManagerId 的映射关系。

在代码清单 4-10 中，receiveWithLogging 作为匹配 BlockManagerMasterActor 接收到消息的偏函数；属性 spark.storage.blockManagerSlaveTimeoutMs 和 spark.executor.heartbeatInterval 共同决定 Slave 节点，即 BlockManager 的超时时间；属性 spark.storage.blockManagerTimeoutIntervalMs 指定检查 BlockManager 超时的时间间隔。

4.3.2 询问 Driver 并获取回复方法

在 Executor 的 BlockManagerMaster 中，所有与 Driver 上 BlockManagerMaster 的交互方法最终都调用了 askDriverWithReply，可见它是一个最基础的方法，它的代码如下。

```
private def askDriverWithReply[T](message: Any): T = {
    AkkaUtils.askWithReply(message, driverActor, AKKA_RETRY_ATTEMPTS, AKKA_RETRY_
        INTERVAL_MS,
        timeout)
}
```

此外，tell 方法作为 askDriverWithReply 的代理也经常被调用，代码如下。

```
private def tell(message: Any) {
    if (!askDriverWithReply[Boolean](message)) {
        throw new SparkException("BlockManagerMasterActor returned false,
            expected true.")
    }
}
```

askDriverWithReply 调用了 AkkaUtils.askWithReply 方法。askWithReply 方法实际使用 ActorSystem 向 BlockManagerMasterActor 发送任何消息。发送每条消息的最大尝试次数是 3 次，间隔为 3000 毫秒，请求超时时间是 30 秒，具体实现见附录 G。BlockManagerMasterActor 接收到消息后将由 receiveWithLogging 函数匹配，并处理具体的逻辑。

4.3.3 向 BlockManagerMaster 注册 BlockManagerId

Executor 或者 Driver 自身的 BlockManager 在初始化时，需要向 Driver 的 BlockManager 注册 BlockManager 信息，代码如下。

```
def registerBlockManager(blockManagerId: BlockManagerId, maxMemSize: Long,
     slaveActor: ActorRef) {
    logInfo("Trying to register BlockManager")
    tell(RegisterBlockManager(blockManagerId, maxMemSize, slaveActor))
    logInfo("Registered BlockManager")
}
```

从上面代码看到，消息内容包括 BlockManagerId、最大内存、BlockManagerSlaveActor。消息体带有 BlockManagerSlaveActor 是为了便于接受 BlockManagerMasterActor 回复的消息。这些信息被封装为 RegisterBlockManager，并调用刚刚在 4.3.2 节介绍的 tell 方法。根据之前的分析，RegisterBlockManager 消息会被 BlockManagerMasterActor 匹配并执行 register 方法注册 BlockManager，并在 register 方法执行结束后向发送者 BlockManagerSlaveActor 发送一个简单的消息 true。注册 BlockManager 的实现见代码清单 4-11。

代码清单4-11 注册BlockManager的实现

```
private def register(id: BlockManagerId, maxMemSize: Long, slaveActor: ActorRef) {
    val time = System.currentTimeMillis()
    if (!blockManagerInfo.contains(id)) {
        blockManagerIdByExecutor.get(id.executorId) match {
            case Some(oldId) =>
                // A block manager of the same executor already exists, so remove
                    it (assumed dead)
                logError("Got two different block manager registrations on same
```

```
                    executor - "
                    + s" will replace old one $oldId with new one $id")
                removeExecutor(id.executorId)
            case None =>
        }
        logInfo("Registering block manager %s with %s RAM, %s".format(
            id.hostPort, Utils.bytesToString(maxMemSize), id))

        blockManagerIdByExecutor(id.executorId) = id

        blockManagerInfo(id) = new BlockManagerInfo(
            id, System.currentTimeMillis(), maxMemSize, slaveActor)
    }
    listenerBus.post(SparkListenerBlockManagerAdded(time, id, maxMemSize))
}
```

register 方法确保 blockManagerInfo 持有消息中的 blockManagerId 及对应信息，并且确保每个 Executor 最多只能有一个 blockManagerId，旧的 blockManagerId 会被移除。最后向 listenerBus 中 post（推送）一个 SparkListenerBlockManagerAdded 事件。

此处以注册 BlockManager 为例，演示了 askDriverWithReply 和 tell 的使用。

4.4 磁盘块管理器 DiskBlockManager

4.4.1 DiskBlockManager 的构造过程

BlockManager 初始化时会创建 DiskBlockManager，DiskBlockManager 的构造步骤如下：

1）调用 createLocalDirs 方法创建本地文件目录，然后创建二维数组 subDirs，用来缓存一级目录 localDirs 及二级目录，其中二级目录的数量根据配置 spark.diskStore.subDirectories 获取，默认为 64。以笔者本地为例，创建的目录为：C:\Users\{username}\AppData\Local\Temp\spark-016f279f-4060-4065-b0cb-c7fad1f616ae\spark-58cdc43a-a39d-4b49-b357-f5ce9cc5c051，其中 spark-016f279f-4060-4065-b0cb-c7fad1f616ae 是一级目录，spark-58cdc43a-a39d-4b49-b357-f5ce9cc5c051 是二级目录，见代码清单 4-12。

代码清单4-12 创建本地文件目录的creatLocalDirs方法

```
private[spark]
val subDirsPerLocalDir = blockManager.conf.getInt("spark.diskStore.subDirectories", 64)

private[spark] val localDirs: Array[File] = createLocalDirs(conf)
if (localDirs.isEmpty) {
    logError("Failed to create any local dir.")
    System.exit(ExecutorExitCode.DISK_STORE_FAILED_TO_CREATE_DIR)
}
private val subDirs = Array.fill(localDirs.length)(new Array[File](subDirsPerLocalDir))
```

```
private def createLocalDirs(conf: SparkConf): Array[File] = {
    Utils.getOrCreateLocalRootDirs(conf).flatMap { rootDir =>
        try {
            val localDir = Utils.createDirectory(rootDir, "blockmgr")
            logInfo(s"Created local directory at $localDir")
            Some(localDir)
        } catch {
            case e: IOException =>
            logError(s"Failed to create local dir in $rootDir. Ignoring this
                directory.", e)
            None
        }
    }
}
```


注意 createLocalDirs 方法具体创建目录的过程实际调用了 Utils 的 getOrCreateLocalRootDirs 和 createDirectory 方法。有关 Utils 的使用请参见附录 A。

2）添加运行时环境结束时的钩子，用于在进程关闭时创建线程，通过调用 DiskBlockManager 的 stop 方法，清除一些临时目录，见代码清单 4-13。

代码清单4-13 addShutdownHook的实现

```
addShutdownHook()

private def addShutdownHook() {
        Runtime.getRuntime.addShutdownHook(new Thread("delete Spark local dirs") {
            override def run(): Unit = Utils.logUncaughtExceptions {
                logDebug("Shutdown hook called")
                DiskBlockManager.this.stop()
            }
        })
    }
/** Cleanup local dirs and stop shuffle sender. */
    private[spark] def stop() {
        // Only perform cleanup if an external service is not serving our shuffle
            files.
        if (!blockManager.externalShuffleServiceEnabled || blockManager.blockManagerId.
            isDriver) {
            localDirs.foreach { localDir =>
                if (localDir.isDirectory() && localDir.exists()) {
                    try {
                        if (!Utils.hasRootAsShutdownDeleteDir(localDir)) Utils.
                            deleteRecursively(localDir)
                    } catch {
                        case e: Exception =>
                        logError(s"Exception while deleting local spark dir: $localDir", e)
                    }
                }
        }
    }
}
```

DiskBlockManager 为什么要创建二级目录结构？这是因为二级目录用于对文件进行散列存储，散列存储可以使所有文件都随机存放，写入或删除文件更方便，存取速度快，节省空间。

4.4.2 获取磁盘文件方法 getFile

很多代码中都使用 DiskBlockManager 的 getFile 方法，获取磁盘上的文件，通过对 getFile 的分析，能够掌握 Spark 磁盘散列文件存储的实现机制。getFile 方法的实现见代码清单 4-14，其处理步骤如下：

1）根据文件名计算哈希值。

2）根据哈希值与本地文件一级目录的总数求余数，记为 dirId。

3）根据哈希值与本地文件一级目录的总数求商数，此商数与二级目录的数目再求余数，记为 subDirId。

4）如果 dirId/subDirId 目录存在，则获取 dirId/subDirId 目录下的文件，否则新建 dirId/subDirId 目录。

代码清单4-14 getFile方法的实现

```
def getFile(blockId: BlockId): File = getFile(blockId.name)
def getFile(filename: String): File = {
    val hash = Utils.nonNegativeHash(filename)
    val dirId = hash % localDirs.length
    val subDirId = (hash / localDirs.length) % subDirsPerLocalDir

    var subDir = subDirs(dirId)(subDirId)
    if (subDir == null) {
        subDir = subDirs(dirId).synchronized {
            val old = subDirs(dirId)(subDirId)
            if (old != null) {
                old
            } else {
                val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
                newDir.mkdir()
                subDirs(dirId)(subDirId) = newDir
                newDir
            }
        }
    }

    new File(subDir, filename)
}
```

4.4.3 创建临时 Block 方法 createTempShuffleBlock

当 ShuffleMapTask 运行结束需要把中间结果临时保存，此时就调用 createTempShuffleBlock 方法创建临时的 Block，并返回 TempShuffleBlockId 与其文件的对偶，见代码清单 4-15。

TempShuffleBlockId 的生成规则："temp_shuffle_" 后加上 UUID 字符串。

代码清单4-15 createTempShuffleBlock方法的实现

```
def createTempShuffleBlock(): (TempShuffleBlockId, File) = {
    var blockId = new TempShuffleBlockId(UUID.randomUUID())
    while (getFile(blockId).exists()) {
        blockId = new TempShuffleBlockId(UUID.randomUUID())
    }
    (blockId, getFile(blockId))
}
```

4.5 磁盘存储 DiskStore

当 MemoryStore 没有足够空间时，就会使用 DiskStore 将块存入磁盘。DiskStore 继承自 BlockStore，并实现了 getBytes、putBytes、putArray、putIterator 等方法。

4.5.1 NIO 读取方法 getBytes

getBytes 方法通过 DiskBlockManager 的 getFile 方法获取文件。然后使用 NIO 将文件读取到 ByteBuffer，见代码清单 4-16。

代码清单4-16 getBytes的实现

```
private def getBytes(file: File, offset: Long, length: Long): Option[ByteBuffer] = {
    val channel = new RandomAccessFile(file, "r").getChannel

    try {
        // For small files, directly read rather than memory map
        if (length < minMemoryMapBytes) {
            val buf = ByteBuffer.allocate(length.toInt)
            channel.position(offset)
            while (buf.remaining() != 0) {
                if (channel.read(buf) == -1) {
                    throw new IOException("Reached EOF before filling buffer\n" +
                        s"offset=$offset\nfile=${file.getAbsolutePath}\nbuf.remaining=
                            ${buf.remaining}")
                }
            }
            buf.flip()
            Some(buf)
        } else {
            Some(channel.map(MapMode.READ_ONLY, offset, length))
        }
    } finally {
        channel.close()
    }
}

override def getBytes(blockId: BlockId): Option[ByteBuffer] = {
```

```
        val file = diskManager.getFile(blockId.name)
        getBytes(file, 0, file.length)
    }
```

4.5.2 NIO 写入方法 putBytes

putBytes 方法的作用是通过 DiskBlockManager 的 getFile 方法获取文件。然后使用 NIO 的 Channel 将 ByteBuffer 写入文件，见代码清单 4-17。

代码清单4-17 putBytes的实现

```
override def putBytes(blockId: BlockId, _bytes: ByteBuffer, level: StorageLevel): PutResult = {
    val bytes = _bytes.duplicate()
    logDebug(s"Attempting to put block $blockId")
    val startTime = System.currentTimeMillis
    val file = diskManager.getFile(blockId)
    val channel = new FileOutputStream(file).getChannel
    while (bytes.remaining > 0) {
        channel.write(bytes)
    }
    channel.close()
    val finishTime = System.currentTimeMillis
    logDebug("Block %s stored as %s file on disk in %d ms".format(
        file.getName, Utils.bytesToString(bytes.limit), finishTime - startTime))
    PutResult(bytes.limit(), Right(bytes.duplicate()))
}
```

4.5.3 数组写入方法 putArray

putArray 内部实际调用了 putIterator，代码如下。

```
override def putArray(
        blockId: BlockId,
        values: Array[Any],
        level: StorageLevel,
        returnValues: Boolean): PutResult = {
    putIterator(blockId, values.toIterator, level, returnValues)
}
```

4.5.4 Iterator 写入方法 putIterator

putIterator 的实现见代码清单 4-18，其处理步骤如下：

1）使用了 DiskBlockManager 的 getFile 获取 blockId 对应的 Block 文件，并封装为 FileOutputStream。

2）调用 BlockManager 的 dataSerializeStream 方法，将 FileOutputStream 序列化并压缩。dataSerializeStream 的实现见 4.8.12 节。

3）如果需要返回写入的数据（即 returnValues 等于 true），则将写入的文件使用 getBytes

读取为 ByteBuffer，与文件的长度一并封装到 PutResult 中并返回，否则只返回文件长度。

代码清单4-18 putIterator的实现

```
val startTime = System.currentTimeMillis
val file = diskManager.getFile(blockId)
val outputStream = new FileOutputStream(file)
try {
    try {
        blockManager.dataSerializeStream(blockId, outputStream, values)
    } finally {
        outputStream.close()
    }
} catch {
    case e: Throwable =>
        if (file.exists()) {
            file.delete()
        }
        throw e
}

val length = file.length
val timeTaken = System.currentTimeMillis - startTime

if (returnValues) {
    val buffer = getBytes(blockId).get
    PutResult(length, Right(buffer))
} else {
    PutResult(length, null)
}
```

4.6 内存存储 MemoryStore

MemoryStore 负责将没有序列化的 Java 对象数组或者序列化的 ByteBuffer 存储到内存中。我们先来看看 MemoryStore 的数据结构，见代码清单 4-19。

代码清单4-19 MemoryStore的数据结构

```
private[spark] class MemoryStore(blockManager: BlockManager, maxMemory: Long)
    extends BlockStore(blockManager) {

    private val conf = blockManager.conf
    private val entries = new LinkedHashMap[BlockId, MemoryEntry](32, 0.75f, true)

    @volatile private var currentMemory = 0L

    // Ensure only one thread is putting, and if necessary, dropping blocks at
        any given time
    private val accountingLock = new Object

    private val unrollMemoryMap = mutable.HashMap[Long, Long]()
```

```
    private val maxUnrollMemory: Long = {
        val unrollFraction = conf.getDouble("spark.storage.unrollFraction", 0.2)
        (maxMemory * unrollFraction).toLong
    }

private val unrollMemoryThreshold: Long =
    conf.getLong("spark.storage.unrollMemoryThreshold", 1024 * 1024)

def freeMemory: Long = maxMemory - currentMemory
```

根据代码清单 4-19，我们先来用一张图来说明 MemoryStore 的内存模型，见图 4-3。

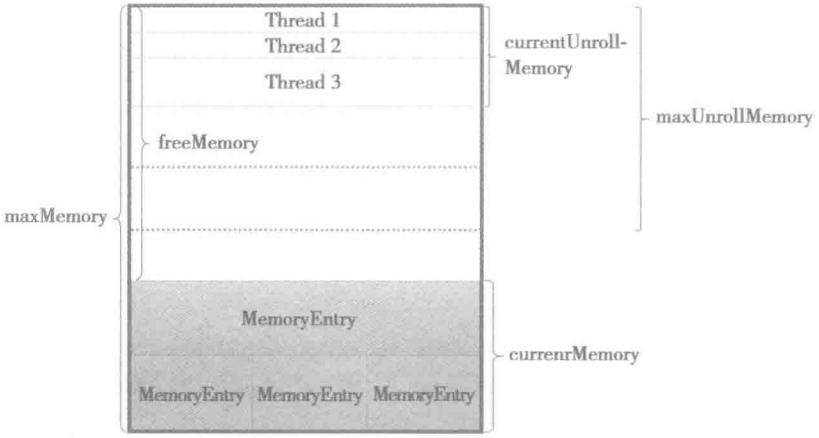


图 4-3 MemoryStore 的内存模型

从图 4-3 中看出，整个 MemoryStore 的存储分为两块：一块是被很多 MemoryEntry 占据的内存 currentMemory，这些 MemoryEntry 实际是通过 entries（即 LinkedHashMap[BlockId, MemoryEntry）持有的；另一块是 unrollMemoryMap 通过占座方式占用的内存 current-UnrollMemory。所谓占座，好比教室里空着的座位，有人在座位上放上书本，以防在需要坐的时候，却发现没有位置了。比起人的行为，unrollMemoryMap 占座的出发点却是“高尚”的，这样可以防止在向内存真正写入数据时，内存不足发生溢出。每个线程实际占用的空间，其实是 vector（即 SizeTrackingVector）占用的大小，但是 unrollMemoryMap 的大小会稍大些。

这里把代码清单 4-19 中的一些概念，结合图 4-3 再解释下：

- maxUnrollMemory：当前 Driver 或者 Executor 最多展开的 Block 所占用的内存，可以修改属性 spark.storage.unrollFraction 改变大小；
- maxMemory：当前 Driver 或者 Executor 的最大内存；
- currentMemory：当前 Driver 或者 Executor 已经使用的内存；
- freeMemory：当前 Driver 或者 Executor 未使用的内存，freeMemory = maxMemory − currentMemory；

- currentUnrollMemory：unrollMemoryMap 中所有展开的 Block 的内存之和，即当前 Driver 或者 Executor 中所有线程展开的 Block 的内存之和；
- unrollMemoryMap：当前 Driver 或者 Executor 中所有线程展开的 Block 都存入此 Map 中，key 为线程 Id，value 为线程展开的所有块的内存的大小总和。

MemoryStore 继承自 BlockStore，并实现了 getBytes、putBytes、putArray、putIterator、getValues 等方法。下面逐个介绍 MemoryStore 中实现的各个方法。

4.6.1 数据存储方法 putBytes

如果 Block 可以被反序列化（即存储级别 StorageLevel.Deserialized 等于 ture），那么先对 Block 序列化，然后调用 putIterator；否则调用 tryToPut 方法，见代码清单 4-20。

代码清单4-20 putBytes的实现

```
override def putBytes(blockId: BlockId, _bytes: ByteBuffer, level: StorageLevel):
    PutResult = {
    // Work on a duplicate - since the original input might be used elsewhere.
    val bytes = _bytes.duplicate()
    bytes.rewind()
    if (level.deserialized) {
        val values = blockManager.dataDeserialize(blockId, bytes)
        putIterator(blockId, values, level, returnValues = true)
    } else {
        val putAttempt = tryToPut(blockId, bytes, bytes.limit, deserialized = false)
        PutResult(bytes.limit(), Right(bytes.duplicate()), putAttempt.droppedBlocks)
    }
}
```

4.6.2 Iterator 写入方法 putIterator 详解

MemoryStore 的 putIterator 方法的实现见代码清单 4-21。调用 unrollSafely 将块在内存中安全展开，如果返回数据的类型匹配 Left(arrayValues)，则说明内存足够并调用 putArray 方法写入内存；如果返回数据的类型匹配 Right(iteratorValues)，则说明内存不足并写入硬盘或者放弃。diskStore.putIterator 的实现见 4.5.4 节。unrollSafely 方法将在 4.6.3 节说明。

代码清单4-21 MemoryStore.putIterator的实现

```
private[storage] def putIterator(
        blockId: BlockId,
        values: Iterator[Any],
        level: StorageLevel,
        returnValues: Boolean,
        allowPersistToDisk: Boolean): PutResult = {
    val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]
    val unrolledValues = unrollSafely(blockId, values, droppedBlocks)
    unrolledValues match {
        case Left(arrayValues) =>
        // Values are fully unrolled in memory, so store them as an array
```

```
            val res = putArray(blockId, arrayValues, level, returnValues)
            droppedBlocks ++= res.droppedBlocks
            PutResult(res.size, res.data, droppedBlocks)
            case Right(iteratorValues) =>
                // Not enough space to unroll this block; drop to disk if applicable
                if (level.useDisk && allowPersistToDisk) {
                logWarning(s"Persisting block $blockId to disk instead.")
                val res = blockManager.diskStore.putIterator(blockId, iteratorValues,
                     level, returnValues)
                PutResult(res.size, res.data, droppedBlocks)
            } else {
                PutResult(0, Left(iteratorValues), droppedBlocks)
            }
        }
    }
```

4.6.3 安全展开方法 unrollSafely

为了防止写入内存的数据过大，导致内存溢出，Spark 采用了一种优化方案：在正式写入内存之前，先用逻辑方式申请内存，如果申请成功，再写入内存，这个过程称为安全展开。表 4-1 列出了安全展开方法 unrollSafely 中一些变量及算法的定义。

表 4-1 unrollSafely 中一些变量及算法的定义

变 量 名	含 义
elementsUnrolled	展开的元素数量
initialMemoryThreshold	每个线程用来展开 Block 的初始内存阈值，可以修改属性 spark.storage.unrollMemoryThreshold 改变大小
memoryCheckPeriod	经过多少次展开内存后，判断是否需要申请更多内存
memoryThreshold	当前线程保留的用于特殊展开操作的内存阈值
memoryGrowthFactor	内存请求因子，与 vector 的大小乘积，减去 memoryThreshold 作为新请求的内存大小
previousMemoryReserved	之前当前线程已经展开的驻留的内存大小，当前线程增加的展开内存，最后会释放
vector	跟踪展开内存信息
maxUnrollMemory	当前 Driver 或者 Executor 最多展开的 Block 所占用的内存
maxMemory	当前 Driver 或者 Executor 的最大内存
currentMemory	当前 Driver 或者 Executor 已经使用的内存
freeMemory	当前 Driver 或者 Executor 未使用的内存：freeMemory = maxMemory – currentMemory
currentUnrollMemory	unrollMemoryMap 中所有展开的 Block 的内存之和，即当前 Driver 或者 Executor 中所有线程展开的 Block 的内存之和
unrollMemoryMap	当前 Driver 或者 Executor 中所有线程展开的 Block 都存入此 Map 中，key 为线程 Id，value 为线程展开的所有块的内存的大小总和
currentSize	vector 中跟踪的对象的总大小
keepUnrolling	标记是否有足够的内存可以继续展开 Block keepUnrolling = freeMemory > currentUnrollMemory + memory（要分配内存）

有了上述概念，下面我们列出展开内存的步骤：

1）申请 memoryThreshold 的初始大小为 initialMemoryThreshold。

2）如果 Iterator[Any] 中有元素并且 keepUnrolling 为 true，则向 vector 中添加 Iterator-[Any] 中的对象，elementsUnrolled 自增 1。如果 Iterator[Any] 中没有元素或者 keepUnrolling 不等于 true，则跳转至第 4）步。

3）如果 elementsUnrolled % memoryCheckPeriod == 0，则开始检查 currentSize 是否已经比 memoryThreshold 大？假如 currentSize 已经超过了 memoryThreshold，则需要再申请内存，申请内存大小 amountToRequest = currentSize * memoryGrowthFactor – memoryThreshold。如果申请失败，但是 maxUnrollMemory > currentUnrollMemory，则要求释放当前 Driver 或者 Executor 的其他内存，具体释放过程在 4.6.4 节讲解。释放内存必然伴随着其他 Block 被移入硬盘或者彻底清除，这些块的状态会在释放后返回。此时再次申请内存，memoryThreshold 增加的大小为 amountToRequest。返回第 2）步。

4）根据是否将 Block 完整地放入内存，以数组或者迭代器形式返回 vector 的数据。

5）最后在 finally 语句块里，还会计算本次展开块实际占用的空间 amountToRelease，并更新 unrollMemoryMap 中当前线程占用的内存大小，并减去 amountToRelease。如果 unrollMemoryMap 中当前线程占用的内存大小小于等于 0，则从 unrollMemoryMap 中完全清除此线程的数据。

对展开 Block 的算法就描述到这里，读者可以结合图 4-3 进行分析。unrollSafely 的实现见代码清单 4-22。

代码清单4-22 unrollSafely的实现

```
def unrollSafely(
        blockId: BlockId,
        values: Iterator[Any],
        droppedBlocks: ArrayBuffer[(BlockId, BlockStatus)])
    : Either[Array[Any], Iterator[Any]] = {

    // Number of elements unrolled so far
    var elementsUnrolled = 0
    // Whether there is still enough memory for us to continue unrolling this
        block
    var keepUnrolling = true
    // Initial per-thread memory to request for unrolling blocks (bytes). Exposed
        for testing.
    val initialMemoryThreshold = unrollMemoryThreshold
    // How often to check whether we need to request more memory
    val memoryCheckPeriod = 16
    // Memory currently reserved by this thread for this particular unrolling
        operation
    var memoryThreshold = initialMemoryThreshold
    // Memory to request as a multiple of current vector size
```

```
    val memoryGrowthFactor = 1.5
    // Previous unroll memory held by this thread, for releasing later (only at
        the very end)
    val previousMemoryReserved = currentUnrollMemoryForThisThread
    // Underlying vector for unrolling the block
    var vector = new SizeTrackingVector[Any]

    // Request enough memory to begin unrolling
    keepUnrolling = reserveUnrollMemoryForThisThread(initialMemoryThreshold)

    // Unroll this block safely, checking whether we have exceeded our threshold
        periodically
    try {
        while (values.hasNext && keepUnrolling) {
            vector += values.next()
            if (elementsUnrolled % memoryCheckPeriod == 0) {
                // If our vector's size has exceeded the threshold, request more memory
                val currentSize = vector.estimateSize()
                if (currentSize >= memoryThreshold) {
                    val amountToRequest = (currentSize * memoryGrowthFactor -
                        memoryThreshold).toLong
                    // Hold the accounting lock, in case another thread concurrently
                        puts a block that
                    // takes up the unrolling space we just ensured here
accountingLock.synchronized {
                        if (!reserveUnrollMemoryForThisThread(amountToRequest)) {
                            // If the first request is not granted, try again after
                                ensuring free space
                            // If there is still not enough space, give up and drop
                                the partition
                            val spaceToEnsure = maxUnrollMemory - currentUnrollMemory
                            if (spaceToEnsure > 0) {
                                val result = ensureFreeSpace(blockId, spaceToEnsure)
                                droppedBlocks ++= result.droppedBlocks
                            }
                            keepUnrolling = reserveUnrollMemoryForThisThread(amo
                                untToRequest)
                        }
                    }
                    // New threshold is currentSize * memoryGrowthFactor
                    memoryThreshold += amountToRequest
                }
            }
            elementsUnrolled += 1
        }

        if (keepUnrolling) {
            // We successfully unrolled the entirety of this block
            Left(vector.toArray)
```

```
            } else {
                // We ran out of space while unrolling the values for this block
                logUnrollFailureMessage(blockId, vector.estimateSize())
                Right(vector.iterator ++ values)
            }
        } finally {
            // If we return an array, the values returned do not depend on the
                underlying vector and
            // we can immediately free up space for other threads. Otherwise, if we
                return an iterator,
            // we release the memory claimed by this thread later on when the task
                finishes.
            if (keepUnrolling) {
                val amountToRelease = currentUnrollMemoryForThisThread -
                    previousMemoryReserved
                releaseUnrollMemoryForThisThread(amountToRelease)
            }
        }
    }
```

unrollSafely 多次用到 reserveUnrollMemoryForThisThread，以便给当前线程申请逻辑内存，它的实现如下。

```
def reserveUnrollMemoryForThisThread(memory: Long): Boolean = {
    accountingLock.synchronized {
        val granted = freeMemory > currentUnrollMemory + memory
        if (granted) {
            val threadId = Thread.currentThread().getId
            unrollMemoryMap(threadId) = unrollMemoryMap.getOrElse(threadId, 0L)
                + memory
        }
        granted
    }
}
```

4.6.4 确认空闲内存方法 ensureFreeSpace

ensureFreeSpace 方法用于确认是否有足够内存，如果不足，会释放被 MemoryEntry 占用的内存。为了叙述方便，通过表 4-2 说明一些变量。

表 4-2 ensureFreeSpace 中的一些变量含义

变 量 名	含 义
actualFreeMemory	实际空闲的内存：actualFreeMemory = freeMemory − currentUnrollMemory
selectedBlocks	已经选择要从内存中腾出的 Block 对应的 BlockId 的数组
selectedMemory	selectedBlocks 中的所有块的总大小
space	需要腾出的内存大小
entries	用于存放 Block，类型为 LinkedHashMap[BlockId, MemoryEntry]
blockIdToAdd	将要添加的 Block 对应的 blockId

有了上述概念，我们现在来看 ensureFreeSpace 的实现，见代码清单 4-23。ensureFreeSpace 的处理步骤如下：

1）space 不能超过 maxMemory 的限制，否则返回。

2）如果 actualFreeMemory 大于等于 space，说明此时已经有充足的内存，不需要释放内存空间，直接返回。如果 actualFreeMemory 小于 space，则说明空闲空间不足，需要释放一部分已经占用的内存。

3）当 actualFreeMemory + selectedMemory < space，则迭代 entries 获得 blockId 和 MemoryEntry。如果 blockIdToAdd 的 rddId 是空或者 blockIdToAdd 的 rddId 不等于 MemoryEntry 对应 blockId 的 rddId[⊖]，则将 blockId 加入 selectedBlocks，selectedMemory 增加 MemoryEntry 的大小。

4）当 actualFreeMemory + selectedMemory ≥ space 时，则说明可以腾出足够的内存空间。此时将 selectedBlocks 中所有的 blockId 对应于 entries 里的 MemoryEntry 取出，通过判断 MemoryEntry 是否可以反序列化，分别转换为 Array[Any] 或者 ByteBuffer，调用 blockManager 的 dropFromMemory 方法，从内存中移除 blockId 及 MemoryEntry，此方法最终返回移除的 Block 的状态。dropFromMemory 的实现将在 4.8.1 节说明。

代码清单4-23 ensureFreeSpace方法的实现

```
private def ensureFreeSpace(
        blockIdToAdd: BlockId,
        space: Long): ResultWithDroppedBlocks = {
    logInfo(s"ensureFreeSpace($space) called with curMem=$currentMemory,
        maxMem=$maxMemory")

    val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]

    if (space > maxMemory) {
        logInfo(s"Will not store $blockIdToAdd as it is larger than our memory
            limit")
        return ResultWithDroppedBlocks(success = false, droppedBlocks)
    }

    // Take into account the amount of memory currently occupied by unrolling
        blocks
    val actualFreeMemory = freeMemory - currentUnrollMemory

    if (actualFreeMemory < space) {
        val rddToAdd = getRddId(blockIdToAdd)
        val selectedBlocks = new ArrayBuffer[BlockId]
        var selectedMemory = 0L

        entries.synchronized {
            val iterator = entries.entrySet().iterator()
```

⊖ 这可以排除当前 RDD 自身在 MemoryStore 中存储的 Block。

```
            while (actualFreeMemory + selectedMemory < space && iterator.hasNext) {
                val pair = iterator.next()
                val blockId = pair.getKey
                if (rddToAdd.isEmpty || rddToAdd != getRddId(blockId)) {
                    selectedBlocks += blockId
                    selectedMemory += pair.getValue.size
                }
            }
        }
        if (actualFreeMemory + selectedMemory >= space) {
            logInfo(s"${selectedBlocks.size} blocks selected for dropping")
            for (blockId <- selectedBlocks) {
                val entry = entries.synchronized { entries.get(blockId) }
                if (entry != null) {
                    val data = if (entry.deserialized) {
                        Left(entry.value.asInstanceOf[Array[Any]])
                    } else {
                        Right(entry.value.asInstanceOf[ByteBuffer].duplicate())
                    }
                    val droppedBlockStatus = blockManager.dropFromMemory(blockId, data)
                    droppedBlockStatus.foreach { status => droppedBlocks +=
                        ((blockId, status)) }
                }
            }
            return ResultWithDroppedBlocks(success = true, droppedBlocks)
        } else {
            logInfo(s"Will not store $blockIdToAdd as it would require dropping
                another block " +
                "from the same RDD")
            return ResultWithDroppedBlocks(success = false, droppedBlocks)
        }
    }
    ResultWithDroppedBlocks(success = true, droppedBlocks)
}
```

4.6.5 内存写入方法 putArray

内存写入方法 putArray 首先对对象大小进行估算，然后写入内存。如果 unrollSafely 返回的数据匹配 Left（arrayValues），根据前面的分析知道，整个 Block 是可以一次性放入内存的。此时调用 putArray 方法，见代码清单 4-24。

代码清单4-24 putArray的实现

```
override def putArray(
        blockId: BlockId,
        values: Array[Any],
        level: StorageLevel,
        returnValues: Boolean): PutResult = {
    if (level.deserialized) {
        val sizeEstimate = SizeEstimator.estimate(values.asInstanceOf[AnyRef])
        val putAttempt = tryToPut(blockId, values, sizeEstimate, deserialized =
            true)
        PutResult(sizeEstimate, Left(values.iterator), putAttempt.droppedBlocks)
```

```
        } else {
            val bytes = blockManager.dataSerialize(blockId, values.iterator)
            val putAttempt = tryToPut(blockId, bytes, bytes.limit, deserialized =
                false)
            PutResult(bytes.limit(), Right(bytes.duplicate()), putAttempt.droppedBlocks)
        }
    }
```

SizeEstimator.estimate 用来估算对象的大小，遍历对象及其属性。估算对象大小见代码清单 4-25。

代码清单4-25 估算对象大小的方法estimate

```
def estimate(obj: AnyRef): Long = estimate(obj, new IdentityHashMap[AnyRef,
    AnyRef])

private def estimate(obj: AnyRef, visited: IdentityHashMap[AnyRef, AnyRef]): Long
    = {
    val state = new SearchState(visited)
    state.enqueue(obj)
    while (!state.isFinished) {
        visitSingleObject(state.dequeue(), state)
    }
    state.size
}

private def visitSingleObject(obj: AnyRef, state: SearchState) {
    val cls = obj.getClass
    if (cls.isArray) {
        visitArray(obj, cls, state)
    } else if (obj.isInstanceOf[ClassLoader] || obj.isInstanceOf[Class[_]]) {
    } else {
        val classInfo = getClassInfo(cls)
        state.size += classInfo.shellSize
        for (field <- classInfo.pointerFields) {
            state.enqueue(field.get(obj))
        }
    }
}
```

4.6.6 尝试写入内存方法 tryToPut

根据 4.6.1 节的内容我们知道，当 Block 不支持序列化时，会调用 tryToPut 方法。在介绍 MemoryStore 的 putArray 方法时，也最终使用此方法。

tryToPut 的实现见代码清单 4-26。可以看到 tryToPut 也调用了 ensureFreeSpace 方法，记得在调用 tryToPut 之前，已经在 unrollSafely 方法中调用过 ensureFreeSpace 了，这难道是重复调用了同一份代码吗？不，因为在展开阶段，即便内存充足，当真正写入数据时依然可能内存不足，所以需要再次确认空闲内存是否充足。

如果内存充足或者迁移其他内存 Block 后有足够内存，则会创建 MemoryEntry 对象，并

将此对象与其 blockId 放入 entries 中，currentMemory 会上浮估算的大小 size。如果此时内存不足，还要把此 blockId 对应的 MemoryEntry 对象迁移到磁盘或者清除。

代码清单4-26 tryToPut的实现

```
private def tryToPut(
        blockId: BlockId,
        value: Any,
        size: Long,
        deserialized: Boolean): ResultWithDroppedBlocks = {

    var putSuccess = false
    val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]

    accountingLock.synchronized {
        val freeSpaceResult = ensureFreeSpace(blockId, size)
        val enoughFreeSpace = freeSpaceResult.success
        droppedBlocks ++= freeSpaceResult.droppedBlocks

        if (enoughFreeSpace) {
            val entry = new MemoryEntry(value, size, deserialized)
            entries.synchronized {
                entries.put(blockId, entry)
                currentMemory += size
            }
            val valuesOrBytes = if (deserialized) "values" else "bytes"
            logInfo("Block %s stored as %s in memory (estimated size %s, free
                %s)".format(
                blockId, valuesOrBytes, Utils.bytesToString(size), Utils.
                    bytesToString(freeMemory)))
            putSuccess = true
        } else {
            // Tell the block manager that we couldn't put it in memory so that it
                can drop it to
            // disk if the block allows disk storage.
            val data = if (deserialized) {
                Left(value.asInstanceOf[Array[Any]])
            } else {
                Right(value.asInstanceOf[ByteBuffer].duplicate())
            }
            val droppedBlockStatus = blockManager.dropFromMemory(blockId, data)
            droppedBlockStatus.foreach { status => droppedBlocks += ((blockId, status)) }
        }
    }
    ResultWithDroppedBlocks(putSuccess, droppedBlocks)
}
```

4.6.7 获取内存数据方法 getBytes

getBytes 方法用于从 entries 中获取 MemoryEntry，见代码清单 4-27。

代码清单4-27 getBytes的实现

```
override def getBytes(blockId: BlockId): Option[ByteBuffer] = {
    val entry = entries.synchronized {
        entries.get(blockId)
    }
    if (entry == null) {
        None
    } else if (entry.deserialized) {
        Some(blockManager.dataSerialize(blockId, entry.value.asInstanceOf
            [Array[Any]].iterator))
    } else {
        Some(entry.value.asInstanceOf[ByteBuffer].duplicate()) // Doesn't
            actually copy the data
    }
}
```

从代码清单4-27看到，如果MemoryEntry支持反序列化，则将MemoryEntry的value反序列化后反回，否则对MemoryEntny的value复制后返回。

4.6.8 获取数据方法getValues

getValues也用于从内存中获取数据，即从entries中获取MemoryEntry，并将blockId和value返回，见代码清单4-28。

代码清单4-28 getValues的实现

```
override def getValues(blockId: BlockId): Option[Iterator[Any]] = {
    val entry = entries.synchronized {
        entries.get(blockId)
    }
    if (entry == null) {
        None
    } else if (entry.deserialized) {
        Some(entry.value.asInstanceOf[Array[Any]].iterator)
    } else {
        val buffer = entry.value.asInstanceOf[ByteBuffer].duplicate() // Doesn't
            actually copy data
        Some(blockManager.dataDeserialize(blockId, buffer))
    }
}
```

4.7 Tachyon存储TachyonStore

为什么要使用Tachyon？原因如下：

- Spark的ShuffleMapTask和ResultTask被划分到不同Stage，ShuffleMapTask执行完毕将中间结果输出到本地磁盘文件系统（如HDFS），然后下一Stage中的ResultTask通

过 shuffleClient 下载 ShuffleMapTask 的输出到本地磁盘文件系统，这种基于磁盘的读写效率较低；

- Spark 的计算引擎与存储体系都位于 Executor 的同一进程中，当计算执行崩溃出错后，存储体系缓存的数据也会全部丢失；
- 不同的 Spark 任务可能会访问同样的数据，例如两个任务都要访问 HDFS 中的某些 Block，每个任务都要自己去磁盘加载数据到内存中。这导致数据被重复加载到内存，数据对象太多会导致 Java GC 时间过长等问题。

4.7.1 Tachyon 简介

Tachyon 也诞生于 UCBerkeley 的 AMP 实验室，是以内存为中心的高容错的分布式文件系统，能够为集群框架（比如 Spark、Map-Reduce 等）提供可靠的内存级的文件共享服务。从软件栈的层次来看，Tachyon 是位于现有大数据计算框架和大数据存储系统之间的独立的一层，如图 4-4 所示。它利用底层文件系统作为备份，对于上层应用来说，Tachyon 就是一个分布式文件系统。

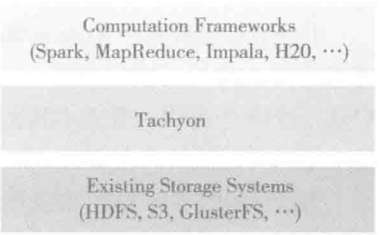


图 4-4 Tachyon 与计算框架、存储系统的层次关系

Tachyon 属于伯克利大数据分析软件栈（Berkeley Data Analytics Stack）中的存储层软件，如图 4-5 所示。

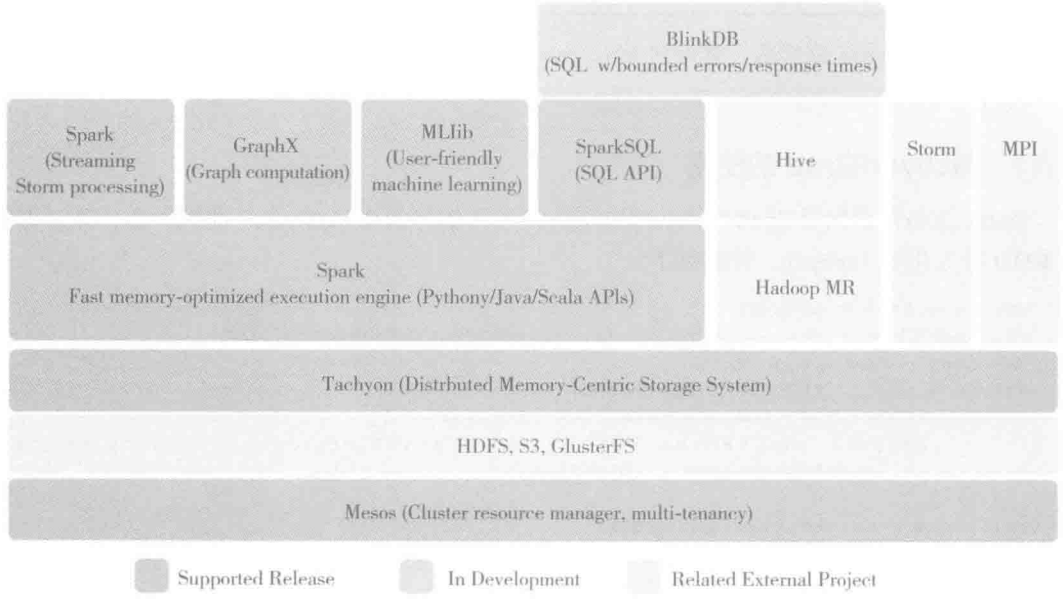


图 4-5 伯克利大数据分析软件栈

Tachyon 的整体架构如图 4-6 所示。

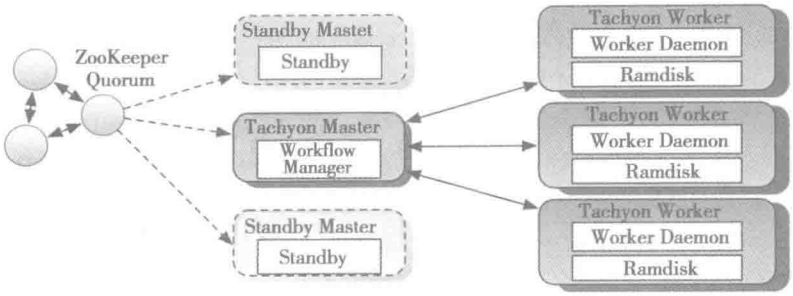


图 4-6 Tachyon 的整体架构

Tachyon 也采用了 Master-Worker 的架构。Tachyon Master 支持 ZooKeeper 进行容错，用于管理全部文件的元数据信息，同时也监控各个 Tachyon Worker 的状态。每个 Tachyon Worker 启动一个守护进程，管理本地的 Ramdisk，Ramdisk 中存储了具体的文件数据。Ramdisk 实际是 Tachyon 集群的内存部分。

Tachyon 的容错处理是怎样的？ Tachyon 内存中的数据如果不落地岂不是依然有丢失问题？ Tachyon 采用与 Spark 的 RDD 相类似（都是基于 RDD 不可变性以及粗粒度操作）的方法，它利用 lineage 信息（lineage-based recovery）和异步记录的 checkpoint 来恢复数据，所以可以放心大胆地使用 Tachyon 管理的内存。


注意 更多 Tachyon 的信息，请访问 http://tachyon-project.org/。

4.7.2 TachyonStore 的使用

Spark 源码自带例子 SparkTachyonHdfsLR 演示了如何使用 Tachyon，此例子在计算过程中将 RDD 持久化到 Tachyon，代码如下。

```
val inputPath = args(0)
val sparkConf = new SparkConf().setAppName("SparkTachyonHdfsLR")
val conf = new Configuration()
val sc = new SparkContext(sparkConf,
    InputFormatInfo.computePreferredLocations(
        Seq(new InputFormatInfo(conf, classOf[org.apache.hadoop.mapred.
            TextInputFormat], inputPath))
    ))
val lines = sc.textFile(inputPath)
val points = lines.map(parsePoint _).persist(StorageLevel.OFF_HEAP)
val ITERATIONS = args(1).toInt

// Initialize w to a random value
```

```
var w = DenseVector.fill(D){2 * rand.nextDouble - 1}
println("Initial w: " + w)

for (i <- 1 to ITERATIONS) {
    println("On iteration " + i)
    val gradient = points.map { p =>
        p.x * (1 / (1 + exp(-p.y * (w.dot(p.x)))) - 1) * p.y
    }.reduce(_ + _)
    w -= gradient
}

println("Final w: " + w)
sc.stop()
```

4.7.3 写入 Tachyon 内存的方法 putIntoTachyonStore

TachyonStore 也实现了 BlockStore 的 getSize、putBytes、putArray、putIterator、getValues、getBytes 等方法。其中 putBytes、putArray、putIterator 实际都调用了 putIntoTachyonStore，PutIntoTachyonStore 用于将数据写入 Tachyon 的分布式内存中。putIntoTachyonStore 的实现见代码清单 4-29。

代码清单4-29 putIntoTachyonStore的实现

```
private def putIntoTachyonStore(
        blockId: BlockId,
        bytes: ByteBuffer,
        returnValues: Boolean): PutResult = {
    val byteBuffer = bytes.duplicate()
    byteBuffer.rewind()
    logDebug(s"Attempting to put block $blockId into Tachyon")
    val startTime = System.currentTimeMillis
    val file = tachyonManager.getFile(blockId)
    val os = file.getOutStream(WriteType.TRY_CACHE)
    os.write(byteBuffer.array())
    os.close()
    val finishTime = System.currentTimeMillis
    logDebug("Block %s stored as %s file in Tachyon in %d ms".format(
        blockId, Utils.bytesToString(byteBuffer.limit), finishTime - startTime))

    if (returnValues) {
        PutResult(bytes.limit(), Right(bytes.duplicate()))
    } else {
        PutResult(bytes.limit(), null)
    }
}
```

4.7.4 获取序列化数据方法 getBytes

getValues 方法实际也调用了 getBytes，getBytes 的实现见代码清单 4-30。其中 Tachyon-

BlockManager 的 getFile 的代码实现与 DiskBlockManager 非常类似，有兴趣的读者可以自己找资料阅读。

代码清单4-30 getBytes的实现

```
override def getBytes(blockId: BlockId): Option[ByteBuffer] = {
    val file = tachyonManager.getFile(blockId)
    if (file == null || file.getLocationHosts.size == 0) {
        return None
    }
    val is = file.getInStream(ReadType.CACHE)
    assert (is != null)
    try {
        val size = file.length
        val bs = new Array[Byte](size.asInstanceOf[Int])
        ByteStreams.readFully(is, bs)
        Some(ByteBuffer.wrap(bs))
    } catch {
        case ioe: IOException =>
            logWarning(s"Failed to fetch the block $blockId from Tachyon", ioe)
            None
    } finally {
        is.close()
    }
}
```

4.8 块管理器 BlockManager

已经介绍了 BlockManager 中的主要组件，现在来看看 BlockManager 自身的实现。

4.8.1 移出内存方法 dropFromMemory

当内存不足时，可能需要腾出部分内存空间。dropFromMemory 实现了这个功能，见代码清单 4-31，它的处理步骤如下。

1）从 blockInfo：TimeStampedHashMap[BlockId, BlockInfo] 中检查是否存在要迁移的 blockId。如果存在，从 BlockInfo 中获取 Block 的 StorageLevel。

2）如果 StorageLevel 允许存入硬盘，并且 DiskStore 中不存在此文件，那么调用 DiskStore 的 putArray 或者 putBytes 方法，将此 Block 存入硬盘。

3）从 MemoryStore 中清除此 BlockId 对应的 Block。

4）使用 getCurrentBlockStatus 方法获取 Block 的最新状态。如果此 Block 的 tellMaster 属性为 true，则调用 reportBlockStatus 方法给 BlockManagerMasterActor 报告状态。reportBlockStatus 在 4.8.2 节描述。

5）从 blockInfo 中清除此 BlockId，并返回 Block 的状态。

代码清单4-31 移出内存的实现

```
def dropFromMemory(
        blockId: BlockId,
        data: Either[Array[Any], ByteBuffer]): Option[BlockStatus] = {

    logInfo(s"Dropping block $blockId from memory")
    val info = blockInfo.get(blockId).orNull

    // If the block has not already been dropped
    if (info != null) {
        info.synchronized {
            var blockIsUpdated = false
            val level = info.level

            // Drop to disk, if storage level requires
            if (level.useDisk && !diskStore.contains(blockId)) {
                logInfo(s"Writing block $blockId to disk")
                data match {
                    case Left(elements) =>
                        diskStore.putArray(blockId, elements, level, returnValues = false)
                    case Right(bytes) =>
                        diskStore.putBytes(blockId, bytes, level)
                }
                blockIsUpdated = true
            }

            // Actually drop from memory store
            val droppedMemorySize =
                if (memoryStore.contains(blockId)) memoryStore.getSize(blockId) else 0L
            val blockIsRemoved = memoryStore.remove(blockId)
            if (blockIsRemoved) {
                blockIsUpdated = true
            } else {
                logWarning(s"Block $blockId could not be dropped from memory as
                    it does not exist")
            }
            val status = getCurrentBlockStatus(blockId, info)
            if (info.tellMaster) {
                reportBlockStatus(blockId, info, status, droppedMemorySize)
            }
            if (!level.useDisk) {
                // The block is completely gone from this node; forget it so we
                     can put() it again later.
                blockInfo.remove(blockId)
            }
            if (blockIsUpdated) {
                return Some(status)
            }
        }
    }
    None
}
```

4.8.2 状态报告方法 reportBlockStatus

reportBlockStatus 用于向 BlockManagerMasterActor 报告 Block 的状态并且重新注册 BlockManager。reportBlockStatus 的实现见代码清单 4-32。它的处理步骤如下。

1）调用 tryToReportBlockStatus 方法，tryToReportBlockStatus 调用了 BlockManagerMaster 的 updateBlockInfo 方法向 BlockManagerMasterActor 发送 UpdateBlockInfo 消息更新 Block 占用的内存大小、磁盘大小、存储级别等信息。

2）如果 BlockManager 还没有向 BlockManagerMasterActor 注册，则调用 asyncReregister 方法，asyncReregister 调用了 reregister，最后 reregister 实际调用了 BlockManagerMaster 的 registerBlockManager 方法和 reportAllBlocks 方法，reportAllBlocks 方法实际也是调用了 tryToReportBlockStatus。

代码清单4-32 状态报告的实现

```
private def reportBlockStatus(blockId: BlockId, info: BlockInfo,
        status: BlockStatus, droppedMemorySize: Long = 0L): Unit = {
    val needReregister = !tryToReportBlockStatus(blockId, info, status,
         droppedMemorySize)
    if (needReregister) {
        logInfo(s"Got told to re-register updating block $blockId")
        asyncReregister()
    }
        logDebug(s"Told master about block $blockId")
}

private def tryToReportBlockStatus(blockId: BlockId, info: BlockInfo,
        status: BlockStatus, droppedMemorySize: Long = 0L): Boolean = {
    if (info.tellMaster) {
        val storageLevel = status.storageLevel
        val inMemSize = Math.max(status.memSize, droppedMemorySize)
        val inTachyonSize = status.tachyonSize
        val onDiskSize = status.diskSize
        master.updateBlockInfo(
            blockManagerId, blockId, storageLevel, inMemSize, onDiskSize,
                inTachyonSize)
    } else {
        true
        }
    }

def reregister(): Unit = {
    logInfo("BlockManager re-registering with master")
    master.registerBlockManager(blockManagerId, maxMemory, slaveActor)
    reportAllBlocks()
}

private def asyncReregister(): Unit = {
    asyncReregisterLock.synchronized {
        if (asyncReregisterTask == null) {
```

```
                asyncReregisterTask = Future[Unit] {
                    reregister()
                    asyncReregisterLock.synchronized {
                        asyncReregisterTask = null
                    }
                }
            }
        }
    }
```

RegisterBlockManager 消息的处理已在 4.3 节介绍过，不再赘述。我们来看看 BlockManagerMaster 的 updateBlockInfo，代码如下。

```
val res = askDriverWithReply[Boolean](
    UpdateBlockInfo(blockManagerId, blockId, storageLevel, memSize, diskSize,
        tachyonSize))
logInfo("Updated info of block " + blockId)
res
```

从上面 updateBlockInfo 方法的实现中发现它也调用了我们熟悉的 askDriverWithReply 方法，只不过消息是 UpdateBlockInfo。BlockManagerMasterActor 接收后会调用 updateBlockInfo 方法更新 blockManagerInfo 及 blockLocations 等信息，比较简单，读者可自行分析。如果不熟悉 askDriverWithReply，可以回顾 4.3.2 节。

4.8.3 单对象块写入方法 putSingle

putSingle 方法用于将由一个对象构成的 Block 写入存储系统。putSingle 经过层层调用，实际调用了 doPut 方法，见代码清单 4-33。doPut 方法将在 4.8.5 节说明。

代码清单4-33 putSingle的实现

```
def putSingle(
        blockId: BlockId,
        value: Any,
        level: StorageLevel,
        tellMaster: Boolean = true): Seq[(BlockId, BlockStatus)] = {
    putIterator(blockId, Iterator(value), level, tellMaster)
}

def putIterator(
        blockId: BlockId, values: Iterator[Any], level: StorageLevel,
        tellMaster: Boolean = true,
        effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId,
            BlockStatus)] = {
    require(values != null, "Values is null")
    doPut(blockId, IteratorValues(values), level, tellMaster, effective-
        StorageLevel)
}
```

4.8.4 序列化字节块写入方法 putBytes

putBytes 方法用于将序列化字节组成的 Block 写入存储系统，putBytes 实际也调用了 doPut 方法，见代码清单 4-34。

代码清单4-34 putBytes的实现

```
def putBytes(
        blockId: BlockId,
        bytes: ByteBuffer,
        level: StorageLevel,
        tellMaster: Boolean = true,
        effectiveStorageLevel: Option[StorageLevel] = None): Seq[(BlockId,
            BlockStatus)] = {
    require(bytes != null, "Bytes is null")
    doPut(blockId, ByteBufferValues(bytes), level, tellMaster, effective-
        StorageLevel)
}
```

4.8.5 数据写入方法 doPut

putSingle、putBytes 等方法真正的数据写入实际由 doPut 实现，doPut 的处理流程如图 4-7 所示。

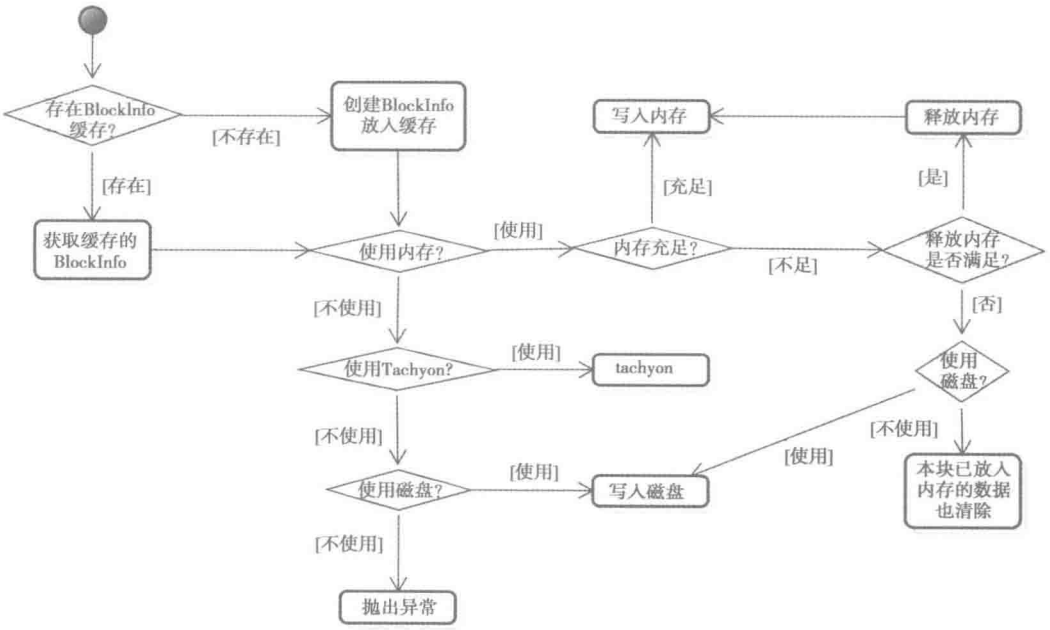


图 4-7 doPut 的处理流程

doPut 的处理步骤如下。

1）获取 putBlockInfo。如果 blockInfo 中已经缓存了 BlockInfo，则使用缓存的 BlockInfo，否则使用新建的 BlockInfo。获取 PutBlockInfo 的实现，见代码清单 4-35。

代码清单4-35 获取putBlockInfo的代码实现

```
val updatedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]

val putBlockInfo = {
    val tinfo = new BlockInfo(level, tellMaster)
    // Do atomically !
    val oldBlockOpt = blockInfo.putIfAbsent(blockId, tinfo)
    if (oldBlockOpt.isDefined) {
        if (oldBlockOpt.get.waitForReady()) {
            logWarning(s"Block $blockId already exists on this machine; not re-
                adding it")
            return updatedBlocks
        }
        oldBlockOpt.get
    } else {
        tinfo
    }
}
```

2）获取块最终使用的存储级别 putLevel，根据 putLevel 判断块写入的 BlockStore，从代码清单 4-36 可以看出，优先使用 MemoryStore，其次是 TachyonStore 和 DiskStore。依据 data 的实际包装类型，分别调用 BlockStore 不同的方法，如 putIterator、putArray、putBytes 等。

代码清单4-36 获取块最终使用的存储级别

```
// The level we actually use to put the block
val putLevel = effectiveStorageLevel.getOrElse(level)
val (returnValues, blockStore: BlockStore) = {
    if (putLevel.useMemory) {
        // Put it in memory first, even if it also has useDisk set to true;
        // We will drop it to disk later if the memory store can't hold it.
        (true, memoryStore)
    } else if (putLevel.useOffHeap) {
        // Use tachyon for off-heap storage
        (false, tachyonStore)
    } else if (putLevel.useDisk) {
        // Don't get back the bytes from put unless we replicate them
        (putLevel.replication > 1, diskStore)
    } else {
        assert(putLevel == StorageLevel.NONE)
        throw new BlockException(
            blockId, s"Attempted to put block $blockId without specifying storage level!")
    }
}

// Actually put the values
val result = data match {
    case IteratorValues(iterator) =>
        blockStore.putIterator(blockId, iterator, putLevel, returnValues)
    case ArrayValues(array) =>
```

```
            blockStore.putArray(blockId, array, putLevel, returnValues)
        case ByteBufferValues(bytes) =>
            bytes.rewind()
            blockStore.putBytes(blockId, bytes, putLevel)
    }
    size = result.size
    result.data match {
        case Left (newIterator) if putLevel.useMemory => valuesAfterPut = newIterator
        case Right (newBytes) => bytesAfterPut = newBytes
        case _ =>
    }
```

3）写入完毕后，将写入操作导致从内存 drop 掉的 Block 更新到 updatedBlocks：ArrayBuffer[(BlockId, BlockStatus)] 中。使用 getCurrentBlockStatus 获取写入 Block 的状态。将 putBlockInfo 设置为允许其他线程读取，调用 reportBlockStatus 将当前 Block 的信息更新到 BlockManagerMasterActor，最后将 putBlockInfo 添加到 updatedBlocks 中，见代码清单 4-37。updatedBlocks 中的 Block 的状态由于都发生了变化，所以都需要向 BlockManagerMasterActor 发送 updateBlockInfo 消息。

代码清单4-37 整理需要更新的Block

```
// Keep track of which blocks are dropped from memory
if (putLevel.useMemory) {
    result.droppedBlocks.foreach { updatedBlocks += _ }
}

val putBlockStatus = getCurrentBlockStatus(blockId, putBlockInfo)
if (putBlockStatus.storageLevel != StorageLevel.NONE) {
    marked = true
    putBlockInfo.markReady(size)
    if (tellMaster) {
        reportBlockStatus(blockId, putBlockInfo, putBlockStatus)
    }
updatedBlocks += ((blockId, putBlockStatus))
```

4）如果 putLevel.replication 大于 1，即为了容错考虑，数据的备份数量大于 1 的时候，需要将 Block 的数据备份到其他节点上，见代码清单 4-38。备份工作由 replicate 完成，请阅读 4.8.6 节。

代码清单4-38 Block的备份

```
if (putLevel.replication > 1) {
    data match {
        case ByteBufferValues(bytes) =>
        if (replicationFuture != null) {
            Await.ready(replicationFuture, Duration.Inf)
        }
        case _ =>
            val remoteStartTime = System.currentTimeMillis
```

```
            // Serialize the block if not already done
            if (bytesAfterPut == null) {
                if (valuesAfterPut == null) {
                throw new SparkException(
                    "Underlying put returned neither an Iterator nor bytes! This
                        shouldn't happen.")
                }
                bytesAfterPut = dataSerialize(blockId, valuesAfterPut)
            }
            replicate(blockId, bytesAfterPut, putLevel)
            logDebug("Put block %s remotely took %s"
                .format(blockId, Utils.getUsedTimeMs(remoteStartTime)))
        }
    }

    BlockManager.dispose(bytesAfterPut)

    updatedBlocks
```

4.8.6 数据块备份方法 replicate

在介绍 replicate 方法之前，先对其中的一些定义进行解释，见表 4-3。

表 4-3 replicate 中的定义

变 量 名	含 义
maxReplicationFailures	最大复制失败次数
numPeersToReplicateTo	需要复制的备份数
peersForReplication	可以作为备份的 BlockManager 的缓存
peersReplicatedTo	已经作为备份的 BlockManager 的缓存
peersFailedToReplicateTo	已经复制失败的 BlockManager 的缓存
replicationFailed	标记复制是否失败
failures	复制失败次数
done	标记复制是否完成

为了容灾，peersForReplication 中缓存的 BlockManager 不应当是当前的 BlockManager。获取其他所有 BlockManager 的方法是 getPeers，见代码清单 4-39。getPeers 方法中的定义见表 4-4。

代码清单4-39 getPeers的实现

```
private def getPeers(forceFetch: Boolean): Seq[BlockManagerId] = {
    peerFetchLock.synchronized {
        val cachedPeersTtl = conf.getInt("spark.storage.cachedPeersTtl", 60 *
            1000) // milliseconds
        val timeout = System.currentTimeMillis - lastPeerFetchTime > cachedPeersTtl
        if (cachedPeers == null || forceFetch || timeout) {
            cachedPeers = master.getPeers(blockManagerId).sortBy(_.hashCode)
            lastPeerFetchTime = System.currentTimeMillis
```

```
            logDebug("Fetched peers from master: " + cachedPeers.mkString("[",
                ",", "]"))
        }
        cachedPeers
    }
}
```

表 4-4 getPeers 方法中的定义

变 量 名	含 义
cachedPeers: Seq[BlockManagerId]	当前 BlockManager 缓存的 BlockManagerId
cachedPeersTtl	cachedPeers 缓存的超时时间，默认为 60 秒，可以修改属性 spark.storage.cachedPeersTtl 改变大小
forceFetch	标记是否强制从 BlockManagerMasterActor 获取最新的 BlockManagerId

当 cachedPeers 为空或者 forceFetch 为 true 或者当前时间超时，则会调用 BlockManagerMaster 的 getPeers 方法，从 BlockManagerMasterActor 获取最新的 BlockManagerId。

BlockManagerMaster 的 getPeers 方法也调用了我们熟悉的 askDriverWithReply，代码如下。

```
def getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId] = {
    askDriverWithReply[Seq[BlockManagerId]](GetPeers(blockManagerId))
}
```

BlockManagerMasterActor 匹配 GetPeers 消息将执行 getPeers 方法，getPeers 从 BlockManagerInfo 中过滤掉 Driver 的 BlockManager 和当前的 Executor 的 BlockManager，将其余的 BlockManagerId 都返回，见代码清单 4-40。

代码清单4-40 BlockManagerMasterActor. getPeers的实现

```
private def getPeers(blockManagerId: BlockManagerId): Seq[BlockManagerId] = {
    val blockManagerIds = blockManagerInfo.keySet
    if (blockManagerIds.contains(blockManagerId)) {
        blockManagerIds.filterNot { _.isDriver }.filterNot { _ == blockManagerId
            }.toSeq
    } else {
        Seq.empty
    }
}
```

replicate 方法的实现见代码清单 4-41。replicate 有个内部函数 getRandomPeer，用于随机获取 BlockManagerId。由于 random : Random(blockId.hashCode) 使用 blockId 的哈希值，这样就保证在同一个节点上多次尝试复制同一个 Block，保证它始终被复制到同一批节点上。特别注意的是，当复制失败并且再次尝试时，会强制从 BlockManagerMasterActor 获取所有最新的 BlockManagerId。并且从 peersForReplication 中排除 peersReplicatedTo 和 peersFailedToReplicateTo，即排除已经使用和已经复制失败的 BlockManager 的 BlockManagerId。

代码清单4-41 replicate方法的实现

```
private def replicate(blockId: BlockId, data: ByteBuffer, level: StorageLevel):
    Unit = {
    // 常量、变量定义省略
    val random = new Random(blockId.hashCode)
    def getRandomPeer(): Option[BlockManagerId] = {
        if (replicationFailed) {
            peersForReplication.clear()
            peersForReplication ++= getPeers(forceFetch = true)
            peersForReplication --= peersReplicatedTo
            peersForReplication --= peersFailedToReplicateTo
        }
        if (!peersForReplication.isEmpty) {
            Some(peersForReplication(random.nextInt(peersForReplication.size)))
        } else {
            None
        }
}
```

现在我们正式讲解备份复制的过程，见代码清单4-42。通过图4-8能够帮助我们理解其处理步骤。

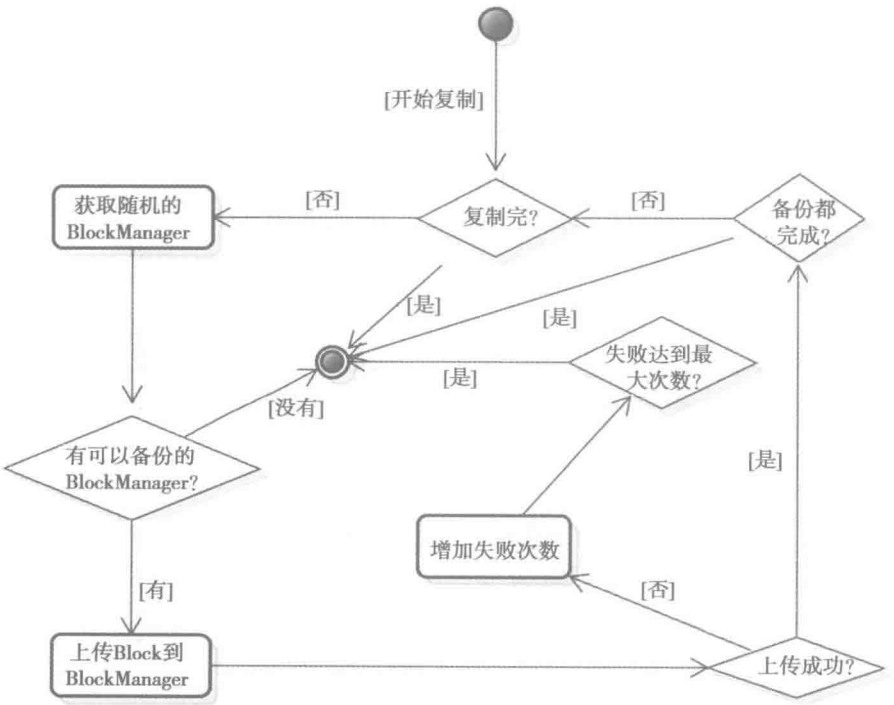


图4-8 Block备份复制的过程

代码清单4-42 备份复制的实现

```
while (!done) {
    getRandomPeer() match {
        case Some(peer) =>
            try {
                val onePeerStartTime = System.currentTimeMillis
                data.rewind()
                logTrace(s"Trying to replicate $blockId of ${data.limit()} bytes
                    to $peer")
                blockTransferService.uploadBlockSync(
                peer.host, peer.port, peer.executorId, blockId, new NioManaged-
                    Buffer(data), tLevel)
                logTrace(s"Replicated $blockId of ${data.limit()} bytes to $peer
                    in %s ms"
                    .format(System.currentTimeMillis - onePeerStartTime))
                peersReplicatedTo += peer
                peersForReplication -= peer
                replicationFailed = false
                if (peersReplicatedTo.size == numPeersToReplicateTo) {
                    done = true  // specified number of peers have been replicated to
                }
            } catch {
                case e: Exception =>
                    logWarning(s"Failed to replicate $blockId to $peer, failure
                        #$failures", e)
                    failures += 1
                    replicationFailed = true
                    peersFailedToReplicateTo += peer
                    if (failures > maxReplicationFailures) { // too many failures
                         in replcating to peers
                        done = true
                }
            }
        case None => // no peer left to replicate to
            done = true
    }
}
```

从 replicate 的代码实现，总结复制的过程如下。

1）调用 getRandomPeer 随机获取 BlockManager。

2）上传 Block 到 BlockManager。

3）如果上传成功，则将此 BlockManager 添加到 peersReplicatedTo，而从 peersForReplication 中移除，设置 replicationFailed 等于 false，done 等于 true；如果上传过程出现异常，则将此 BlockManager 添加到 peersFailedToReplicateTo，failures 自增，设置 replicationFailed 等于 true，done 等于 false。

如果上传失败，以上过程会迭代多次，直到失败次数 failures 超过最大失败次数 maxReplicationFailures。

异步上传方法 uploadBlockSync 实际是通过调用 blockTransferService. uploadBlock 来完成

的，代码如下。NettyBlockTransferService 的 uploadBlock 方法已在 4.2.6 节介绍过。

```
def uploadBlockSync(hostname: String, port: Int, execId: String, blockId:
BlockId,
        blockData: ManagedBuffer, level: StorageLevel): Unit = {
    Await.result(uploadBlock(hostname, port, execId, blockId, blockData, level),
        Duration.Inf)
}
```

4.8.7 创建 DiskBlockObjectWriter 的方法 getDiskWriter

getDiskWriter 用于创建 DiskBlockObjectWriter，见代码清单 4-43。属性 spark.shuffle.sync 决定写操作是同步还是异步。其中有关 wrapForCompression 方法的内容请阅读 4.8.12 节。

代码清单4-43 创建DiskBlockObjectWriter

```
def getDiskWriter(
        blockId: BlockId,
        file: File,
        serializer: Serializer,
        bufferSize: Int,
        writeMetrics: ShuffleWriteMetrics): BlockObjectWriter = {
    val compressStream: OutputStream => OutputStream = wrapForCompression(blockId,
        _)
    val syncWrites = conf.getBoolean("spark.shuffle.sync", false)
    new DiskBlockObjectWriter(blockId, file, serializer, bufferSize,
        compressStream, syncWrites,
        writeMetrics)
}
```

4.8.8 获取本地 Block 数据方法 getBlockData

getBlockData 用于从本地获取 Block 的数据，见代码清单 4-44。

代码清单4-44 从本地获取Block

```
override def getBlockData(blockId: BlockId): ManagedBuffer = {
    if (blockId.isShuffle) {
        shuffleManager.shuffleBlockManager.getBlockData(blockId.asInstanceOf
            [ShuffleBlockId])
    } else {
        val blockBytesOpt = doGetLocal(blockId, asBlockResult = false)
            .asInstanceOf[Option[ByteBuffer]]
        if (blockBytesOpt.isDefined) {
            val buffer = blockBytesOpt.get
            new NioManagedBuffer(buffer)
        } else {
            throw new BlockNotFoundException(blockId.toString)
        }
    }
}
```

getBlockData 的处理实现如下。

1）如果 Block 是 ShuffleMapTask 的输出，那么多个 partition 的中间结果都写入了同一个文件，怎样读取不同 partition 的中间结果？ IndexShuffleBlockManager 的 getBlockData 方法解决了这个问题，请阅读 4.13 节。

2）如果 Block 是 ResultTask 的输出，则使用 doGetLocal 来获取本地中间结果数据，请参阅 4.8.9 节。

4.8.9 获取本地 shuffle 数据方法 doGetLocal

当 reduce 任务与 map 任务处于同一节点时，不需要远程拉取，只需调取 doGetLocal 方法从本地获取中间处理结果即可。doGetLocal 的实现见代码清单 4-45，其处理步骤如下：

1）如果 Block 允许使用内存，则调用 MemoryStore 的 getValues 或者 getBytes 方法获取。getValues 和 getBytes 参阅 4.6 节。

2）如果 Block 允许使用 Tachyon，则调用 TachyonStore 的 getBytes 方法获取，请参阅 4.7.4 节。

3）如果 Block 允许使用 DiskStore，则调用 DiskStore 的 getBytes 方法获取，请参阅 4.5.1 节。

代码清单4-45 doGetLocal的实现

```
private def doGetLocal(blockId: BlockId, asBlockResult: Boolean): Option[Any] = {
    val info = blockInfo.get(blockId).orNull
    if (info != null) {
        info.synchronized {
            // 省略部分代码
            // Look for the block in memory
            if (level.useMemory) {
                logDebug(s"Getting block $blockId from memory")
                val result = if (asBlockResult) {
                    memoryStore.getValues(blockId).map(new BlockResult(_,
                        DataReadMethod.Memory, info.size))
                } else {
                    memoryStore.getBytes(blockId)
                }
                result match {
                    case Some(values) =>
                        return result
                    case None =>
                        logDebug(s"Block $blockId not found in memory")
                }
            }

            // Look for the block in Tachyon
            if (level.useOffHeap) {
                logDebug(s"Getting block $blockId from tachyon")
                if (tachyonStore.contains(blockId)) {
                    tachyonStore.getBytes(blockId) match {
                    case Some(bytes) =>
                        if (!asBlockResult) {
```

```
                    return Some(bytes)
                } else {
                    return Some(new BlockResult(
                        dataDeserialize(blockId, bytes), DataReadMethod.
                            Memory, info.size))
                }
            case None =>
            logDebug(s"Block $blockId not found in tachyon")
        }
    }
}
// Look for block on disk, potentially storing it back in memory if required
if (level.useDisk) {
    logDebug(s"Getting block $blockId from disk")
    val bytes: ByteBuffer = diskStore.getBytes(blockId) match {
        case Some(b) => b
        case None =>
            throw new BlockException(
                blockId, s"Block $blockId not found on disk, though it should
                    be")
        }
        assert(0 == bytes.position())

        // 次要代码此处省略
```

4.8.10 获取远程 Block 数据方法 doGetRemote

doGetRemote 用于从远端节点上获取 Block 数据，见代码清单 4-46。其处理步骤如下：

1）向 BlockManagerMasterActor 发送 GetLocations 消息获取 Block 数据存储的 BlockManagerId。如果 Block 数据复制份数多于 1 个，则会返回多个 BlockManagerId，对这些 BlockManagerId 洗牌，避免总是从一个远程 BlockManager 获取 Block 数据。发送 GetLocations 消息使用了 getLocations 方法，代码如下。

```
def getLocations(blockId: BlockId): Seq[BlockManagerId] = {
    askDriverWithReply[Seq[BlockManagerId]](GetLocations(blockId))
}
```

2）根据返回的 BlockManagerId 信息，使用 BlockTransferService 远程同步获取 Block 数据。

代码清单4-46 获取远程Block数据

```
private def doGetRemote(blockId: BlockId, asBlockResult: Boolean): Option[Any] = {
    require(blockId != null, "BlockId is null")
    val locations = Random.shuffle(master.getLocations(blockId))
    for (loc <- locations) {
        logDebug(s"Getting remote block $blockId from $loc")
        val data = blockTransferService.fetchBlockSync(
            loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer()
```

```
            if (data != null) {
                if (asBlockResult) {
                    return Some(new BlockResult(
                        dataDeserialize(blockId, data),
                        DataReadMethod.Network,
                        data.limit()))
                } else {
                    return Some(data)
                }
            }
            logDebug(s"The value of block $blockId is null")
        }
        logDebug(s"Block $blockId not found")
        None
    }
```

4.8.11 获取 Block 数据方法 get

get 方法用于通过 BlockId 获取 Block。get 方法在实现上首先从本地获取，如果没有则去远程获取，见代码清单 4-47。

代码清单4-47 get方法的实现

```
def get(blockId: BlockId): Option[BlockResult] = {
    val local = getLocal(blockId)
    if (local.isDefined) {
        logInfo(s"Found block $blockId locally")
        return local
    }
    val remote = getRemote(blockId)
    if (remote.isDefined) {
        logInfo(s"Found block $blockId remotely")
        return remote
    }
    None
}
```

getLocal 方法实际调用了 doGetLocal 方法，代码如下。

```
def getLocal(blockId: BlockId): Option[BlockResult] = {
    logDebug(s"Getting local block $blockId")
    doGetLocal(blockId, asBlockResult = true).asInstanceOf[Option[BlockResult]]
}
```

getRemote 方法实际调用了 doGetRemote 方法，代码如下。

```
def getRemote(blockId: BlockId): Option[BlockResult] = {
    logDebug(s"Getting remote block $blockId")
    doGetRemote(blockId, asBlockResult = true).asInstanceOf[Option[BlockResult]]
}
```

4.8.12 数据流序列化方法 dataSerializeStream

如果写入存储体系的数据本身是序列化的，那么读取时应该对其反序列化。dataSerializeStream 方法使用 compressionCodec 对文件输入流进行压缩和序列化处理，见代码清单4-48。compressionCodec 的代码分析见 4.11 节。

代码清单4-48 dataSerializeStream的实现

```
def dataSerializeStream(
        blockId: BlockId,
        outputStream: OutputStream,
        values: Iterator[Any],
        serializer: Serializer = defaultSerializer): Unit = {
    val byteStream = new BufferedOutputStream(outputStream)
    val ser = serializer.newInstance()
    ser.serializeStream(wrapForCompression(blockId, byteStream)).writeAll
        (values).close()
}

def wrapForCompression(blockId: BlockId, s: OutputStream): OutputStream = {
    if (shouldCompress(blockId)) compressionCodec.compressedOutputStream(s) else s
}
```

4.9 metadataCleaner 和 broadcastCleaner

为了有效利用磁盘空间和内存，metadataCleaner 和 broadcastCleaner 分别用于清除 blockInfo（TimeStampedHashMap[BlockId, BlockInfo]）中很久不用的非广播和广播 Block 信息。

注意 此处的 metadataCleaner 与 3.3 节介绍的 metadataCleaner 命名相同，作用却不一样，读者不要混淆。

3.3 节中已经介绍过 metadataCleaner，由此知道每个 metadataCleaner 的关键在于函数参数 cleanupFunc: (Long) => Unit。此处的 metadataCleaner 的函数参数是 dropOldNonBroadcastBlocks，broadcastCleaner 的函数参数是 dropOldBroadcastBlocks。

```
private def dropOldNonBroadcastBlocks(cleanupTime: Long): Unit = {
    logInfo(s"Dropping non broadcast blocks older than $cleanupTime")
    dropOldBlocks(cleanupTime, !_.isBroadcast)
}

private def dropOldBroadcastBlocks(cleanupTime: Long): Unit = {
    logInfo(s"Dropping broadcast blocks older than $cleanupTime")
    dropOldBlocks(cleanupTime, _.isBroadcast)
}
```

这两个函数都调用了 dropOldBlocks，dropOldBlocks 的实现见代码清单 4-49。遍历 blockInfo，将很久不用的 Block 从 MemoryStore、DiskStore、TachyonStore 中清除。

代码清单4-49 删除很久不用的Block

```
private def dropOldBlocks(cleanupTime: Long, shouldDrop: (BlockId => Boolean)): Unit = {
    val iterator = blockInfo.getEntrySet.iterator
    while (iterator.hasNext) {
        val entry = iterator.next()
        val (id, info, time) = (entry.getKey, entry.getValue.value, entry.
            getValue.timestamp)
        if (time < cleanupTime && shouldDrop(id)) {
            info.synchronized {
                val level = info.level
                if (level.useMemory) { memoryStore.remove(id) }
                if (level.useDisk) { diskStore.remove(id) }
                if (level.useOffHeap) { tachyonStore.remove(id) }
                iterator.remove()
                logInfo(s"Dropped block $id")
            }
            val status = getCurrentBlockStatus(id, info)
            reportBlockStatus(id, info, status)
        }
    }
}
```

4.10 缓存管理器 CacheManager

CacheManager 用于缓存 RDD 某个分区计算后的中间结果。读者可能误以为 RDD 都缓存在 CacheManager 的某个存储部分中，实际上 CacheManager 只是对 BlockManager 的代理，真正的缓存依然使用 BlockManager。在任务迭代计算的过程中，当判断存储级别使用了缓存，就会调用 CacheManager 的 getOrCompute 方法。getOrCompute 的实现见代码清单 4-50。

代码清单4-50 getOrCompute的实现

```
def getOrCompute[T](
        rdd: RDD[T],
        partition: Partition,
        context: TaskContext,
        storageLevel: StorageLevel): Iterator[T] = {

    val key = RDDBlockId(rdd.id, partition.index)
    logDebug(s"Looking for partition $key")
    blockManager.get(key) match {
        case Some(blockResult) =>
            // Partition is already materialized, so just return its values
            context.taskMetrics.inputMetrics = Some(blockResult.inputMetrics)
            new InterruptibleIterator(context, blockResult.data.asInstanceOf
                [Iterator[T]])

        case None =>
        // Acquire a lock for loading this partition
```

```
        // If another thread already holds the lock, wait for it to finish return
             its results
        val storedValues = acquireLockForPartition[T](key)
        if (storedValues.isDefined) {
            return new InterruptibleIterator[T](context, storedValues.get)
        }

        // Otherwise, we have to load the partition ourselves
        try {
            logInfo(s"Partition $key not found, computing it")
            val computedValues = rdd.computeOrReadCheckpoint(partition, context)

            // If the task is running locally, do not persist the result
            if (context.isRunningLocally) {
                return computedValues
            }

            // Otherwise, cache the values and keep track of any updates in block
                 statuses
            val updatedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]
            val cachedValues = putInBlockManager(key, computedValues,
                 storageLevel, updatedBlocks)
            val metrics = context.taskMetrics
            val lastUpdatedBlocks = metrics.updatedBlocks.getOrElse(Seq[(BlockId,
                 BlockStatus)]())
            metrics.updatedBlocks = Some(lastUpdatedBlocks ++ updatedBlocks.toSeq)
            new InterruptibleIterator(context, cachedValues)

        } finally {
            loading.synchronized {
                loading.remove(key)
                loading.notifyAll()
            }
        }
    }
}
```

从 getOrCompute 的实现分析其处理逻辑如下：

1）从存储体系获取 Block；

2）如果确实获取到了 Block，那么将它封装为 InterruptibleIterator 并返回。如果还没有缓存 Block，则重新计算或者从 CheckPoint 中获取数据，并调用 putInBlockManager 方法将数据写入缓存后封装为 InterruptibleIterator 并返回。其中 RDD 的 computeOrReadCheckpoint 方法将在 6.1 节说明。

putInBlockManager 的实现见代码清单 4-51。

代码清单4-51 putInBlockManager的实现

```
private def putInBlockManager[T](
        key: BlockId,
        values: Iterator[T],
```

```
        level: StorageLevel,
        updatedBlocks: ArrayBuffer[(BlockId, BlockStatus)],
        effectiveStorageLevel: Option[StorageLevel] = None): Iterator[T] = {

    val putLevel = effectiveStorageLevel.getOrElse(level)
    if (!putLevel.useMemory) {
        updatedBlocks ++=
            blockManager.putIterator(key, values, level, tellMaster = true,
                effectiveStorageLevel)
        blockManager.get(key) match {
            case Some(v) => v.data.asInstanceOf[Iterator[T]]
            case None =>
                logInfo(s"Failure to store $key")
                throw new BlockException(key, s"Block manager failed to return
                    cached value for $key!")
        }
    } else {
        blockManager.memoryStore.unrollSafely(key, values, updatedBlocks) match {
            case Left(arr) =>
            // We have successfully unrolled the entire partition, so cache it in
                memory
            updatedBlocks ++=
                blockManager.putArray(key, arr, level, tellMaster = true,
                    effectiveStorageLevel)
            arr.iterator.asInstanceOf[Iterator[T]]
        case Right(it) =>
            // There is not enough space to cache this partition in memory
            val returnValues = it.asInstanceOf[Iterator[T]]
            if (putLevel.useDisk) {
                logWarning(s"Persisting partition $key to disk instead.")
                val diskOnlyLevel = StorageLevel(useDisk = true, useMemory = false,
                    useOffHeap = false, deserialized = false, putLevel.replication)
                putInBlockManager[T](key, returnValues, level, updatedBlocks,
                    Some(diskOnlyLevel))
            } else {
                returnValues
            }
        }
    }
}
```

从 putInBlockManager 的实现，总结它的处理步骤如下。

1）获取实际的存储级别。

2）如果存储级别不允许使用内存，那么直接调用 BlockManager 的 putIterator 方法。在 doPut 方法的处理中，由于存储级别不允许使用内存，所以数据实际被直接写入了磁盘或者 Tachyon。

3）如果存储级别允许使用内存，那么首先尝试展开。如果展开成功，说明有足够内存可以存储数据，因此将数据存入内存；如果展开失败，则将数据存入磁盘。

4.11 压缩算法

为了节省磁盘存储空间，有些情况下需要对Block进行压缩。根据配置属性spark.io.compression.codec来确定要使用的压缩算法（默认为snappy，此压缩算法在牺牲少量压缩比例的条件下，却极大地提高了压缩速度），并生成SnappyCompressionCodec的实例，见代码清单4-52。

代码清单4-52 CompressionCodec的实现

```
private[spark] object CompressionCodec {

    private val shortCompressionCodecNames = Map(
        "lz4" -> classOf[LZ4CompressionCodec].getName,
        "lzf" -> classOf[LZFCompressionCodec].getName,
        "snappy" -> classOf[SnappyCompressionCodec].getName)

def createCodec(conf: SparkConf): CompressionCodec = {
    createCodec(conf, conf.get("spark.io.compression.codec", DEFAULT_COMPRESSION_
        CODEC))
}

def createCodec(conf: SparkConf, codecName: String): CompressionCodec = {
    val codecClass = shortCompressionCodecNames.getOrElse(codecName.toLowerCase,
         codecName)
    val ctor = Class.forName(codecClass, true, Utils.getContextOrSparkClassLoader)
        .getConstructor(classOf[SparkConf])
    ctor.newInstance(conf).asInstanceOf[CompressionCodec]
}

    val DEFAULT_COMPRESSION_CODEC = "snappy"
    val ALL_COMPRESSION_CODECS = shortCompressionCodecNames.values.toSeq
}
```

4.12 磁盘写入实现DiskBlockObjectWriter

DiskBlockObjectWriter被用于输出Spark任务的中间计算结果。DiskBlockObjectWriter的fileSegment方法用于创建文件分片FileSegment（FileSegment记录分片的起始、结束偏移量），代码如下。

```
override def fileSegment(): FileSegment = {
    new FileSegment(file, initialPosition, finalPosition - initialPosition)
}
```

下面我们逐个讲解DiskBlockObjectWriter的其他方法，包括open、write、close、commitAndClose。

1. 打开一个文件输出流

DiskBlockObjectWriter 的 open 方法，利用 NIO、压缩、缓存、序列化方式打开一个文件输出流，见代码清单 4-53。

代码清单4-53 DiskBlockObjectWriter的open方法

```
override def open(): BlockObjectWriter = {
    fos = new FileOutputStream(file, true)
    ts = new TimeTrackingOutputStream(fos)
    channel = fos.getChannel()
    bs = compressStream(new BufferedOutputStream(ts, bufferSize))
    objOut = serializer.newInstance().serializeStream(bs)
    initialized = true
    this
}
```

2. 写入文件

DiskBlockObjectWriter 的 write 方法用于将数据写入文件，并更新测量信息，见代码清单 4-54。

代码清单4-54 DiskBlockObjectWriter的write方法

```
override def write(value: Any) {
    if (!initialized) {
        open()
    }

    objOut.writeObject(value)

    if (writesSinceMetricsUpdate == 32) {
        writesSinceMetricsUpdate = 0
        updateBytesWritten()
    } else {
        writesSinceMetricsUpdate += 1
    }
}

private def updateBytesWritten() {
    val pos = channel.position()
    writeMetrics.shuffleBytesWritten += (pos - reportedPosition)
    reportedPosition = pos
}
```

3. 关闭文件输出流

DiskBlockObjectWriter 的 close 方法用于关闭文件输出流，并更新测量信息，见代码清单 4-55。

代码清单4-55 关闭文件输出流

```
override def close() {
    if (initialized) {
```

```
            if (syncWrites) {
                // Force outstanding writes to disk and track how long it takes
                objOut.flush()
                def sync = fos.getFD.sync()
                callWithTiming(sync)
            }
            objOut.close()

            channel = null
            bs = null
            fos = null
            ts = null
            objOut = null
            initialized = false
        }
    }

    private def callWithTiming(f: => Unit) = {
        val start = System.nanoTime()
        f
        writeMetrics.shuffleWriteTime += (System.nanoTime() - start)
    }
```

4. 缓存数据提交

DiskBlockObjectWriter 的 commitAndClose 方法将缓存数据写入磁盘并关闭缓存，然后更新测量数据，见代码清单 4-56。

代码清单4-56 commitAndClose提交缓存数据

```
override def commitAndClose(): Unit = {
    if (initialized) {
        objOut.flush()
        bs.flush()
        close()
    }
    finalPosition = file.length()
    // In certain compression codecs, more bytes are written after close() is
        called
    writeMetrics.shuffleBytesWritten += (finalPosition - reportedPosition)
}
```

4.13 块索引 shuffle 管理器 IndexShuffleBlockManager

IndexShuffleBlockManager 通常用于获取 Block 索引文件，并根据索引文件读取 Block 文件的数据。

1. 获取 shuffle 文件方法 getBlockData

有时候我们不知道 Block 的 BlockId，所以无法使用 BlockManager 的 get 方法获取 Block。

如果知道 ShuffleBlockId，我们依然可以通过 ShuffleBlockId 记录的 shuffleId 和 mapId 获取 Block。ShuffleBlockId 的格式如下。

```
case class ShuffleBlockId(shuffleId: Int, mapId: Int, reduceId: Int) extends
    BlockId {
    def name = "shuffle_" + shuffleId + "_" + mapId + "_" + reduceId
}
```

按照此格式生成的 ShuffleBlockId 能够关联 shuffleId、mapId 和 reduceId 的信息，例如，shuffle_0_0_0、shuffle_0_1_0 等。

getBlockData 方法根据 shuffleId 和 mapId（即 partitionId）读取索引文件，从索引文件中获得 partition 计算中间结果写入文件的偏移量和中间结果的大小，根据此偏移量和大小读取文件中 partition 的中间计算结果，见代码清单 4-57。

代码清单4-57 getBlockData的实现

```
override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = {
    val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)

    val in = new DataInputStream(new FileInputStream(indexFile))
    try {
        ByteStreams.skipFully(in, blockId.reduceId * 8)
        val offset = in.readLong()
        val nextOffset = in.readLong()
        new FileSegmentManagedBuffer(
            transportConf,
            getDataFile(blockId.shuffleId, blockId.mapId),
            offset,
            nextOffset - offset)
        } finally {
        in.close()
    }
}
```

2. 获取 shuffle 数据文件方法 getDataFile

getDataFile 的实现代码如下。

```
def getDataFile(shuffleId: Int, mapId: Int): File = {
    blockManager.diskBlockManager.getFile(ShuffleDataBlockId(shuffleId, mapId, 0))
}
```

getDataFile 的实质是调用 diskBlockManager 的 getFile 方法，请参阅 4.4.2 节。

3. 索引文件偏移量记录方法 writeIndexFile

writeIndexFile 方法用于在 Block 索引文件中记录各个 partition 的偏移量信息，便于下游 Stage 的任务读取，见代码清单 4-58。

代码清单4-58 索引文件写入方法writeIndexFile

```
def writeIndexFile(shuffleId: Int, mapId: Int, lengths: Array[Long]) = {
    val indexFile = getIndexFile(shuffleId, mapId)
```

```
        val out = new DataOutputStream(new BufferedOutputStream(new FileOutput-
          Stream(indexFile)))
        try {
            var offset = 0L
            out.writeLong(offset)

            for (length <- lengths) {
                offset += length
                out.writeLong(offset)
            }
        } finally {
            out.close()
        }
    }
```

4.14 shuffle 内存管理器 ShuffleMemoryManager

ShuffleMemoryManager 用于为执行 shuffle 操作的线程分配内存池。每种磁盘溢出集合（如 ExternalAppendOnlyMap 和 ExternalSorter）都能从这个内存池获得内存。当溢出集合的数据已经输出到存储系统，获得的内存会释放。当线程执行的任务结束，整个内存池都会被 Executor 释放。ShuffleMemoryManager 会保证每个线程都能合理地共享内存，而不会使得一些线程获得了很大的内存，导致其他线程经常不得不将溢出的数据写入磁盘。

尝试获得内存方法 tryToAcquire

此方法用于当前线程尝试获得 numBytes 大小的内存，并返回实际获得的内存大小，见代码清单 4-59。

代码清单4-59 尝试获得内存的实现

```
def tryToAcquire(numBytes: Long): Long = synchronized {
    val threadId = Thread.currentThread().getId
    assert(numBytes > 0, "invalid number of bytes requested: " + numBytes)

    if (!threadMemory.contains(threadId)) {
        threadMemory(threadId) = 0L
        notifyAll()   // Will later cause waiting threads to wake up and check
            numThreads again
    }

    while (true) {
        val numActiveThreads = threadMemory.keys.size
        val curMem = threadMemory(threadId)
        val freeMemory = maxMemory - threadMemory.values.sum

        val maxToGrant = math.min(numBytes, math.max(0, (maxMemory /
             numActiveThreads) - curMem))

        if (curMem < maxMemory / (2 * numActiveThreads)) {
```

```
            if (freeMemory >= math.min(maxToGrant, maxMemory / (2 * numActive-
                Threads) - curMem)) {
                val toGrant = math.min(maxToGrant, freeMemory)
                threadMemory(threadId) += toGrant
                return toGrant
            } else {
                logInfo(s"Thread $threadId waiting for at least 1/2N of shuffle memory
                    pool to be free")
                wait()
            }
        } else {
            // Only give it as much memory as is free, which might be none if it
                reached 1 / numThreads
            val toGrant = math.min(maxToGrant, freeMemory)
            threadMemory(threadId) += toGrant
            return toGrant
        }
        }
        0L  // Never reached
    }
```

根据 ShuffleMemoryManager 的实现，它的处理逻辑：假设当前有 N 个线程，必须保证每个线程在溢出之前至少获得$\frac{1}{2N}$的内存，并且每个线程最多获得$\frac{1}{N}$的内存。由于 N 是动态变化的变量，所以要持续对这些线程进行跟踪，以便无论何时在这些线程发生变化时重新按照$\frac{1}{2N}$和$\frac{1}{N}$计算。

4.15 小结

本章一开始介绍了 BlockStore 的接口定义，目前虽然只有 MemoryStore、DiskStore 和 TachyonStore 三种实现，但是随着技术的发展，当有更优秀的存储中间件出现时，随时可以实现 BlockStore，完成集成（Tachyon 就是例子）。DiskStore 对磁盘文件按照散列存储节省空间的同时提高了文件访问的效率。MemoryStore 基于内存做了大量优化，其构建的内存模型值得任何大数据存储引擎借鉴。TachyonStore 解决了 Spark 中共享磁盘文件系统性能差、计算引擎出错导致存储体系的数据丢失、内存中大量重复数据导致 GC 时间长等问题。

此外，FileSegment 与 Block 索引文件共同解决了分区数据读写同一文件的问题；shuffle 服务基于 Netty 实现的 Block 上传下载服务；shuffle 任务通过 $\left(\frac{1}{2N}, \frac{1}{N}\right)$ 的区间管理获得内存的大小保证每个线程都能合理地共享内存并减少磁盘 I/O 操作；Executor 的 BlockManager 与 Driver 的 BlockManager 上的 BlockManagerMasterActor 在 ActorSystem 中通信，保证 Driver 获取实时的 Block 状态信息的实现都值得读者回味。

第 5 章 Chapter 5

任务提交与执行

即欲捭之，贵周；即欲阖之，贵密。周密之贵微，而与道相追。

——《鬼谷子》

本章导读

记得几年前，笔者阅读 Hadoop 的相关资料时，基本是以 word Count 的例子展开的。word Count 的例子是最常见、最易于理解、最便于讲解的大数据应用场景。类似于任何介绍编程语言的书籍，hello world 能用最短的时间，带给读者最直观的感受，从而降低语言学习的曲线，也便于作者著述。作为大数据的入门实例，笔者也不能免俗，自然是经典的 word Count 例子。

任务的提交与执行构建在存储体系与计算引擎之上，存储体系已在第 4 章介绍，计算引擎将在第 6 章讲述。任务的配置信息、jar 包依赖、中间计算结果缓存等信息都离不开存储体系，而任务的执行则要依靠计算引擎的能力。

5.1 任务概述

为了对整个任务提交与执行过程有个整体认识，请读者从阅读图 5-1 开始。

这里对图 5-1 中任务提交与执行过程进行简短介绍：

1）build operator DAG：此阶段主要完成 RDD 的转换及 DAG 的构建。

2）split graph into stages of tasks：此阶段主要完成 finalStage 的创建与 Stage 的划分，做好 Stage 与 Task 的准备工作后，最后提交 Stage 与 Task。

3）launch tasks via cluster manager：使用集群管理器（Cluster manager）分配资源与任务调

度，对于失败的任务还会有一定的重试与容错机制。

4）execute tasks：执行任务，并将任务中间结果和最终结果存入存储体系。

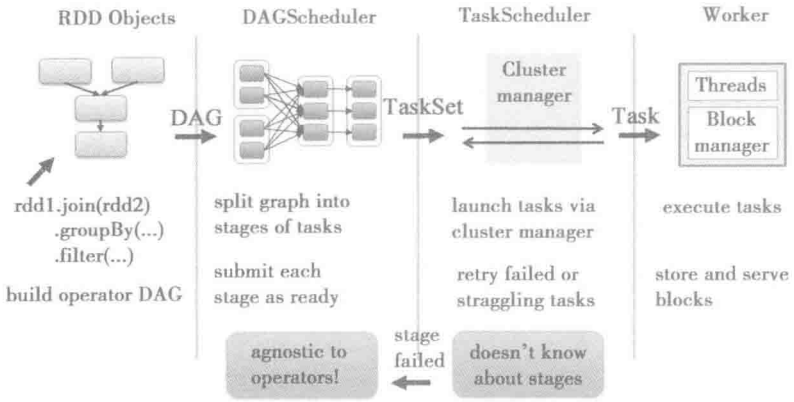


图 5-1 任务提交与执行过程

Spark 源码自带的 example 项目中有很多给学习或者研究 Spark 的读者准备的例子，为了使熟悉 Java 但是对 Scala 不了解的读者也能变得轻松，笔者选择其中的 JavaWordCount 为例，来开始本章的学习历程。例子 JavaWordCount 是用 Java 实现的，请看代码清单 5-1。

代码清单5-1 JavaWordCount的实现

```
public final class JavaWordCount {
    private static final Pattern SPACE = Pattern.compile(" ");
    public static void main(String[] args) throws Exception {

        if (args.length < 1) {
            System.err.println("Usage: JavaWordCount <file>");
            System.exit(1);
        }

        SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount").
            setMaster("local");
        JavaSparkContext ctx = new JavaSparkContext(sparkConf);
        JavaRDD<String> lines = ctx.textFile(args[0], 1);

        JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public Iterable<String> call(String s) {
            return Arrays.asList(SPACE.split(s));
        }
    });

    JavaPairRDD<String, Integer> ones = words.mapToPair(new PairFunction<String,
        String, Integer>() {
        @Override
        public Tuple2<String, Integer> call(String s) {
```

```
            return new Tuple2<String, Integer>(s, 1);
        }
    });

    JavaPairRDD<String, Integer> counts = ones.reduceByKey(new Function2<Integer,
        Integer, Integer>() {
        @Override
        public Integer call(Integer i1, Integer i2) {
            return i1 + i2;
        }
    });

    List<Tuple2<String, Integer>> output = counts.collect();
        for (Tuple2<?,?> tuple : output) {
            System.out.println(tuple._1() + ": " + tuple._2());
        }
        ctx.stop();
    }
}
```

JavaSparkContext 的 textFile 实际只是代理了 SparkContext 的 textFile 方法而已，代码如下。

```
def textFile(path: String, minPartitions: Int): JavaRDD[String] =
  sc.textFile(path, minPartitions)
```

SparkContext 的 textFile 方法中，调用了 hadoopFile 方法。熟悉 Hadoop1.0 版本的同学可能已经注意到了 TextInputFormat、LongWritable、Text 这几个类型，此处分别获取了它们的 Class 实例，见代码清单 5-2。

代码清单5-2　textFile的实现

```
def textFile(path: String, minPartitions: Int = defaultMinPartitions): RDD-
    [String] = {
    assertNotStopped()
    hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
        minPartitions).map(pair => pair._2.toString).setName(path)
}
```

MRv1 有新旧两套 MapReduce 编程接口，Spark 为什么要选择 MRv1 的老 API 作为编程模型？MRv2 向前兼容 MRv1 的应用程序时，老 API 编写的应用程序几乎不用改造就可以运行在 MRv2 上；但是用 MRv1 的新 API 编写的应用程序则不具备良好的兼容性，需要使用 MRv2 重新编译对参数、返回值调整后才能运行在 MRv2 上。笔者认为这可能是 Spark 做出选择的原因。

hadoopFile 方法的功能主要是为了构建 HadoopRDD，见代码清单 5-3，其处理步骤如下：

1）将 Hadoop 的 Configuration 封装为 SerializableWritable 用于序列化读写操作，然后广播 Hadoop 的 Configuration。Hadoop 的 Configuration 通常只有 10 KB，所以不会对性能有影响；

2）定义偏函数 (jobConf: JobConf) => FileInputFormat.setInputPaths（jobConf, path）用于

以后设置输入路径；

3）构建 HadoopRDD。

代码清单5-3 hadoopFile的实现

```
def hadoopFile[K, V](
    path: String,
    inputFormatClass: Class[_ <: InputFormat[K, V]],
    keyClass: Class[K],
    valueClass: Class[V],
    minPartitions: Int = defaultMinPartitions
    ): RDD[(K, V)] = {
assertNotStopped()
// A Hadoop configuration can be about 10 KB, which is pretty big, so broad-cast it.
val confBroadcast = broadcast(new SerializableWritable(hadoopConfiguration))
val stInputPathsFunc = (jobConf: JobConf) => FileInputFormat.setInputPaths
     (jobConf, path)
new HadoopRDD(
    this,
    confBroadcast,
    Some(setInputPathsFunc),
    inputFormatClass,
    keyClass,
    valueClass,
    minPartitions).setName(path)
}
```

5.2 广播 Hadoop 的配置信息

SparkContext 的 broadcast 方法用于广播 Hadoop 的配置信息，其实现见代码清单 5-4。

代码清单5-4 广播Hadoop的配置信息

```
def broadcast[T: ClassTag](value: T): Broadcast[T] = {
    assertNotStopped()
    if (classOf[RDD[_]].isAssignableFrom(classTag[T].runtimeClass)) {
        logWarning("Can not directly broadcast RDDs; instead, call collect() and "
            + "broadcast the result (see SPARK-5063)")
    }
    val bc = env.broadcastManager.newBroadcast[T](value, isLocal)
    val callSite = getCallSite
    logInfo("Created broadcast " + bc.id + " from " + callSite.shortForm)
    cleaner.foreach(_.registerBroadcastForCleanup(bc))
    bc
}
```

上面的代码通过使用 BroadcastManager 发送广播，广播结束将广播对象注册到 ContextCleanner 中，以便清理。根据 3.2.9 节的内容，我们知道 BroadcastManager 的 newBroadcast 方法实际代理了 broadcastFactory 的 newBroadcast 方法。通过 TorrentBroadcastFactory 生成

TorrentBroadcast 对象的代码如下。

```
override def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean, id: Long) = {
    new TorrentBroadcast[T](value_, id)
}
```

从代码清单 5-5 看到构造 TorrentBroadcast 的过程分为三步。

1）设置广播配置信息。根据 spark.broadcast.compress 配置属性确认是否对广播消息进行压缩，并且生成 CompressionCodec 对象。CompressionCodec 的具体介绍见 4.11 节。根据 spark.broadcast.blockSize 配置属性确认块的大小，默认为 4 MB。

2）生成 BroadcastBlockId。

3）块的写入操作，返回广播变量包含的块数。

代码清单5-5　TorrentBroadcast的实现

```
@transient private lazy val _value: T = readBroadcastBlock()
@transient private var compressionCodec: Option[CompressionCodec] = _
@transient private var blockSize: Int = _

private def setConf(conf: SparkConf) {
    compressionCodec = if (conf.getBoolean("spark.broadcast.compress", true)) {
        Some(CompressionCodec.createCodec(conf))
    } else {
        None
    }
    blockSize = conf.getInt("spark.broadcast.blockSize", 4096) * 1024
}
setConf(SparkEnv.get.conf)

private val broadcastId = BroadcastBlockId(id)

private val numBlocks: Int = writeBlocks(obj)
```

块的写入操作 writeBlocks

从代码清单 5-6，看到块写入操作 writeBlocks 的工作分三步。

1）将要写入的对象在本地的存储体系中备份一份，以便于 Task 也可以在本地的 Driver 上运行。

2）给 ByteArrayChunkOutputStream 指定压缩算法，并且将对象以序列化方式写入 ByteArrayChunkOutputStream 后转换为 Array[ByteBuffer]。

3）将每一个 ByteBuffer 作为一个 Block，使用 putBytes 方法写入存储体系。

代码清单5-6　writeBlocks的实现

```
private def writeBlocks(value: T): Int = {
    SparkEnv.get.blockManager.putSingle(broadcastId, value, StorageLevel.MEMORY_
        AND_DISK,
        tellMaster = false)
```

```
        val blocks =
            TorrentBroadcast.blockifyObject(value, blockSize, SparkEnv.get.serializer,
                compressionCodec)
            blocks.zipWithIndex.foreach { case (block, i) =>
        SparkEnv.get.blockManager.putBytes(
            BroadcastBlockId(id, "piece" + i),
            block,
            StorageLevel.MEMORY_AND_DISK_SER,
            tellMaster = true)
        }
        blocks.length
    }
```

BlockManager 的 putSingle 和 putBytes 已在第 4 章介绍过，不再赘述。

TorrentBroadcast.blockifyObject 方法用于将对象序列化写入 ByteArrayChunkOutputStream，并用 CompressionCodec 压缩，最终将 ByteArrayChunkOutputStream 转换为 Array[ByteBuffer]。blockifyObject 的实现见代码清单 5-7。

代码清单5-7 blockifyObject方法的实现

```
def blockifyObject[T: ClassTag](
        obj: T,
        blockSize: Int,
        serializer: Serializer,
        compressionCodec: Option[CompressionCodec]): Array[ByteBuffer] = {
    val bos = new ByteArrayChunkOutputStream(blockSize)
    val out: OutputStream = compressionCodec.map(c => c.compressedOutputStream(bos)).
     getOrElse(bos)
    val ser = serializer.newInstance()
    val serOut = ser.serializeStream(out)
    serOut.writeObject[T](obj).close()
    bos.toArrays.map(ByteBuffer.wrap)
}
```

5.3 RDD 转换及 DAG 构建

5.3.1 为什么需要 RDD

以下从数据处理模型、依赖划分原则、数据处理效率及容错处理 4 个方面解释 Spark 发明 RDD 的原因。

1. 数据处理模型

RDD 是一个容错的、并行的数据结构，可以控制将数据存储到磁盘或内存，能够获取数据的分区。RDD 提供了一组类似于 Scala 的操作，比如 map、flatMap、filter 等，这些操作实际是对 RDD 进行转换（transformation）。此外，RDD 还提供了 join、groupBy、reduceByKey

等操作完成数据计算（注意，reduceByKey 是 action，而非 transformation）。

当前的大数据应用场景非常丰富，如流式计算、图计算、机器学习等。它们既有相似之处，又各有不同。为了能够对所有场景下的数据处理使用统一的方式，抽象出 RDD 这一模型。

通常数据处理的模型包括：迭代计算、关系查询、MapReduce、流式处理等。Hadoop 采用 MapReduce 模型，Storm 采用流式处理模型，而 Spark 则实现了以上所有模型。

2. 依赖划分原则

一个 RDD 包含一个或者多个分区，每个分区实际是一个数据集合的片段。在构建 DAG 的过程中，会将 RDD 用依赖关系串联起来。每个 RDD 都有其依赖（除了最顶级 RDD 的依赖是空列表），这些依赖分为 NarrowDependency 和 ShuffleDependency 两种。为什么要对依赖进行区分？从功能角度讲它们是不一样的。NarrowDependency 会被划分到同一个 Stage 中，这样它们就能以管道的方式迭代执行。ShuffleDependency 由于依赖的上游 RDD 不止一个，所以往往需要跨节点传输数据。从容灾角度讲，它们恢复计算结果的方式不同。NarrowDependency 只需要重新执行父 RDD 的丢失分区的计算即可恢复。而 ShuffleDependency 则需要考虑恢复所有父 RDD 的丢失分区。

解释了依赖划分的原因，实际也解释了为什么要划分 Stage 这个问题。

3. 数据处理效率

ShuffleDependency 所依赖的上游 RDD 的计算过程允许在多个节点并发执行，如图 5-2 ⊖所示，实际也就是后面将会讲到的 ShuffleMapTask 在多个节点上的多个实例。如果数据量很大，可以适当增加分区数量，这种根据硬件条件对并发任务数量的控制，能更好地利用各种资源，也能有效提高 Spark 的数据处理效率。

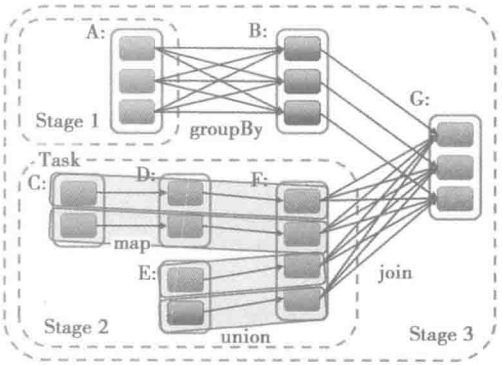


图 5-2　RDD 并行计算示意图

4. 容错处理

传统关系型数据库往往采用日志记录的方式来容灾容错，数据恢复都依赖于重新执行日志中的 SQL。Hadoop 为了避免单机故障概率较高的问题，通过将数据备份到其他机器容灾。由于所有备份机器同时出故障的概率比单机故障概率低很多，从而在宕机等问题发生时，从备份机读取数据。RDD 本身是一个不可变的（Scala 中称为 immutable）数据集，当某个 Worker 节点上的任务失败时，可以利用 DAG 重新调度计算这个失败的任务。由于不用复制数据，也大大降低了网络通信。在流式计算的场景中，Spark 需要记录日志和检查点（CheckPoint），以便利用 CheckPoint 和日志对数据进行恢复。

⊖ 图片引用自 http://www.zhihu.com/question/26568496。

5.3.2 RDD 实现分析

代码清单 5-3 中最后创建了 HadoopRDD，此时的 DAG 如图 5-3 所示。

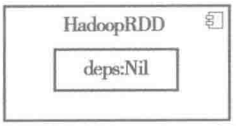


图 5-3 只有 HadoopRDD 时的 DAG

hadoopFile 方法创建完 HadoopRDD 后，会调用 RDD 的 map 方法，见代码清单 5-2。

map 方法将 HadoopRDD 封装为 MappedRDD，见代码清单 5-8。

代码清单5-8 map方法的实现

```
def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))
```

这里调用了 SparkContext 的 clean 方法，实现如下。

```
private[spark] def clean[F <: AnyRef](f: F, checkSerializable: Boolean = true): F = {
    ClosureCleaner.clean(f, checkSerializable)
    f
}
```

clean 方法实际调用了 ClosureCleaner 的 clean 方法，这里意在清除闭包中的不能序列化的变量，防止 RDD 在网络传输过程中反序列化失败。

构造 MappedRDD 的步骤如下：

1）调用 MappedRDD 的父类 RDD 的辅助构造器，RDD 的辅助构造器实现如下。

```
def this(@transient oneParent: RDD[_]) =
    this(oneParent.context , List(new OneToOneDependency(oneParent)))
```

辅助构造器首先将 oneParent 封装为 OneToOneDependency，OneToOneDependency 继承自 NarrowDependency，其实现如下。

```
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
    override def getParents(partitionId: Int) = List(partitionId)
}
```

2）调用 RDD 的主构造器，主构造器实现如下。

```
abstract class RDD[T: ClassTag](
    @transient private var _sc: SparkContext,
    @transient private var deps: Seq[Dependency[_]]
) extends Serializable with Logging {

protected def getDependencies: Seq[Dependency[_]] = deps
```


注意 getDependencies 方法虽然只是简单地返回依赖信息，但是它将在接下来的内容中发挥巨大的作用。

构建完 MappedRDD 后，此时的 DAG 如图 5-4 所示。

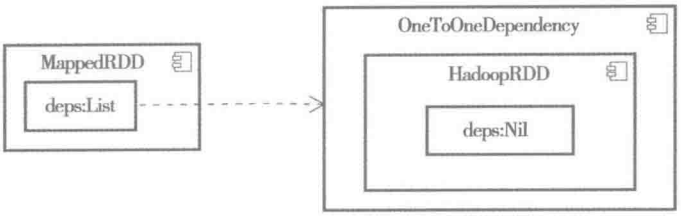


图 5-4　MappedRDD 与 HadoopRDD 组成的 DAG

MappedRDD 在 JavaSparkContext 中会被隐式转换为 JavaRDD。代码清单 5-1 接着执行 JavaRDD 的 flatMap 方法，由于 JavaRDD 实现了 JavaRDDLike 特质，所以实际调用了 JavaRDDLike 的 flatMap 方法，它的实现如下。

```
def flatMap[U](f: FlatMapFunction[T, U]): JavaRDD[U] = {
    import scala.collection.JavaConverters._
    def fn = (x: T) => f.call(x).asScala
    JavaRDD.fromRDD(rdd.flatMap(fn)(fakeClassTag[U]))(fakeClassTag[U])
}
```

此时，JavaRDD 内部的 rdd 属性实质上还是 MappedRDD，调用 MappedRDD 的 flatMap 方法，其实现如下。

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] =
    new FlatMappedRDD(this, sc.clean(f))
```

MappedRDD 被封装为 FlatMappedRDD，构造 FlatMappedRDD 也会调用父类 RDD 的辅助构造器，并为其设置 OneToOneDependency，这与 MappedRDD 的构造过程是一样的。此时的 DAG 如图 5-5 所示。

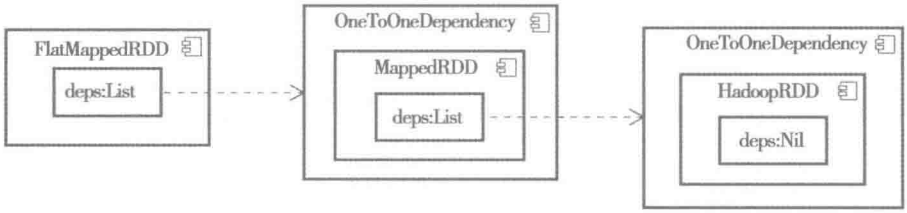


图 5-5　FlatMappedRDD、MappedRDD 与 HadoopRDD 组成的 DAG

FlatMappedRDD 创建完后调用了 JavaRDD 的 fromRDD 方法，将 FlatMappedRDD 也封装为 JavaRDD，代码如下。

```
object JavaRDD {
    implicit def fromRDD[T: ClassTag](rdd: RDD[T]): JavaRDD[T] = new JavaRDD[T](rdd)
    implicit def toRDD[T](rdd: JavaRDD[T]): RDD[T] = rdd.rdd
}
```

接着执行 JavaRDD 的 mapToPair 方法时（见代码清单 5-1），JavaRDD 由于实现了 Java-

RDDLike 特质，所以实际调用了 JavaRDDLike 的 mapToPair 方法，代码实现如下。

```
def mapToPair[K2, V2](f: PairFunction[T, K2, V2]): JavaPairRDD[K2, V2] = {
    def cm = implicitly[ClassTag[(K2, V2)]]
    new JavaPairRDD(rdd.map[(K2, V2)](f)(cm))(fakeClassTag[K2], fakeClassTag[V2])
}
```

此时，JavaRDD 内部的 rdd 属性实际上还是 FlatMappedRDD，此时调用 RDD 的 map，又被封装为 MappedRDD，见代码清单 5-8。此时的 DAG 如图 5-6 所示。

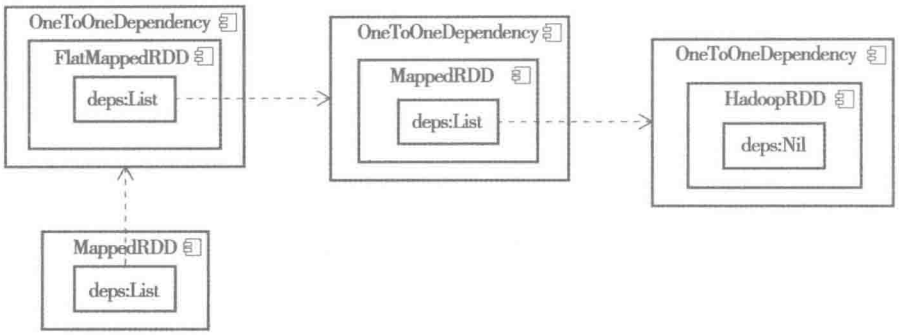


图 5-6 FlatMappedRDD、2 个 MappedRDD 与 HadoopRDD 组成的 DAG

然后 MappedRDD 又被封装为 JavaPairRDD。执行 JavaPairRDD 的 reduceByKey 方法，其实现见代码清单 5-9。

代码清单5-9 reduceByKey的实现

```
def reduceByKey(func: JFunction2[V, V, V]): JavaPairRDD[K, V] = {
    fromRDD(reduceByKey(defaultPartitioner(rdd), func))
}
```

defaultPartitioner 方法的实现见代码清单 5-10，其功能实现如下。

1）将 RDD 转换为 Seq，然后对 Seq 按照 RDD 的 partitions_：Array[Partition] 的 size 倒序排列。

2）创建 HashPartitioner 对象。如果配置了 spark.default.parallelism 属性，则用此属性值作为分区数量。否则使用 Seq 中所有 RDD 的 partitions 函数返回值的最大值作为分区数量。

代码清单5-10 defaultPartitioner方法的实现

```
def defaultPartitioner(rdd: RDD[_], others: RDD[_]*): Partitioner = {
    val bySize = (Seq(rdd) ++ others).sortBy(_.partitions.size).reverse
    for (r <- bySize if r.partitioner.isDefined) {
        return r.partitioner.get
    }
    if (rdd.context.conf.contains("spark.default.parallelism")) {
        new HashPartitioner(rdd.context.defaultParallelism)
    } else {
```

```
            new HashPartitioner(bySize.head.partitions.size)
        }
    }
```

RDD 的 partitions 方法的实现见代码清单 5-11。

代码清单5-11　partitions方法的实现

```
final def partitions: Array[Partition] = {
    checkpointRDD.map(_.partitions).getOrElse {
        if (partitions_ == null) {
            partitions_ = getPartitions
        }
        partitions_
    }
}
```

本例中，partitions 方法实际调用了 MappedRDD 的 getPartitions 方法。MappedRDD 的 getPartitions 方法调用了 RDD 的 firstParent，见代码清单 5-12。

代码清单5-12　getPartitions方法的实现

```
override def getPartitions: Array[Partition] = firstParent[T].partitions
```

firstParent 用于返回依赖的第一个父 RDD，见代码清单 5-13。我们知道 MappedRDD 的第一个依赖 RDD 是 FlatMappedRDD，然后调用 FlatMappedRDD 的 partitions 方法。FlatMappedRDD 没有 partitions 方法，所以调用了 RDD 的 partitions 方法。

代码清单5-13　firstParent的实现

```
protected[spark] def firstParent[U: ClassTag] = {
    dependencies.head.rdd.asInstanceOf[RDD[U]]
}
```

FlatMappedRDD 的 getPartitions 与 MappedRDD 的 getPartitions 完全一样，仍然会调用 firstParent 方法。FlatMappedRDD 的第一个父 RDD 是最早封装的那个 MappedRDD，MappedRDD 的第一个父 RDD 是 HadoopRDD。HadoopRDD 也没有 partitions 方法，实际也是调用了 RDD 的 partitions 方法。partitions 最终调用 HadoopRDD 的 getPartitions 方法，见代码清单 5-14。

代码清单5-14　getPartitions方法的实现

```
override def getPartitions: Array[Partition] = {
    val jobConf = getJobConf()
    SparkHadoopUtil.get.addCredentials(jobConf)
    val inputFormat = getInputFormat(jobConf)
    if (inputFormat.isInstanceOf[Configurable]) {
        inputFormat.asInstanceOf[Configurable].setConf(jobConf)
    }
    val inputSplits = inputFormat.getSplits(jobConf, minPartitions)
    val array = new Array[Partition](inputSplits.size)
    for (i <- 0 until inputSplits.size) {
```

```
            array(i) = new HadoopPartition(id, i, inputSplits(i))
        }
        array
    }
```

阅读代码清单 5-14 发现，最后是通过 Hadoop 的 TextInputFormat.getSplits(jobConf, minPartitions) 方法决定分区个数的。

回到代码清单 5-9，最终调用 RDD 的 reduceByKey，见代码清单 5-15。

代码清单5-15 JavaPairRDD. reduceByKey的实现

```
def reduceByKey(partitioner: Partitioner, func: JFunction2[V, V, V]):
    JavaPairRDD[K, V] =
    fromRDD(rdd.reduceByKey(partitioner, func))
```

无论是 RDD 还是 MappedRDD 都没有 reduceByKey。这是怎么回事？笔者刚才的结论只是阅读代码清单 5-15 的直观感觉而已。实际上这里发生了隐式转换，将 RDD 封装成了 PairRDDFunctions，隐式转换函数如下。

```
implicit def rddToPairRDDFunctions[K, V](rdd: RDD[(K, V)])
        (implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null) = {
    new PairRDDFunctions(rdd)
}
```

implicit 是 Scala 中的关键字，语法含义是隐式执行。在发生隐式转换之前，还需要将 Java 里的方法转换为 Scala 的参数，这里主要做了两样工作：

1）用户自定义实现所需方法的匿名对象，将对象作为参数。正如本例中代码清单 5-1 的 Function2 匿名对象。

2）隐式转换将对象转换为 Scala 的函数，转换函数如下。

```
private[spark]
    implicit def toScalaFunction2[T1, T2, R](fun: JFunction2[T1, T2, R]): Function2-
        [T1, T2, R] = {
        (x: T1, x1: T2) => fun.call(x, x1)
}
```

经过了多次转换，终于可以调用 PairRDDFunctions 的 reduceByKey 方法了，其实现如下。

```
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = {
    combineByKey[V]((v: V) => v, func, func, partitioner)
}
```

combineByKey 方法的实现见代码清单 5-16，其处理步骤如下：

1）创建 Aggregator（其中 mergeValue 就是代码清单 5-1 中的 Function2）。

2）由于本例中当前 RDD 还没有设置 partitioner，self.partitioner ! = Some（partitioner），因而创建 ShuffledRDD。

代码清单5-16　combineByKey方法的实现

```
val aggregator = new Aggregator[K, V, C](
    self.context.clean(createCombiner),
    self.context.clean(mergeValue),
    self.context.clean(mergeCombiners))
if (self.partitioner == Some(partitioner)) {
    self.mapPartitions(iter => {
        val context = TaskContext.get()
        new InterruptibleIterator(context, aggregator.combineValuesByKey(iter,
            context))
    }, preservesPartitioning = true)
} else {
    new ShuffledRDD[K, V, C](self, partitioner)
        .setSerializer(serializer)
        .setAggregator(aggregator)
        .setMapSideCombine(mapSideCombine)
}
```

ShuffledRDD 的构造器及 setSerializer、setAggregator、setMapSideCombine 等方法的实现，见代码清单 5-17。从中看出 ShuffledRDD 的依赖是 ShuffleDependency。FlatMappedRDD 和 MappedRDD 的依赖都是在构造器被调用时创建的，而 ShuffledRDD 的依赖 ShuffleDependency 则是在其 getDependencies 方法被调用时才创建的。

代码清单5-17　ShuffledRDD及setSerializer等方法的实现

```
class ShuffledRDD[K, V, C](
    @transient var prev: RDD[_ <: Product2[K, V]],
    part: Partitioner)
extends RDD[(K, C)](prev.context, Nil) {

private var serializer: Option[Serializer] = None
private var keyOrdering: Option[Ordering[K]] = None
private var aggregator: Option[Aggregator[K, V, C]] = None
private var mapSideCombine: Boolean = false

/** Set a serializer for this RDD's shuffle, or null to use the default (spark.serializer) */
    def setSerializer(serializer: Serializer): ShuffledRDD[K, V, C] = {
        this.serializer = Option(serializer)
        this
    }
    /** Set key ordering for RDD's shuffle. */
    def setKeyOrdering(keyOrdering: Ordering[K]): ShuffledRDD[K, V, C] = {
        this.keyOrdering = Option(keyOrdering)
        this
    }
    /** Set aggregator for RDD's shuffle. */
    def setAggregator(aggregator: Aggregator[K, V, C]): ShuffledRDD[K, V, C] = {
        this.aggregator = Option(aggregator)
        this
    }
    /** Set mapSideCombine flag for RDD's shuffle. */
```

```
def setMapSideCombine(mapSideCombine: Boolean): ShuffledRDD[K, V, C] = {
    this.mapSideCombine = mapSideCombine
    this
}
override def getDependencies: Seq[Dependency[_]] = {
    List(new ShuffleDependency(prev, part, serializer, keyOrdering, aggregator,
        mapSideCombine))
}
override val partitioner = Some(part)
```

ShuffledRDD 被 fromRDD 方法重新封装为 JavaPairRDD，代码如下。

```
def fromRDD[K: ClassTag, V: ClassTag](rdd: RDD[(K, V)]): JavaPairRDD[K, V] = {
    new JavaPairRDD[K, V](rdd)
}
```

此时的 DAG 如图 5-7 所示。

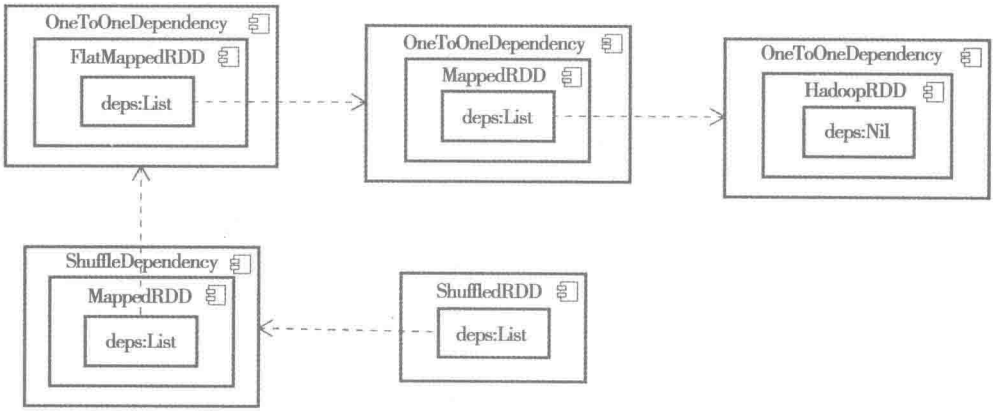


图 5-7 最终的 DAG

5.4 任务提交

5.4.1 任务提交的准备

经过 5.3 节对 RDD 的层层转换以及 DAG 的构建，现在要执行 JavaPairRDD 的 word count 例子方法了，见代码清单 5-1。collect 中调用了 RDD 的 collect 方法后转成 Seq，并封装 Seq 为 ArrayList，其代码实现如下。

```
def collect(): JList[T] = {
    import scala.collection.JavaConversions._
    val arr: java.util.Collection[T] = rdd.collect().toSeq
    new java.util.ArrayList(arr)
}
```

RDD 的 collect 方法调用了 SparkContext 的 runJob，见代码清单 5-18。

代码清单5-18　collect的实现

```
def collect(): Array[T] = {
    val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
    Array.concat(results: _*)
}
```

SparkContext 的 runJob 又调用了重载的 runJob，见代码清单 5-19。

代码清单5-19　runJob的实现

```
 def runJob[T, U: ClassTag](rdd: RDD[T], func: Iterator[T] => U): Array[U] = {
    runJob(rdd, func, 0 until rdd.partitions.size, false)
}
```

接着又调用两个重载的 runJob，见代码清单 5-20。

代码清单5-20　runJob的重载

```
def runJob[T, U: ClassTag](
        rdd: RDD[T],
        func: Iterator[T] => U,
        partitions: Seq[Int],
        allowLocal: Boolean
        ): Array[U] = {
    runJob(rdd, (context: TaskContext, iter: Iterator[T]) => func(iter), partitions, allowLocal)
}

def runJob[T, U: ClassTag](
        rdd: RDD[T],
        func: (TaskContext, Iterator[T]) => U,
        partitions: Seq[Int],
        allowLocal: Boolean
        ): Array[U] = {
    val results = new Array[U](partitions.size)
    runJob[T, U](rdd, func, partitions, allowLocal, (index, res) => results(index) = res)
    results
}
```

最终调用的 runJob 方法里又一次调用 clean 方法防止闭包的反序列化错误，然后运行 dagScheduler 的 runJob，见代码清单 5-21。

代码清单5-21　最终调用的runJob方法

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit) {
    if (stopped) {
        throw new IllegalStateException("SparkContext has been shutdown")
```

```
    }
    val callSite = getCallSite
    val cleanedFunc = clean(func)
    logInfo("Starting job: " + callSite.shortForm)
    dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
        resultHandler, localProperties.get)
    progressBar.foreach(_.finishAll())
    rdd.doCheckpoint()
}
```

dagScheduler 的 runJob 方法主要调用 submitJob 方法，之后的 waiter.awaitResult() 说明了任务的运行是异步的，见代码清单 5-22。

代码清单5-22 dagScheduler的runJob方法

```
def runJob[T, U: ClassTag](
        rdd: RDD[T],
        func: (TaskContext, Iterator[T]) => U,
        partitions: Seq[Int],
        callSite: CallSite,
        allowLocal: Boolean,
        resultHandler: (Int, U) => Unit,
        properties: Properties): Unit = {
    val start = System.nanoTime
    val waiter = submitJob(rdd, func, partitions, callSite, allowLocal, result-
         Handler, properties)
    waiter.awaitResult() match {
        case JobSucceeded => {
            logInfo("Job %d finished: %s, took %f s".format
                (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        }
        case JobFailed(exception: Exception) =>
            logInfo("Job %d failed: %s, took %f s".format
                (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
            throw exception
    }
}
```

1. 提交 Job

submitJob 方法用来将一个 Job 提交到 job scheduler，见代码清单 5-23。

代码清单5-23 submitJob方法的实现

```
val maxPartitions = rdd.partitions.length
partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>
    throw new IllegalArgumentException(
        "Attempting to access a non-existent partition: " + p + ". " +
        "Total number of partitions: " + maxPartitions)
}

val jobId = nextJobId.getAndIncrement()
if (partitions.size == 0) {
```

```
        return new JobWaiter[U](this, jobId, 0, resultHandler)
    }

    assert(partitions.size > 0)
    val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
    val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
    eventProcessActor ! JobSubmitted(
        jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties)
    waiter
```

根据代码清单 5-23 分析，submitJob 的处理步骤如下：

1）调用 RDD 的 partitions 函数来获取当前 Job 的最大分区数，即 maxPartitions。根据 maxPartitions，确认我们没有在一个不存在的 partition 上运行任务。

2）生成当前 Job 的 jobId。

3）创建 JobWaiter，望文生义，即 Job 的服务员。JobWaiter 的实现见代码清单 5-24。从代码清单 5-24 可以了解，此 JobWaiter 被阻塞，直到 job 完成或者被取消。

4）向 eventProcessActor 发送 JobSubmitted 事件（这里的 eventProcessActor 在 3.7 节中已经介绍过，就是 DAGSchedulerEventProcessActor）。

5）返回 JobWaiter。

代码清单5-24　JobWaiter的实现

```
private[spark] class JobWaiter[T](
        dagScheduler: DAGScheduler,
        val jobId: Int,
        totalTasks: Int,
        resultHandler: (Int, T) => Unit)
    extends JobListener {

    private var finishedTasks = 0
    @volatile
    private var _jobFinished = totalTasks == 0
    def jobFinished = _jobFinished
    private var jobResult: JobResult = if (jobFinished) JobSucceeded else null

    def cancel() {
        dagScheduler.cancelJob(jobId)
    }

    override def taskSucceeded(index: Int, result: Any): Unit = synchronized {
        if (_jobFinished) {
            throw new UnsupportedOperationException("taskSucceeded() called on a
                finished JobWaiter")
        }
        resultHandler(index, result.asInstanceOf[T])
        finishedTasks += 1
        if (finishedTasks == totalTasks) {
```

```
                _jobFinished = true
                jobResult = JobSucceeded
                this.notifyAll()
            }
        }

    override def jobFailed(exception: Exception): Unit = synchronized {
            _jobFinished = true
            jobResult = JobFailed(exception)
            this.notifyAll()
        }

        def awaitResult(): JobResult = synchronized {
            while (!_jobFinished) {
                this.wait()
            }
            return jobResult
        }
    }
```

2. 处理 Job 提交

DAGSchedulerEventProcessActor 收到 JobSubmitted 事件，会调用 dagScheduler 的 handleJobSubmitted 方法，见代码清单 3-35。handleJobSubmitted 的具体实现见代码清单 5-25，其执行过程如下。

1）创建 finalStage 及 Stage 的划分。创建 Stage 的过程可能发生异常。比如，运行在 HadoopRDD 上的任务所依赖的底层 HDFS 文件被删除了。所以当异常发生时需要主动调用 JobWaiter 的 jobFailed 方法。

2）创建 ActiveJob 并更新 jobIdToActiveJob = new HashMap[Int, ActiveJob]、activeJobs = new HashSet[ActiveJob] 和 finalStage.resultOfJob。

3）向 listenerBus 发送 SparkListenerJobStart 事件。

4）提交 finalStage。

5）提交等待中的 Stage。

代码清单5-25 handleJobSubmitted的实现

```
private[scheduler] def handleJobSubmitted(jobId: Int, finalRDD: RDD[_],
func: (TaskContext, Iterator[_]) => _, partitions: Array[Int], allowLocal: Boolean,
        callSite: CallSite, listener: JobListener, properties: Properties) {
    var finalStage: Stage = null
    try {
        finalStage = newStage(finalRDD, partitions.size, None, jobId, callSite)
    } catch {
        case e: Exception =>
            logWarning("Creating new stage failed due to exception - job: " + jobId, e)
            listener.jobFailed(e)
            return
    }
```

```
        if (finalStage != null) {
            val job = new ActiveJob(jobId, finalStage, func, partitions, callSite, listener, properties)
            clearCacheLocs()
            val shouldRunLocally =
                localExecutionEnabled && allowLocal && finalStage.parents.isEmpty &&
                    partitions.length == 1
            if (shouldRunLocally) {
                // Compute very short actions like first() or take() with no parent stages locally.
                listenerBus.post(SparkListenerJobStart(job.jobId, Seq.empty, properties))
                runLocally(job)
            } else {
                jobIdToActiveJob(jobId) = job
                activeJobs += job
                finalStage.resultOfJob = Some(job)
                val stageIds = jobIdToStageIds(jobId).toArray
                val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(_.latestInfo))
                listenerBus.post(SparkListenerJobStart(job.jobId, stageInfos, properties))
                submitStage(finalStage)
            }
        }
        submitWaitingStages()
    }
```

5.4.2　finalStage 的创建与 Stage 的划分

在 Spark 中，一个 Job 可能被划分为一个或多个 Stage，各个之间存在依赖关系，其中最下游的 Stage 也称为最终的 Stage，用来处理 Job 最后阶段的工作。

1. newStage 的实现分析

handleJobSubmitted 方法使用 newStage 方法创建 finalStage，newStage 的实现见代码清单 5-26，它的处理步骤如下：

1）调用 getParentStages 获取所有的父 Stage 的列表，父 Stage 主要是宽依赖（如 ShuffleDependency）对应的 Stage，此列表内的 Stage 包含以下几种：

① 当前 RDD 的直接或间接的依赖是 ShuffleDependency 且已经注册过的 Stage。

② 当前 RDD 的直接或间接的依赖是 ShuffleDependency 且没有注册过 Stage 的，则根据 ShuffleDependency 本身的 RDD，找到它的直接或间接的依赖是 ShuffleDependency 且没有注册过 Stage 的所有 ShuffleDependency，为他们生成 Stage 并注册。

③ 当前 RDD 的直接或间接的依赖是 ShuffleDependency 且没有注册过 Stage 的，为它们生成 Stage 并注册，最后也添加此 Stage 到 List。

2）生成 Stage 的 Id，并创建 Stage（创建 Stage 在后面会详述）。

3）将 Stage 注册到 stageIdToStage ＝ new HashMap[Int, Stage] 中。

4）调用 updateJobIdStageIdMaps 方法 Stage 及其祖先 Stage 与 jobId 的对应关系。

代码清单5-26 newStage的实现

```
private def newStage(
        rdd: RDD[_],
        numTasks: Int,
        shuffleDep: Option[ShuffleDependency[_, _, _]],
        jobId: Int,
        callSite: CallSite)
    : Stage =
{
    val parentStages = getParentStages(rdd, jobId)
    val id = nextStageId.getAndIncrement()
    val stage = new Stage(id, rdd, numTasks, shuffleDep, parentStages, jobId, callSite)
    stageIdToStage(id) = stage
    updateJobIdStageIdMaps(jobId, stage)
    stage
}
```

2. 获取父 Stage 列表

Spark 中 Job 会被划分为一到多个 Stage，这些 Stage 的划分是从 finalStage 开始，从后往前边划分边创建的。getParentStages 方法（见代码清单 5-27）用于获取或者创建给定 RDD 的所有父 Stage，这些 Stage 将被分配给 jobId 对应的 job，其处理步骤如下。

1）通过调用 RDD 的 dependencies 方法（见代码清单 5-28）获取 RDD 的所有 Dependency 的序列。

2）逐个访问每个 RDD 及其依赖的非 Shuffle 的 RDD，遍历每个 RDD 的 ShuffleDependency 依赖，并调用 getShuffleMapStage 获取或者创建 Stage，并将这些返回的 Stage 都放入 parents：HashSet[Stage]。由此可见，Stage 的划分是以 ShuffleDependency 为分界线的。

代码清单5-27 getParentStages方法的实现

```
private def getParentStages(rdd: RDD[_], jobId: Int): List[Stage] = {
    val parents = new HashSet[Stage]
    val visited = new HashSet[RDD[_]]
    val waitingForVisit = new Stack[RDD[_]]
    def visit(r: RDD[_]) {
        if (!visited(r)) {
            visited += r
            for (dep <- r.dependencies) {
                dep match {
                    case shufDep: ShuffleDependency[_, _, _] =>
                        parents += getShuffleMapStage(shufDep, jobId)
                    case _ =>
                        waitingForVisit.push(dep.rdd)
                }
            }
        }
    }
    waitingForVisit.push(rdd)
    while (!waitingForVisit.isEmpty) {
```

```
            visit(waitingForVisit.pop())
        }
        parents.toList
    }
```

代码清单5-28　dependencies方法的实现

```
    final def dependencies: Seq[Dependency[_]] = {
        checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
            if (dependencies_ == null) {
                dependencies_ = getDependencies
            }
            dependencies_
        }
    }
```

3. 获取 map 任务对应 Stage

getShuffleMapStage 方法（见代码清单 5-29）用于获取或者创建 Stage 并注册到 shuffleToMapStage：HashMap[Int, Stage] 中，处理步骤如下。

1）如果已经注册了 ShuffleDependency 对应的 Stage，则直接返回此 Stage。

2）否则调用 registerShuffleDependencies 方法找到所有祖先中，还没有为其注册过 Stage 的 ShuffleDependency，调用方法 newOrUsedStage 创建 Stage 并注册。最后还会为当前 ShuffleDependency，调用方法 newOrUsedStage 创建、注册并返回此 Stage。

代码清单5-29　getShuffleMapStage方法的实现

```
    private def getShuffleMapStage(shuffleDep: ShuffleDependency[_, _, _], jobId: Int):
        Stage = {
        shuffleToMapStage.get(shuffleDep.shuffleId) match {
            case Some(stage) => stage
            case None =>
                registerShuffleDependencies(shuffleDep, jobId)
                val stage =
                    newOrUsedStage(
                        shuffleDep.rdd, shuffleDep.rdd.partitions.size, shuffleDep, jobId,
                        shuffleDep.rdd.creationSite)
                shuffleToMapStage(shuffleDep.shuffleId) = stage

                stage
            }
        }
        private def registerShuffleDependencies(shuffleDep: ShuffleDependency[_, _, _],
            jobId: Int) = {
            val parentsWithNoMapStage = getAncestorShuffleDependencies(shuffleDep.rdd)
            while (!parentsWithNoMapStage.isEmpty) {
                val currentShufDep = parentsWithNoMapStage.pop()
                val stage =
                    newOrUsedStage(
                currentShufDep.rdd, currentShufDep.rdd.partitions.size, currentShufDep, jobId,
```

```
                currentShufDep.rdd.creationSite)
            shuffleToMapStage(currentShufDep.shuffleId) = stage
        }
    }
```

getAncestorShuffleDependencies 用来找到 RDD 直接或者间接依赖的所有祖先中，还没有为其注册过 Stage 的 ShuffleDependency，见代码清单 5-30。

代码清单5-30 getAncestorShuffleDependencies的实现

```
private def getAncestorShuffleDependencies(rdd: RDD[_]): Stack[Shuffle-
    Dependency[_, _, _]] = {
    val parents = new Stack[ShuffleDependency[_, _, _]]
    val visited = new HashSet[RDD[_]]
    // We are manually maintaining a stack here to prevent StackOverflowError
    // caused by recursively visiting
    val waitingForVisit = new Stack[RDD[_]]
    def visit(r: RDD[_]) {
        if (!visited(r)) {
            visited += r
            for (dep <- r.dependencies) {
                dep match {
                    case shufDep: ShuffleDependency[_, _, _] =>
                        if (!shuffleToMapStage.contains(shufDep.shuffleId)) {
                            parents.push(shufDep)
                        }

                        waitingForVisit.push(shufDep.rdd)
                    case _ =>
                        waitingForVisit.push(dep.rdd)
                }
            }
        }
    }

    waitingForVisit.push(rdd)
    while (!waitingForVisit.isEmpty) {
        visit(waitingForVisit.pop())
    }
    parents
}
```

newOrUsedStage 方法（见代码清单 5-31）：首先调用 newStage 创建 Stage（前面已有说明，不再赘述），然后将 ShuffleDependency 的 shuffleId 和 partitions 的 size 注册到 MapOutputTrackerMaster 的 mapStatuses = new Time-StampedHashMap[Int, Array[MapStatus]]() 中。

代码清单5-31 newOrUsedStage方法的实现

```
private def newOrUsedStage(
        rdd: RDD[_],
```

```
        numTasks: Int,
        shuffleDep: ShuffleDependency[_, _, _],
        jobId: Int,
        callSite: CallSite)
    : Stage =
{
    val stage = newStage(rdd, numTasks, Some(shuffleDep), jobId, callSite)
    if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {
        val serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)
        val locs = MapOutputTracker.deserializeMapStatuses(serLocs)
        for (i <- 0 until locs.size) {
            stage.outputLocs(i) = Option(locs(i)).toList   // locs(i) will be null if missing
        }
        stage.numAvailableOutputs = locs.count(_ != null)
    } else {
        mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size)
    }
    stage
}
```

很多地方都调用了 newStage 创建 Stage，从代码清单 5-32 来看看 Stage 的数据结构。

代码清单5-32　Stage的数据结构

```
private[spark] class Stage(val id: Int, val rdd: RDD[_], val numTasks: Int,
    val shuffleDep: Option[ShuffleDependency[_, _, _]],// Output shuffle if stage is
        a map stage
    val parents: List[Stage], val jobId: Int, val callSite: CallSite) extends Logging {

    val isShuffleMap = shuffleDep.isDefined
    val numPartitions = rdd.partitions.size
    val outputLocs = Array.fill[List[MapStatus]](numPartitions)(Nil)
    var numAvailableOutputs = 0

    val jobIds = new HashSet[Int]

    var resultOfJob: Option[ActiveJob] = None
    var pendingTasks = new HashSet[Task[_]]

    private var nextAttemptId = 0

    val name = callSite.shortForm
    val details = callSite.longForm

    var latestInfo: StageInfo = StageInfo.fromStage(this)
```

Stage 的构造过程中调用了 StageInfo 的 fromStage 方法（见代码清单 5-33）创建 StageInfo。

代码清单5-33　fromStage方法的实现

```
def fromStage(stage: Stage, numTasks: Option[Int] = None): StageInfo = {
    val ancestorRddInfos = stage.rdd.getNarrowAncestors.map(RDDInfo.fromRdd)
    val rddInfos = Seq(RDDInfo.fromRdd(stage.rdd)) ++ ancestorRddInfos
```

```
        new StageInfo(
            stage.id,
            stage.attemptId,
            stage.name,
            numTasks.getOrElse(stage.numTasks),
            rddInfos,
            stage.details)
    }
```

创建 StageInfo 的步骤如下：

1）调用 getNarrowAncestors 方法获取 RDD 的所有直接或者间接的 NarrowDependency 的 RDD，见代码清单 5-34。

代码清单5-34 getNarrowAncestors的实现

```
private[spark] def getNarrowAncestors: Seq[RDD[_]] = {
    val ancestors = new mutable.HashSet[RDD[_]]

    def visit(rdd: RDD[_]) {
        val narrowDependencies = rdd.dependencies.filter(_.isInstanceOf[NarrowDependency[_]])
        val narrowParents = narrowDependencies.map(_.rdd)
        val narrowParentsNotVisited = narrowParents.filterNot(ancestors.contains)
        narrowParentsNotVisited.foreach { parent =>
            ancestors.add(parent)
            visit(parent)
        }
    }

    visit(this)
    ancestors.filterNot(_ == this).toSeq
}
```

返回的 Seq[RDD[_]] 全部 map 到 RDDInfo.fromRdd 方法，生成 RDDInfo，代码如下。

```
def fromRdd(rdd: RDD[_]): RDDInfo = {
    val rddName = Option(rdd.name).getOrElse(rdd.id.toString)
    new RDDInfo(rdd.id, rddName, rdd.partitions.size, rdd.getStorageLevel)
}
```

2）对当前 Stage 的 RDD 调用 RDDInfo.fromRdd 方法，也生成 RDDInfo，然后所有生成的 RDDInfo 都合入 rddInfos 中。

3）创建当前 Stage 的 StageInfo。

回头看看代码清单 5-26 中调用的 updateJobIdStageIdMaps 方法，它的功能如下：

通过迭代调用内部的 updateJobIdStageIdMapsList 函数，最终将 jobId 添加到 Stage 及它的所有祖先 Stage 的映射 jobIds = new HashSet[Int] 中，将 jobId 和 Stage 及它的所有祖先 Stage 的 id，更新到 jobIdToStageIds = new HashMap[Int, HashSet[Int]] 中。updateJobIdStageIdMaps 的实现见代码清单 5-35。

代码清单5-35　updateJobIdStageIdMaps的实现

```
private def updateJobIdStageIdMaps(jobId: Int, stage: Stage) {
    def updateJobIdStageIdMapsList(stages: List[Stage]) {
        if (stages.nonEmpty) {
            val s = stages.head
            s.jobIds += jobId
            jobIdToStageIds.getOrElseUpdate(jobId, new HashSet[Int]()) += s.id
            val parents: List[Stage] = getParentStages(s.rdd, jobId)
            val parentsWithoutThisJobId = parents.filter { ! _.jobIds.contains(jobId) }
            updateJobIdStageIdMapsList(parentsWithoutThisJobId ++ stages.tail)
        }
    }
    updateJobIdStageIdMapsList(List(stage))
}
```

本例中，最终划分的 Stage 可以用图 5-8 表示。

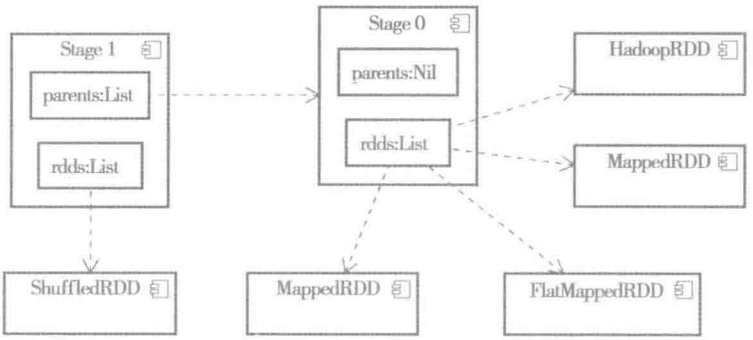


图 5-8　本例最终划分的 Stage

5.4.3　创建 Job

ActiveJob 的定义见代码清单 5-36。这里对其中的一些定义做些解释：

- numPartitions：任务的分区数量。
- finished：标识每个 partition 相关的任务是否完成。
- numFinished：已经完成的任务数。

代码清单5-36　ActiveJob的定义

```
private[spark] class ActiveJob(
        val jobId: Int,
        val finalStage: Stage,
        val func: (TaskContext, Iterator[_]) => _,
        val partitions: Array[Int],
        val callSite: CallSite,
        val listener: JobListener,
        val properties: Properties) {

    val numPartitions = partitions.length
```

```
    val finished = Array.fill[Boolean](numPartitions)(false)
    var numFinished = 0
}
```

我们回头看看 SparkListenerJobStart 事件的处理，从代码清单 3-16 可以看到，SparkListenerBus 的 sparkListeners（比如 JobProgressListener）中，凡是实现了 onJobStart 方法的，将被处理。

5.4.4 提交 Stage

在提交 finalStage 之前，如果存在没有提交的祖先 Stage，则需要先提交所有没有提交的祖先 Stage。每个 Stage 提交之前，如果存在没有提交的祖先 Stage，都会先提交祖先 Stage，并且将子 Stage 放入 waitingStages = new HashSet[Stage] 中等待，如果不存在没有提交的祖先 Stage，则提交所有未提交的 Task。提交 Stage 的实现见代码清单 5-37。

代码清单5-37 提交Stage的实现

```
private def submitStage(stage: Stage) {
    val jobId = activeJobForStage(stage)
    if (jobId.isDefined) {
        logDebug("submitStage(" + stage + ")")
        if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
            val missing = getMissingParentStages(stage).sortBy(_.id)
            logDebug("missing: " + missing)
            if (missing == Nil) {
                logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
                submitMissingTasks(stage, jobId.get)
            } else {
                for (parent <- missing) {
                    submitStage(parent)
                }
                waitingStages += stage
            }
        }
    } else {
        abortStage(stage, "No active job for stage " + stage.id)
    }
}
```

getMissingParentStages 方法用来找到 Stage 的所有不可用的祖先 Stage，见代码清单 5-38。

代码清单5-38 getMissingParentStages方法的实现

```
private def getMissingParentStages(stage: Stage): List[Stage] = {
    val missing = new HashSet[Stage]
    val visited = new HashSet[RDD[_]]
    val waitingForVisit = new Stack[RDD[_]]
    def visit(rdd: RDD[_]) {
        if (!visited(rdd)) {
            visited += rdd
```

```
                if (getCacheLocs(rdd).contains(Nil)) {
                    for (dep <- rdd.dependencies) {
                        dep match {
                        case shufDep: ShuffleDependency[_, _, _] =>
                            val mapStage = getShuffleMapStage(shufDep, stage.jobId)
                            if (!mapStage.isAvailable) {
                                missing += mapStage
                            }
                        case narrowDep: NarrowDependency[_] =>
                            waitingForVisit.push(narrowDep.rdd)
                        }
                    }
                }
            }
        }
        waitingForVisit.push(stage.rdd)
        while (!waitingForVisit.isEmpty) {
            visit(waitingForVisit.pop())
        }
        missing.toList
    }
```

如何判断 Stage 可用？它的判断十分简单：如果 Stage 不是 Map 任务，那么它是可用的；否则它的已经输出计算结果的分区任务数量要和分区数一样，即所有分区上的子任务都要完成。判断逻辑如下。

```
def isAvailable: Boolean = {
    if (!isShuffleMap) {
        true
    } else {
        numAvailableOutputs == numPartitions
    }
}
```

回头看看 handleJobSubmitted 方法中调用的 submitWaitingStages 方法，submitWaitingStages 实际上循环 waitingStages 中的 Stage 并调用 submitStage，实现如下。

```
val waitingStagesCopy = waitingStages.toArray
waitingStages.clear()
for (stage <- waitingStagesCopy.sortBy(_.jobId)) {
        submitStage(stage)
}
```

5.4.5　提交 Task

提交 Task 的入口是 submitMissingTasks 函数，此函数在 Stage 没有不可用的祖先 Stage 时，被调用处理当前 Stage 未提交的任务。

1. 提交还未计算的任务

submitMissingTasks 用于提交还未计算的任务。在分析 submitMissingTasks 之前，先对一些定义进行描述：

- pendingTasks：类型是 HashSet[Task[_]]，存储有待处理的 Task。
- MapStatus：包括执行 Task 的 BlockManager 的地址和要传给 reduce 任务的 Block 的估算大小。
- outputLocs：如果 Stage 是 map 任务，则 outputLocs 记录每个 Partition 的 MapStatus。

通过对代码清单 5-39 的分析，submitMissingTasks 的执行过程总结如下。

1）清空 pendingTasks。由于当前 Stage 的任务刚开始提交，所以需要清空，便于记录需要计算的任务。

2）找出还未计算的 partition（如果 Stage 是 map 任务，那么 outputLocs 中 partition 对应的 List[MapStatus] 为 Nil，说明此 partition 还未计算。如果 Stage 不是 map 任务，那么需要获取 Stage 的 finalJob，并调用 finished 方法判断每个 partition 的任务是否完成）。

3）将当前 Stage 加入运行中的 Stage 集合（runningStages：HashSet[Stage]）中。

4）使用 StageInfo.fromStage 方法创建当前 Stage 的 latestInfo（StageInfo）。

5）向 listenerBus 发送 SparkListenerStageSubmitted 事件。

6）如果 Stage 是 map 任务，那么序列化 Stage 的 RDD 及 ShuffleDependency。如果 Stage 不是 map 任务，那么序列化 Stage 的 RDD 及 resultOfJob 的处理函数。这些序列化得到的字节数组最后需要使用 sc.broadcast 进行广播。

7）如果 Stage 是 map 任务，则创建 ShuffleMapTask，否则创建 ResultTask。还未计算的 partition 个数决定了最终创建的 Task 个数。并将创建的所有 Task 都添加到 Stage 的 pendingTasks 中。

8）利用上一步创建的所有 Task、当前 Stage 的 id、jobId 等信息创建 TaskSet，并调用 taskScheduler 的 submitTasks，批量提交 Stage 及其所有 Task。

代码清单5-39 submitMissingTasks的实现

```
private def submitMissingTasks(stage: Stage, jobId: Int) {
    stage.pendingTasks.clear()

    val partitionsToCompute: Seq[Int] = {
        if (stage.isShuffleMap) {
            (0 until stage.numPartitions).filter(id => stage.outputLocs(id) == Nil)
        } else {
            val job = stage.resultOfJob.get
            (0 until job.numPartitions).filter(id => !job.finished(id))
        }
    }

    val properties = if (jobIdToActiveJob.contains(jobId)) {
        jobIdToActiveJob(stage.jobId).properties
    } else {
        null
    }

runningStages += stage
```

```
stage.latestInfo = StageInfo.fromStage(stage, Some(partitionsToCompute.size))
listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))

var taskBinary: Broadcast[Array[Byte]] = null
try {
        val taskBinaryBytes: Array[Byte] =
        if (stage.isShuffleMap) {
            closureSerializer.serialize((stage.rdd, stage.shuffleDep.get) : AnyRef).
                array()
        } else {
            closureSerializer.serialize((stage.rdd, stage.resultOfJob.get.func)
                : AnyRef).array()
        }
    taskBinary = sc.broadcast(taskBinaryBytes)
} catch {
    case e: NotSerializableException =>
        abortStage(stage, "Task not serializable: " + e.toString)
        runningStages -= stage
        return
    case NonFatal(e) =>
        abortStage(stage, s"Task serialization failed: $e\n${e.getStackTraceString}")
        runningStages -= stage
        return
    }

    val tasks: Seq[Task[_]] = if (stage.isShuffleMap) {
        partitionsToCompute.map { id =>
        val locs = getPreferredLocs(stage.rdd, id)
        val part = stage.rdd.partitions(id)
        new ShuffleMapTask(stage.id, taskBinary, part, locs)
    }
} else {
    val job = stage.resultOfJob.get
    partitionsToCompute.map { id =>
        val p: Int = job.partitions(id)
        val part = stage.rdd.partitions(p)
        val locs = getPreferredLocs(stage.rdd, p)
        new ResultTask(stage.id, taskBinary, part, locs, id)
    }
}

if (tasks.size > 0) {
    try {
        closureSerializer.serialize(tasks.head)
    } catch {
        case e: NotSerializableException =>
            abortStage(stage, "Task not serializable: " + e.toString)
            runningStages -= stage
            return
        case NonFatal(e) => // Other exceptions, such as IllegalArgumentException
            from Kryo.
            abortStage(stage, s"Task serialization failed: $e\n${e.getStackTraceString}")
```

```
                runningStages -= stage
                return
            }

            logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " ("
                + stage.rdd + ")")
            stage.pendingTasks ++= tasks
            logDebug("New pending tasks: " + stage.pendingTasks)
            taskScheduler.submitTasks(
                new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.
                    jobId, properties))
            stage.latestInfo.submissionTime = Some(clock.getTime())
        } else {
            listenerBus.post(SparkListenerStageCompleted(stage.latestInfo))
            logDebug("Stage " + stage + " is actually done; %b %d %d".format(
                stage.isAvailable, stage.numAvailableOutputs, stage.numPartitions))
            runningStages -= stage
        }
    }
```

submitTasks 方法（见代码清单 5-40）提交任务主要分为以下步骤。

1）构建任务集管理器。即将 TaskScheduler、TaskSet 及最大失败次数（maxTaskFailures）封装为 TaskSetManager。

2）设置任务集调度策略（调度模式有 FAIR 和 FiFO 两种，此处以默认的 FiFO 为例）。将 TaskSetManager 添加到 FiFOSchedulableBuilder 中，代码如下。

```
override def addTaskSetManager(manager: Schedulable, properties: Properties) {
    rootPool.addSchedulable(manager)
}
```

实际上是把 TaskSetManager 加入 rootPool 的先进先出（FiFO）的调度队列 schedulableQueue 和 schedulableNameToSchedulable 中，并且设置 TaskSetManager 的 parent 是 Pool，见代码清单 5-41。

由于同时可能有多个任务提交，所以需要一种调度策略来决定究竟先提交哪个任务集，例如本例中的 FiFO 调度策略。

3）资源分配。调用 LocalBackend 的 reviveOffers 方法（见代码清单 3-30），实际向 localActor 发送 ReviveOffers 消息。localActor 对 ReviveOffers 消息的匹配执行 reviveOffers 方法（见代码清单 3-36）。

代码清单5-40　submitTasks方法的实现

```
override def submitTasks(taskSet: TaskSet) {
    val tasks = taskSet.tasks
    logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
    this.synchronized {
        val manager = new TaskSetManager(this, taskSet, maxTaskFailures)
```

```
            activeTaskSets(taskSet.id) = manager
            schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)
            // 省略部分代码
        }
        backend.reviveOffers()
    }
```

代码清单5-41　addSchedulable方法的实现

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

reviveOffers（见代码清单 5-42）的处理步骤如下：

1）使用 ExecutorId、ExecutorHostName、freeCores（空闲 CPU 核数）创建 WorkerOffer；

2）调用 TaskSchedulerImpl 的 resourceOffers 方法分配资源；

3）调用 Executor 的 launchTask 方法运行任务。

代码清单5-42　LocalBackend. reviveOffers的实现

```
def reviveOffers() {
    val offers = Seq(new WorkerOffer(localExecutorId, localExecutorHostname, freeCores))
    val tasks = scheduler.resourceOffers(offers).flatten
    for (task <- tasks) {
        freeCores -= scheduler.CPUS_PER_TASK
        executor.launchTask(executorBackend, task.taskId, task.name, task.
            serializedTask)
    }
    if (tasks.isEmpty && scheduler.activeTaskSets.nonEmpty) {
        // Try to reviveOffer after 1 second, because scheduler may wait for
            locality timeout
        context.system.scheduler.scheduleOnce(1000 millis, self, ReviveOffers)
    }
}
```

2. 资源分配

resourceOffers 方法（见代码清单 5-43）用于 Task 任务的资源分配，其处理步骤如下。

1）标记 Executor 与 host 的关系，增加激活的 Executor 的 id，按照 host 对 Executor 分组，并向 DAGSchedulerEventProcessActor 发送 ExecutorAdded 事件等。


注意　这里的 activeExecutorIds、executorsByHost 及 hostsByRack 是为了在后续计算本地化时使用。

2）计算资源的分配与计算。对所有 WorkerOffer 随机洗牌，避免将任务总是分配给同样的 WorkerOffer。

3）根据每个 WorkerOffer 的可用的 CPU 核数创建同等尺寸的任务描述（TaskDescription）数组。

4）将每个 WorkerOffer 的可用的 CPU 核数统计到可用 CPU（availableCpus）数组中。

5）对 rootPool 中的所有 TaskSetManager 按照调度算法排序（本例中为 FiFO 调度算法）。

6）调用每个 TaskSetManager 的 resourceOffer 方法，根据 WorkerOffer 的 ExecutorId 和 host 找到需要执行的任务并进一步进行资源处理。

7）任务分配到相应的 host 和 Executor 后，将 taskId 与 TaskSetId 的关系、taskId 与 ExecutorId 的关系、executors 与 Host 的分组关系等更新并且将 availableCpus 数目减去每个任务分配的 CPU 核数（CPUS_PER_TASK）。

8）返回第 3）步生成的 TaskDescription 列表。

代码清单5-43 Task任务的资源分配实现

```
def resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {
    var newExecAvail = false
    for (o <- offers) {
        executorIdToHost(o.executorId) = o.host
        activeExecutorIds += o.executorId
        if (!executorsByHost.contains(o.host)) {
            executorsByHost(o.host) = new HashSet[String]()
            executorAdded(o.executorId, o.host)
            newExecAvail = true
        }
        for (rack <- getRackForHost(o.host)) {
            hostsByRack.getOrElseUpdate(rack, new HashSet[String]()) += o.host
        }
    }

    val shuffledOffers = Random.shuffle(offers)
    val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))
    val availableCpus = shuffledOffers.map(o => o.cores).toArray
    val sortedTaskSets = rootPool.getSortedTaskSetQueue
    for (taskSet <- sortedTaskSets) {
        logDebug("parentName: %s, name: %s, runningTasks: %s".format(
            taskSet.parent.name, taskSet.name, taskSet.runningTasks))
        if (newExecAvail) {
            taskSet.executorAdded()
        }
    }
    var launchedTask = false
    for (taskSet <- sortedTaskSets; maxLocality <- taskSet.myLocalityLevels) {
        do {
            launchedTask = false
            for (i <- 0 until shuffledOffers.size) {
                val execId = shuffledOffers(i).executorId
                val host = shuffledOffers(i).host
                if (availableCpus(i) >= CPUS_PER_TASK) {
                    for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {
                        tasks(i) += task
```

```
                        val tid = task.taskId
                        taskIdToTaskSetId(tid) = taskSet.taskSet.id
                        taskIdToExecutorId(tid) = execId
                        executorsByHost(host) += execId
                        availableCpus(i) -= CPUS_PER_TASK
                        assert(availableCpus(i) >= 0)
                        launchedTask = true
                    }
                }
            }
        } while (launchedTask)
    }

    if (tasks.size > 0) {
        hasLaunchedTask = true
    }
    return tasks
}
```

DAGSchedulerEventProcessActor 会将 ExecutorAdded 事件匹配执行 DagScheduler 的 handleExecutorAdded 方法，用于将跟踪失败的节点重新恢复正常和提交等待中的 Stage，见代码清单 3-35 和代码清单 5-44。

代码清单5-44　handleExecutorAdded的实现

```
private[scheduler] def handleExecutorAdded(execId: String, host: String) {
    if (failedEpoch.contains(execId)) {
        logInfo("Host added was in lost list earlier: " + host)
        failedEpoch -= execId
    }
    submitWaitingStages()
}
```

3．Worker 任务分配

resourceOffer 方法（见代码清单 5-45）用于给 Worker 分配 Task，其处理步骤如下：

1）获取当前任务集允许使用的本地化级别。

2）调用 findTask 寻找 Executor、Host、pendingTasksWithNoPrefs 中有待运行的 task。

3）创建 TaskInfo，并对 task、addedFiles、addedJars（file 和 jar 是如何添加到 SparkContext 的，在 3.12 节中介绍过）进行序列化。

4）调用 DagScheduler 的 taskStarted 方法，笔者认为此处方法名不当，因为 taskStarted 的功能是向 DAGSchedulerEventProcessActor 发送 BeginEvent 事件，它的实现如下。

```
def taskStarted(task: Task[_], taskInfo: TaskInfo) {
    eventProcessActor ! BeginEvent(task, taskInfo)
}
```

DAGSchedulerEventProcessActor 在接收 BeginEvent 事件后，调用了 dagScheduler 的方法 handleBeginEvent，见代码清单 3-35。handleBeginEvent 方法通过发送 SparkListenerTaskStart

事件给 listenerBus，用以各种监听器更新 SparkUI 的显示，见代码清单 5-46。

5）最终封装 TaskDescription 对象并返回。

代码清单5-45 resourceOffer的实现

```
def resourceOffer(
        execId: String,
        host: String,
        maxLocality: TaskLocality.TaskLocality)
    : Option[TaskDescription] =
{
    if (!isZombie) {
        val curTime = clock.getTime()

        var allowedLocality = maxLocality

        if (maxLocality != TaskLocality.NO_PREF) {
            allowedLocality = getAllowedLocalityLevel(curTime)
            if (allowedLocality > maxLocality) {
                // We're not allowed to search for farther-away tasks
                allowedLocality = maxLocality
            }
        }

        findTask(execId, host, allowedLocality) match {
            case Some((index, taskLocality, speculative)) => {
                val task = tasks(index)
                val taskId = sched.newTaskId()
                copiesRunning(index) += 1
                val attemptNum = taskAttempts(index).size
                val info = new TaskInfo(taskId, index, attemptNum, curTime,
                    execId, host, taskLocality, speculative)
                taskInfos(taskId) = info
                taskAttempts(index) = info :: taskAttempts(index)
                if (maxLocality != TaskLocality.NO_PREF) {
                    currentLocalityIndex = getLocalityIndex(taskLocality)
                    lastLaunchTime = curTime
                }
                val startTime = clock.getTime()
                val serializedTask = Task.serializeWithDependencies(
                    task, sched.sc.addedFiles, sched.sc.addedJars, ser)
                if (serializedTask.limit > TaskSetManager.TASK_SIZE_TO_WARN_KB * 1024 &&
                        !emittedTaskSizeWarning) {
                    emittedTaskSizeWarning = true
                    logWarning(s"Stage ${task.stageId} contains a task of very large size " +
                        s"(${serializedTask.limit / 1024} KB). The maximum recommended
                            task size is " +
                        s"${TaskSetManager.TASK_SIZE_TO_WARN_KB} KB.")
                }
                addRunningTask(taskId)

                val taskName = s"task ${info.id} in stage ${taskSet.id}"
```

```
                logInfo("Starting %s (TID %d, %s, %s, %d bytes)".format(
                    taskName, taskId, host, taskLocality, serializedTask.limit))

                sched.dagScheduler.taskStarted(task, info)
                return Some(new TaskDescription(taskId, execId, taskName, index,
                    serializedTask))
            }
            case _ =>
        }
    }
    None
}
```

代码清单5-46　handleBeginEvent的实现

```
private[scheduler] def handleBeginEvent(task: Task[_], taskInfo: TaskInfo) {
    val stageAttemptId = stageIdToStage.get(task.stageId).map(_.latestInfo.
        attemptId).getOrElse(-1)
    listenerBus.post(SparkListenerTaskStart(task.stageId, stageAttemptId, taskInfo))
    submitWaitingStages()
}
```

local 模式下，任务提交的过程可以用图 5-9 来表示。

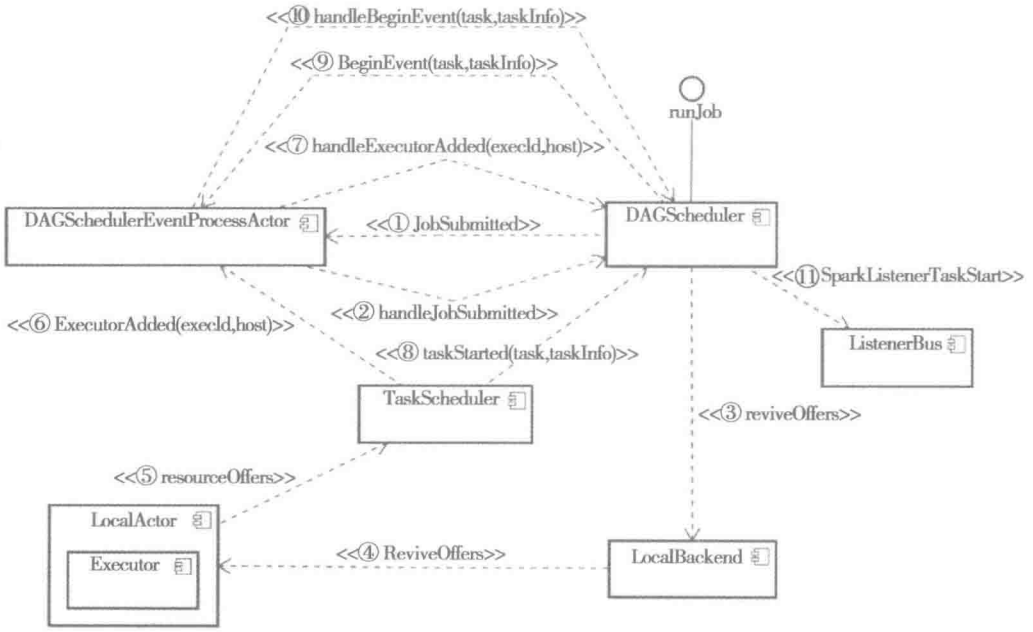


图 5-9　local 模式下的任务提交

4. 本地化分析

与 Hadoop 类似，Spark 中任务的处理也要考虑数据的本地性（Locality）。Spark 目前支持 PROCESS_LOCAL（本地进程）、NODE_LOCAL（本地节点）、NO_PREF（没有喜好）、

RACK_LOCAL（本地机架）、ANY（任何）几种。

Spark 涉及本地性的数据只有两种，HadoopRDD 和数据源于存储体系的 RDD（即由 CacheManager 从 BlockManager 中读取，或者 Streaming 数据源 RDD）。

除了 NO_PREF，其他定义都比较好理解。什么是 NO_PREF？

当 Driver 应用程序刚刚启动，Driver 分配获得的 Executor 很可能还没有初始化完毕。所以会有一部分任务的本地化级别被设置为 NO_PREF。如果是 ShuffleRDD，其本地性始终为 NO_PREF。对于这两种本地化级别是 NO_PREF 的情况，在任务分配时会被优先分配到非本地节点执行，达到一定的优化效果。

代码清单 5-45 中调用 getAllowedLocalityLevel 方法来获取任务集允许使用的本地化级别。在讲解 getAllowedLocalityLevel 之前，我们先介绍本地化的几个概念。

- myLocalityLevels：当前 TaskSetManager 允许使用的本地化级别。

myLocalityLevels 实际是对函数 computeValidLocalityLevels 的引用，代码如下。

```
var myLocalityLevels = computeValidLocalityLevels()
```

computeValidLocalityLevels 方法（见代码清单 5-47）用于计算有效的本地化级别。以 PROCESS_LOCAL 为例，如果存在 Executor 中有待执行的任务（pendingTasksForExecutor 不为空）且 PROCESS_LOCAL 本地化的等待时间不为 0（调用 getLocalityWait 方法获得）且存在 Executor 已被激活（pendingTasksForExecutor 中的 ExecutorId 有存在于 TaskScheduler 的 activeExecutorIds 中的），那么允许的本地化级别里包括 PROCESS_LOCAL。

代码清单5-47 computeValidLocalityLevels方法的实现

```
private def computeValidLocalityLevels(): Array[TaskLocality.TaskLocality] = {
    import TaskLocality.{PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL, ANY}
    val levels = new ArrayBuffer[TaskLocality.TaskLocality]
    if (!pendingTasksForExecutor.isEmpty && getLocalityWait(PROCESS_LOCAL) != 0 &&
            pendingTasksForExecutor.keySet.exists(sched.isExecutorAlive(_))) {
        levels += PROCESS_LOCAL
    }
    if (!pendingTasksForHost.isEmpty && getLocalityWait(NODE_LOCAL) != 0 &&
            pendingTasksForHost.keySet.exists(sched.hasExecutorsAliveOnHost(_))) {
        levels += NODE_LOCAL
    }
    if (!pendingTasksWithNoPrefs.isEmpty) {
        levels += NO_PREF
    }
    if (!pendingTasksForRack.isEmpty && getLocalityWait(RACK_LOCAL) != 0 &&
            pendingTasksForRack.keySet.exists(sched.hasHostAliveOnRack(_))) {
        levels += RACK_LOCAL
    }
    levels += ANY
    logDebug("Valid locality levels for " + taskSet + ": " + levels.mkString(", "))
    levels.toArray
}
```

getLocalityWait 方法（见代码清单 5-48）用于获取各个本地化级别的等待时间，这些配置如表 5-1 所示。

表 5-1　getLocalityWait 方法的配置

变 量 名	描　述	默 认 值
spark.locality.wait	本地化级别的默认等待时间	3 000
spark.locality.wait.process	本地进程的等待时间	3 000
spark.locality.wait.node	本地节点的等待时间	3 000
spark.locality.wait.rack	本地机架的等待时间	3 000

提示　在任务的运行时间很长且数量较多的情况下，适当调高这些参数可以显著提高性能。然而当这些参数值都已经超过任务的运行时长时，则需要调小这些参数。

代码清单5-48　getLocalityWait方法的实现

```
private def getLocalityWait(level: TaskLocality.TaskLocality): Long = {
    val defaultWait = conf.get("spark.locality.wait", "3000")
    level match {
        case TaskLocality.PROCESS_LOCAL =>
            conf.get("spark.locality.wait.process", defaultWait).toLong
        case TaskLocality.NODE_LOCAL =>
            conf.get("spark.locality.wait.node", defaultWait).toLong
        case TaskLocality.RACK_LOCAL =>
            conf.get("spark.locality.wait.rack", defaultWait).toLong
        case _ => 0L
    }
}
```

❑ localityWaits：本地化级别等待时间。

localityWaits 实际是对 myLocalityLevels 应用 getLocalityWait 方法获得，代码如下。

```
var localityWaits = myLocalityLevels.map(getLocalityWait) // Time to wait at each level
```

现在一起分析 getAllowedLocalityLevel 方法（见代码清单 5-49），它的处理步骤如下：

1）根据当前本地化级别索引（currentLocalityIndex 刚开始为 0），获取此本地化级别的等待时长；

2）如果当前时间与上次运行本地化时间（lastLaunchTime）之差大于等于上一步获得的时长并且当前本地化级别索引小于 myLocalityLevels 的索引范围，那么将第 1) 步的时长增加到 lastLaunchTime 中，然后使 currentLocalityIndex 增加 1，最后重新从第 1) 步开始执行。（这个过程也称为本地化级别跳级。）

代码清单5-49　getAllowedLocalityLevel的实现

```
private def getAllowedLocalityLevel(curTime: Long): TaskLocality.TaskLocality = {
```

```
        while (curTime - lastLaunchTime >= localityWaits(currentLocalityIndex) &&
            currentLocalityIndex < myLocalityLevels.length - 1)
        {
            lastLaunchTime += localityWaits(currentLocalityIndex)
            currentLocalityIndex += 1
        }
        myLocalityLevels(currentLocalityIndex)
    }
```

经过对 Spark 任务本地化的分析后，读者可能觉得这样的代码实现过于复杂，并且在获取本地化级别的时候竟然每次都要等待一段本地化级别的等待时长，这种实现未免太过奇怪。正如刚开始说的，任何任务都希望被分配到可以从本地读取数据的节点上以得到最大的性能提升。然而每个任务的运行时长都不是事先可以预料的，当一个任务在分配时，如果没有满足最佳本地化（PROCESS_LOCAL）的资源时，如果一直固执地期盼得到最佳的资源，很有可能被已经占用最佳资源但是运行时间很长的任务耽搁，所以这些代码实现了当没有最佳本地化时，退而求其次选择稍微差点的资源。

5.5 执行任务

调用 Executor 的 launchTask 方法（见代码清单 5-50）时，标志着任务执行阶段的开始。launchTask 的执行过程如下。

1）创建 TaskRunner，并将其与 taskId、taskName 及 serializedTask 添加到 runningTasks = new ConcurrentHashMap[Long, TaskRunner] 中。

2）TaskRunner 实现了 Runnable 接口（Scala 中称为继承 Runnable 特质），最后使用线程池（即为 3.8.1 节所述的线程池）执行 TaskRunner。

代码清单5-50 launchTask的实现

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

我们知道线程执行时，会调用 TaskRunner 的 run 方法。run 方法的处理动作包括状态更新、任务反序列化、任务运行。

5.5.1 状态更新

调用 execBackend 的 statusUpdate 方法更新任务状态，代码如下。

```
execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER)
```

以 LocalBackend 为例，实际向 LocalActor 发送 StatusUpdate 消息，代码如下。

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

LocalActor 在接收到 StatusUpdate 事件时，匹配执行 TaskSchedulerImpl 的 statusUpdate 方法，并根据 Task 的最新状态做一系列处理，见代码清单 3-36。

5.5.2　任务还原

所谓任务还原就是将 Driver 提交的 Task 在 Executor 上通过反序列化、更新依赖达到 Task 还原效果的过程。

对代码清单 5-45 中序列化的 serializedTask 执行反序列化操作，代码如下。

```
val (taskFiles, taskJars, taskBytes) = Task.deserializeWithDependencies(serializ
    edTask)
```

更新依赖的文件或者 jar 包，代码如下。

```
updateDependencies(taskFiles, taskJars)
```

updateDependencies 方法（见代码清单 5-51）获取依赖是利用了 Utils.fetchFile 方法实现的（fetchFile 的具体介绍请读者阅读附录 A）。下载的 jar 文件还会添加到 Executor 自身类加载器的 URL 中。

代码清单5-51　获取依赖的实现

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

```
                logInfo("Adding " + url + " to class loader")
                urlClassLoader.addURL(url)
            }
        }
    }
}
```

最后将 Task 的 ByteBuffer 反序列化为 Task 实例，实现如下。

```
task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContext-ClassLoader)
```

5.5.3 任务运行

TaskRunner 最终调用 Task 的 run 方法来运行任务，实现如下。

```
val value = task.run(taskId.toInt)
```

run 方法中创建了 TaskContextImpl，并且设置到 TaskContext 的 ThreadLocal 中。最后调用 runTask 方法，见代码清单 5-52。

代码清单5-52 Task. run的实现

```
final def run(attemptId: Long): T = {
    context = new TaskContextImpl(stageId, partitionId, attemptId, runningLocally
        = false)
    TaskContextHelper.setTaskContext(context)
    context.taskMetrics.hostname = Utils.localHostName()
    taskThread = Thread.currentThread()
    if (_killed) {
        kill(interruptThread = false)
    }
    try {
        runTask(context)
    } finally {
        context.markTaskCompleted()
        TaskContextHelper.unset()
    }
}
```

在 word count 的例子中，首先执行的 Task 是 ShuffleMapTask，那么 ShuffleMapTask 的 runTask 方法（见代码清单 5-53）都做了什么？曾经介绍 submitMissingTasks 的时候，其中对任务的 RDD 和 ShuffleDependency 进行过序列化操作，现在是时候反序列化了，这样可以得到 RDD 和 ShuffleDependency。接下来调用 SortShuffleManager 的 getWriter 方法获取 partitionId 指定分区的 SortShuffleWriter。之后便利用此 Writer 将计算的中间结果写入文件。

代码清单5-53 ShuffleMapTask. runTask的实现

```
override def runTask(context: TaskContext): MapStatus = {
    val ser = SparkEnv.get.closureSerializer.newInstance()
    val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
        ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
```

```
        metrics = Some(context.taskMetrics)
        var writer: ShuffleWriter[Any, Any] = null
        try {
            val manager = SparkEnv.get.shuffleManager
            writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId,
                context)
            writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <:
                Product2[Any, Any]]])
            return writer.stop(success = true).get
        } catch {
            case e: Exception =>
            try {
                if (writer != null) {
                writer.stop(success = false)
                }
            } catch {
                case e: Exception =>
                    log.debug("Could not stop writer", e)
                }
                throw e
            }
    }
```

SortShuffleManager 的 getWriter 实现，见代码清单 5-54。参数 mapId 实际传入的是 partitionId，由此我们可以看到 partition 与 map 任务的关系。

SortShuffleWriter 负责计算结果的缓存处理及持久化，其内容将在第 6 章中展开。我们暂时只需理解的是 map 任务的 Stage 的任务执行结果将通过 SortShuffleManager 持久化到存储体系即可。RDD 的 iterator 方法触发任务计算，笔者也将在第 6 章详述。

代码清单5-54　SortShuffleManager. getWriter的实现

```
override def getWriter[K, V](handle: ShuffleHandle, mapId: Int, context:
    TaskContext)
        : ShuffleWriter[K, V] = {
    val baseShuffleHandle = handle.asInstanceOf[BaseShuffleHandle[K, V, _]]
    shuffleMapNumber.putIfAbsent(baseShuffleHandle.shuffleId, baseShuffleHandle.
        numMaps)
    new SortShuffleWriter(
        shuffleBlockManager, baseShuffleHandle, mapId, context)
}
```

ResultTask 的 runTask 方法实现，见代码清单 5-62。

5.6　任务执行后续处理

5.6.1　计量统计与执行结果序列化

分析代码清单 5-55，可以看到任务执行结束后，还会有以下处理。

1）任务执行结果的简单序列化。

2）计量统计，需要更新的指标有：

- Executor 反序列化消耗的时间；
- Executor 实际执行任务消耗的时间；
- Executor 执行垃圾回收消耗的时间；
- Executor 执行结果序列化消耗的时间。

3）将前两步得到的简单序列化结果和计量统计内容封装为 DirectTaskResult，然后序列化。

代码清单5-55 计量统计及执行结果序列化

```
val taskFinish = System.currentTimeMillis()

    if (task.killed) {
        throw new TaskKilledException
    }

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

    for (m <- task.metrics) {
        m.executorDeserializeTime = taskStart - deserializeStartTime
        m.executorRunTime = taskFinish - taskStart
        m.jvmGCTime = gcTime - startGCTime
        m.resultSerializationTime = afterSerialization - beforeSerialization
    }

    val accumUpdates = Accumulators.values

    val directResult = new DirectTaskResult(valueBytes, accumUpdates, task.
     metrics.orNull)
    val serializedDirectResult = ser.serialize(directResult)
    val resultSize = serializedDirectResult.limit
```

5.6.2 内存回收

TaskRunner 的 run 方法最后还会在 finally（见代码清单 5-56）中做一些清理工作，包括：

1）释放当前线程通过 ShuffleMemoryManager 获得的内存；

2）释放当前线程在 MemoryStore 的 unrollMemoryMap 中展开占用的内存；

3）释放当前线程用于聚合计算占用的内存；

4）将当前 Task 从 runningTasks 中移除。

代码清单5-56 内存回收

```
finally {
    env.shuffleMemoryManager.releaseMemoryForThisThread()
```

```
        env.blockManager.memoryStore.releaseUnrollMemoryForThisThread()
        Accumulators.clear()
        runningTasks.remove(taskId)
    }
```

5.6.3　执行结果处理

任务完成的时候会发送一次 statusUpdate 消息，LocalActor 会先匹配执行 TaskSchedulerImpl 的 statusUpdate 方法，然后调用 reviveOffers 方法调用其他的任务，见代码清单 3-36。

TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 5-57）会从 taskIdToTaskSetId、taskIdToExecutorId 中移除此任务，并且调用 taskResultGetter 的 enqueueSuccessfulTask 方法。

代码清单5-57　执行结果处理

```
taskIdToTaskSetId.get(tid) match {
    case Some(taskSetId) =>
        if (TaskState.isFinished(state)) {
            taskIdToTaskSetId.remove(tid)
            taskIdToExecutorId.remove(tid)
        }
        activeTaskSets.get(taskSetId).foreach { taskSet =>
            if (state == TaskState.FiNISHED) {
                taskSet.removeRunningTask(tid)
                taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)
            } else if (Set(TaskState.FAILED, TaskState.KILLED, TaskState.LOST).
                  contains(state)) {
                taskSet.removeRunningTask(tid)
                taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)
            }
        }
    case None =>
}
```

taskResultGetter 的 enqueueSuccessfulTask 和 enqueueFailedTask 方法，分别用于处理执行成功任务的返回结果和执行失败任务的返回结果。我们以 enqueueSuccessfulTask 方法（见代码清单 5-58）为例。

代码清单5-58　enqueueSuccessfulTask的实现

```
def enqueueSuccessfulTask(
    taskSetManager: TaskSetManager, tid: Long, serializedData: ByteBuffer) {
    getTaskResultExecutor.execute(new Runnable {
        override def run(): Unit = Utils.logUncaughtExceptions {
            try {
                val (result, size) = serializer.get().deserialize[TaskResult[_]]
                      (serializedData) match {
                case directResult: DirectTaskResult[_] =>
                if (!taskSetManager.canFetchMoreResults(serializedData.limit())) {
                    return
                }
```

```
                (directResult, serializedData.limit())
            case IndirectTaskResult(blockId, size) =>
                if (!taskSetManager.canFetchMoreResults(size)) {
                    sparkEnv.blockManager.master.removeBlock(blockId)
                    return
                }
                logDebug("Fetching indirect task result for TID %s".format(tid))
                scheduler.handleTaskGettingResult(taskSetManager, tid)
                val serializedTaskResult = sparkEnv.blockManager.getRemote-
                    Bytes(blockId)
                if (!serializedTaskResult.isDefined) {
                    scheduler.handleFailedTask(
                        taskSetManager, tid, TaskState.FiNISHED, TaskResultLost)
                    return
                }
                val deserializedResult = serializer.get().deserialize[Direct-
                    TaskResult[_]](
                    serializedTaskResult.get)
                sparkEnv.blockManager.master.removeBlock(blockId)
                (deserializedResult, size)
            }

            result.metrics.resultSize = size
            scheduler.handleSuccessfulTask(taskSetManager, tid, result)
        } catch {
            //异常处理代码省略
            }
        }
    })
}
```

从 enqueueSuccessfulTask 的实现不难看出其中另起的线程，主要调用了 TaskSchedulerImpl 的 handleSuccessfulTask 方法。TaskSchedulerImpl 的 handleSuccessfulTask 方法的实现如下。

```
def handleSuccessfulTask(
    taskSetManager: TaskSetManager,
    tid: Long,
    taskResult: DirectTaskResult[_]) = synchronized {
    taskSetManager.handleSuccessfulTask(tid, taskResult)
}
```

TaskSetManager 的 handleSuccessfulTask 方法（见代码清单 5-59）对 TaskSet 中的任务信息进行成功状态标记，然后调用 DagScheduler 的 taskEnded 方法。

代码清单5-59 TaskSetManager. handleSuccessfulTask的实现

```
def handleSuccessfulTask(tid: Long, result: DirectTaskResult[_]) = {
    val info = taskInfos(tid)
    val index = info.index
    info.markSuccessful()
    removeRunningTask(tid)
    sched.dagScheduler.taskEnded(
```

```
            tasks(index), Success, result.value(), result.accumUpdates, info, result.
                metrics)
        if (!successful(index)) {
            tasksSuccessful += 1
            logInfo("Finished task %s in stage %s (TID %d) in %d ms on %s (%d/%d)".format(
                info.id, taskSet.id, info.taskId, info.duration, info.host, tasks-
                    Successful, numTasks))
            // Mark successful and stop if all the tasks have succeeded.
            successful(index) = true
            if (tasksSuccessful == numTasks) {
                isZombie = true
            }
        } else {
            logInfo("Ignoring task-finished event for " + info.id + " in stage " + task-
                Set.id +
                " because task " + index + " has already completed successfully")
        }
        failedExecutors.remove(index)
        maybeFinishTaskSet()
    }
```

DagScheduler 的 taskEnded 方法的实现如下。

```
def taskEnded(
        task: Task[_],
        reason: TaskEndReason,
        result: Any,
        accumUpdates: Map[Long, Any],
        taskInfo: TaskInfo,
        taskMetrics: TaskMetrics) {
    eventProcessActor! CompletionEvent(task, reason, result, accumUpdates,
        taskInfo, taskMetrics)
}
```

DAGSchedulerEventProcessActor 接收 CompletionEvent 消息，将处理交给了 handleTaskCompletion，见代码清单 3-35。handleTaskCompletion 方法首先向 listenerBus 发送 SparkListenerTaskEnd，代码如下。

```
event.reason match {
    case Success =>
        listenerBus.post(SparkListenerTaskEnd(stageId, stage.latestInfo.attemptId,
            taskType,
        event.reason, event.taskInfo, event.taskMetrics))
    stage.pendingTasks -= task
```

之后的处理因 Task 类型而不同。

1. ResultTask 任务的结果处理

如果是 ResultTask，那么将执行代码清单 5-60 所示的代码分支，其处理步骤如下：

1）标识 ActiveJob 的 finished 里对应分区的任务为完成状态，并且将已完成的任务数 numFinished 加 1。

2）如果 ActiveJob 的所有任务都完成，则标记当前 Stage 完成并向 listenerBus 发送 SparkListenerJobEnd 事件。

3）调用 JobWaiter 的 taskSucceeded 方法，以便通知 JobWaiter 有任务成功。

代码清单5-60 ResultTask的结果处理

```
task match {
    case rt: ResultTask[_, _] =>
        stage.resultOfJob match {
            case Some(job) =>
                if (!job.finished(rt.outputId)) {
                    updateAccumulators(event)
                    job.finished(rt.outputId) = true
                    job.numFinished += 1
                    // If the whole job has finished, remove it
                if (job.numFinished == job.numPartitions) {
                    markStageAsFinished(stage)
                    cleanupStateForJobAndIndependentStages(job)
                    listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
                }

                try {
                    job.listener.taskSucceeded(rt.outputId, event.result)
                } catch {
                    case e: Exception =>
                        job.listener.jobFailed(new SparkDriverExecutionException(e))
                    }
                }
            case None =>
                logInfo("Ignoring result from " + rt + " because its job has
                    finished")
}
```

JobWaiter 的 taskSucceeded 方法（见代码清单 5-24），其处理步骤如下：

1）JobWaiter 中的 resultHandler 实际是代码清单 5-20 里的匿名函数 (index, res) => results(index) = res，通过回调此匿名函数，将当前任务的结果加入最终结果集。

2）finishedTasks 自增，当完成任务数 finishedTasks 等于全部任务数 totalTasks 时，标记 job 完成，并且唤醒等待的线程，即执行代码清单 5-22 中调用 awaitResult 方法的线程。

2. ShuffleMapTask 任务的结果处理

如果是 ShuffleMapTask，那么将执行代码清单 5-61 所示的代码分支，其处理步骤如下。

1）将 Task 的 partitionId 和 MapStatus 追加到 Stage 的 outputLocs 中。

2）将当前 Stage 标记为完成，然后将当前 Stage 的 shuffleId 和 outputLocs 中的 MapStatus 注册到 mapOutputTracker。根据 3.2.3 节的内容，这里注册的 map 任务状态将最终被 reduce 任务所用。

3）如果 Stage 的 outputLocs 中某个分区的输出为 Nil，那么说明有任务失败了，这时需要

再次提交此 Stage。

4）如果不存在 Stage 的 outputLocs 中某个分区的输出为 Nil，那么说明所有任务执行成功了，这时需要遍历 waitingStages 中的 Stage 并将它们放入 runningStages，最后调用 submitMissingTasks 方法逐个提交这些准备运行的 Stage 的任务。在 word count 例子里，由于 map 任务的 Stage 已经运行完成，现在运行的是 reduce 任务的 Stage，所以此时调用 submitMissingTasks 方法则创建了 ResultTask。

代码清单5-61　ShuffleMapTask的结果处理

```
case smt: ShuffleMapTask =>
    updateAccumulators(event)
    val status = event.result.asInstanceOf[MapStatus]
    val execId = status.location.executorId
    logDebug("ShuffleMapTask finished on " + execId)
    if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {
        logInfo("Ignoring possibly bogus ShuffleMapTask completion from " +
            execId)
    } else {
        stage.addOutputLoc(smt.partitionId, status)
    }
    if (runningStages.contains(stage) && stage.pendingTasks.isEmpty) {
        markStageAsFinished(stage)
        logInfo("looking for newly runnable stages")
        logInfo("running: " + runningStages)
        logInfo("waiting: " + waitingStages)
        logInfo("failed: " + failedStages)
        if (stage.shuffleDep.isDefined) {
            mapOutputTracker.registerMapOutputs(
                stage.shuffleDep.get.shuffleId,
                stage.outputLocs.map(list => if (list.isEmpty) null else list.
                    head).toArray,changeEpoch = true)
            }
            clearCacheLocs()
            if (stage.outputLocs.exists(_ == Nil)) {
                logInfo("Resubmitting " + stage + " (" + stage.name +
                    ") because some of its tasks had failed: " +
                    stage.outputLocs.zipWithIndex.filter(_._1 == Nil).map(_._2).
                        mkString(", "))
                submitStage(stage)
            } else {
                val newlyRunnable = new ArrayBuffer[Stage]
                for (stage <- waitingStages) {
                    logInfo("Missing parents for " + stage + ": " + getMissing-
                        ParentStages(stage))
                }
                for (stage <- waitingStages if getMissingParentStages(stage) ==
                     Nil) {
                    newlyRunnable += stage
                }
                waitingStages --= newlyRunnable
```

```
                runningStages ++= newlyRunnable
                for {
                    stage <- newlyRunnable.sortBy(_.id)
                    jobId <- activeJobForStage(stage)
                } {
                    logInfo("Submitting " + stage + " (" + stage.rdd + "), which
                        is now runnable")
                    submitMissingTasks(stage, jobId)
                }
            }
        }
    }
```

ResultTask 的 runTask 方法与 ShuffleMapTask 有很多不同，见代码清单 5-62。

代码清单5-62 ResultTask.runTask的实现

```
override def runTask(context: TaskContext): U = {
    // Deserialize the RDD and the func using the broadcast variables.
    val ser = SparkEnv.get.closureSerializer.newInstance()
    val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](
        ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContext-
            ClassLoader)

    metrics = Some(context.taskMetrics)
    func(context, rdd.iterator(partition, context))
}
```

此处调用 RDD 的 iterator 方法完成计算，请参阅 6.5 节。最后回调代码清单 5-18 中的偏函数（iter：Iterator[T]）=> iter.toArray。

word count 例子最终执行的结果如下。

5.7　小结

本章首先从 Spark 为什么设计 RDD 入手，依次讲解 RDD 的实现分析、Stage 的划分、提交 Stage、提交 Task、任务执行、执行结果处理等内容。

对于资源分配中涉及的本地化实现，本章也做了较为详细的分析，Spark 通过一种阶梯式的本地化策略，在有效利用资源、节省网络 I/O 的同时提高了系统执行的效率。

容错能力方面，Spark 通过 DAG 构成的有向无环图可以在其中某些任务执行失败的情况下，通过重新提交任务达到容错。而那些执行成功的任务由于其结果数据已经在缓存中，所以不用重复计算。

Chapter 6 第6章

计算引擎

捭阖者，道之大化，说之变也。必豫审其变化。

——《鬼谷子》

本章导读

Spark 的计算是一个层层迭代的过程。本章将 5.5.3 节与 5.6.3 两节中有关计算的内容单独抽出来，是为了让本书的内容及轮廓更清晰。读者可以先阅读第 5 章，也可以直接阅读本章内容，了解计算的有趣过程。

RDD 作为 Spark 对各种数据计算模型的统一抽象，被用于迭代计算过程以及任务输出结果的缓存读写。在所有 MapReduce 框架中，shuffle 是连接 map 任务和 reduce 任务的桥梁。map 任务的中间输出要作为 reduce 任务的输入，就必须经过 shuffle，shuffle 的性能优劣直接决定了整个计算引擎的性能和吞吐量。相比于 Hadoop 的 MapReduce，我们可以看到 Spark 提供多种计算结果处理的方式，对 shuffle 过程进行了优化。

本章将继续以 word count 为例讲解。

6.1 迭代计算

MappedRDD 的 iterator 方法实际是父类 RDD 的 iterator 方法，见代码清单 6-1。如果分区任务初次执行，此时还没有缓存，所以会调用 computeOrReadCheckpoint 方法。

这里需要说一下 iterator 方法的容错处理过程：如果某个分区任务执行失败，但是其他分区任务执行成功，可以利用 DAG 重新调度。失败的分区任务将从检查点恢复状态，而那些执行成功的分区任务由于其执行结果已经缓存到存储体系，所以调用 CacheManager 的 getOrCompute 方法获取即可，不需要再次执行。

代码清单6-1 iterator方法实现

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
    if (storageLevel != StorageLevel.NONE) {
        SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
    } else {
        computeOrReadCheckpoint(split, context)
    }
}
```

CacheManager 的有关内容已在 4.10 节介绍过，我们主要分析 computeOrReadCheckpoint 方法。computeOrReadCheckpoint 在存在检查点时直接获取中间结果，否则需要调用 compute 继续计算，代码如下。

```
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext):
    Iterator[T] =
{
    if (isCheckpointed) firstParent[T].iterator(split, context) else compute(split,context)
}
```

MappedRDD 的 compute 方法实现如下。

```
override def compute(split: Partition, context: TaskContext) =
    firstParent[T].iterator(split, context).map(f)
```

MappedRDD 的 compute 方法首先调用 firstParent 找到其父 RDD（有关 firstParent 方法的内容，请阅读 5.3.2 节），本例中 MappedRDD 的父 RDD 为 FlatMappedRDD。FlatMappedRDD 的 iterator 方法，其实也是 RDD 的 iterator。

经过 RDD 管道中对 iterator 和 computeOrReadCheckpoint 的层层调用，最终到达 HadoopRDD。查看此时的线程栈更直观，如图 6-1 所示。

图 6-1 word count 例子的线程栈

HadoopRDD的compute方法用来创建NextIterator的匿名内部类，然后将其封装为InterruptibleIterator，见代码清单6-2。

代码清单6-2 HadoopRDD.compute的实现

```
override def compute(theSplit: Partition, context: TaskContext): Interruptible-
    Iterator[(K, V)] = {
    val iter = new NextIterator[(K, V)] {

        val split = theSplit.asInstanceOf[HadoopPartition]
        val jobConf = getJobConf()

        val inputMetrics = new InputMetrics(DataReadMethod.Hadoop)
        val bytesReadCallback = if (split.inputSplit.value.isInstanceOf[FileSplit]) {
            SparkHadoopUtil.get.getFSBytesReadOnThreadCallback(
                split.inputSplit.value.asInstanceOf[FileSplit].getPath, jobConf)
        } else {
            None
        }
        if (bytesReadCallback.isDefined) {
            context.taskMetrics.inputMetrics = Some(inputMetrics)
        }

        var reader: RecordReader[K, V] = null
        val inputFormat = getInputFormat(jobConf)
        HadoopRDD.addLocalConfiguration(new SimpleDateFormat("yyyyMMddHHmm").
            format(createTime),
            context.stageId, theSplit.index, context.attemptId.toInt, jobConf)
        reader = inputFormat.getRecordReader(split.inputSplit.value, jobConf,
            Reporter.NULL)

        context.addTaskCompletionListener{ context => closeIfNeeded() }
        val key: K = reader.createKey()
        val value: V = reader.createValue()

        var recordsSinceMetricsUpdate = 0

        override def getNext() = {
            try {
                finished = !reader.next(key, value)
            } catch {
                case eof: EOFException =>
                    finished = true
            }

            if (recordsSinceMetricsUpdate == HadoopRDD.RECORDS_BETWEEN_BYTES_
                 READ_METRIC_UPDATES
                    && bytesReadCallback.isDefined) {
                recordsSinceMetricsUpdate = 0
                val bytesReadFn = bytesReadCallback.get
            inputMetrics.bytesRead = bytesReadFn()
            } else {
                recordsSinceMetricsUpdate += 1
```

```
                }
                (key, value)
            }
            // 省略关闭RecordReader的代码
        }
        new InterruptibleIterator[(K, V)](context, iter)
    }
```

构造 NextIterator 的过程如下：

1）从 broadcast 中获取 jobConf，此处的 jobConf 正是代码清单 5-3 中的 hadoopConfiguration。

2）创建 InputMetrics 用于计算字节读取的测量信息，然后在 RecordReader 正式读取数据之前创建 bytesReadCallback。bytesReadCallback 用于获取当前线程从文件系统读取的字节数。

3）获取 inputFormat，此处的 inputFormat 正是代码清单 5-2 中的 TextInputFormat。

4）使用 addLocalConfiguration 给 JobConf 添加 Hadoop 任务相关配置。addLocalConfiguration 的实现见代码清单 6-3。

代码清单6-3 HadoopRDD.addLocalConfiguration的实现

```
def addLocalConfiguration(jobTrackerId: String, jobId: Int, splitId: Int,
    attemptId: Int,
                              conf: JobConf) {
    val jobID = new JobID(jobTrackerId, jobId)
    val taId = new TaskAttemptID(new TaskID(jobID, true, splitId), attemptId)

    conf.set("mapred.tip.id", taId.getTaskID.toString)
    conf.set("mapred.task.id", taId.toString)
    conf.setBoolean("mapred.task.is.map", true)
    conf.setInt("mapred.task.partition", splitId)
    conf.set("mapred.job.id", jobID.toString)
}
```

5）创建 RecordReader，调用 reader.createKey() 和 reader.createValue() 得到的正是代码清单 5-2 中的 LongWritable 和 Text。NextIterator 的 getNext 实际是代理了 RecordReader 的 next 方法并且每读取一些记录后使用 bytesReadCallback 更新 InputMetrics 的 bytesRead 字段。

6）将 NextIterator 封装为 InterruptibleIterator。

InterruptibleIterator 只是对 NextIterator 的代理，见代码清单 6-4。

代码清单6-4 InterruptibleIterator的实现

```
class InterruptibleIterator[+T](val context: TaskContext, val delegate:
    Iterator[T])
    extends Iterator[T] {

    def hasNext: Boolean = {
        if (context.isInterrupted) {
            throw new TaskKilledException
        } else {
            delegate.hasNext
```

```
            }
        }

        def next(): T = delegate.next()
    }
```

根据 5.5.3 节的内容，我们知道整个 rdd.iterator 调用结束，最后返回 InterruptibleIterator 对象后，会调用 SortShuffleWriter 的 write 方法（见代码清单 6-5），其功能如下：

1）创建 ExternalSorter，然后调用 insertAll 将计算结果写入缓存。

2）调用 shuffleBlockManager.getDataFile 方法获取当前任务要输出的文件路径，请参阅 4.13 节。

3）调用 shuffleBlockManager.consolidateId 创建 blockId，请参阅 4.13 节所讲的 ShuffleBlockId。

4）调用 ExternalSorter 的 writePartitionedFile 将中间结果持久化。

5）调用 shuffleBlockManager.writeIndexFile 方法创建索引文件。

6）创建 MapStatus。

代码清单6-5 SortShuffleWriter.write的实现

```
override def write(records: Iterator[_ <: Product2[K, V]]): Unit = {
    if (dep.mapSideCombine) {
        require(dep.aggregator.isDefined, "Map-side combine without Aggregator
            specified!")
        sorter = new ExternalSorter[K, V, C](
            dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)
        sorter.insertAll(records)
    } else {
        sorter = new ExternalSorter[K, V, V](
            None, Some(dep.partitioner), None, dep.serializer)
        sorter.insertAll(records)
    }

    val outputFile = shuffleBlockManager.getDataFile(dep.shuffleId, mapId)
    val blockId = shuffleBlockManager.consolidateId(dep.shuffleId, mapId)
    val partitionLengths = sorter.writePartitionedFile(blockId, context,
        outputFile)
    shuffleBlockManager.writeIndexFile(dep.shuffleId, mapId, partitionLengths)

    mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)
}
```

6.2 什么是 shuffle

shuffle 是所有 MapReduce 计算框架所必须经过的阶段，shuffle 用于打通 map 任务的输出与 reduce 任务的输入，map 任务的中间输出结果按照 key 值哈希后分配给某一个 reduce 任务，这个过程如图 6-2 所示。

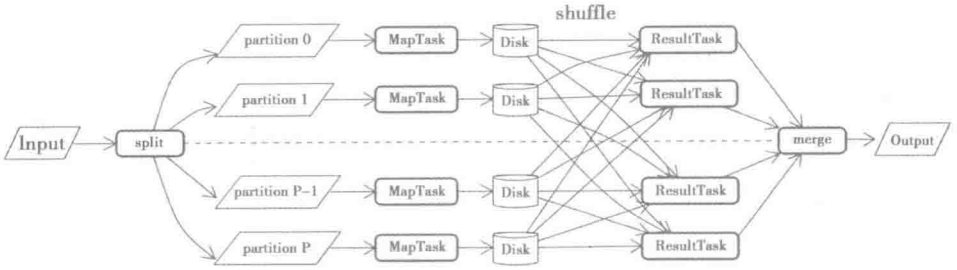


图 6-2 MapReduce 计算框架的 shuffle 过程示意图

在具体分析源码之前，我们先看看 Spark 早期版本的 shuffle 是怎样的，如图 6-3 所示。

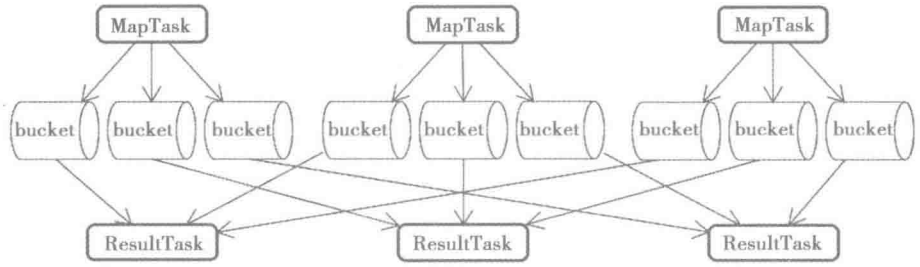


图 6-3 Spark 早期版本的 shuffle 过程

这里对图 6-3 做一些解释：

1）map 任务会为每一个 reduce 任务创建一个 bucket。假设有 M 个 map 任务，R 个 reduce 任务，则 map 阶段一共会创建 $M \times R$ 个 bucket；

2）map 任务会将产生的中间结果按照 partition 写入不同的 bucket 中；

3）reduce 任务从本地或者远端的 map 任务所在的 BlockManager 获取相应的 bucket 作为输入。

Spark 早期的 shuffle 过程存在以下问题：

1）map 任务的中间结果首先存入内存，然后才写入磁盘。这对于内存的开销很大，当一个节点上 map 任务的输出结果集很大时，很容易导致内存紧张，进而发生内存溢出（out of memory，OOM）；

2）每个 map 任务都会输出 R（reduce 任务数量）个 bucket。假如 M 等于 1000，R 也等于 1000，那么共计生成 100 万个 bucket，在 bucket 本身不大，但是 shuffle 很频繁的情况下，磁盘 I/O 将成为性能瓶颈。

熟悉 Hadoop 的读者应该知道，Hadoop MapReduce 的 shuffle 过程存在以下问题：

1）reduce 任务获取到 map 任务的中间输出后，会对这些数据在磁盘上进行 merge sort，虽然不怎么占用内存，但是却产生了更多的磁盘 I/O；

2）当数据量很小，但是 map 任务和 reduce 任务数目很多时，会产生很多网络 I/O。

为了解决以上 Hadoop MapReduce 和早期 Spark 在 shuffle 过程中的性能问题，目前 Spark 的 shuffle 已经做了多种性能优化，主要解决方法包括：

1）将 map 任务给每个 partition 的 reduce 任务输出的 bucket 合并到同一个文件中，这解决了 bucket 数量很多，但是本身数据体积不大时，造成 shuffle 很频繁，磁盘 I/O 成为性能瓶颈的问题；

2）map 任务逐条输出计算结果，而不是一次性输出到内存，并使用 AppendOnlyMap 缓存及其聚合算法对中间结果进行聚合，这大大减小了中间结果所占的内存大小；

3）对 SizeTrackingAppendOnlyMap 和 SizeTrackingPairBuffer 等缓存进行溢出判断，当超出 myMemoryThreshold 的大小时，将数据写入磁盘，防止内存溢出；

4）reduce 任务对拉取到的 map 任务中间结果逐条读取，而不是一次性读入内存，并在内存中进行聚合和排序（其本质上也使用了 AppendOnlyMap 缓存），这也大大减小了数据占用的内存；

5）reduce 任务将要拉取的 Block 按照 BlockManager 地址划分，然后将同一 BlockManager 地址中的 Block 累积为少量网络请求，减少网络 I/O。

在接下来的源码分析过程中，我们一起来看看这些问题是如何解决的。

6.3 map 端计算结果缓存处理

在详细介绍 map 端对中间计算结果的细节之前，先理解两个概念：

- bypassMergeThreshold：传递到 reduce 端再做合并（merge）操作的阈值。如果 partition 的数量小于 bypassMergeThreshold 的值，则不需要在 Executor 执行聚合和排序操作，只需要将各个 partition 直接写到 Executor 的存储文件，最后在 reduce 端再做串联。通过配置 spark.shuffle.sort.bypassMergeThreshold 可以修改 bypassMergeThreshold 的大小，在分区数量小的时候提升计算引擎的性能。bypassMergeThreshold 的默认值是 200。
- bypassMergeSort：标记是否传递到 reduce 端再做合并和排序，即是否直接将各个 partition 直接写到 Executor 的存储文件。当没有定义 aggregator、ordering 函数，并且 partition 的数量小于等于 bypassMergeThreshold 时，bypassMergeSort 为 true。如果 bypassMergeSort 为 true，map 中间结果将直接输出到磁盘，此时不会占用太多内存，避免了内存撑爆问题。

map 端计算结果缓存（见代码清单 6-6）有三种处理方式：

- map 端对计算结果在缓存中执行聚合和排序。
- map 不使用缓存，也不执行聚合和排序，直接调用 spillToPartitionFiles 将各个 partition 直接写到自己的存储文件（即 bypassMergeSort 为 true 的情况），最后由 reduce 端对计算结果执行合并和排序。spillToPartitionFiles 的实现，请参阅 6.4.1 节。
- map 端对计算结果简单缓存。

代码清单6-6 ExternalSorter.insertAll的实现

```
def insertAll(records: Iterator[_ <: Product2[K, V]]): Unit = {
    val shouldCombine = aggregator.isDefined

    if (shouldCombine) {
        // Combine values in-memory first using our AppendOnlyMap
        val mergeValue = aggregator.get.mergeValue
        val createCombiner = aggregator.get.createCombiner
        var kv: Product2[K, V] = null
        val update = (hadValue: Boolean, oldValue: C) => {
            if (hadValue) mergeValue(oldValue, kv._2) else createCombiner(kv._2)
        }
        while (records.hasNext) {
            addElementsRead()
            kv = records.next()
            map.changeValue((getPartition(kv._1), kv._1), update)
            maybeSpillCollection(usingMap = true)
        }
    } else if (bypassMergeSort) {
        if (records.hasNext) {
            spillToPartitionFiles(records.map { kv =>
                ((getPartition(kv._1), kv._1), kv._2.asInstanceOf[C])
            })
        }
    } else {
        while (records.hasNext) {
            addElementsRead()
            val kv = records.next()
            buffer.insert((getPartition(kv._1), kv._1), kv._2.asInstanceOf[C])
            maybeSpillCollection(usingMap = false)
        }
    }
}
```

我们先来分析两种需要缓存的方式。

6.3.1 map 端计算结果缓存聚合

一个任务的分区数量通常很多，如果只是简单地将数据存储到 Executor 上。在执行 reduce 任务时会存在大量的网络 I/O 操作，这时网络 I/O 将成为系统性能的瓶颈，reduce 任务读取 map 任务的计算结果变慢，导致其他想要分配到被这些 map 任务占用的节点的任务不得不等待或者降低本地化选择分配到更远的节点上。对于更远节点的 I/O 本身会更慢，因此还会导致更多的任务得不到分配或者无法高效本地化。经过这样的恶性循环，整个集群将变得迟钝，新的任务长时间得不到执行或者执行变慢。

通过在 map 端对计算结果在缓存中执行聚合和排序，能够节省 I/O 操作，进而提升系统性能。这种情况下，必须要定义聚合器（aggregator）函数，以便于对计算结果按照 partitionID 和 key 聚合后排序。

ExternalSorter 的 insertAll 方法（见代码清单 6-6）中，如果定义了 aggregator，则 shouldCombine 为 true。此分支执行过程如下：

1）由于设置了聚合函数 aggregator，则从聚合函数获取 mergeValue（word count 例子中为 Function2）、createCombiner（word count 例子中为 PairFunction）等函数。

2）定义 update 偏函数，此函数用于操作 mergeValue 和 createCombiner。

3）迭代之前创建的 iterator，每读取一条 Product2[K, V]（此时真正执行代码清单 5-1 中的 FlatMapFunction 和 PairFunction），将每行字符串按照空格切分，并且给每个文本设置 1，比如 (#,1)、(Apache,1)、(Spark,1)…。

4）以（分区索引，Product2[K, V]._1）为参数调用 SizeTrackingAppendOnlyMap 的 changeValue 函数（见代码清单 6-7），与 update 函数配合，按照 key 值叠加 value。

5）调用 maybeSpillCollection 方法，来处理 SizeTrackingAppendOnlyMap 溢出（当 SizeTrackingAppendOnlyMap 的大小超过 myMemoryThreshold 时，将集合中的数据写入磁盘并新建 SizeTrackingAppendOnlyMap）。这样做是为了防止内存溢出，解决了 Spark 早期版本 shuffle 的内存撑爆问题。

代码清单6-7　SizeTrackingAppendOnlyMap.changeValue方法的实现

```
override def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
    val newValue = super.changeValue(key, updateFunc)
    super.afterUpdate()
    newValue
}
```

SizeTrackingAppendOnlyMap 的 changeValue 方法的处理包括三步：

1）调用父类 AppendOnlyMap 的 changeValue 函数，应用缓存聚合算法。

2）调用继承特质 SizeTracker 的 afterUpdate 函数，增加对 AppendOnlyMap 大小的采样。

3）返回第 1) 步计算的结果。

1. AppendOnlyMap 的缓存聚合算法

SizeTrackingAppendOnlyMap 的父类 AppendOnlyMap 的 changeValue 函数（见代码清单 6-8）用于回调 update 函数进行聚合操作。其实现可以说明，AppendOnlyMap 支持 null 值的缓存，而 Java 的 map 默认是不支持的。changeValue 方法利用一种使用数据缓存的算法完成聚合。在介绍此算法前先弄清一些定义：

- LOAD_FACTOR：负载因子，常量值等于 0.7。
- initialCapacity：初始容量值 64。
- capacity：容量，初始时等于 initialCapacity。
- curSize：记录当前已经放入 data 的 key 与聚合值的数量。
- data：数组，初始大小为 2 * capacity，data 数组的实际大小之所以是 capacity 的 2 倍，是因为 key 和聚合值各占一位。

- growThreshold：data 数组容量增加的阈值，表达式为 growThreshold = LOAD_FACTOR * capacity。
- mask：计算数据存放位置的掩码值，表达式为 capacity − 1。
- k：要放入 data 的 key。
- pos：k 将要放入 data 的索引值。索引值等于 k 的哈希值再次计算哈希值的结果与 mask 按位 & 运算的值。表达式为 pos = rehash（k.hashCode）& mask。
- curKey：data（2 * pos）位置的当前 key。
- newValue：key 的聚合值。

在掌握以上概念的前提下，给出以下算法描述：

条件 1：如果 curKey 等于 null，那么 newValue 等于 1；

条件 2：如果 curKey 不等于 null 并且不等于 k，那么从 pos 当前位置向后找，直到此位置的索引值与 mask 按位 & 运算后的新位置的 key 符合条件 1 或者条件 3；

条件 3：如果 curKey 不等于 null 并且等于 k，那么 newValue 等于 data(2 * pos + 1) 与 k 对应的值按照 mergeValue 定义的函数运算（在 word count 例子中，mergeValue 是代码清单 5-1 中的 Function2 函数）。

代码清单6-8 AppendOnlyMap.changeValue方法的实现

```
def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
    assert(!destroyed, destructionMessage)
    val k = key.asInstanceOf[AnyRef]
    if (k.eq(null)) {
        if (!haveNullValue) {
            incrementSize()
        }
        nullValue = updateFunc(haveNullValue, nullValue)
        haveNullValue = true
        return nullValue
    }
    var pos = rehash(k.hashCode) & mask
    var i = 1
    while (true) {
        val curKey = data(2 * pos)
        if (k.eq(curKey) || k.equals(curKey)) {
            val newValue = updateFunc(true, data(2 * pos + 1).asInstanceOf[V])
            data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
            return newValue
        } else if (curKey.eq(null)) {
            val newValue = updateFunc(false, null.asInstanceOf[V])
            data(2 * pos) = k
            data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
            incrementSize()
            return newValue
        } else {
        val delta = i
            pos = (pos + delta) & mask
```

```
            i += 1
        }
    }
    null.asInstanceOf[V] // Never reached but needed to keep compiler happy
}
```

2. AppendOnlyMap 的容量增长

incrementSize 方法（见代码清单 6-9）用于扩充 AppendOnlyMap 的容量。当 curSize > growThreshold 时，调用 growTable 方法将 capacity 容量扩大一倍，即 newCapacity = capacity * 2。

growTable 方法（见代码清单 6-10）先创建 newCapacity 的两倍大小的新数组，将老数组中的元素复制到新数组中，新数组索引位置采用新的 mask 重新使用 rehash（k.hashCode）&mask 计算。

代码清单6-9 AppendOnlyMap.incrementSize方法代码

```
private def incrementSize() {
    curSize += 1
    if (curSize > growThreshold) {
        growTable()
    }
}
```

代码清单6-10 AppendOnlyMap.growTable方法代码

```
protected def growTable() {
    val newCapacity = capacity * 2
    if (newCapacity >= (1 << 30)) {
        throw new Exception("Can't make capacity bigger than 2^29 elements")
    }
    val newData = new Array[AnyRef](2 * newCapacity)
    val newMask = newCapacity - 1

    var oldPos = 0
    while (oldPos < capacity) {
        if (!data(2 * oldPos).eq(null)) {
            val key = data(2 * oldPos)
            val value = data(2 * oldPos + 1)
            var newPos = rehash(key.hashCode) & newMask
            var i = 1
            var keepGoing = true
            while (keepGoing) {
                val curKey = newData(2 * newPos)
                if (curKey.eq(null)) {
                    newData(2 * newPos) = key
                    newData(2 * newPos + 1) = value
                    keepGoing = false
                } else {
                    val delta = i
                    newPos = (newPos + delta) & newMask
                    i += 1
                }
            }
        }
        oldPos += 1
```

```
    }
    data = newData
    capacity = newCapacity
    mask = newMask
    growThreshold = (LOAD_FACTOR * newCapacity).toInt
}
```

经过以上算法的运算，word count 例子的数据集合中间计算结果会变为 ((0, site,), 1)、((0, which), 2)、((0, Hadoop), 4) 的样子，证明确实发生了聚合。

3. AppendOnlyMap 大小采样

代码清单 6-10 列出了 AppendOnlyMap 的容量增长实现方法 growTable，那是不是意味着 AppendOnlyMap 的容量可以无限制增长呢？当然不是，我们需要对 AppendOnlyMap 大小进行限制。很明显我们可以在 AppendOnlyMap 的每次更新操作之后计算它的大小，这应该没有什么问题。Spark 为大数据平台需要提供实时计算能力，无论是数据量还是对 CPU 的开销都很大，每当发生 update 或者 insert 操作就计算一次大小会严重影响 Spark 的性能，因此 Spark 实际采用了采样并利用这些采样对 AppendOnlyMap 未来的大小进行估算或推测的方式。

SizeTrackingAppendOnlyMap 继承了特质 SizeTracker，其 afterUpdate（见代码清单 6-11）用于每次更新 AppendOnlyMap 的缓存后进行采样，采样前提是已经到达设定的采样间隔（nextSampleNum），其处理步骤如下：

1）将 AppendOnlyMap 所占的内存进行估算并且与当前编号（numUpdates）一起作为样本数据更新到 samples ＝ new mutable.Queue[Sample] 中。

2）如果当前采样数量大于 2，则使 samples 执行一次出队操作，保证样本总数等于 2。

3）计算每次更新增加的大小，公式如下：

$$\text{bytesPerUpdate} = \frac{\text{本次采集大小} - \text{上次采样大小}}{\text{本次采集编号} - \text{上次采样编号}}$$

如果样本数小于 2，那么 bytesPerUpdate ＝ 0。

4）计算下次采样的间隔 nextSampleNum。

代码清单6-11 SizeTracker.afterUpdate的实现

```
protected def afterUpdate(): Unit = {
    numUpdates += 1
    if (nextSampleNum == numUpdates) {
        takeSample()
    }
}

private def takeSample(): Unit = {
    samples.enqueue(Sample(SizeEstimator.estimate(this), numUpdates))
    // Only use the last two samples to extrapolate
    if (samples.size > 2) {
        samples.dequeue()
    }
```

```
        val bytesDelta = samples.toList.reverse match {
            case latest :: previous :: tail =>
                (latest.size - previous.size).toDouble / (latest.numUpdates - previous.numUpdates)
            // If fewer than 2 samples, assume no change
            case _ => 0
        }
        bytesPerUpdate = math.max(0, bytesDelta)
        nextSampleNum = math.ceil(numUpdates * SAMPLE_GROWTH_RATE).toLong
    }
```

AppendOnlyMap 的大小采样数据用于推测 AppendOnlyMap 未来的大小，推测的实现见代码清单 6-12。由于 SizeTrackingPairBuffer 也继承了 SizeTracker，所以 estimateSize 方法不但对 AppendOnlyMap 也对 SizeTrackingPairBuffer 在内存中的容量进行限制以防内存溢出时发挥其作用。有关 SizeTrackingPairBuffer 的介绍请看 6.3.2 节。

代码清单6-12 SizeTracker.estimateSize方法代码

```
def estimateSize(): Long = {
    assert(samples.nonEmpty)
    val extrapolatedDelta = bytesPerUpdate * (numUpdates - samples.last.numUpdates)
    (samples.last.size + extrapolatedDelta).toLong
}
```

6.3.2 map 端计算结果简单缓存

ExternalSorter 的 insertAll 方法中，如果没有定义 aggregator，那么 shouldCombine 为 false，见代码清单 6-6。这时会调用 SizeTrackingPairBuffer 的 insert 方法（见代码清单 6-13），从其实现可以知道，它只不过是把计算结果简单地缓存到数组中了。

代码清单6-13 SizeTrackingPairBuffer.insert代码实现

```
def insert(key: K, value: V): Unit = {
    if (curSize == capacity) {
        growArray()
    }
    data(2 * curSize) = key.asInstanceOf[AnyRef]
    data(2 * curSize + 1) = value.asInstanceOf[AnyRef]
    curSize += 1
    afterUpdate()
  }
```

下面我们来介绍 SizeTrackingPairBuffer 的容量增长。

SizeTrackingPairBuffer 的容量增长是通过 growArray 方法实现的。growArray 实现增长 data 数组容量的方式非常简单，只是新建 2 倍大小的新数组，然后简单复制而已，见代码清单 6-14。

新建的数组大小有可能超过 Int 类型的最大值，所以会抛出异常。

代码清单6-14 SizeTrackingPairBuffer.growArray代码实现

```
private def growArray(): Unit = {
    if (capacity == (1 << 29)) {
        throw new Exception("Can't grow buffer beyond 2^29 elements")
    }
    val newCapacity = capacity * 2
    val newArray = new Array[AnyRef](2 * newCapacity)
    System.arraycopy(data, 0, newArray, 0, 2 * capacity)
    data = newArray
    capacity = newCapacity
    resetSamples()
}
```

Spark 使用 SizeTrackingPairBuffer 的过程中，也会调用 maybeSpillCollection 方法，来处理 SizeTrackingPairBuffer 溢出（当 SizeTrackingPairBuffer 的大小超过 myMemoryThreshold 时，将集合中的数据写入磁盘并新建 SizeTrackingPairBuffer）。这样做是为了防止内存溢出，解决了 Spark 早期版本 shuffle 的内存撑爆问题。

6.3.3 容量限制

既然 AppendOnlyMap 和 SizeTrackingPairBuffer 的容量都可以增长，那么数据量不大的时候不会有问题。但由于大数据处理的数据量往往都很大，全部都放入内存会将系统的内存撑爆。Spark 为了防止这个问题的发生，提供了函数 maybeSpillCollection，见代码清单 6-15。

代码清单6-15 ExternalSorter.maybeSpillCollection代码实现

```
private def maybeSpillCollection(usingMap: Boolean): Unit = {
    if (!spillingEnabled) {
        return
    }

    if (usingMap) {
        if (maybeSpill(map, map.estimateSize())) {
            map = new SizeTrackingAppendOnlyMap[(Int, K), C]
        }
    } else {
        if (maybeSpill(buffer, buffer.estimateSize())) {
            buffer = new SizeTrackingPairBuffer[(Int, K), C]
        }
    }
}
```

1. 集合溢出判定

maybeSpillCollection 判定集合是否溢出主要由 maybeSpill 函数来决定，见代码清单 6-16。maybeSpill 函数的处理步骤如下：

1）为当前线程尝试获取 amountToRequest 大小的内存（amountToRequest = 2 * current-Memory − myMemoryThreshold）。shuffleMemoryManager 的 tryToAcquire 方法已在 4.14 节详细

介绍过。

2）如果获得的内存依然不足（myMemoryThreshold <= currentMemory），则调用 spill，执行溢出操作。内存不足可能是申请到的内存为 0 或者已经申请得到的内存大小超过了 myMemoryThreshold。

3）溢出后续处理，如 elementsRead 归零，已溢出内存字节数（memoryBytesSpilled）增加线程当前内存大小（currentMemory），释放当前线程占用的内存。

代码清单6-16 Spillable.maybeSpill的实现

```
protected def maybeSpill(collection: C, currentMemory: Long): Boolean = {
    if (elementsRead > trackMemoryThreshold && elementsRead % 32 == 0 &&
            currentMemory >= myMemoryThreshold) {
        val amountToRequest = 2 * currentMemory - myMemoryThreshold
        val granted = shuffleMemoryManager.tryToAcquire(amountToRequest)
        myMemoryThreshold += granted
        if (myMemoryThreshold <= currentMemory) {
            _spillCount += 1
            logSpillage(currentMemory)

            spill(collection)

            _elementsRead = 0
            _memoryBytesSpilled += currentMemory
            releaseMemoryForThisThread()
            return true
        }
    }
    false
}
```

2. 溢出

spill 方法的实现，见代码清单 6-17。如果 bypassMergeSort 为真，则调用 spillToPartitionFiles 将内存中的数据溢出到分区文件，具体细节请参阅 6.4.1 节。如果 bypassMergeSort 不为真，则调用 spillToMergeableFile。

代码清单6-17 ExternalSorter.spill方法的实现

```
override protected[this] def spill(collection: SizeTrackingPairCollection[(Int,
    K), C]): Unit = {
    if (bypassMergeSort) {
        spillToPartitionFiles(collection)
    } else {
        spillToMergeableFile(collection)
    }
}
```

spillToMergeableFile 方法（见代码清单 6-18）的处理步骤如下：

1）调用 createTempShuffleBlock 创建临时文件，createTempShuffleBlock 的实现已在 4.4.3 节讲述过。

2）新建 ShuffleWriteMetrics 用于测量。

3）调用 getDiskWriter 方法创建 DiskBlockObjectWriter，具体内容参阅 4.8.7 节。

4）调用 destructiveSortedIterator 方法对集合元素排序，在 6.4.2 节将会详细介绍。

5）将集合内容写入临时文件。写入的时机有两个：

- 集合遍历完的时候，执行 flush。
- 遍历过程中，每当写入 DiskBlockObjectWriter 的元素个数（objectsWritten）达到批量序列化尺寸（serializerBatchSize）时，也会执行 flush，然后重新创建 DiskBlockObjectWriter。

代码清单6-18 ExternalSorter.spillToMergeableFile方法的实现

```
private def spillToMergeableFile(collection: SizeTrackingPairCollection[(Int, K),
    C]): Unit = {
    val (blockId, file) = diskBlockManager.createTempShuffleBlock()
    curWriteMetrics = new ShuffleWriteMetrics()
    var writer = blockManager.getDiskWriter(blockId, file, ser, fileBufferSize,
        curWriteMetrics)
    var objectsWritten = 0   // Objects written since the last flush
    val batchSizes = new ArrayBuffer[Long]
    val elementsPerPartition = new Array[Long](numPartitions)

    def flush() = {
        val w = writer
        writer = null
        w.commitAndClose()
        _diskBytesSpilled += curWriteMetrics.shuffleBytesWritten
        batchSizes.append(curWriteMetrics.shuffleBytesWritten)
        objectsWritten = 0
    }

    var success = false
    try {
        val it = collection.destructiveSortedIterator(partitionKeyComparator)
        while (it.hasNext) {
            val elem = it.next()
            val partitionId = elem._1._1
            val key = elem._1._2
            val value = elem._2
            writer.write(key)
            writer.write(value)
            elementsPerPartition(partitionId) += 1
            objectsWritten += 1

            if (objectsWritten == serializerBatchSize) {
                flush()
                curWriteMetrics = new ShuffleWriteMetrics()
                writer = blockManager.getDiskWriter(blockId, file, ser,
                    fileBufferSize, curWriteMetrics)
            }
        }

        if (objectsWritten > 0) {
            flush()
```

```
        } else if (writer != null) {
            val w = writer
            writer = null
            w.revertPartialWritesAndClose()
        success = true
    } finally {
        if (!success) {
            if (writer != null) {
                writer.revertPartialWritesAndClose()
            }
            if (file.exists()) {
                file.delete()
            }
        }

    spills.append(SpilledFile(file, blockId, batchSizes.toArray, elementsPerPartition))
```

6.4 map 端计算结果持久化

writePartitionedFile（见代码清单 6-19）用于持久化计算结果。此方法有两个分支：

- 溢出到分区文件后合并：将内存中缓存的多个 partition 的计算结果分别写入多个临时 Block 文件，然后将这些 Block 文件的内容全部写入正式的 Block 输出文件中。
- 内存中排序合并：将缓存的中间计算结果按照 partition 分组后写入 Block 输出文件。此种方式还需要更新此任务与内存、磁盘有关的测量数据。

无论哪种排序方式，每个 partition 都会最终写入一个正式的 Block 文件，所以每个 map 任务实际上最后只会生成一个磁盘文件，最终解决了 Spark 早期版本中一个 map 任务输出的 bucket 文件过多和磁盘 I/O 成为性能瓶颈的问题。此外，无论哪种排序方式，每输出完一个 partition 的中间结果时，都会记录当前 partition 的长度，此长度将记录在索引文件中，以便下游任务的读取。

writePartitionedFile 中有关 DiskBlockObjectWriter 的实现，请参阅 4.12 节。

代码清单6-19 ExternalSorter.writePartitionedFile的实现

```
def writePartitionedFile(blockId: BlockId, context: TaskContext,
        outputFile: File): Array[Long] = {
    val lengths = new Array[Long](numPartitions)

    if (bypassMergeSort && partitionWriters != null) {
        spillToPartitionFiles(if (aggregator.isDefined) map else buffer)
        partitionWriters.foreach(_.commitAndClose())
        var out: FileOutputStream = null
        var in: FileInputStream = null
        try {
            out = new FileOutputStream(outputFile, true)
            for (i <- 0 until numPartitions) {
            in = new FileInputStream(partitionWriters(i).fileSegment().file)
```

```
                val size = org.apache.spark.util.Utils.copyStream(in, out, false,
                    transferToEnabled)
                in.close()
                in = null
                lengths(i) = size
            }
        } finally {
            if (out != null) {
                out.close()
            }
            if (in != null) {
                in.close()
            }
        }
    } else {
        for ((id, elements) <- this.partitionedIterator) {
            if (elements.hasNext) {
                val writer = blockManager.getDiskWriter(
                    blockId, outputFile, ser, fileBufferSize, context.taskMetrics.
                        shuffle-WriteMetrics.get)
                for (elem <- elements) {
                    writer.write(elem)
                }
                writer.commitAndClose()
                val segment = writer.fileSegment()
                lengths(id) = segment.length
            }
        }
    }

    context.taskMetrics.memoryBytesSpilled += memoryBytesSpilled
        context.taskMetrics.diskBytesSpilled += diskBytesSpilled
        context.taskMetrics.shuffleWriteMetrics.filter(_ => bypassMergeSort).foreach { m =>
            if (curWriteMetrics != null) {
                m.shuffleBytesWritten += curWriteMetrics.shuffleBytesWritten
                m.shuffleWriteTime += curWriteMetrics.shuffleWriteTime
            }
        }

        lengths
    }
```

6.4.1 溢出分区文件

spillToPartitionFiles（见代码清单 6-20）用于将内存中的集合数据按照每个 partition 创建一个临时 Block 文件，为每个临时 Block 文件生成一个 DiskBlockObjectWriter，并且用 DiskBlockObjectWriter 将计算结果分别写入这些临时 Block 文件中。createTempShuffleBlock 方法创建临时的 Block，具体内容请参阅 4.4.3 节。getDiskWriter 方法创建 DiskBlockObjectWriter，具体内容请参阅 4.8.7 节。

代码清单6-20 ExternalSorter的溢出分区文件实现

```
private def spillToPartitionFiles(collection: SizeTrackingPairCollection[(Int, K),
    C]): Unit = {
```

```
        spillToPartitionFiles(collection.iterator)
    }

    private def spillToPartitionFiles(iterator: Iterator[((Int, K), C)]): Unit = {
        assert(bypassMergeSort)

        if (partitionWriters == null) {
            curWriteMetrics = new ShuffleWriteMetrics()
            partitionWriters = Array.fill(numPartitions) {
                val (blockId, file) = diskBlockManager.createTempShuffleBlock()
                blockManager.getDiskWriter(blockId, file, ser, fileBufferSize,
                    curWrite-Metrics).open()
            }
        }

        while (iterator.hasNext) {
            val elem = iterator.next()
            val partitionId = elem._1._1
            val key = elem._1._2
            val value = elem._2
            partitionWriters(partitionId).write((key, value))
        }
    }
```

代码清单 6-19 的持久化分支（即满足 bypassMergeSort && partitionWriters != null 条件的代码分支）说明 spillToPartitionFiles 方法为每个 partition 生成的临时文件最后会逐个读取并统一写入正式的 Block 文件，如图 6-4 所示。spillToPartitionFiles 方法在 bypass-MergeSort 为 true，SizeTrackingAppend-OnlyMap 或者 SizeTrackingPairBuffer 的大小超过 myMemoryThreshold 时被调用，以防止内存撑爆问题。此外，由于每个 partition 生成的临时文件最后会逐个读取并统一写入正式的 Block 文件，所以每个 map 任务实际上最后只会生成一个磁盘文件（相当于多个 bucket 被合并到一个文件中），最终解决了产生 bucket 文件过多和磁盘 I/O 成为性能瓶颈的问题。

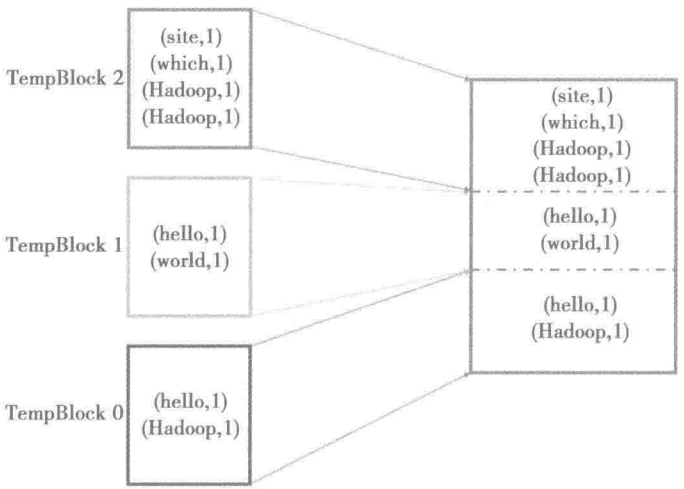


图 6-4 溢出到分区文件后合并

6.4.2 排序与分区分组

partitionedIterator（见代码清单 6-21）通过对集合按照指定的比较器进行排序，并且按照 partition id 分组，生成迭代器。

代码清单6-21 ExternalSorter.partitionedIterator的实现

```
def partitionedIterator: Iterator[(Int, Iterator[Product2[K, C]])] = {
    val usingMap = aggregator.isDefined
    val collection: SizeTrackingPairCollection[(Int, K), C] = if (usingMap) map
         else buffer
    if (spills.isEmpty && partitionWriters == null) {
        if (!ordering.isDefined) {
            groupByPartition(collection.destructiveSortedIterator(partitionComparator))
        } else {
            groupByPartition(collection.destructiveSortedIterator(partitionKeyComparator))
        }
    } else if (bypassMergeSort) {
        val collIter = groupByPartition(collection.destructiveSortedIterator(par
             titionComparator))
        collIter.map { case (partitionId, values) =>
            (partitionId, values ++ readPartitionFile(partitionWriters(partitionId)))
        }
    } else {
        merge(spills, collection.destructiveSortedIterator(partitionKeyComparator))
    }
}
```

1. 比较器

代码清单 6-22 列出了目前的三种比较器：

- keyComparator：按照指定的 key 进行排序；
- partitionComparator：按照 partition id 进行比较；
- partitionKeyComparator：先按照 partition id 进行比较，再按照指定的 key 进行第二级排序。当没有指定排序字段并且没有指定聚合函数时会退化为 partitionComparator。

代码清单6-22 ExternalSorter中提供的几种比较器

```
private val keyComparator: Comparator[K] = ordering.getOrElse(new Comparator[K] {
    override def compare(a: K, b: K): Int = {
        val h1 = if (a == null) 0 else a.hashCode()
        val h2 = if (b == null) 0 else b.hashCode()
        if (h1 < h2) -1 else if (h1 == h2) 0 else 1
    }
})

private val partitionComparator: Comparator[(Int, K)] = new Comparator[(Int, K)] {
    override def compare(a: (Int, K), b: (Int, K)): Int = {
        a._1 - b._1
    }
}
```

```
private val partitionKeyComparator: Comparator[(Int, K)] = {
    if (ordering.isDefined || aggregator.isDefined) {
        new Comparator[(Int, K)] {
            override def compare(a: (Int, K), b: (Int, K)): Int = {
                val partitionDiff = a._1 - b._1
                if (partitionDiff != 0) {
                    partitionDiff
                } else {
                    keyComparator.compare(a._2, b._2)
                }
            }
        }
    } else {
        partitionComparator
    }
}
```

由于 partitionedIterator 方法实际是通过调用 destructiveSortedIterator 和 groupByPartition 来实现，下面详细分析这两个方法。

2. 排序

destructiveSortedIterator 方法（见代码清单 6-23）的处理步骤如下：

1）将 data 数组向左整理排列。

2）利用 Sorter、KVArraySortDataFormat 以及指定的比较器进行排序。这其中用到了 TimSort，也就是优化版的归并排序。

3）生成新的迭代器。

代码清单6-23 AppendOnlyMap.destructiveSortedIterator的排序实现

```
def destructiveSortedIterator(keyComparator: Comparator[K]): Iterator[(K, V)] = {
    destroyed = true
    var keyIndex, newIndex = 0
    while (keyIndex < capacity) {
        if (data(2 * keyIndex) != null) {
            data(2 * newIndex) = data(2 * keyIndex)
            data(2 * newIndex + 1) = data(2 * keyIndex + 1)
            newIndex += 1
        }
        keyIndex += 1
    }
    assert(curSize == newIndex + (if (haveNullValue) 1 else 0))

    new Sorter(new KVArraySortDataFormat[K, AnyRef]).sort(data, 0, newIndex,
        keyComparator)

    new Iterator[(K, V)] {
        var i = 0
        var nullValueReady = haveNullValue
        def hasNext: Boolean = (i < newIndex || nullValueReady)
        def next(): (K, V) = {
```

```
                if (nullValueReady) {
                    nullValueReady = false
                    (null.asInstanceOf[K], nullValue)
                } else {
                    val item = (data(2 * i).asInstanceOf[K], data(2 * i + 1).asInstanceOf[V])
                    i += 1
                    item
                }
            }
        }
    }
```

3. 分区分组

groupByPartition（见代码清单 6-24）主要用于对 destructiveSortedIterator 生成的迭代器按照 partition id 分组。

代码清单6-24 ExternalSorter.groupByPartition的分区分组代码

```
private def groupByPartition(data: Iterator[((Int, K), C)])
    : Iterator[(Int, Iterator[Product2[K, C]])] =
{
    val buffered = data.buffered
    (0 until numPartitions).iterator.map(p => (p, new IteratorForPartition(p,
        buffered)))
}
```

IteratorForPartition 如何区分 partition 呢？见代码清单 6-25。可见其 hasNext 会判断数据的 partitionId。

代码清单6-25 ExternalSorter.IteratorForPartition的实现

```
private[this] class IteratorForPartition(partitionId: Int, data: Buffered-
    Iterator[((Int, K), C)])
    extends Iterator[Product2[K, C]]
{
    override def hasNext: Boolean = data.hasNext && data.head._1._1 == partitionId

    override def next(): Product2[K, C] = {
        if (!hasNext) {
            throw new NoSuchElementException
        }
        val elem = data.next()
        (elem._1._2, elem._2)
    }
}
```

6.4.3 分区索引文件

根据代码清单 6-19 我们知道，无论采用哪种缓存处理，在持久化的时候都会被写入同一文件，那么 reduce 任务如何从此文件中按照分区读取数据呢？还记得在代码清单 6-5 中调用 IndexShuffleBlockManager 的 writeIndexFile 方法生成的分区索引文件吗？此文件使用偏移

量来区分各个分区的计算结果，偏移量来自于合并排序过程中记录的各个 partition 的长度。writeIndexFile 的详细介绍请阅读 4.13 节。

这里用图 6-5 展示内存中排序、分组后生成分区索引文件的全过程。

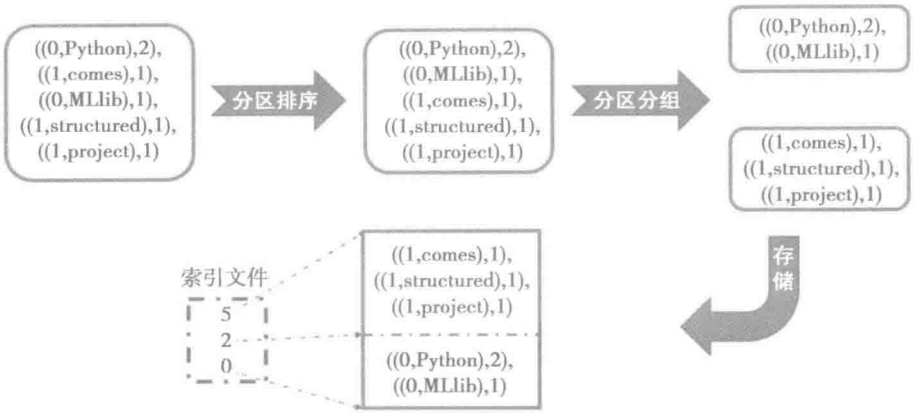


图 6-5 排序、分组、生成分区索引文件的全过程

6.5 reduce 端读取中间计算结果

先简单说下，当 map 任务相关 Stage 的任务都执行完毕后，会唤起下游 Stage 的提交及任务的执行。上游任务的执行结果必然是下游任务的输入，我们就下游任务如何读取上游任务计算结果来展开。

根据 5.6.3 节中“ResultTask 任务的结果处理”部分的内容，我们知道 ResultTask 的计算也是由 RDD 的 iterator 方法驱动，这在介绍 ShuffleMapTask 的时候已经介绍过。其计算过程最终会落实到 ShuffledRDD 的 compute 方法。ShuffledRDD 的 compute 方法（见代码清单 6-26）首先调用 SortShuffleManager 的 getReader 方法创建 HashShuffleReader，然后执行 HashShuffleReader 的 read 方法读取依赖任务的中间计算结果。

代码清单6-26 ShuffledRDD.compute代码

```
override def compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {
    val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]
    SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.
        index + 1, context)
        .read()
        .asInstanceOf[Iterator[(K, C)]]
}
```

SortShuffleManager 的 getReader 方法的实现如下。

```
override def getReader[K, C](
        handle: ShuffleHandle,
```

```
        startPartition: Int,
        endPartition: Int,
        context: TaskContext): ShuffleReader[K, C] = {
    new HashShuffleReader(
        handle.asInstanceOf[BaseShuffleHandle[K, _, C]], startPartition, endPartition,
            context)
}
```

HashShuffleReader 用来读取上游任务计算结果，它的 read 方法（见代码清单 6-27）的处理步骤如下：

1）从远端节点或者本地读取中间计算结果。

2）对 InterruptibleIterator 执行聚合。

3）对 InterruptibleIterator 排序，由于使用 ExternalSorter 的 insertAll，不再赘述。

代码清单6-27 HashShuffleReader的read方法

```
override def read(): Iterator[Product2[K, C]] = {
    val ser = Serializer.getSerializer(dep.serializer)
    val iter = BlockStoreShuffleFetcher.fetch(handle.shuffleId, startPartition,
        context, ser)

    val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {
        if (dep.mapSideCombine) {
            new InterruptibleIterator(context, dep.aggregator.get.combineCombiners-
                ByKey(iter, context))
        } else {
            new InterruptibleIterator(context, dep.aggregator.get.combineValues-
                ByKey(iter, context))
        }
    } else {
        require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")
        iter.asInstanceOf[Iterator[Product2[K, C]]].map(pair => (pair._1, pair._2))
    }

    dep.keyOrdering match {
        case Some(keyOrd: Ordering[K]) =>
            val sorter = new ExternalSorter[K, C, C](ordering = Some(keyOrd),
                serializer = Some(ser))
            sorter.insertAll(aggregatedIter)
            context.taskMetrics.memoryBytesSpilled += sorter.memoryBytesSpilled
            context.taskMetrics.diskBytesSpilled += sorter.diskBytesSpilled
            sorter.iterator
        case None =>
            aggregatedIter
    }
}
```

从远端节点或者本地读取中间计算结果通过调用 BlockStoreShuffleFetcher 的 fetch 方法（见代码清单 6-28）实现，它的处理步骤如下：

1）获取 map 任务执行的状态信息。

2）按照中间结果所在节点划分各个 Block。

3）创建 ShuffleBlockFetcherIterator（即从远端节点或者本地读取中间计算结果）。

4）将 ShuffleBlockFetcherIterator 封装为 InterruptibleIterator。

代码清单6-28 BlockStoreShuffleFetcher的fetch方法

```
def fetch[T](shuffleId: Int, reduceId: Int, context: TaskContext,
    serializer: Serializer) : Iterator[T] =
    {
        logDebug("Fetching outputs for shuffle %d, reduce %d".format(shuffleId, reduceId))
        val blockManager = SparkEnv.get.blockManager

        val startTime = System.currentTimeMillis
        val statuses = SparkEnv.get.mapOutputTracker.getServerStatuses(shuffleId, reduceId)
        logDebug("Fetching map output location for shuffle %d, reduce %d took %d ms".format(
            shuffleId, reduceId, System.currentTimeMillis - startTime))

        val splitsByAddress = new HashMap[BlockManagerId, ArrayBuffer[(Int, Long)]]
        for (((address, size), index) <- statuses.zipWithIndex) {
            splitsByAddress.getOrElseUpdate(address, ArrayBuffer()) += ((index, size))
        }

        val blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])] =
             splitsByAddress.toSeq.map {
            case (address, splits) =>
                (address, splits.map(s => (ShuffleBlockId(shuffleId, s._1, reduceId), s._2)))
    }

    def unpackBlock(blockPair: (BlockId, Try[Iterator[Any]])) : Iterator[T] = {
        val blockId = blockPair._1
        val blockOption = blockPair._2
        blockOption match {
            case Success(block) => {
                block.asInstanceOf[Iterator[T]]
            }
            case Failure(e) => {
                //异常省略
            }
        }
    }

    val blockFetcherItr = new ShuffleBlockFetcherIterator(context,
        SparkEnv.get.blockManager.shuffleClient, blockManager, blocksByAddress,
        serializer,
        SparkEnv.get.conf.getLong("spark.reducer.maxMbInFlight", 48) * 1024 * 1024)
    val itr = blockFetcherItr.flatMap(unpackBlock)

    val completionIter = CompletionIterator[T, Iterator[T]](itr, {
        context.taskMetrics.updateShuffleReadMetrics()
    })

    new InterruptibleIterator[T](context, completionIter)
}
```

6.5.1 获取 map 任务状态

Spark 通过调用 MapOutputTracker 的 getServerStatuses（见代码清单 6-29）来获取 map 任务执行的状态信息，其处理步骤如下：

1）从当前 BlockManager 的 MapOutputTracker 中获取 MapStatus，若没有就进入第 2) 步，否则直接到第 4) 步。

2）如果获取列表（fetching）中已经存在要取的 shuffleId，那么就等待其他线程获取。如果获取列表中不存在要取的 shuffleId，那么就将 shuffleId 放入获取列表。

3）调用 askTracker 方法（见代码清单 6-30）向 MapOutputTrackerMasterActor 发送 GetMapOutputStatuses 消息获取 map 任务的状态信息。MapOutputTrackerMasterActor 接收到 GetMapOutputStatuses 消息后，将请求的 map 任务状态信息序列化后发送给请求方，见代码清单 6-31。请求方接收到 map 任务状态信息后进行反序列化操作，然后放入本地的 mapStatuses 中。

4）调用 MapOutputTracker 的 convertMapStatuses 方法（见代码清单 6-32）将获得的 MapStatus 转换为 map 任务所在的地址（即 BlockManagerId）和 map 任务输出中分配给当前 reduce 任务的 Block 大小。

代码清单6-29 MapOutputTracker.getServerStatuses的代码

```
def getServerStatuses(shuffleId: Int, reduceId: Int): Array[(BlockManagerId, Long)]
    = {
    val statuses = mapStatuses.get(shuffleId).orNull
    if (statuses == null) {
        logInfo("Don't have map outputs for shuffle " + shuffleId + ", fetching them")
        var fetchedStatuses: Array[MapStatus] = null
        fetching.synchronized {
            // Someone else is fetching it; wait for them to be done
            while (fetching.contains(shuffleId)) {
                try {
                    fetching.wait()
                } catch {
                    case e: InterruptedException =>
                }
            }

            fetchedStatuses = mapStatuses.get(shuffleId).orNull
            if (fetchedStatuses == null) {
                fetching += shuffleId
            }
        }
        if (fetchedStatuses == null) {
            try {
                val fetchedBytes =
                    askTracker(GetMapOutputStatuses(shuffleId)).asInstance-
                        Of[Array[Byte]]
                fetchedStatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes)
```

```
                logInfo("Got the output locations")
                mapStatuses.put(shuffleId, fetchedStatuses)
            } finally {
                fetching.synchronized {
                    fetching -= shuffleId
                    fetching.notifyAll()
                }
            }
        }
        if (fetchedStatuses != null) {
            fetchedStatuses.synchronized {
                return MapOutputTracker.convertMapStatuses(shuffleId, reduceId,
                    fetchedStatuses)
            }
        } else {
            logError("Missing all output locations for shuffle " + shuffleId)
            throw new MetadataFetchFailedException(
                shuffleId, reduceId, "Missing all output locations for shuffle " +
                    shuffleId)
        }
    } else {
        statuses.synchronized {
            return MapOutputTracker.convertMapStatuses(shuffleId, reduceId, statuses)
        }
    }
}
```

代码清单6-30 MapOutputTracker.askTracker的实现

```
protected def askTracker(message: Any): Any = {
    try {
        val future = trackerActor.ask(message)(timeout)
        Await.result(future, timeout)
    } catch {
        case e: Exception =>
            logError("Error communicating with MapOutputTracker", e)
            throw new SparkException("Error communicating with MapOutputTracker", e)
    }
}
```

代码清单6-31 MapOutputTrackerMasterActor处理GetMapOutputStatuses消息

```
case GetMapOutputStatuses(shuffleId: Int) =>
        val hostPort = sender.path.address.hostPort
        logInfo("Asked to send map output locations for shuffle " + shuffleId + "
            to " + hostPort)
        val mapOutputStatuses = tracker.getSerializedMapOutputStatuses(shuffleId)
        val serializedSize = mapOutputStatuses.size
        if (serializedSize > maxAkkaFrameSize) {
        val msg = s"Map output statuses were $serializedSize bytes which " +
            s"exceeds spark.akka.frameSize ($maxAkkaFrameSize bytes)."

        val exception = new SparkException(msg)
```

```
            logError(msg, exception)
            throw exception
        }
        sender ! mapOutputStatuses
```

代码清单6-32 map任务地址转换

```
private def convertMapStatuses(
    shuffleId: Int,
    reduceId: Int,
    statuses: Array[MapStatus]): Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
assert (statuses != null)
val splitsByAddress = new HashMap[BlockManagerId, ArrayBuffer[(BlockId, Long)]]
for ((status, mapId) <- statuses.zipWithIndex) {
    if (status == null) {
        val errorMessage = s"Missing an output location for shuffle $shuffleId"
        logError(errorMessage)
        throw new MetadataFetchFailedException(shuffleId, reduceId, errorMessage)
    } else {
        splitsByAddress.getOrElseUpdate(status.location, ArrayBuffer()) +=
            ((ShuffleBlockId(shuffleId, mapId, reduceId), status.getSizeForBlock(reduceId)))
        }
    }

    splitsByAddress.toSeq
}
```

6.5.2 划分本地与远程 Block

无论从本地还是从 MapOutputTrackerMasterActor 获取的状态信息，都需要按照地址划分并且转换为 BlockId。ShuffleBlockFetcherIterator 是读取中间结果的关键。构造 ShuffleBlockFetcherIterator 的时候会调用到 initialize 方法（见代码清单 6-33），它的初始化过程如下：

1）使用 splitLocalRemoteBlocks 方法划分本地读取和远程读取的 Block 的请求。

2）将 FetchRequest 随机排序后存入 fetchRequests：newQueue[FetchRequest]。

3）遍历 fetchRequests 中的所有 FetchRequest，远程请求 Block 中间结果。

4）调用 fetchLocalBlocks 获取本地 Block。

代码清单6-33 ShuffleBlockFetcherIterator的初始化

```
private[this] def initialize(): Unit = {
    context.addTaskCompletionListener(_ => cleanup())

    val remoteRequests = splitLocalRemoteBlocks()
    fetchRequests ++= Utils.randomize(remoteRequests)

    while (fetchRequests.nonEmpty &&
        (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <= maxBytes-
            InFlight)) {
        sendRequest(fetchRequests.dequeue())
```

```
    }

    val numFetches = remoteRequests.size - fetchRequests.size
    logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs(startTime))

    fetchLocalBlocks()
    logDebug("Got local blocks in " + Utils.getUsedTimeMs(startTime))
}
```

splitLocalRemoteBlocks 方法（见代码清单 6-34）用于划分哪些 Block 从本地获取，哪些需要远程拉取，是获取中间计算结果的关键。为便于描述，先解释以下定义：

- targetRequestSize：每个远程请求的最大尺寸。
- totalBlocks：统计 Block 总数。
- localBlocks：ArrayBuffer[BlockId]，缓存可以在本地获取的 Block 的 blockId。
- remoteBlocks：HashSet[BlockId]，缓存需要远程获取的 Block 的 blockId。
- curBlocks：ArrayBuffer[(BlockId, Long)]，远程获取的累加缓存，用于保证每个远程请求的尺寸不超过 targetRequestSize。为什么要累加缓存？如果向一个机器节点频繁地请求字节数很小的 Block，那么势必造成网络拥塞并增加节点负担。将多个小数据量的请求合并为一个大的请求将避免这些问题，提高系统性能。
- curRequestSize：当前累加到 curBlocks 中的所有 Block 的大小之和，用于保证每个远程请求的尺寸不超过 targetRequestSize。
- remoteRequests：new ArrayBuffer[FetchRequest]，缓存需要远程请求的 FetchRequest 对象。
- numBlocksToFetch：一共要获取的 Block 数量。
- maxBytesInFlight：单次航班请求的最大字节数。什么叫航班？其实就是一批请求，这批请求的字节总数不能超过 maxBytesInFlight，而且每个请求的字节数不能超过 maxBytesInFlight 的五分之一。可以通过参数 spark.reducer.maxMbInFlight 来控制大小。为什么每个请求的字节数不能超过 maxBytesInFlight 的五分之一？这样做是为了提高请求的并发度，允许 5 个请求分别从 5 个节点获取数据，最大限度利用各节点的资源。

明白了这些定义，我们一起来看看 splitLocalRemoteBlocks 的处理逻辑吧。

1）遍历已经在代码清单 6-28 中按照 BlockManagerId 分组的 blockInfo，如果 blockInfo 所在的 Executor 与当前 Executor 相同，则将它的 BlockId 存入 localBlocks；否则，将 blockInfo 的 BlockId 和 size 累加到 curBlocks，将 blockId 存入 remoteBlocks，curRequestSize 增加 size 的大小，每当 curRequestSize >= targetRequestSize，则新建 FetchRequest 放入 remoteRequests，并且为生成下一个 FetchRequest 做一些准备（如新建 curBlocks，curRequestSize 置为 0）。

2）遍历结束，curBlocks 中如果仍然有缓存的（BlockId, Long），新建 FetchRequest 放入 remoteRequests。此次请求不受 maxBytesInFlight 和 targetRequestSize 的影响。

代码清单6-34 ShuffleBlockFetcherIterator.splitLocalRemoteBlocks方法

```
private[this] def splitLocalRemoteBlocks(): ArrayBuffer[FetchRequest] = {
    val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)
    val remoteRequests = new ArrayBuffer[FetchRequest]
    var totalBlocks = 0
    for ((address, blockInfos) <- blocksByAddress) {
        totalBlocks += blockInfos.size
        if (address.executorId == blockManager.blockManagerId.executorId) {
            localBlocks ++= blockInfos.filter(_._2 != 0).map(_._1)
            numBlocksToFetch += localBlocks.size
        } else {
            val iterator = blockInfos.iterator
            var curRequestSize = 0L
            var curBlocks = new ArrayBuffer[(BlockId, Long)]
            while (iterator.hasNext) {
                val (blockId, size) = iterator.next()
                if (size > 0) {
                    curBlocks += ((blockId, size))
                    remoteBlocks += blockId
                    numBlocksToFetch += 1
                    curRequestSize += size
                } else if (size < 0) {
                    throw new BlockException(blockId, "Negative block size " + size)
                }
                if (curRequestSize >= targetRequestSize) {
                    remoteRequests += new FetchRequest(address, curBlocks)
                    curBlocks = new ArrayBuffer[(BlockId, Long)]
                    logDebug(s"Creating fetch request of $curRequestSize at $address")
                    curRequestSize = 0
                }
            }
            if (curBlocks.nonEmpty) {
                remoteRequests += new FetchRequest(address, curBlocks)
            }
        }
    }
    remoteRequests
}
```

6.5.3 获取远程 Block

sendRequest 方法（见代码清单 6-35）用于远程请求中间结果。sendRequest 利用 FetchRequest 里封装的 blockId、size、address 等信息，调用 shuffleClient 的 fetchBlocks 方法获取其他节点上的中间计算结果。shuffleClient.fetchBlocks 方法可以参阅 4.2.5 节。

代码清单6-35 ShuffleBlockFetcherIterator.sendRequest方法

```
private[this] def sendRequest(req: FetchRequest) {
    logDebug("Sending request for %d blocks (%s) from %s".format(
        req.blocks.size, Utils.bytesToString(req.size), req.address.hostPort))
    bytesInFlight += req.size
```

```
        val sizeMap = req.blocks.map { case (blockId, size) => (blockId.toString, size)
            }.toMap
        val blockIds = req.blocks.map(_._1.toString)

        val address = req.address
        shuffleClient.fetchBlocks(address.host, address.port, address.executorId,
            blockIds.toArray,
            new BlockFetchingListener {
                override def onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit = {
                    if (!isZombie) {
                        buf.retain()
                        results.put(new SuccessFetchResult(BlockId(blockId), sizeMap(blockId), buf))
                        shuffleMetrics.remoteBytesRead += buf.size
                        shuffleMetrics.remoteBlocksFetched += 1
                    }
                    logTrace("Got remote block " + blockId + " after " + Utils.getUsed-
                        TimeMs(startTime))
                }

                override def onBlockFetchFailure(blockId: String, e: Throwable): Unit = {
                    logError(s"Failed to get block(s) from ${req.address.host}:${req.
                        address.port}", e)
                    results.put(new FailureFetchResult(BlockId(blockId), e))
                }
            }
        )
    }
```

6.5.4 获取本地 Block

fetchLocalBlocks（见代码清单 6-36）用于对本地中间计算结果的获取。fetchLocalBlocks 方法很简单，利用熟悉的 BlockManager 的 getBlockData 方法获取本地 Block，最后将取到的中间结果存入 results = new LinkedBlockingQueue[FetchResult] 中。

代码清单6-36 ShuffleBlockFetcherIterator. fetchLocalBlocks方法

```
    private[this] def fetchLocalBlocks() {
        val iter = localBlocks.iterator
        while (iter.hasNext) {
            val blockId = iter.next()
            try {
                val buf = blockManager.getBlockData(blockId)
                shuffleMetrics.localBlocksFetched += 1
                buf.retain()
                results.put(new SuccessFetchResult(blockId, 0, buf))
            } catch {
                case e: Exception =>
                    logError(s"Error occurred while fetching local blocks", e)
                    results.put(new FailureFetchResult(blockId, e))
                    return
            }
        }
    }
```

6.6 reduce 端计算

6.6.1 如何同时处理多个 map 任务的中间结果

reduce 任务的上游 map 任务可能有多个，根据之前的分析，知道这些中间结果的 Block 及数据缓存在 ShuffleBlockFetcherIterator 的 results：new LinkedBlockingQueue[FetchResult] 中。ShuffleBlockFetcherIterator 作为迭代器，它的实现见代码清单 6-37。从其实现可知，每次迭代 ShuffleBlockFetcherIterator，会先从 results：new LinkedBlockingQueue[FetchResult] 中取出一个 FetchResult，并构造此 FetchResult 的迭代器 iteratorTry，具体迭代的数据就是从 iteratorTry 中获取。每当 iteratorTry 迭代结束，才会再次迭代 ShuffleBlockFetcherIterator。

代码清单6-37 ShuffleBlockFetcherIterator.scala

```
override def hasNext: Boolean = numBlocksProcessed < numBlocksToFetch

override def next(): (BlockId, Try[Iterator[Any]]) = {
    numBlocksProcessed += 1
    val startFetchWait = System.currentTimeMillis()
    currentResult = results.take()
    val result = currentResult
    val stopFetchWait = System.currentTimeMillis()
    shuffleMetrics.fetchWaitTime += (stopFetchWait - startFetchWait)

    result match {
        case SuccessFetchResult(_, size, _) => bytesInFlight -= size
        case _ =>
    }
    while (fetchRequests.nonEmpty &&
        (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <=
            maxBytesInFlight)) {
        sendRequest(fetchRequests.dequeue())
    }

    val iteratorTry: Try[Iterator[Any]] = result match {
        case FailureFetchResult(_, e) =>
            Failure(e)
        case SuccessFetchResult(blockId, _, buf) =>
            Try(buf.createInputStream()).map { is0 =>
                val is = blockManager.wrapForCompression(blockId, is0)
                val iter = serializer.newInstance().deserializeStream(is).asIterator
                CompletionIterator[Any, Iterator[Any]](iter, {
                    currentResult = null
                    buf.release()
                })
            }
        }

        (result.blockId, iteratorTry)
    }
}
```


注意 由于之前远程获取 Block 时，一小部分请求可能就达到了 maxBytesInFlight 的限制，所以很有可能会剩余很多请求没有发送。所以每此迭代 ShuffleBlockFetcher-Iterator 的时候还有个附加动作用于发送剩余请求。

6.6.2 reduce 端在缓存中对中间计算结果执行聚合和排序

reduce 端获取 map 端任务计算中间结果后，将 ShuffleBlockFetcherIterator 封装为 InterruptibleIterator 并聚合。聚合操作主要依赖 aggregator 的 combineCombinersByKey 方法，见代码清单 6-38。如果 isSpillEnabled 为 false，会再次使用 AppendOnlyMap 的 changeValue 方法，这些内容已在 6.2 节介绍 AppendOnlyMap 的缓存聚合算法时做过详细介绍。isSpillEnabled 默认是 true，此时会使用 ExternalAppendOnlyMap 完成聚合。

代码清单6-38 Aggregator.combineCombinersByKey的实现

```
def combineCombinersByKey(iter: Iterator[_ <: Product2[K, C]], context: TaskContext)
      : Iterator[(K, C)] =
    {
        if (!isSpillEnabled) {
            val combiners = new AppendOnlyMap[K,C]
            var kc: Product2[K, C] = null
            val update = (hadValue: Boolean, oldValue: C) => {
                if (hadValue) mergeCombiners(oldValue, kc._2) else kc._2
            }
            while (iter.hasNext) {
                kc = iter.next()
                combiners.changeValue(kc._1, update)
            }
            combiners.iterator
        } else {
            val combiners = new ExternalAppendOnlyMap[K, C, C](identity,
                 mergeCombiners, mergeCombiners)
            while (iter.hasNext) {
                val pair = iter.next()
                combiners.insert(pair._1, pair._2)
            }
            Option(context).foreach { c =>
                c.taskMetrics.memoryBytesSpilled += combiners.memoryBytesSpilled
                c.taskMetrics.diskBytesSpilled += combiners.diskBytesSpilled
        }
        combiners.iterator
    }
}
```

ExternalAppendOnlyMap 的 insert 方法的实际工作是由 insertAll 完成的，见代码清单 6-39。从代码实现可以看到其实质也是使用 SizeTrackingAppendOnlyMap，已在介绍 AppendOnlyMap 的缓存聚合算法时做过详细介绍。

代码清单6-39 ExternalAppendOnlyMap的insert方法

```
def insert(key: K, value: V): Unit = {
```

```
        insertAll(Iterator((key, value)))
    }

    def insertAll(entries: Iterator[Product2[K, V]]): Unit = {
        var curEntry: Product2[K, V] = null
        val update: (Boolean, C) => C = (hadVal, oldVal) => {
            if (hadVal) mergeValue(oldVal, curEntry._2) else createCombiner(curEntry._2)
        }

        while (entries.hasNext) {
            curEntry = entries.next()
            if (maybeSpill(currentMap, currentMap.estimateSize())) {
                currentMap = new SizeTrackingAppendOnlyMap[K, C]
            }
            currentMap.changeValue(curEntry._1, update)
            addElementsRead()
        }
    }
```

经过以上处理，数据结果为类似（##, 8），（N, 1），（set, 2），（use, 3），（Hadoop-supported, 1）的样子。

6.7 map 端与 reduce 端组合分析

这一节主要对计算引擎部分的内容进行串联，用图来展示最常见的几种组合，以便大家对计算引擎有个宏观的认识。其中的具体执行过程，已在本章之前的内容中介绍过，此处不再赘述。

6.7.1 在 map 端溢出分区文件，在 reduce 端合并组合

bypassMergeSort 标记是否传递到 reduce 端再做合并和排序，此种情况不使用缓存，而是先将数据按照 partition 写入不同文件，最后按 partition 顺序合并写入同一文件。当没有指定聚合、排序函数，且 partition 数量较小时，一般采用这种方式。此种方式将多个 bucket 合并到同一个文件，通过减少 map 输出的文件数量，节省了磁盘 I/O，最终提升了性能，见图 6-6。

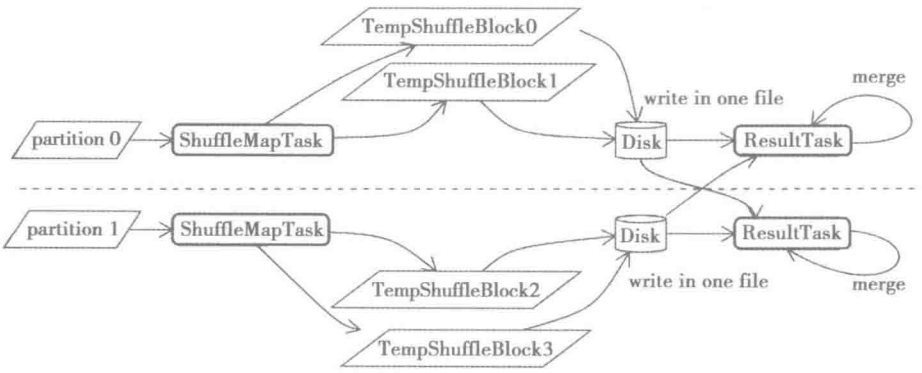


图 6-6 map 端溢出分区文件与 reduce 端合并组合

6.7.2 在 map 端简单缓存、排序分组，在 reduce 端合并组合

此种情况在缓存中利用指定的排序函数对数据按照 partition 或者 key 进行排序，最后按 partition 顺序合并写入同一文件。当没有指定聚合函数，且 partition 数量大时，一般采用这种方式，见图 6-7。此种方式将多个 bucket 合并到同一个文件，通过减少 map 输出的文件数量，节省了磁盘 I/O，提升了性能；对 SizeTrackingPairBuffer 的缓存进行溢出判断，当超出 myMemoryThreshold 的大小时，将数据写入磁盘，防止内存溢出。

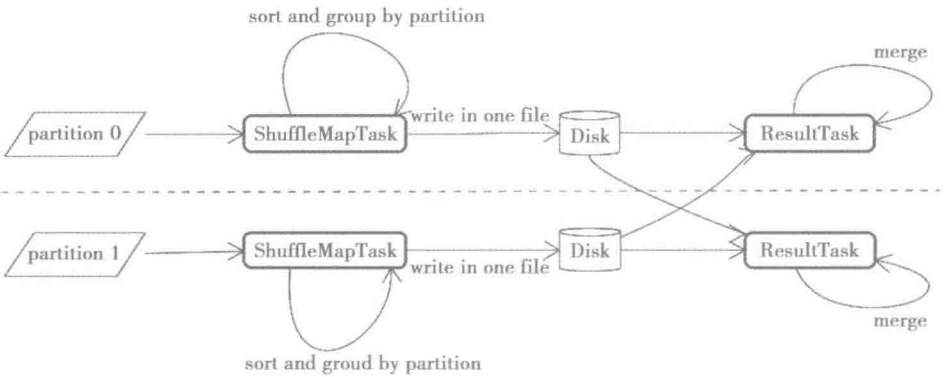


图 6-7 map 端简单缓存、排序分组与 reduce 端合并组合

6.7.3 在 map 端缓存中聚合、排序分组，在 reduce 端组合

此种情况在缓存中对数据按照 key 聚合，并且利用指定的排序函数对数据按照 partition 或者 key 进行排序，最后按 partition 顺序合并写入同一文件。当指定了聚合函数时，一般采用这种方式，见图 6-8。此种方式将多个 bucket 合并到同一个文件，通过减少 map 输出的文件数量，节省了磁盘 I/O，提升了性能；对中间输出数据不是一次性读取，而是逐条放入 AppendOnlyMap 的缓存，并对数据进行聚合，减少了中间结果占用的内存大小；对 AppendOnlyMap 的缓存进行溢出判断，当超出 myMemoryThreshold 的大小时，将数据写入磁盘，防止内存溢出。

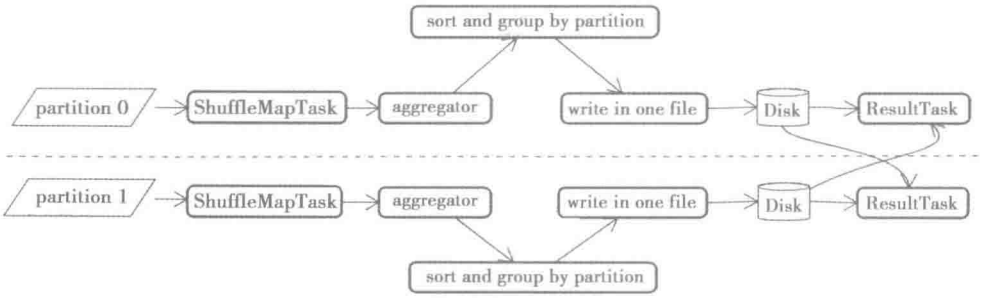


图 6-8 map 端缓存中聚合、排序分组与 reduce 端组合

6.8 小结

本章从迭代计算的层层剥离开始，分别分析了 map 和 reduce 任务的处理逻辑。这其中不乏值得读者学习的代码实现，比如：

- RDD 迭代计算是如何实现容错的？基于缓存和检查点。
- map 任务是如何聚合的？使用 AppendOnlyMap 提供的数据结构来实现聚合。
- 为什么有时候需要在 map 端聚合？为了降低网络 I/O，提升性能。
- map 任务如何输出？按照分区排序或者分组后生成分区文件，并创建分区索引文件标记文件中各个分区数据的偏移量和长度。
- reduce 任务是如何获取 map 任务的输出的？通过 mapOutputTracker 获取到 map 任务所在 Executor 的 BlockManagerId 和 Block 的大小，然后使用 shuffleClient 下载。
- 发送请求时有哪些性能优化？对请求分批发送，限制分批请求的大小，并行发送请求以及将多个请求数据小的请求合并等。

Chapter 7 第 7 章

部署模式

事在四方，要在中央。圣人执要，四方来效。

——《韩非子》

本章导读

之前的章节，笔者为了突出讲解 Spark 各个组件的原理，主要以 local 部署模式进行源码剖析。其实无论什么模式，大多数代码逻辑都是通用的。由于各种模式在部署上有很多差异，所以笔者在本章专门讲解各个部署模式的差异以及部署的容错。

Spark 目前支持的部署方式如下：

- 本地部署模式：local、local[N] 或者 local[N, maxRetries]。主要用于代码调试和跟踪。不具备容错能力，所以不适用于生产环境。
- 本地集群部署模式：local-cluster[N, cores, memory]。也主要用于代码调试和测试，是源码学习常用的模式。不具备容错能力，不能用于生产环境。
- Standalone 部署模式：spark://。具备容错能力并且支持分布式部署，所以可用于实际的生产。
- 第三方部署模式：yarn-standalone、yarn-cluster、mesos://、zk://、simr:// 等。

在正式开始本章内容之前，先介绍一些概念：

- Driver：应用驱动程序，可以理解是老板的客户。
- Master：Spark 的主控节点，可以理解为集群的老板。
- Worker：Spark 的工作节点，可以理解为集群的各个主管。
- Executor：Spark 的工作进程，由 Worker 监管，负责具体任务的执行。

7.1 local 部署模式

本书在讲解各章内容时，主要都以 local 模式为例。local 部署模式只有 Driver，没有 Master 和 Worker，执行任务的 Executor 与 Driver 在同一个 JVM 进程内。为了更直观地感受 local 模式，我们以 local 模式下的任务提交与执行过程为例，见图 7-1。

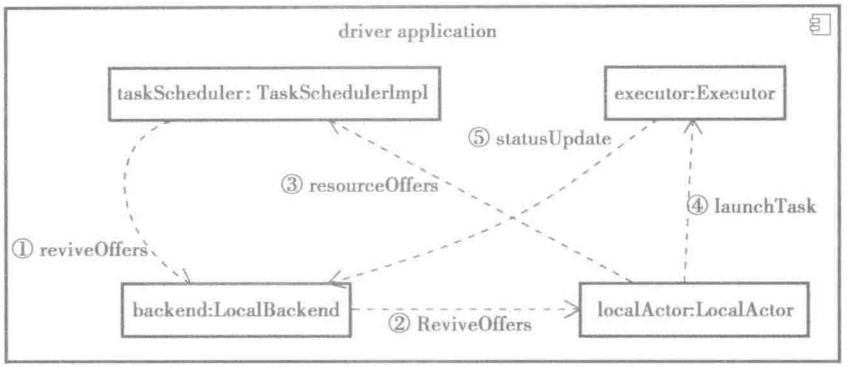


图 7-1 local 模式下的任务提交与执行过程

图 7-1 中 local 模式下的任务提交与执行过程如下：

1）local 模式下 ExecutorBackend 的实现类是 LocalBackend，当有任务要提交时，由 TaskSchedulerImpl 调用 LocalBackend 的 reviveOffers 方法申请资源；

2）LocalBackend 向 LocalActor 发送 ReviveOffers 消息，为任务申请资源；

3）LocalActor 收到 ReviveOffers 消息后，调用 TaskSchedulerImpl 的 resourceOffers 方法申请资源，TaskSchedulerImpl 将根据任务申请的 CPU 核数和内存及本地化等条件为其分配资源；

4）任务获得资源后，调用 Executor 的 launchTask 方法运行任务；

5）在任务运行过程中，Executor 中运行的 TaskRunner 通过调用 LocalBackend 的 statusUpdate 方法，实现向 LocalActor 发送 StatusUpdate 消息。LocalActor 接收到 StatusUpdate 消息，再调用 TaskSchedulerImpl 的 statusUpdate 不断更新任务的状态。任务的状态有 LAUNCHING、RUNNING、FiNISHED、FAILED、KILLED 和 LOST 等。整个过程在第 5 章已经做了详细描述。

7.2 local-cluster 部署模式

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

- 使用 LocalSparkCluster 启动集群。

- SparkDeploySchedulerBackend 的启动过程不同。
- AppClient 的启动与调度。
- local-cluster 模式的任务执行。

在 3.6 节，笔者曾经以 local 模式为例，讲解了 TaskSchedulerImpl 和 LocalBackend。本节我们设置 master 为“local-cluster[2,1,1024]”，那么在创建 TaskSchedulerImpl 时就会匹配 local-cluster 模式。“local-cluster[2,1,1024]”中的 2 指定了 Worker 数量（numSlaves），1 指定了每个 Worker 占用的 CPU 核数（coresPerSlave），1024 指定的是每个 Worker 占用的内存大小（memoryPerSlave）。memoryPerSlave 必须比 executorMemory 大，因为 Worker 的内存大小包括 Executor 占用的内存。

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


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

SparkContext 匹配 local-cluster 模式的代码见代码清单 7-1。

代码清单7-1 SparkContext匹配local-cluster模式的代码

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

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

7.2.1 LocalSparkCluster 的启动

在介绍 LocalSparkCluster 的启动之前，先介绍下 LocalSparkCluster 中定义的一些数据

结构：

- masterActorSystems：用于缓存所有的 Master 的 ActorSystem；
- workerActorSystems：维护所有的 Worker 的 ActorSystem。

LocalSparkCluster 的 start 方法用于创建、启动 Master 的 ActorSystem 与多个 Worker 的 ActorSystem，LocalSparkCluster 的 stop 方法关闭、清理 Master 的 ActorSystem 与多个 Worker 的 ActorSystem，见代码清单 7-2。

代码清单7-2 LocalSparkCluster的实现

```
private[spark]
class LocalSparkCluster(numWorkers: Int, coresPerWorker: Int, memoryPerWorker: Int)
    extends Logging {

    private val localHostname = Utils.localHostName()
    private val masterActorSystems = ArrayBuffer[ActorSystem]()
    private val workerActorSystems = ArrayBuffer[ActorSystem]()

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

        val conf = new SparkConf(false)
        val (masterSystem, masterPort, _) = Master.startSystemAndActor(localHost
             name, 0, 0, conf)
        masterActorSystems += masterSystem
        val masterUrl = "spark://" + localHostname + ":" + masterPort
        val masters = Array(masterUrl)

        for (workerNum <- 1 to numWorkers) {
            val (workerSystem, _) = Worker.startSystemAndActor(localHostname, 0,
                 0, coresPerWorker,
                memoryPerWorker, masters, null, Some(workerNum))
            workerActorSystems += workerSystem
        }

        masters
    }

    def stop() {
        logInfo("Shutting down local Spark cluster.")
        workerActorSystems.foreach(_.shutdown())
        masterActorSystems.foreach(_.shutdown())
        masterActorSystems.clear()
        workerActorSystems.clear()
    }
}
```

1. 启动 Master

startSystemAndActor 方法（见代码清单 7-3）用于创建、启动 Master 的 ActorSystem，然后将 Master 注册到 ActorSystem。关于 ActorSystem 的创建细节，已在 3.2.2 节详细介绍过，

此处不再赘述。Master 的 ActorSystem 的 Akka 访问地址是 akka://sparkMaster，Akka 端口在每次启动时会不同。笔者本地当前注册的 Master 的访问地址为 Actor[akka://sparkMaster/user/Master#1813834079]。

代码清单7-3 Master.startSystemAndActor的实现

```
def startSystemAndActor(
    host: String,
    port: Int,
        webUiPort: Int,
        conf: SparkConf): (ActorSystem, Int, Int) = {
    val securityMgr = new SecurityManager(conf)
    val (actorSystem, boundPort) = AkkaUtils.createActorSystem(systemName, host,
         port, conf = conf,
        securityManager = securityMgr)
    val actor = actorSystem.actorOf(Props(classOf[Master], host, boundPort,
         webUiPort, securityMgr), actorName)
    val timeout = AkkaUtils.askTimeout(conf)
    val respFuture = actor.ask(RequestWebUIPort)(timeout)
    val resp = Await.result(respFuture, timeout).asInstanceOf[WebUIPortResponse]
    (actorSystem, boundPort, resp.webUIBoundPort)
}
```

向 Master 的 ActorSystem 注册 Master 时，会先触发其 preStart 方法（见代码清单 7-4）。

代码清单7-4 Master的preStart方法代码

```
override def preStart() {
    logInfo("Starting Spark master at " + masterUrl)
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
    webUi.bind()
    masterWebUiUrl = "http://" + masterPublicAddress + ":" + webUi.boundPort
    context.system.scheduler.schedule(0 millis, WORKER_TIMEOUT millis, self,
        CheckForWorkerTimeOut)

    masterMetricsSystem.registerSource(masterSource)
    masterMetricsSystem.start()
    applicationMetricsSystem.start()
    masterMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
    applicationMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)

    persistenceEngine = RECOVERY_MODE match {
        case "ZOOKEEPER" =>
            logInfo("Persisting recovery state to ZooKeeper")
            new ZooKeeperPersistenceEngine(SerializationExtension(context.system), conf)
        case "FILESYSTEM" =>
            logInfo("Persisting recovery state to directory: " + RECOVERY_DIR)
            new FileSystemPersistenceEngine(RECOVERY_DIR, SerializationExtension-
(context.system))
        case _ =>
            new BlackHolePersistenceEngine()
    }
    leaderElectionAgent = RECOVERY_MODE match {
```

```
            case "ZOOKEEPER" =>
                context.actorOf(Props(classOf[ZooKeeperLeaderElectionAgent], self,
                    masterUrl, conf))
            case _ =>
                context.actorOf(Props(classOf[MonarchyLeaderAgent], self))
        }
    }
```

preStart 的处理步骤如下：

1）订阅 RemotingLifecycleEvent，监听远程客户端断开连接的事件。

2）给 ActorSystem 增加定时调度，向自身发送 CheckForWorkerTimeOut 消息。Master 接收到 CheckForWorkerTimeOut 消息后，会匹配调用 timeOutDeadWorkers 方法处理，代码如下。

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

timeOutDeadWorkers 方法（见代码清单 7-5）的处理步骤如下：

① 过滤出所有超时的 Worker。即使用当前时间减去 Worker 最大超时时间仍然大于 lastHeartbeat 的 Worker 节点。

② 如果 WorkerInfo 的状态是 DEAD，则等待足够长的时间后将它从 workers 列表中移除。足够长的时间的计算公式为：属性 spark.dead.worker.persistence 的值乘以 Worker 最大超时时间。spark.dead.worker.persistence 默认等于 15。如果 WorkerInfo 的状态不是 DEAD，则调用 removeWorker 方法（见代码清单 7-6）将 WorkerInfo 的状态置为 DEAD，从 idToWorker 缓存中移除 Worker 的 id，从 addressToWorker 的缓存中移除 WorkerInfo，最后向此 WorkerInfo 的所有 Executor 所服务的 Drvier application 发送 ExecutorUpdated 消息，更新 Executor 的状态为 LOST。最后使用 removeDriver（见代码清单 7-7）重新调度之前调度给此 Worker 的 Driver。

代码清单7-5 Master.timeOutDeadWorkers的实现

```
def timeOutDeadWorkers() {
    val currentTime = System.currentTimeMillis()
    val toRemove = workers.filter(_.lastHeartbeat < currentTime - WORKER_TIMEOUT).
        toArray
    for (worker <- toRemove) {
        if (worker.state != WorkerState.DEAD) {
            logWarning("Removing %s because we got no heartbeat in %d seconds".format(
                worker.id, WORKER_TIMEOUT/1000))
            removeWorker(worker)
        } else {
            if (worker.lastHeartbeat < currentTime - ((REAPER_ITERATIONS + 1) *
                WORKER_TIMEOUT)) {
                workers -= worker
            }
        }
    }
}
```

代码清单7-6 删除Worker的代码

```
def removeWorker(worker: WorkerInfo) {
    logInfo("Removing worker " + worker.id + " on " + worker.host + ":" + worker.port)
    worker.setState(WorkerState.DEAD)
    idToWorker -= worker.id
    addressToWorker -= worker.actor.path.address
    for (exec <- worker.executors.values) {
        logInfo("Telling app of lost executor: " + exec.id)
        exec.application.driver ! ExecutorUpdated(
            exec.id, ExecutorState.LOST, Some("worker lost"), None)
        exec.application.removeExecutor(exec)
    }
    for (driver <- worker.drivers.values) {
        if (driver.desc.supervise) {
            logInfo(s"Re-launching ${driver.id}")
            relaunchDriver(driver)
        } else {
            logInfo(s"Not re-launching ${driver.id} because it was not supervised")
            removeDriver(driver.id, DriverState.ERROR, None)
        }
    }
    persistenceEngine.removeWorker(worker)
}
```

代码清单7-7 删除Driver的实现

```
def removeDriver(driverId: String, finalState: DriverState, exception: Option[Exception]) {
    drivers.find(d => d.id == driverId) match {
        case Some(driver) =>
            logInfo(s"Removing driver: $driverId")
            drivers -= driver
            if (completedDrivers.size >= RETAINED_DRIVERS) {
                val toRemove = math.max(RETAINED_DRIVERS / 10, 1)
                completedDrivers.trimStart(toRemove)
            }
            completedDrivers += driver
            persistenceEngine.removeDriver(driver)
            driver.state = finalState
            driver.exception = exception
            driver.worker.foreach(w => w.removeDriver(driver))
            schedule()
        case None =>
            logWarning(s"Asked to remove unknown driver: $driverId")
    }
}
```

3）启动 webUI、masterMetricsSystem 和 applicationMetricsSystem，然后给 masterMetricsSystem 和 applicationMetricsSystem 创建 ServletContextHandler 并注册到 webUI。这些内容读者可以回顾 3.4 及 3.9 几节的内容。

4）选择故障恢复的持久化引擎（persistenceEngine），这部分内容将在 7.4 节讲解。

5）选择领导选举代理（leaderElectionAgent），local-cluster模式中将向Master的ActorSystem注册MonarchyLeaderAgent，因此触发MonarchyLeaderAgent的preStart方法向Master发送ElectedLeader消息，代码如下。

```
override def preStart() {
    masterActor ! ElectedLeader
}
```

Master接收到ElectedLeader消息后会进行选举操作，由于local-cluster模式中只有一个Master，所以persistenceEngine没有持久化的App、Driver、Worker的信息，所以当前Master即为激活（ALIVE）状态的，见代码清单7-8。这里涉及Master的故障恢复，也将在7.4节讲解。

代码清单7-8 Master对选举消息的处理

```
case ElectedLeader => {
    val (storedApps, storedDrivers, storedWorkers) = persistenceEngine.
         readPersistedData()
    state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.
        isEmpty) {
        RecoveryState.ALIVE
    } else {
        RecoveryState.RECOVERING
    }
    logInfo("I have been elected leader! New state: " + state)
    if (state == RecoveryState.RECOVERING) {
        beginRecovery(storedApps, storedDrivers, storedWorkers)
        recoveryCompletionTask = context.system.scheduler.scheduleOnce(WORKER_
            TIMEOUT millis, self,
          CompleteRecovery)
    }
}
```

最后将刚刚创建的masterSystem注册到缓存masterActorSystems，并将Master的Akka地址注册到缓存masters。

2. 启动Worker

创建完masterSystem，开始创建、启动Worker的ActorSystem，见代码清单7-9。所有Worker的ActorSystem的akka的访问地址以akka://sparkWorker加Worker编号来访问，例如：akka://sparkWorker1、akka://sparkWorker2等。每个Worker的ActorSystem都需要注册自身的Worker。同时每个Worker的ActorSystem都需要注册到workerActorSystems缓存。

代码清单7-9 Worker.startSystemAndActor的实现

```
def startSystemAndActor(
        host: String,
        port: Int,
        webUiPort: Int,
```

```
        cores: Int,
        memory: Int,
        masterUrls: Array[String],
        workDir: String,
        workerNumber: Option[Int] = None): (ActorSystem, Int) = {
    val conf = new SparkConf
    val systemName = "sparkWorker" + workerNumber.map(_.toString).getOrElse("")
    val actorName = "Worker"
    val securityMgr = new SecurityManager(conf)
    val (actorSystem, boundPort) = AkkaUtils.createActorSystem(systemName, host, port,
        conf = conf, securityManager = securityMgr)
    actorSystem.actorOf(Props(classOf[Worker], host, boundPort, webUiPort, cores, memory,
        masterUrls, systemName, actorName,  workDir, conf, securityMgr), name = actorName)
    (actorSystem, boundPort)
}
```

注册 Worker 时触发它的 preStart 方法（见代码清单 7-10），其处理步骤如下：

1）创建工作目录。

2）订阅 RemotingLifecycleEvent，监听远程客户端断开连接的事件。

3）启动 shuffleService，此处的 shuffleService 虽然是 StandaloneWorkerShuffleService，但是其原理和 4.2 节讲解的 ShuffleClient 一样使用了 Netty 的异步网络框架，因此不再赘述。

4）创建 WorkerWebUI 并启动，可以参考 3.4 节。

5）将 Worker 注册到 Master。

6）启动 metricsSystem，并且给 metricsSystem 的测量信息创建 ServletContextHandler 后注册到 webUI，读者可以参考 3.4 和 3.9 两节中介绍的内容，其过程都是类似的。

代码清单7-10　Worker.preStart的代码实现

```
override def preStart() {
    assert(!registered)
    logInfo("Starting Spark worker %s:%d with %d cores, %s RAM".format(
        host, port, cores, Utils.megabytesToString(memory)))
    logInfo("Spark home: " + sparkHome)
    createWorkDir()
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
    shuffleService.startIfEnabled()
    webUi = new WorkerWebUI(this, workDir, webUiPort)
    webUi.bind()
    registerWithMaster()

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

Worker 的 registerWithMaster 方法（见代码清单 7-11）用于将 Worker 注册到 Master。其中调用了 tryRegisterAllMasters 方法（见代码清单 7-12）向所有 Master 发送 RegisterWorker 消息。

代码清单7-11 将Worker注册到Master的代码

```
def registerWithMaster() {
    registrationRetryTimer match {
        case None =>
            registered = false
            tryRegisterAllMasters()
            connectionAttemptCount = 0
            registrationRetryTimer = Some {
                context.system.scheduler.schedule(INITIAL_REGISTRATION_RETRY_INTERVAL,
                    INITIAL_REGISTRATION_RETRY_INTERVAL, self, ReregisterWithMaster)
            }
        case Some(_) =>
            logInfo("Not spawning another attempt to register with the master,
                since there is an" +
                " attempt scheduled already.")
    }
}
```

代码清单7-12 Worker.tryRegisterAllMasters的实现

```
private def tryRegisterAllMasters() {
    for (masterUrl <- masterUrls) {
        logInfo("Connecting to master " + masterUrl + "...")
        val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
        actor ! RegisterWorker(workerId, host, port, cores, memory, webUi.
            boundPort, publicAddress)
    }
}
```

代码清单 7-13 列出了 Master 收到 RegisterWorker 消息后的处理步骤：

1）创建 WorkerInfo。

2）注册 WorkerInfo。

3）向 Worker 发送 RegisteredWorker 消息，表示注册完成。

4）调用 schedule 方法进行资源调度。

代码清单7-13 Master处理RegisterWorker消息的代码

```
case RegisterWorker(id, workerHost, workerPort, cores, memory, workerUiPort,
    publicAddress) =>
{
    logInfo("Registering worker %s:%d with %d cores, %s RAM".format(
        workerHost, workerPort, cores, Utils.megabytesToString(memory)))
    if (state == RecoveryState.STANDBY) {
        // ignore, don't send response
    } else if (idToWorker.contains(id)) {
        sender ! RegisterWorkerFailed("Duplicate worker ID")
    } else {
        val worker = new WorkerInfo(id, workerHost, workerPort, cores, memory,
            sender, workerUiPort, publicAddress)
        if (registerWorker(worker)) {
            persistenceEngine.addWorker(worker)
```

```
                sender ! RegisteredWorker(masterUrl, masterWebUiUrl)
                schedule()
            } else {
                val workerAddress = worker.actor.path.address
                logWarning("Worker registration failed. Attempted to re-register worker
                    at same " +
                    "address: " + workerAddress)
                sender ! RegisterWorkerFailed("Attempted to re-register worker at same
                    address: "
                    + workerAddress)
            }
        }
    }
```

注册 WorkerInfo，其实就是将其添加到 workers：HashSet[WorkerInfo] 中，并且更新 worker id 与 worker 以及 worker address 与 worker 的映射关系，见代码清单 7-14。

代码清单7-14 Master.registerWorker的实现

```
def registerWorker(worker: WorkerInfo): Boolean = {
    workers.filter { w =>
        (w.host == worker.host && w.port == worker.port) && (w.state ==
            WorkerState.DEAD)
    }.foreach { w =>
        workers -= w
    }

    val workerAddress = worker.actor.path.address
    if (addressToWorker.contains(workerAddress)) {
        val oldWorker = addressToWorker(workerAddress)
        if (oldWorker.state == WorkerState.UNKNOWN) {
            removeWorker(oldWorker)
        } else {
            logInfo("Attempted to re-register worker at same address: " + workerAddress)
            return false
        }
    }

    workers += worker
    idToWorker(worker.id) = worker
    addressToWorker(workerAddress) = worker
    true
}
```

代码清单 7-15 是 Worker 接受 RegisteredWorker 消息的处理逻辑，步骤如下：

1）标记注册成功。

2）调用 changeMaster 方法（见代码清单 7-16）更新 activeMasterUrl、activeMasterWebUiUrl、master、masterAddress 等信息。

3）启动定时调度给自己发送 SendHeartbeat 消息。

代码清单7-15 Worker处理RegisteredWorker的代码

```
case RegisteredWorker(masterUrl, masterWebUiUrl) =>
    logInfo("Successfully registered with master " + masterUrl)
    registered = true
    changeMaster(masterUrl, masterWebUiUrl)
    context.system.scheduler.schedule(0 millis, HEARTBEAT_MILLIS millis, self,
        SendHeartbeat)
    if (CLEANUP_ENABLED) {
        logInfo(s"Worker cleanup enabled; old application directories will be
            deleted in: $workDir")
        context.system.scheduler.schedule(CLEANUP_INTERVAL_MILLIS millis,
            CLEANUP_INTERVAL_MILLIS millis, self, WorkDirCleanup)
    }
```

代码清单7-16 changeMaster的实现

```
def changeMaster(url: String, uiUrl: String) {
    activeMasterUrl = url
    activeMasterWebUiUrl = uiUrl
    master = context.actorSelection(Master.toAkkaUrl(activeMasterUrl))
    masterAddress = activeMasterUrl match {
        case Master.sparkUrlRegex(_host, _port) =>
            Address("akka.tcp", Master.systemName, _host, _port.toInt)
        case x =>
            throw new SparkException("Invalid spark URL: " + x)
    }
    connected = true
    // Cancel any outstanding re-registration attempts because we found a new
        master
    registrationRetryTimer.foreach(_.cancel())
    registrationRetryTimer = None
}
```

Worker 收到 SendHeartbeat 消息后，会向 Master 转发 Heartbeat 消息，代码如下。

```
case SendHeartbeat =>
    if (connected) { master ! Heartbeat(workerId) }
```

Master 收到 Heartbeat 消息后的实现（见代码清单 7-17）用于更新 WorkerInfo 的 lastHeartbeat，即最后一次接收到心跳的时间戳。如果 Worker 的 id 与 Worker 的映射关系（idToWorker）中找不到匹配的 Worker，但是 Worker 的缓存（workers）中却存在此 id，那么向 Worker 发送 ReconnectWorker 消息。

代码清单7-17 Master接收Heartbeat消息的代码

```
case Heartbeat(workerId) => {
    idToWorker.get(workerId) match {
        case Some(workerInfo) =>
            workerInfo.lastHeartbeat = System.currentTimeMillis()
        case None =>
            if (workers.map(_.id).contains(workerId)) {
                logWarning(s"Got heartbeat from unregistered worker $workerId." +
```

```
                " Asking it to re-register.")
            sender ! ReconnectWorker(masterUrl)
        } else {
            logWarning(s"Got heartbeat from unregistered worker $workerId." +
                " This worker was never registered, so ignoring the heartbeat.")
        }
    }
}
```

经过以上源码分析，我们得出以下结论：local-cluster模式下有一个Master和多个Worker，它们位于同一个JVM内，通过各自启动的ActorSystem通信。

7.2.2 CoarseGrainedSchedulerBackend的启动

local-cluster模式中，除了创建TaskScheduler的时候与local模式不同，启动taskScheduler时，也会不同。根据3.8节我们知道，最终会调用backend的start方法。在local-cluster模式中，backend为SparkDeploySchedulerBackend。SparkDeploySchedulerBackend的start方法（见代码清单7-18）的执行过程如下：

1）调用父类CoarseGrainedSchedulerBackend的start方法，注册DriverActor。

2）进行一些参数、Java选项、类路径的设置。这些配置被封装为Command，然后由ApplicationDescription持有。ApplicationDescription作为AppClient的构造参数，在AppClient启动时传递给Worker。Worker最终利用ApplicationDescription里封装的Command启动Executor。

3）启动AppClient。

代码清单7-18 SparkDeploySchedulerBackend的启动实现

```
override def start() {
    super.start()

    val driverUrl = "akka.tcp://%s@%s:%s/user/%s".format(
        SparkEnv.driverActorSystemName,
        conf.get("spark.driver.host"),
        conf.get("spark.driver.port"),
        CoarseGrainedSchedulerBackend.ACTOR_NAME)
    val args = Seq(driverUrl, "{{EXECUTOR_ID}}", "{{HOSTNAME}}", "{{CORES}}",
        "{{APP_ID}}",
        "{{WORKER_URL}}")
    val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")
        .map(Utils.splitCommandString).getOrElse(Seq.empty)
    val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath").
        toSeq.flatMap { cp =>
        cp.split(java.io.File.pathSeparator)
    }
    val libraryPathEntries =
        sc.conf.getOption("spark.executor.extraLibraryPath").toSeq.flatMap { cp =>
            cp.split(java.io.File.pathSeparator)
        }
```

```
        // Start executors with a few necessary configs for registering with the scheduler
        val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
        val javaOpts = sparkJavaOpts ++ extraJavaOpts
        val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
            args, sc.executorEnvs, classPathEntries, libraryPathEntries, javaOpts)
        val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
        val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory,
             command,
            appUIAddress, sc.eventLogDir)

        client = new AppClient(sc.env.actorSystem, masters, appDesc, this, conf)
        client.start()

        waitForRegistration()
    }
```

注意 此处的 CoarseGrainedSchedulerBackend 是 org.apache.spark.scheduler.cluster 包下的 CoarseGrainedSchedulerBackend。

CoarseGrainedSchedulerBackend 的 start 方法（见代码清单 7-19）从 sc.conf 中复制 Spark 属性，然后注册并持有 DriverActor 的引用，这与 LocalBackend 持有 LocalActor 的引用不同。

代码清单7-19 CoarseGrainedSchedulerBackend的start方法

```
override def start() {
    val properties = new ArrayBuffer[(String, String)]
    for ((key, value) <- scheduler.sc.conf.getAll) {
        if (key.startsWith("spark.")) {
            properties += ((key, value))
        }
    }
    driverActor = actorSystem.actorOf(
        Props(new DriverActor(properties)), name = CoarseGrainedSchedulerBackend.
            ACTOR_NAME)
}
```

7.2.3 启动 AppClient

AppClient 主要用于代表 Application 和 Master 通信。AppClient 在启动时，会向 Driver 的 ActorSystem 注册 ClientActor，代码如下。

```
def start() {
    actor = actorSystem.actorOf(Props(new ClientActor))
}
```

由于向 ActorSystem 注册 Actor 时，ActorSystem 会首先回调 Actor 的 preStart 方法，所以 ClientActor 在正式启动前会触发其 preStart 方法，见代码清单 7-20。preStart 方法首先向 Driver 的 ActorSystem 订阅 RemotingLifecycleEvent，然后调用 registerWithMaster。

代码清单7-20 ClientActor的preStart方法

```
override def preStart() {
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
    try {
        registerWithMaster()
    } catch {
        case e: Exception =>
            logWarning("Failed to connect to master", e)
            markDisconnected()
            context.stop(self)
    }
}
```

registerWithMaster方法（见代码清单7-21）调用tryRegisterAllMasters方法，向所有的Master注册当前Application。其中Master依然使用system.actorSelection方式获得，可参阅附录B。注册Application实际是通过向Master发送RegisterApplication消息实现。注册Application时，如果失败，最多会尝试3次。

代码清单7-21 ClientActor的registerWithMaster实现

```
    def tryRegisterAllMasters() {
        for (masterUrl <- masterUrls) {
            logInfo("Connecting to master " + masterUrl + "...")
            val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
            actor ! RegisterApplication(appDescription)
        }
    }

    def registerWithMaster() {
        tryRegisterAllMasters()
        import context.dispatcher
        var retries = 0
        registrationRetryTimer = Some {
            context.system.scheduler.schedule(REGISTRATION_TIMEOUT, REGISTRATION_
                TIMEOUT) {
                Utils.tryOrExit {
                    retries += 1
                    if (registered) {
                    registrationRetryTimer.foreach(_.cancel())
                } else if (retries >= REGISTRATION_RETRIES) {
                    markDead("All masters are unresponsive! Giving up.")
                } else {
                    tryRegisterAllMasters()
                }
            }
        }
    }
}
```

Master 接收到 RegisterApplication 消息后，其处理步骤（见代码清单 7-22）如下。

1）创建 ApplicationInfo。

2）注册 ApplicationInfo。

3）向 ClientActor 发送 RegisteredApplication 消息。

4）调用 schedule 方法，执行调度。

代码清单7-22 Master对RegisterApplication消息的处理

```
case RegisterApplication(description) => {
    if (state == RecoveryState.STANDBY) {
        // ignore, don't send response
    } else {
        logInfo("Registering app " + description.name)
        val app = createApplication(description, sender)
        registerApplication(app)
        logInfo("Registered app " + description.name + " with ID " + app.id)
        persistenceEngine.addApplication(app)
        sender ! RegisteredApplication(app.id, masterUrl)
        schedule()
    }
}
```

创建 ApplicationInfo 的实现如下。

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

创建 ApplicationInfo 时，调用了 init 方法（见代码清单 7-23）。其中创建了 ApplicationSource，与创建 ExecutorSource 是一样的，请参阅 3.8 节。另外还声明了 executors 用于缓存分配给 Application 的 Executor。

代码清单7-23 ApplicationInfo的初始化代码

```
private def init() {
    state = ApplicationState.WAITING
    executors = new mutable.HashMap[Int, ExecutorInfo]
    coresGranted = 0
    endTime = -1L
    appSource = new ApplicationSource(this)
    nextExecutorId = 0
    removedExecutors = new ArrayBuffer[ExecutorInfo]
}
```

注册 ApplicationInfo 的过程（见代码清单 7-24）如下：

1）向 applicationMetricsSystem 注册 ApplicationSource，与注册 ExecutorSource 是一样的，

请参阅 3.8.2 节。

2）注册 ApplicationInfo，并且更新它与 app id、app driver、app address 的关系。

代码清单7-24 Master.registerApplication的实现

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

    applicationMetricsSystem.registerSource(app.appSource)
    apps += app
    idToApp(app.id) = app
    actorToApp(app.driver) = app
    addressToApp(appAddress) = app
    waitingApps += app
}
```

向 ClientActor 发送的 RegisteredApplication 消息中，包含 app id 和 masterUrl。ClientActor 接收 RegisteredApplication 消息后的处理步骤（见代码清单 7-25）如下：

1）更新 appId，并标识当前 Application 已经注册到 Master；

2）调用 changeMaster 更新 activeMasterUrl、master、masterAddress 等信息；

3）调用 SparkDeploySchedulerBackend 的 connected 方法（见代码清单 7-26）更新 appId 并且调用 notifyContext 方法（见代码清单 7-27）标识 Application 注册完成。

代码清单7-25 ClientActor接收RegisteredApplication消息的代码

```
case RegisteredApplication(appId_, masterUrl) =>
    appId = appId_
    registered = true
    changeMaster(masterUrl)
    listener.connected(appId)
```

代码清单7-26 SparkDeploySchedulerBackend.connected方法

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

代码清单7-27 SparkDeploySchedulerBackend.notifyContext的实现

```
private def notifyContext() = {
    registrationLock.synchronized {
        registrationDone = true
```

```
            registrationLock.notifyAll()
        }
    }
```

到这里，整个 local-cluster 模式的启动过程就介绍完了，可以用图 7-2 来直观表示。

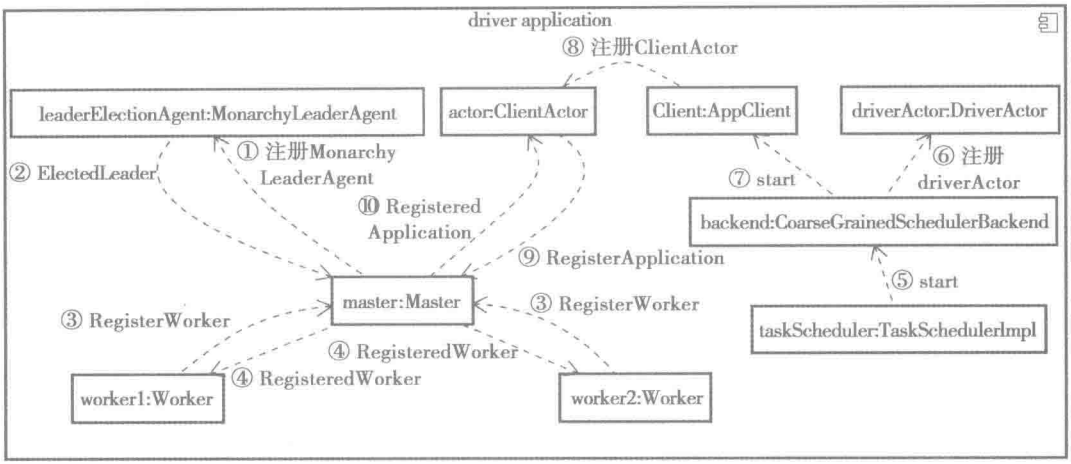


图 7-2　local-cluster 模式的启动过程

这里对图 7-2 中 local-cluster 模式的启动过程进行简短介绍：

1）LocalSparkCluster 首先启动 Master。Master 会完成订阅 RemotingLifecycleEvent，启动检测 Worker 是否死亡的定时调度，启动 webUI，启动测量系统，选择故障恢复的持久化引擎，最后向 Master 的 ActorSystem 注册领导选举代理（默认为 MonarchyLeaderAgent）。

2）注册 MonarchyLeaderAgent 时，ActorSystem 会回调 MonarchyLeaderAgent 的 preStart 方法向 Master 发送 ElectedLeader 消息进行 Master 的领导选举。由于 local-cluster 模式只有一个 Master，因此它将被选举为激活状态的 Master。

3）LocalSparkCluster 启动多个 Worker。每个 Worker 都会完成创建工作目录，订阅 RemotingLifecycleEvent，启动 shuffleService，启动 WebUI，启动测量系统，最后向 Master 注册 Worker 等工作。

4）Master 收到 RegisterWorker 消息后完成创建与注册 WorkerInfo，向 Worker 发送 RegisteredWorker 消息以表示注册完成，最后调用 schedule 方法进行资源调度。

5）启动 taskScheduler 时调用 SparkDeploySchedulerBackend 的 start 方法。

6）SparkDeploySchedulerBackend 调用父类 CoarseGrainedSchedulerBackend 的 start 方法，向 ActorSystem 注册 DriverActor。

7）进行一些参数、Java 选项、类路径的设置。这些配置被封装为 Command，然后由 ApplicationDescription 持有，最后启动 AppClient。这个 Command 用于在 Worker 上启动 CoarseGrainedExecutorBackend 进程，此进程将创建 Executor 执行任务。

8）启动 AppClient 时，向 ActorSystem 注册 ClientActor。

9）注册 ClientActor 时，ActorSystem 回调其 preStart 方法，完成订阅 RemotingLifecycleEvent，并且调用 registerWithMaster 方法向 Master 发送 RegisterApplication 消息。

10）Master 接收到 RegisterApplication 消息后，完成创建 ApplicationInfo，注册 ApplicationInfo，向 ClientActor 发送 RegisteredApplication 消息，最后执行调度。

7.2.4 资源调度

至此我们看到了 Master、Worker 及 Application 的启动与注册，Executor 是作为计算资源看待的，但是一直不见 Executor 的踪影，Executor 是何时创建的呢？ Application 又是何时与 Executor 取得联系的呢？换言之，Executor 是什么时候分配给 Application 处理任务的呢？下面我们一起来看看 Executor 是如何创建和调度的。

无论是注册 Worker 还是注册 Application，最后都会调用 schedule 方法（见代码清单 7-28）对资源调度。资源调度的过程分为两步：逻辑分配和物理分配。

代码清单7-28 Master的schedule方法代码

```
for (app <- waitingApps if app.coresLeft > 0) {
    val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
        .filter(canUse(app, _)).sortBy(_.coresFree).reverse
    val numUsable = usableWorkers.length
    val assigned = new Array[Int](numUsable) // Number of cores to give on each
     node
    var toAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)
    var pos = 0
    while (toAssign > 0) {
        if (usableWorkers(pos).coresFree - assigned(pos) > 0) {
            toAssign -= 1
            assigned(pos) += 1
        }
        pos = (pos + 1) % numUsable
    }
    for (pos <- 0 until numUsable) {
        if (assigned(pos) > 0) {
            val exec = app.addExecutor(usableWorkers(pos), assigned(pos))
            launchExecutor(usableWorkers(pos), exec)
            app.state = ApplicationState.RUNNING
        }
    }
}
```

1. 计算资源逻辑分配

对资源的逻辑分配主要指对 Worker 的 CPU 核数的分配，即将当前 Application 的 CPU 核数需求分配到所有的 Worker 上。而那些内存不满足 ApplicationDescription 中指定的 memoryPerSlave 变量的大小的会被过滤掉。给 Application 分配 CPU 核数的整个处理步骤如下。

1）过滤出所有可用的 Worker，条件如下：

- 处于激活（ALIVE）状态；
- 空闲内存大于等于 Application 在每个 Worker 上需要的内存（memoryPerSlave）；
- 还没有为此 Application 运行过 Executor。

2）对于过滤得到的 Worker 按照其空闲内核数倒序排列。

3）实际需要分配的内核数，从 Application 需要的内核数与所有过滤后的 Worker 的空闲内核数的总和两者中取最小值。

4）如果需要分配的内核数大于 0，则逐个从各个 Worker 中给 Application 分配 1 个内核。当所有 Worker 都分配过后，需要分配的内核数依然大于 0，则从头一个 Worker 再次分配，如此往复，直到 Application 需要的内核数为 0。

为了方便，假设满足过滤条件并且已经按照内核数排序的 Worker 为 Worker0、Worker1、Worker2，Application 还需要的内核数（coresLeft）等于 7。Worker0 的内核数为 4，Worker1 的内核数为 3，Worker2 的内核数为 2。那么此时需要分配的内核数 toAssign 等于 math.min（7，（4 + 3 + 2）），即 toAssign 为 7。此时的内核分配可以用图 7-3 表示。

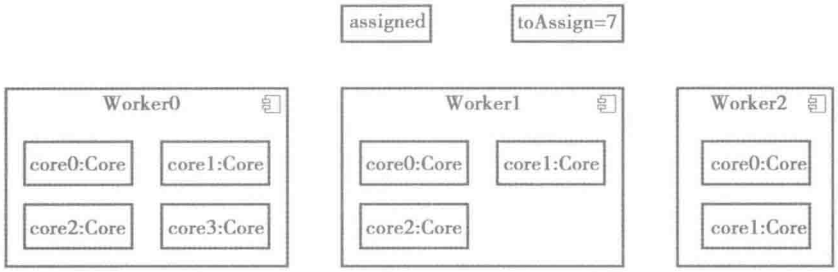


图 7-3 资源逻辑分配前

进行一轮内核分配后如图 7-4 所示。

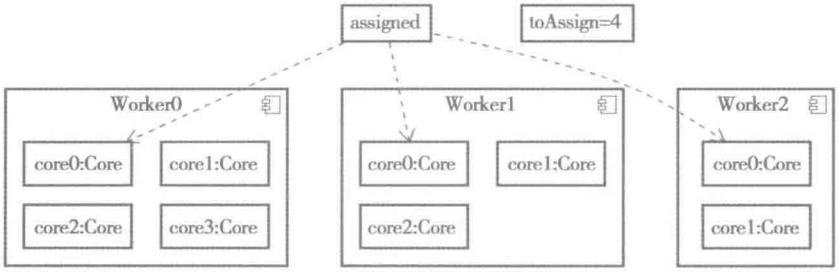


图 7-4 第一轮资源逻辑分配后

第二轮内核分配过后如图 7-5 所示。

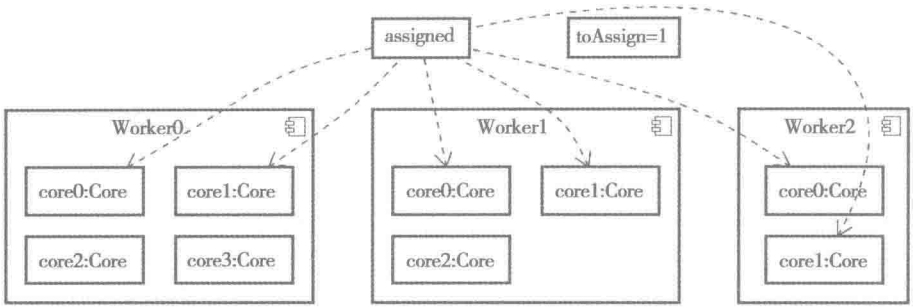


图 7-5 第二轮资源逻辑分配后

内核分配完毕时如图 7-6 所示。

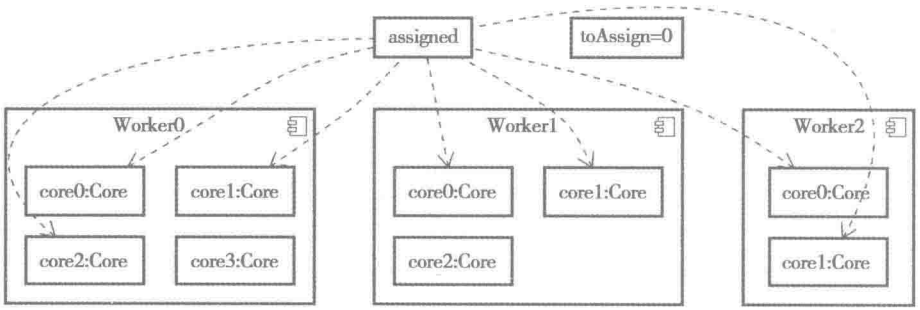


图 7-6 资源逻辑分配完毕

以上介绍的内容，只是逻辑上给 Application 分配各个 Worker 的内核及内存，下面一起看看计算资源真正的分配。

2. 计算资源物理分配

计算资源物理分配是指给 Application 物理分配 Worker 的内存以及核数。由于在逻辑分配的时候已经确定了每个 Worker 分配给 Application 的核数，并且这些 Worker 也都满足 Application 的内存需要，所以可以放心地进行物理分配了。物理分配的步骤如下：

步骤 1 首先使用 WorkerInfo、逻辑分配的 CPU 核数及内存大小创建 ExecutorInfo，然后将此 ExecutorInfo 添加到 ApplicationInfo 的 executors 缓存中，最后增加已经授权得到的内核数，实现如下。

```
def addExecutor(worker: WorkerInfo, cores: Int, useID: Option[Int] = None):
     ExecutorInfo = {
    val exec = new ExecutorInfo(newExecutorId(useID), this, worker, cores, desc.
         memoryPerSlave)
    executors(exec.id) = exec
    coresGranted += cores
    exec
}
```

步骤 2 物理分配通过调用 Master 的 launchExecutor 方法（见代码清单 7-29）来实现，其功能如下：

1）将 ExecutorInfo 添加到 WorkerInfo 的 executors 缓存中，并更新 Worker 已经使用的 CPU 核数和内存大小，见代码清单 7-30。

2）向 Worker 发送 LaunchExecutor 消息，运行 Executor。

3）向 ClientActor 发送 ExecutorAdded 消息，ClientActor 收到 ExecutorAdded 消息后，向 Master 发送 ExecutorStateChanged 消息，见代码清单 7-31。Master 收到 ExecutorStateChanged 消息（见代码清单 7-49）后将向 DriverActor 发送 ExecutorUpdated 消息，用于更新 Driver 上有关 Executor 的状态。

代码清单7-29 Master.launchExecutor的实现

```
def launchExecutor(worker: WorkerInfo, exec: ExecutorInfo) {
    logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
    worker.addExecutor(exec)
    worker.actor ! LaunchExecutor(masterUrl,
        exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory)
    exec.application.driver ! ExecutorAdded(
        exec.id, worker.id, worker.hostPort, exec.cores, exec.memory)
}
```

代码清单7-30 WorkerInfo.addExecutor的实现

```
def addExecutor(exec: ExecutorInfo) {
    executors(exec.fullId) = exec
    coresUsed += exec.cores
    memoryUsed += exec.memory
}
```

代码清单7-31 ClientActor收到ExecutorAdded消息的处理

```
case ExecutorAdded(id: Int, workerId: String, hostPort: String, cores: Int,
    memory: Int) =>
    val fullId = appId + "/" + id
    logInfo("Executor added: %s on %s (%s) with %d cores".format(fullId, workerId,
        hostPort,
          cores))
    master ! ExecutorStateChanged(appId, id, ExecutorState.RUNNING, None, None)
    listener.executorAdded(fullId, workerId, hostPort, cores, memory)
```

Worker 接到 LaunchExecutor 消息后的处理逻辑（见代码清单 7-32）如下：

1）创建 Executor 的工作目录。

2）创建 Application 的本地目录，当 Application 完成时，此目录会被删除。

3）创建并启动 ExecutorRunner。

4）向 Master 发送 ExecutorStateChanged 消息。

代码清单7-32 Worker接到LaunchExecutor消息后的处理

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

val appLocalDirs = appDirectories.get(appId).getOrElse {
    Utils.getOrCreateLocalRootDirs(conf).map { dir =>
        Utils.createDirectory(dir).getAbsolutePath()
    }.toSeq
}
appDirectories(appId) = appLocalDirs

val manager = new ExecutorRunner(appId, execId, appDesc, cores_, memory_,
        self, workerId, host, sparkHome, executorDir, akkaUrl, conf, appLocalDirs,
        ExecutorState.LOADING)
    executors(appId + "/" + execId) = manager
    manager.start()
    coresUsed += cores_
    memoryUsed += memory_
    master ! ExecutorStateChanged(appId, execId, manager.state, None, None)
```

启动 ExecutorRunner 的时候实际创建了线程 workerThread 和 shutdownHook，见代码清单 7-33。shutdownHook 用于在 Worker 关闭时杀掉所有的 Executor 进程。

代码清单7-33 启动ExecutorRunner

```
def start() {
    workerThread = new Thread("ExecutorRunner for " + fullId) {
        override def run() { fetchAndRunExecutor() }
    }
    workerThread.start()
    shutdownHook = new Thread() {
        override def run() {
            killProcess(Some("Worker shutting down"))
        }
    }
    Runtime.getRuntime.addShutdownHook(shutdownHook)
}
```

workerThread 执行过程中主要调用 fetchAndRunExecutor 方法（见代码清单 7-34），其执行步骤如下。

1）构造 ProcessBuilder，进程主类是 org.apache.spark.executor.CoarseGrainedExecutorBackend，有关 buildProcessBuilder 的实现，请参阅附录 F。

2）为 ProcessBuilder 设置执行目录、环境变量。

3）启动 ProcessBuilder，生成进程。

4）重定向进程的文件输出流与错误流为 executorDir 目录下的文件 stdout 与 stderr。以笔者本地为例，分别为：D:\git\spark\work\app-20150707144151-0000\0\stdout 和 D:\git\spark\work\

app-20150707144151-0000\0\stderr。

5）等待获取进程的退出状态，一旦收到退出状态，则向 Worker 发送 ExecutorStateChanged 消息。

代码清单7-34 ExecutorRunner的fetchAndRunExecutor方法

```
def fetchAndRunExecutor() {
    try {
        val builder = CommandUtils.buildProcessBuilder(appDesc.command, memory,
            sparkHome.getAbsolutePath, substituteVariables)
        val command = builder.command()
        logInfo("Launch command: " + command.mkString("\"", "\" \"", "\""))

        builder.directory(executorDir)
        builder.environment.put("SPARK_LOCAL_DIRS", appLocalDirs.mkString(","))
        builder.environment.put("SPARK_LAUNCH_WITH_SCALA", "0")
        process = builder.start()
        val header = "Spark Executor Command: %s\n%s\n\n".format(
            command.mkString("\"", "\" \"", "\""), "=" * 40)

        val stdout = new File(executorDir, "stdout")
        stdoutAppender = FileAppender(process.getInputStream, stdout, conf)

        val stderr = new File(executorDir, "stderr")
        Files.write(header, stderr, UTF_8)
        stderrAppender = FileAppender(process.getErrorStream, stderr, conf)

        val exitCode = process.waitFor()
        state = ExecutorState.EXITED
        val message = "Command exited with code " + exitCode
        worker ! ExecutorStateChanged(appId, execId, state, Some(message),
            Some(exitCode))
    } catch {
        case interrupted: InterruptedException => {
            logInfo("Runner thread for executor " + fullId + " interrupted")
            state = ExecutorState.KILLED
            killProcess(None)
        }
        case e: Exception => {
            logError("Error running executor", e)
            state = ExecutorState.FAILED
            killProcess(Some(e.toString))
            }
        }
    }
}
```

一旦执行了 builder.start()，我们利用 jdk 自带的工具 VisualVM 就会发现此时已经产生了一个新的进程，笔者对本地的 CoarseGrainedExecutorBackend 进程的截图如图 7-7 所示。

图 7-7 CoarseGrainedExecutorBackend 进程的截图

它的系统属性如图 7-8 所示。

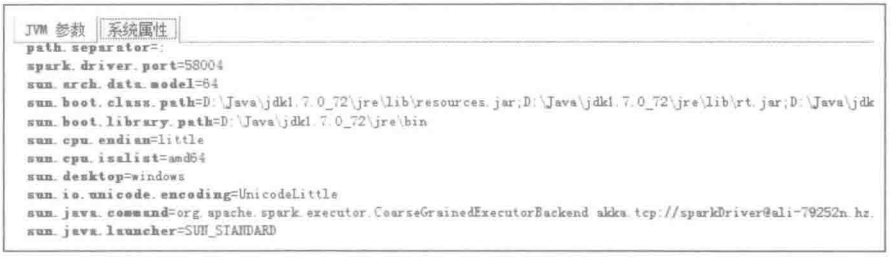


图 7-8 CoarseGrainedExecutorBackend 进程的系统属性

我们看到系统属性 sun.java.command 正是 CommandUtils.buildProcessBuilder 构建的 command，由于截图无法完整显示，笔者给出文本形式的属性值：sun.java.command = org.apache.spark.executor.CoarseGrainedExecutorBackend akka.tcp://sparkDriver@ 主机名 :58004/user/CoarseGrainedScheduler 0 主机名 1 app-20150707144151-0000 akka.tcp://sparkWorker1@ 主机名 :58037/user/Worker。

现在我们来看看 CoarseGrainedExecutorBackend 的 main 方法，见代码清单 7-35。其中调用的 run 方法（见代码清单 7-36）的处理过程如下：

1）给 Driver 发送 RetrieveSparkProps 消息获取 Spark 属性。

2）使用获取的 Spark 属性创建自身需要的 ActorSystem。

3）注册 CoarseGrainedExecutorBackend 到 ActorSystem 中。

4）注册 WorkerWatcher 到 ActorSystem 中。WorkerWatcher 用于 Worker 向 Master 发送心跳以证明 Worker 运行正常。

代码清单7-35 CoarseGrainedExecutorBackend的main方法

```
def main(args: Array[String]) {
    args.length match {
        case x if x < 5 =>
```

```
            System.err.println(
                "Usage: CoarseGrainedExecutorBackend <driverUrl> <executorId>
                   <hostname> " +
                "<cores> <appid> [<workerUrl>] ")
            System.exit(1)

        case 5 =>
            run(args(0), args(1), args(2), args(3).toInt, args(4), None)
        case x if x > 5 =>
            run(args(0), args(1), args(2), args(3).toInt, args(4), Some(args(5)))
    }
}
```

注意 此处的 CoarseGrainedSchedulerBackend 是 org.apache.spark.executor 包下的 CoarseGrainedSchedulerBackend。

代码清单7-36 CoarseGrainedExecutorBackend的run方法

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

        val executorConf = new SparkConf
        val port = executorConf.getInt("spark.executor.port", 0)
        val (fetcher, _) = AkkaUtils.createActorSystem(
            "driverPropsFetcher", hostname, port, executorConf, new Security-
               Manager(executorConf))
        val driver = fetcher.actorSelection(driverUrl)
        val timeout = AkkaUtils.askTimeout(executorConf)
        val fut = Patterns.ask(driver, RetrieveSparkProps, timeout)
        val props = Await.result(fut, timeout).asInstanceOf[Seq[(String, String)]] ++
            Seq[(String, String)](("spark.app.id", appId))
        fetcher.shutdown()

        val driverConf = new SparkConf().setAll(props)
        val (actorSystem, boundPort) = AkkaUtils.createActorSystem(
            SparkEnv.executorActorSystemName,
            hostname, port, driverConf, new SecurityManager(driverConf))
        val sparkHostPort = hostname + ":" + boundPort
        actorSystem.actorOf(
            Props(classOf[CoarseGrainedExecutorBackend],
                driverUrl, executorId, sparkHostPort, cores, props, actorSystem),
            name = "Executor")
        workerUrl.foreach { url =>
            actorSystem.actorOf(Props(classOf[WorkerWatcher], url), name =
                "WorkerWatcher")
        }
```

```
            actorSystem.awaitTermination()
        }
    }
```

注册 CoarseGrainedExecutorBackend 会触发 preStart 方法，其实现如下：

```
override def preStart() {
    logInfo("Connecting to driver: " + driverUrl)
    driver = context.actorSelection(driverUrl)
    driver ! RegisterExecutor(executorId, hostPort, cores)
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
}
```

preStart 方法主要向 DriverActor 发送 RegisterExecutor 消息，DriverActor 接到 RegisterExecutor 消息后的处理步骤（见代码清单 7-37）如下：

1）先向 CoarseGrainedExecutorBackend 发送 RegisteredExecutor 消息，CoarseGrainedExecutorBackend 收到 RegisteredExecutor 消息后创建 Executor，见代码清单 7-38。至此，Executor 终于出现了！

2）更新 Executor 所在地址与 Executor 的映射关系（addressToExecutorId）、Driver 获取的总共 CPU 核数（totalCoreCount）、注册到 Driver 的 Executor 的总数（totalRegisteredExecutors）等信息。

3）创建 ExecutorData 并且注册到 executorDataMap 中。

4）调用 makeOffers 方法执行任务，将会在 7.2.5 节详细介绍。

代码清单7-37 DriverActor接到RegisterExecutor消息后的处理

```
case RegisterExecutor(executorId, hostPort, cores) =>
    Utils.checkHostPort(hostPort, "Host port expected " + hostPort)
    if (executorDataMap.contains(executorId)) {
        sender ! RegisterExecutorFailed("Duplicate executor ID: " + executorId)
    } else {
        logInfo("Registered executor: " + sender + " with ID " + executorId)
        sender ! RegisteredExecutor

        addressToExecutorId(sender.path.address) = executorId
        totalCoreCount.addAndGet(cores)
        totalRegisteredExecutors.addAndGet(1)
        val (host, _) = Utils.parseHostPort(hostPort)
        val data = new ExecutorData(sender, sender.path.address, host, cores, cores)
        CoarseGrainedSchedulerBackend.this.synchronized {
            executorDataMap.put(executorId, data)
            if (numPendingExecutors > 0) {
                numPendingExecutors -= 1
                logDebug(s"Decremented number of pending executors ($numPending-
                    Executors left)")
            }
        }
        makeOffers()
    }
```

代码清单7-38 CoarseGrainedExecutorBackend处理RegisteredExecutor消息的代码

```
case RegisteredExecutor =>
    logInfo("Successfully registered with driver")
    val (hostname, _) = Utils.parseHostPort(hostPort)
    executor = new Executor(executorId, hostname, sparkProperties, cores, isLocal
        = false,
        actorSystem)
```

注册 WorkerWatcher 的时候会触发 preStart 方法（见代码清单 7-39），它会向 Worker 发送 SendHeartbeat 消息初始化连接。

代码清单7-39 WorkerWatcher的preStart方法

```
override def preStart() {
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])

    logInfo(s"Connecting to worker $workerUrl")
    val worker = context.actorSelection(workerUrl)
    worker ! SendHeartbeat // need to send a message here to initiate connection
}
```

Worker 收到 SendHeartbeat 消息后，会向 Master 发送 Heartbeat 消息，代码如下。

```
case SendHeartbeat =>
    if (connected) { master ! Heartbeat(workerId) }
```

根据代码清单 7-17 中 Master 收到 Heartbeat 消息后的实现。如果 Worker 的 id 与 Worker 的映射关系（idToWorker）中找不到匹配的 Worker，但是 Worker 的缓存（workers）中却存在此 id，那么向 Worker 发送 ReconnectWorker 消息。Worker 接收到 ReconnectWorker 消息后将重新向 Master 注册，代码如下。

```
case ReconnectWorker(masterUrl) =>
    logInfo(s"Master with url $masterUrl requested this worker to reconnect.")
    registerWithMaster()
```

在只有一个 Worker 的情况下，AppClient 的计算资源物理分配过程如图 7-9 所示。

这里对图 7-9 中计算资源的物理分配过程进行简短介绍：

1）调用 Master 的 launchExecutor 方法时，向 Worker 发送 LaunchExecutor 消息。

2）Worker 接到 LaunchExecutor 消息后，创建 Executor 的工作目录，创建 Application 的本地目录，创建并启动 ExecutorRunner，最后向 Master 发送 ExecutorStateChanged 消息。

3）ExecutorRunner 创建并运行线程 workerThread，workerThread 在执行过程中调用 fetch-AndRunExecutor 完成对 CoarseGrainedExecutorBackend 进程进行构造。

4）CoarseGrainedExecutorBackend 进程向 Driver 发送 RetrieveSparkProps 消息。

5）Driver 收到 RetrieveSparkProps 消息后向 CoarseGrainedExecutorBackend 进程发送 Spark 属性，CoarseGrainedExecutorBackend 进程最后创建自身需要的 ActorSystem。

6）CoarseGrainedExecutorBackend 进程向刚刚启动的 ActorSystem 注册 CoarseGrained-

ExecutorBackend（实现了 Actor 特质），所以触发其 preStart 方法。CoarseGrainedExecutorBackend 的 preStart 方法向 DriverActor 发送 RegisterExecutor 消息。

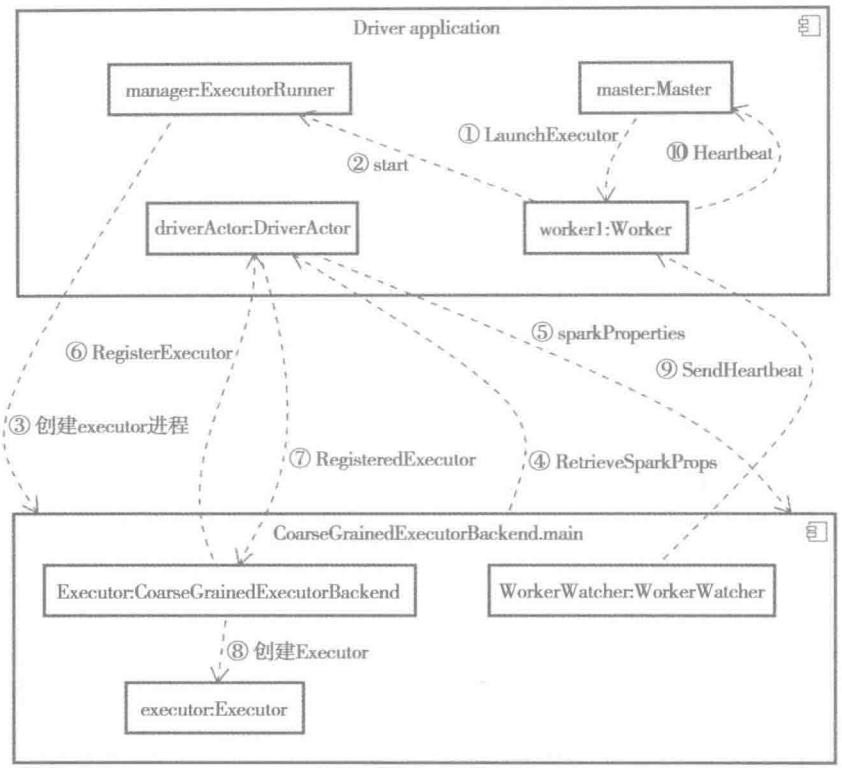


图 7-9 AppClient 的计算资源物理分配过程

7）DriverActor 接到 RegisterExecutor 消息后，先向 CoarseGrainedExecutorBackend 发送 RegisteredExecutor 消息，然后更新 Executor 所在地址与 Executor 的映射关系（addressTo-ExecutorId）、Driver 获取的总共 CPU 核数（totalCoreCount）、注册到 Driver 的 Executor 的总数（totalRegisteredExecutors）等信息，最后创建 ExecutorData 并且注册到 executorDataMap 中。

8）CoarseGrainedExecutorBackend 进程收到 RegisteredExecutor 消息后创建 Executor。

9）CoarseGrainedExecutorBackend 进程向刚刚启动的 ActorSystem 注册 WorkerWatcher，注册 WorkerWatcher 的时候会触发 preStart 方法，preStart 方法会向 Worker 发送 SendHeartbeat 消息初始化连接。

10）Worker 收到 SendHeartbeat 消息后会向 Master 发送 Heartbeat 消息，Master 收到 Heartbeat 消息后如果发现 Worker 没有注册过，则向 Worker 发送 ReconnectWorker 消息，要求 Worker 重新向 Master 注册。

7.2.5 local-cluster 模式的任务执行

在 5.4.5 节，我们知道 TaskSchedulerImpl 的 submitTasks 方法最后会调用 backend 的 reviveOffers 方法，见代码清单 5-40。在 local-cluster 模式下即为 SparkDeploySchedulerBackend 的父类 CoarseGrainedSchedulerBackend 的 reviveOffers 方法，它的实现如下。

```
override def reviveOffers() {
    driverActor ! ReviveOffers
}
```

reviveOffers 方法用于向 DriverActor 发送 ReviveOffers 消息。DriverActor 接收到 ReviveOffers 消息后调用 makeOffers，代码如下。

```
case ReviveOffers =>
    makeOffers()
```

makeOffers 方法（见代码清单 7-40）的处理逻辑如下：

1）将 executorDataMap 中的 ExecutorData 都转换为 WorkerOffer。

2）调用 TaskSchedulerImpl 的 resourceOffers 方法（在 5.4.5 节已经详细介绍过）给当前任务分配 Executor，然后调用 launchTasks。

代码清单7-40 CoarseGrainedSchedulerBackend的makeOffers方法

```
def makeOffers() {
    launchTasks(scheduler.resourceOffers(executorDataMap.map { case (id,
        executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
    }.toSeq))
}
```

launchTasks 方法（见代码清单 7-41）的处理步骤如下：

1）序列化 TaskDescription。

2）取出 TaskDescription 所描述任务分配的 ExecutorData 信息，并且将 ExecutorData 描述的空闲 CPU 核数减去任务占用的核数。

3）向 Executor 所在的 CoarseGrainedExecutorBackend 进程中的 CoarseGrainedExecutorBackend 发送 LaunchTask 消息。

代码清单7-41 CoarseGrainedSchedulerBackend的launchTasks方法

```
def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
    for (task <- tasks.flatten) {
        val ser = SparkEnv.get.closureSerializer.newInstance()
        val serializedTask = ser.serialize(task)
        if (serializedTask.limit >= akkaFrameSize - AkkaUtils.reservedSizeBytes) {
            val taskSetId = scheduler.taskIdToTaskSetId(task.taskId)
            scheduler.activeTaskSets.get(taskSetId).foreach { taskSet =>
                try {
                    var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +
                    "spark.akka.frameSize (%d bytes) - reserved (%d bytes). Consider
```

```
                        increasing " +
                         "spark.akka.frameSize or using broadcast variables for large values."
                    msg = msg.format(task.taskId, task.index, serializedTask.
                         limit, akkaFrameSize,
                        AkkaUtils.reservedSizeBytes)
                    taskSet.abort(msg)
                } catch {
                    case e: Exception => logError("Exception in error callback", e)
                }
            }
        }
        else {
            val executorData = executorDataMap(task.executorId)
            executorData.freeCores -= scheduler.CPUS_PER_TASK
            executorData.executorActor ! LaunchTask(new SerializableBuffer(serializ
                edTask))
        }
    }
}
```

CoarseGrainedExecutorBackend 收到 LaunchTask 消息后，反序列化 TaskDescription，使用 TaskDescription 的 taskId、name、serializedTask 调用 Executor 的方法 launchTask，实现如下。

```
case LaunchTask(data) =>
    if (executor == null) {
        logError("Received LaunchTask command but executor was null")
        System.exit(1)
    } else {
        val ser = SparkEnv.get.closureSerializer.newInstance()
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        executor.launchTask(this, taskDesc.taskId, taskDesc.name, taskDesc.
            serializedTask)
    }
```

Executor 的方法 launchTask 已在 5.5 节详细描述。

现在我们来总结下 local-cluster 模式的任务执行过程，笔者本地设置的 partition 数量为 2，会另起两个 CoarseGrainedExecutorBackend 进程，两个 ShuffleMapTask 分别被分配到两个进程中执行，其执行过程如图 7-10 所示。

图 7-10 中 local-cluster 模式的任务执行过程与 local 模式类似，区别是 local-cluster 模式的每个 Worker 会启动多个 CoarseGrainedExecutorBackend 进程，ExecutorBackend（此模式下为 CoarseGrainedExecutorBackend）和 Executor 都位于此 CoarseGrainedExecutorBackend 的 JVM 进程中。local-cluster 模式的任务执行过程可以参考对图 7-1 中 local 模式的任务执行过程分析。

ResultTask 的执行过程类似，不再赘述。

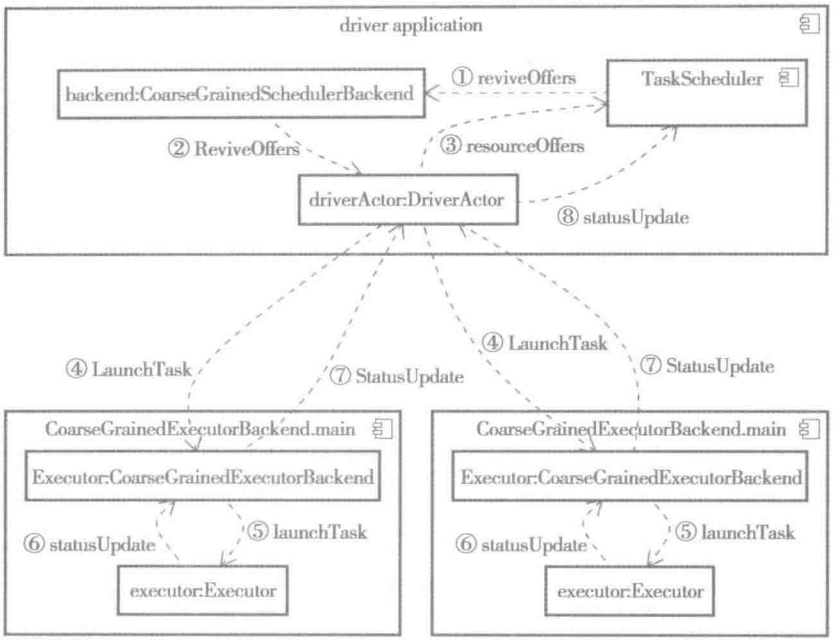


图 7-10 local-cluster 模式的任务执行过程

7.3 Standalone 部署模式

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

- Driver 在集群之外，可以是任意的客户端应用程序。
- Master 部署于单独的进程，甚至应该在单独的机器节点上。Master 有多个，但同时最多只有一个处于激活状态。
- Worker 部署于单独的进程，也推荐在单独的机器节点上部署。

7.3.1 启动 Standalone 模式

我们以 Linux 环境下启动 Standalone 模式为例。启动 Standalone 模式需要保证一定的顺序，即需要先启动 Master，再逐个启动 Worker。Linux 环境启动 Master 的命令如图 7-11 所示。

```
[          @v218142118 ~/install/spark-1.2.0-bin-hadoop1/sbin]$ sh start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/jiaan.gja/install/spark-1.2.0-
        -org.apache.spark.deploy.master.Master-1-v218142118.sqa.zmf.out
```

图 7-11 Linux 环境启动 Master 的命令

Master 进程的默认端口是 7077，webui 的端口是 8080。可以用 jps 命令查看 Master 进程的信息，输出如图 7-12 所示。

```
[jiaan.gja@v218142118 ~/install/spark-1.2.0-bin-hadoop1/sbin]$ jps -lmv
25691 sun.tools.jps.Jps -lmv -Dapplication.home=/opt/taobao/install/jdk-1.7.0_51 -Xms8m
23137 org.apache.spark.deploy.master.Master --ip 127.0.0.1 --port 7077 --webui-port 8080 -XX:MaxPer
fecycleEvents=true -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=10207 -Dcom.s
ntication=false -Dcom.sun.management.jmxremote.ssl=false -Xms512m -Xmx512m
```

图 7-12 jps 命令查看 Master 进程的信息

用浏览器访问 http://10.218.142.118:8080/，可以看到其 web 界面如图 7-13 所示。

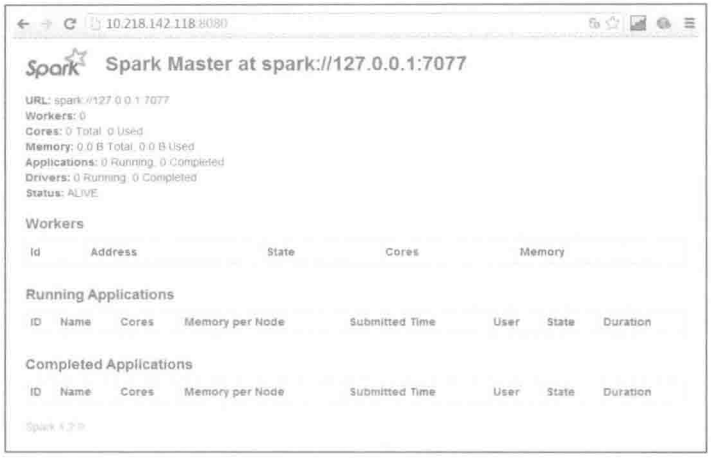


图 7-13 Master 的 Web 界面

启动完 Master，就可以启动 Worker 了，启动 Worker 时需要指定 Master 的连接地址。在命令行输入：./spark-class org.apache.spark.deploy.worker.Worker spark://127.0.0.1:7077，启动过程如图 7-14 所示。

```
[ @v218142118 ~/install/spark-1.2.0-bin-hadoop1/bin]$ ./spark-class org.apache.spark.deploy.worker.worker spa
0.1:7077
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/07/15 12:14:54 INFO Worker: Registered signal handlers for [TERM, HUP, INT]
15/07/15 12:14:54 INFO SecurityManager: Changing view acls to:
15/07/15 12:14:54 INFO SecurityManager: Changing modify acls to:
15/07/15 12:14:54 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view p
 set( ); users with modify permissions: Set( )
15/07/15 12:14:55 INFO Slf4jLogger: Slf4jLogger started
15/07/15 12:14:55 INFO Remoting: Starting remoting
15/07/15 12:14:55 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkworker@localhost:53728]
15/07/15 12:14:55 INFO Remoting: Remoting now listens on addresses: [akka.tcp://sparkworker@localhost:53728]
15/07/15 12:14:55 INFO Utils: Successfully started service 'sparkworker' on port 53728.
15/07/15 12:14:55 INFO Worker: Starting Spark worker localhost:53728 with 2 cores, 2.9 GB RAM
15/07/15 12:14:55 INFO Worker: Spark home: /home/ /install/spark-1.2.0-bin-hadoop1
15/07/15 12:14:55 INFO Utils: Successfully started service 'WorkerUI' on port 8081.
15/07/15 12:14:55 INFO WorkerWebUI: Started WorkerWebUI at http://localhost:8081
15/07/15 12:14:55 INFO Worker: Connecting to master spark://127.0.0.1:7077...
15/07/15 12:14:56 INFO Worker: Successfully registered with master spark://127.0.0.1:7077
```

图 7-14 启动 Worker

从启动信息中可以看出，Worker 创建并启动了自己的 ActorSystem：akka.tcp://sparkWorker@localhost:53728，WorkerUI 的端口为 8081，最后向 Master 注册 Worker 等信息。

用 jps 命令查看 Worker 进程的信息，看到 spark-class 默认会在启动 Worker 进程时，增加一些参数：-XX:MaxPermSize=128m -Dspark.akka.logLifecycleEvents=true -Xms512m -Xmx512m。

如果有很多 Worker，一台一台启动会很麻烦，可以使用 $SPARK_HOME/sbin/start-slaves.sh 来启动多个 Worker。start-slaves.sh 的部分脚本内容如下。

```
# Launch the slaves
if [ "$SPARK_WORKER_INSTANCES" = "" ]; then
    exec "$sbin/slaves.sh" cd "$SPARK_HOME" \; "$sbin/start-slave.sh" 1 "spark://$SPARK_
        MASTER_IP:$SPARK_MASTER_PORT"
else
    if [ "$SPARK_WORKER_WEBUI_PORT" = "" ]; then
        SPARK_WORKER_WEBUI_PORT=8081
    fi
    for ((i=0; i<$SPARK_WORKER_INSTANCES; i++)); do
        "$sbin/slaves.sh" cd "$SPARK_HOME" \; "$sbin/start-slave.sh" $(( $i + 1 ))
          "spark://$SPARK_MASTER_IP:$SPARK_MASTER_PORT" --webui-port $(( $SPARK_
          WORKER_WEBUI_PORT + $i ))
    done
fi
```

7.3.2 启动 Master 分析

Scala 语法允许在 object 中定义 def main（argStrings：Array[String]）函数作为应用的启动入口，可以认为与 Java 中 main 方法一样。在 Master.scala 文件中定义了 object Master，见代码清单 7-42。main 函数完成的内容如下：

1）创建 SparkConf。

2）Master 参数解析。

3）创建、启动 ActorSystem，并向 ActorSystem 注册 Master。

代码清单7-42 Master的main方法

```
private[spark] object Master extends Logging {
    val systemName = "sparkMaster"
    private val actorName = "Master"
    val sparkUrlRegex = "spark://([^:]+):([0-9]+)".r

    def main(argStrings: Array[String]) {
        SignalLogger.register(log)
        val conf = new SparkConf
        val args = new MasterArguments(argStrings, conf)
        val (actorSystem, _, _) = startSystemAndActor(args.host, args.port, args.
             webUiPort, conf)
        actorSystem.awaitTermination()
    }
}
```

1. Master 参数解析

MasterArguments 用于解析系统环境变量和启动 Master 时指定的命令行参数，见代码清单 7-43。

代码清单7-43 MasterArguments解析Master参数

```
private[spark] class MasterArguments(args: Array[String], conf: SparkConf) {
    var host = Utils.localHostName()
    var port = 7077
    var webUiPort = 8080
    var propertiesFile: String = null

    if (System.getenv("SPARK_MASTER_HOST") != null) {
        host = System.getenv("SPARK_MASTER_HOST")
    }
    if (System.getenv("SPARK_MASTER_PORT") != null) {
        port = System.getenv("SPARK_MASTER_PORT").toInt
    }
    if (System.getenv("SPARK_MASTER_WEBUI_PORT") != null) {
        webUiPort = System.getenv("SPARK_MASTER_WEBUI_PORT").toInt
    }

    parse(args.toList)

    propertiesFile = Utils.loadDefaultSparkProperties(conf, propertiesFile)

    if (conf.contains("spark.master.ui.port")) {
        webUiPort = conf.get("spark.master.ui.port").toInt
    }
```

MasterArguments 中解析的参数如表 7-1 所示。

表 7-1 MasterArguments 中解析的参数

属 性	含 义	默 认 值	系统环境变量	命令行参数
host	Master 的监听地址	机器名	SPARK_MASTER_HOST	--host 或者 -h
port	Master 的监听端口	7077	SPARK_MASTER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8080	SPARK_MASTER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf		--properties-file

系统环境变量 SPARK_MASTER_WEBUI_PORT 可以被属性 spark.master.ui.port 所覆盖。

解析命令行参数，是用其 parse 函数（见代码清单 7-44）实现的，命令行参数的值会覆盖系统环境变量的值。解析的命令行参数如表 7-2 所示。

表 7-2 parse 函数解析的命令行参数

参 数 名	参 数 含 义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件
--help	帮助

代码清单7-44 MasterArguments.parse的实现

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

    case ("--host" | "-h") :: value :: tail =>
        Utils.checkHost(value, "Please use hostname " + value)
        host = value
        parse(tail)

    case ("--port" | "-p") :: IntParam(value) :: tail =>
        port = value
        parse(tail)

    case "--webui-port" :: IntParam(value) :: tail =>
        webUiPort = value
        parse(tail)

    case ("--properties-file") :: value :: tail =>
        propertiesFile = value
        parse(tail)

    case ("--help") :: tail =>
        printUsageAndExit(0)

    case Nil => {}

    case _ =>
        printUsageAndExit(1)
}
```

系统变量中如果指定了 spark.master.ui.port，会覆盖环境变量 SPARK_MASTER_WEBUI_PORT 和命令行参数 --webui-port 指定的值。

2. 创建、启动 ActorSystem，并向 ActorSystem 注册 Master

Master 的 startSystemAndActor 方法已在 7.2.1 节详细介绍，不再赘述。

7.3.3 启动 Worker 分析

Worker 也通过 main 函数来启动，见代码清单 7-45。main 函数完成的内容如下：

1）创建 SparkConf。

2）Worker 参数解析。

3）创建、启动 ActorSystem，并向 ActorSystem 注册 Worker。

代码清单7-45 Worker的main方法

```
private[spark] object Worker extends Logging {
    def main(argStrings: Array[String]) {
```

```
        SignalLogger.register(log)
        val conf = new SparkConf
        val args = new WorkerArguments(argStrings, conf)
        val (actorSystem, _) = startSystemAndActor(args.host, args.port, args.
            webUiPort, args.cores,
            args.memory, args.masters, args.workDir)
        actorSystem.awaitTermination()
    }
}
```

1. Worker 参数解析

WorkerArguments 用于解析系统环境变量和启动 Worker 时指定的命令行参数，其实现方式与 MasterArguments 大同小异，读者可以自己阅读其源码。WorkerArguments 相比于 MasterArguments，增加了内核数量（cores）、内存大小（memory）、Master 的监听地址列表（masters）、工作目录（workDir）的解析。WorkerArguments 解析的参数如表 7-3 所示。

表 7-3 WorkerArguments 解析的参数

属 性	含 义	默 认 值	系统环境变量	命令行参数
host	Worker 的监听地址	机器名		--host 或者 -h
port	Worker 的监听端口	0	SPARK_WORKER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8081	SPARK_WORKER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf		--properties-file
cores	Worker 的内核数	系统内核数	SPARK_WORKER_CORES	--cores 或者 -c
memory	Worker 的内存大小	系统最大内存减 1 GB	SPARK_WORKER_MEMORY	--memory 或者 -m
masters	Master 地址列表	null		
workDir	Worker 的工作目录	null	SPARK_WORKER_DIR	--work-dir 或者 -d

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

WorkerArguments 的 parse 函数与 MasterArguments 中的 parse 函数的实现类似，读者可自行阅读其源码。WorkerArguments 的 parse 函数要求必须在启动 Worker 时指定 spark:// 的参数作为 master 的 url。命令行参数的值会覆盖系统环境变量的值，解析的命令行参数如表 7-4 所示。

表 7-4 WorkerArguments 的 parse 函数解析的命令行参数

参 数 名	参 数 含 义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件

（续）

参 数 名	参数含义
--cores 或者 -c	指定内核数
--memory 或者 -m	指定内存大小
--work-dir 或者 -d	指定工作目录
--help	帮助


注意 系统变量中如果指定了 spark.worker.ui.port，会覆盖环境变量 SPARK_WORKER_WEBUI_PORT 和命令行参数 --webui-port 指定的值。

2. 创建、启动 ActorSystem，并向 ActorSystem 注册 Worker

Worker 的 startSystemAndActor 方法已在 7.2.1 节详细介绍，不再赘述。

7.3.4 启动 Driver Application 分析

我们首先自己动手在 Spark 项目中增加一个小应用程序 MasterTest，其实现非常简单，我们将使用它来扮演 Driver application 的角色，连接地址为 spark://10.218.142.118:7077 的 Master，代码如下。

```
object MasterTest {

    def main(args: Array[String]) {
        val sparkConf = new SparkConf().setAppName("Master Test")
        .setMaster("spark://10.218.142.118:7077");
        val sc = new SparkContext(sparkConf)
    }

}
```

由于我们设置 master 为 spark://10.218.142.118:7077，使它匹配 spark://(.*) 模式，见代码清单 7-46。

代码清单7-46 SparkContext匹配spark://(.*)模式的代码

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

有关 SparkDeploySchedulerBackend 的内容，已在 7.2 节详细讲解，此处不再赘述。

我们以 2 个 Worker、1 个 Master、1 个 Driver application 的情况，展示整个 Standalone 部署模式的启动过程，如图 7-15 所示。

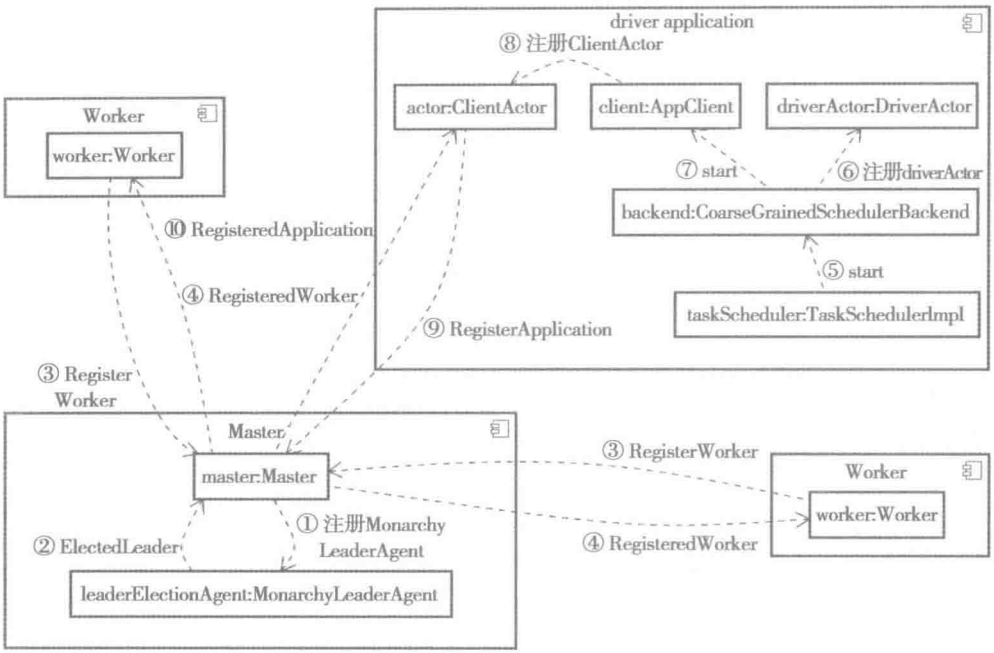


图 7-15 Standalone 模式的启动过程

图 7-15 中展示的 Standalone 模式的启动过程与 local-cluster 模式的启动过程十分相似，读者可以参考图 7-2 中对 local-cluster 模式的启动过程的介绍。不过两种模式的启动过程依然有以下区别：

1）集群是真正的分布式部署，所有 Master 和 Worker 都位于独立的 JVM 进程甚至是不同的机器节点上；

2）Standalone 模式下可以存在多个 Master，这些 Master 之间通过持久化引擎和领导选举机制解决生成环境下 Master 的单点问题，使得 Master 在异常退出后，能够重新选举激活状态的 Master，并从故障中恢复集群。

以 1 个 Worker、1 个 Master、1 个 Driver application 的情况，展示整个 Standalone 部署模式的资源调度，如图 7-16 所示。

图 7-16 中展示的 Standalone 模式的计算资源物理分配过程与 local-cluster 模式的计算资源物理分配过程十分相似，读者可以参考图 7-9 中对 local-cluster 模式的计算资源物理分配过程的介绍。不过两种模式的计算资源物理分配过程依然有以下区别：

1）集群是真正的分布式部署，所有 Master 和 Worker 都位于独立的 JVM 进程甚至是不同的机器节点上；

2）Standalone 模式下可以存在多个 Master，这些 Master 之间通过持久化引擎和领导选举机制解决生成环境下 Master 的单点问题，使得 Master 在异常退出后，能够重新选举激活状态的 Master，并从故障中恢复集群。

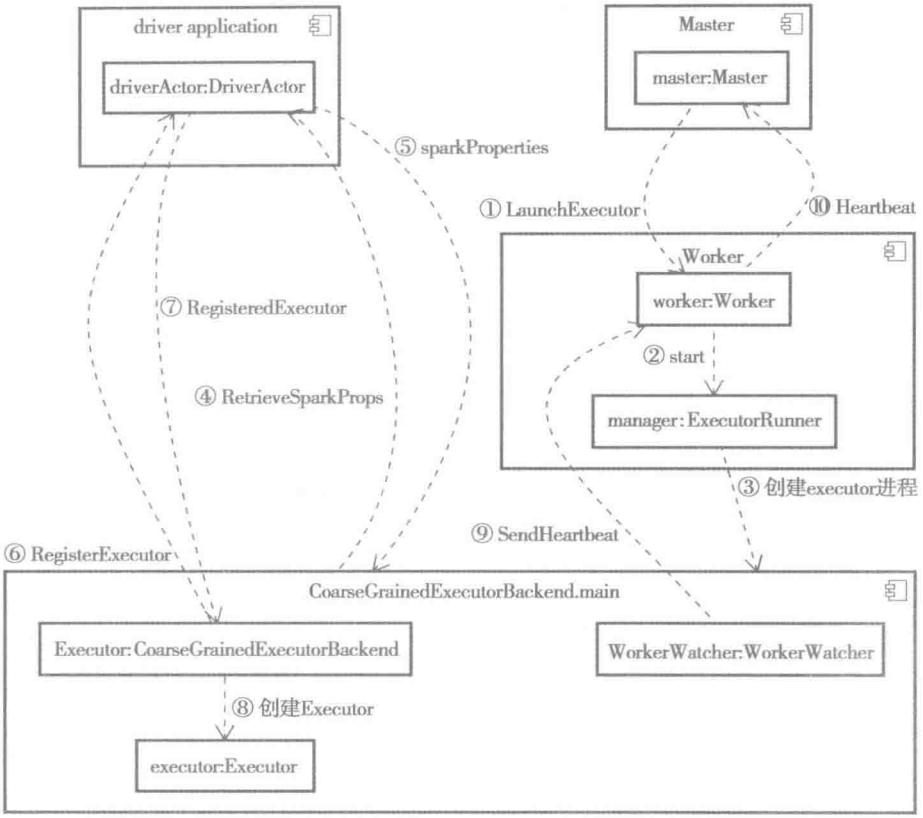


图 7-16 Standalone 模式的计算资源物理分配过程

7.3.5 Standalone 模式的任务执行

以 2 个 Worker、1 个 Master、1 个 Driver application 的情况，展示整个 Standalone 部署模式的任务执行，如图 7-17 所示。

7.3.6 资源回收

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

Spark 中有两种处理方式：一种是离别之前先打声招呼，一种是不辞而别。

1. 离别之前先打声招呼

这种方式很好理解，SparkContext 也提供了 stop 方法（见代码清单 7-47）用于告别，告别就需要停止各种服务和回收资源。

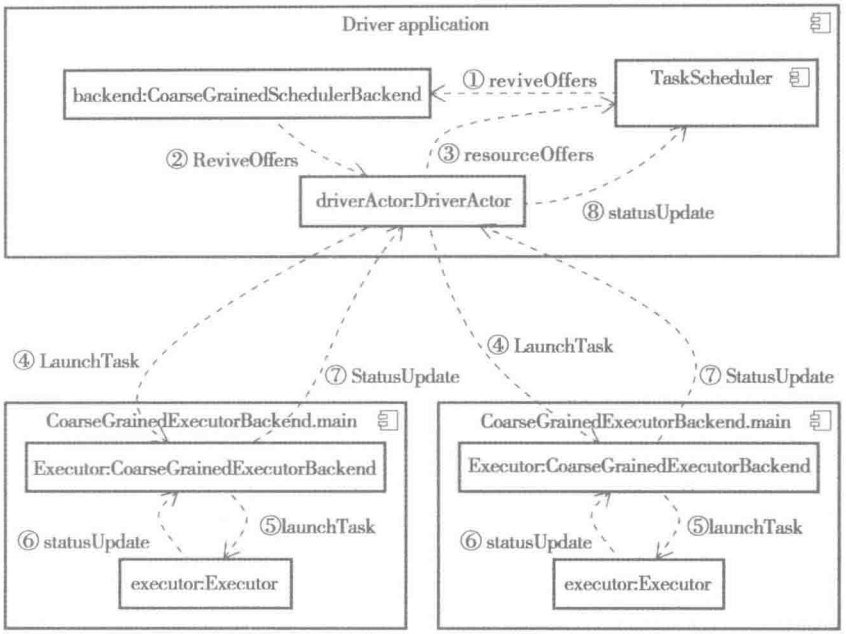


图 7-17 Standalone 模式的任务执行过程

代码清单7-47 SparkContext的stop方法

```
def stop() {
    SparkContext.SPARK_CONTEXT_CONSTRUCTOR_LOCK.synchronized {
        if (!stopped) {
            stopped = true
            postApplicationEnd()
            ui.foreach(_.stop())
            env.metricsSystem.report()
            metadataCleaner.cancel()
            env.actorSystem.stop(heartbeatReceiver)
            cleaner.foreach(_.stop())
            dagScheduler.stop()
            dagScheduler = null
            taskScheduler = null
            env.stop()
            SparkEnv.set(null)
            listenerBus.stop()
            eventLogger.foreach(_.stop())
            logInfo("Successfully stopped SparkContext")
            SparkContext.clearActiveContext()
        } else {
            logInfo("SparkContext already stopped")
        }
    }
}
```

我们主要关心计算资源的回收，所以看看 DagScheduler 的 stop 方法，其代码如下。

```
def stop() {
    logInfo("Stopping DAGScheduler")
    dagSchedulerActorSupervisor ! PoisonPill
    taskScheduler.stop()
}
```

TaskSchedulerImpl 的 stop 方法的实现如下。

```
override def stop() {
    if (backend != null) {
        backend.stop()
    }
    if (taskResultGetter != null) {
        taskResultGetter.stop()
    }
    starvationTimer.cancel()
}
```

CoarseGrainedSchedulerBackend 的 stop 方法中调用了 stopExecutors 方法，实现如下。

```
def stopExecutors() {
    try {
        if (driverActor != null) {
            val future = driverActor.ask(StopExecutors)(timeout)
    //省略部分代码
}
override def stop() {
    stopExecutors()
    //省略部分代码
}
```

stopExecutors 方法向 DriverActor 发送了 StopExecutors 消息，DriverActor 收到 StopExecutors 消息后的处理逻辑如下。

```
case StopExecutors =>
    logInfo("Asking each executor to shut down")
    for ((_, executorData) <- executorDataMap) {
        executorData.executorActor ! StopExecutor
    }
    sender ! true
```

上面的代码遍历 executorDataMap 中的 ExecutorData，向每个 CoarseGrainedExecutorBackend 进程发送 StopExecutor 消息。CoarseGrainedExecutorBackend 进程收到 StopExecutor 消息后的资源回收处理代码如下。

```
case StopExecutor =>
    logInfo("Driver commanded a shutdown")
    executor.stop()
    context.stop(self)
    context.system.shutdown()
```

处理 StopExecutor 消息时调用了 Executor 的 stop 方法关闭线程、停止 SparkEnv 等，代码实现如下。

```
def stop() {
    env.metricsSystem.report()
    env.actorSystem.stop(executorActor)
    isStopped = true
    threadPool.shutdown()
    if (!isLocal) {
        env.stop()
    }
}
```

2. 不辞而别

哦，不！上面的分析发现 Application 只记得跟 Executor 打声招呼，却忘记了 Master。这该怎么办？

Akka 的通信机制保证当相互通信的任意一方异常退出，另一方都会收到 DisassociatedEvent。Master 正是在处理 DisassociatedEvent 消息时移除已经停止的 Driver Application，见代码清单 7-48。

代码清单7-48 Master处理DisassociatedEvent消息的代码

```
case DisassociatedEvent(_, address, _) => {
    // The disconnected client could've been either a worker or an app; remove
        whichever it was
    logInfo(s"$address got disassociated, removing it.")
    addressToWorker.get(address).foreach(removeWorker)
    addressToApp.get(address).foreach(finishApplication)
    if (state == RecoveryState.RECOVERING && canCompleteRecovery) {
        completeRecovery() }
}
```

如果编写任务时忘记了调用 SparkContext 的 stop 方法，Executor 的资源虽然不会被主动回收，但由于 CoarseGrainedExecutorBackend 也会收到 DisassociatedEvent 消息，直接退出 CoarseGrainedExecutorBackend 进程，代码如下。

```
case x: DisassociatedEvent =>
    logError(s"Driver $x disassociated! Shutting down.")
    System.exit(1)
```

7.4 容错机制

在分布式系统中，由于机器数量众多，所以机器发生故障的概率很高，在设计任何分布式系统时都应考虑容错。本节主要针对 Standalone 部署模式的容错能力进行分析。

7.4.1 Executor 异常退出

如果 Executor 异常退出，那么势必导致在此 Executor 上执行的任务无法运行。曾经在

介绍ExecutorRunner启动CoarseGrainedExecutorBackend进程的代码清单7-33中说过，一旦收到退出状态（EXITED），则会向Worker发送ExecutorStateChanged消息。Worker收到ExecutorStateChanged消息后向Master转发ExecutorStateChanged消息，代码如下。

```
case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
    master ! ExecutorStateChanged(appId, execId, state, message, exitStatus)
```

Master收到ExecutorStateChanged消息后对退出状态的处理（见代码清单7-49）步骤如下：

1）找到占有Executor的Application的ApplicationInfo，以及Executor对应的ExecutorInfo。

2）将ExecutorInfo的状态改为EXITED。

3）EXITED也属于Executor完成状态，所以会将ExecutorInfo从ApplicationInfo和WorkerInfo中移除。

4）由于Executor是非正常退出，所以重新调用schedule给Application进行资源调度。

代码清单7-49 Master处理ExecutorStateChanged的代码

```
case ExecutorStateChanged(appId, execId, state, message, exitStatus) => {
    val execOption = idToApp.get(appId).flatMap(app => app.executors.get(execId))
    execOption match {
        case Some(exec) => {
            val appInfo = idToApp(appId)
            exec.state = state
            if (state == ExecutorState.RUNNING) { appInfo.resetRetryCount() }
            exec.application.driver ! ExecutorUpdated(execId, state, message,
                exitStatus)
            if (ExecutorState.isFinished(state)) {
                // Remove this executor from the worker and app
                logInfo(s"Removing executor ${exec.fullId} because it is $state")
                appInfo.removeExecutor(exec)
                exec.worker.removeExecutor(exec)

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

Executor 异常退出的容错处理可以用图 7-18 表示。

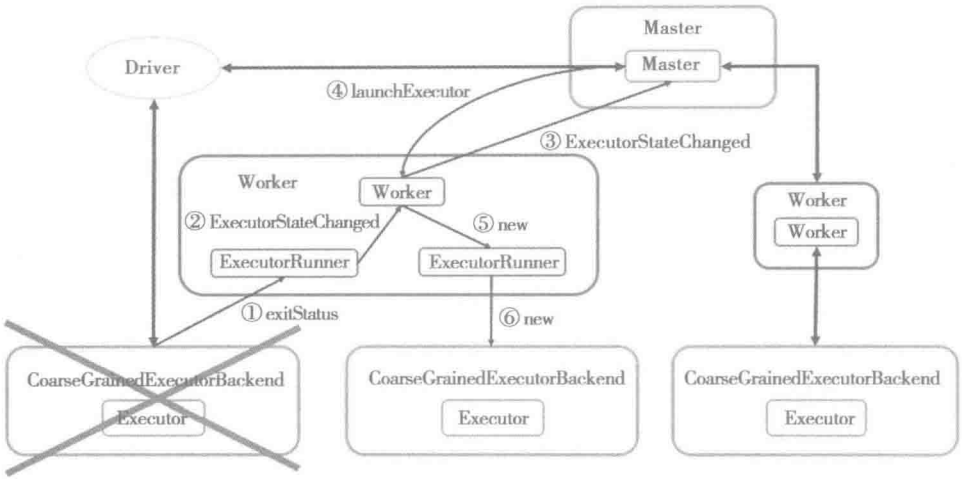


图 7-18 Executor 异常退出的容错处理

第⑥步之后 CoarseGrainedExecutorBackend 将向 Driver 注册 Executor，注册过程已在前面讲过。

7.4.2 Worker 异常退出

当 Worker 进程退出时，会发生什么呢？还记得代码清单 7-33 中的 shutdownHook 线程吗？它将在 Worker 进程退出前调用 killProcess 方法（见代码清单 7-50）主动杀死 CoarseGrainedExecutorBackend 进程，然后收到进程返回的退出状态（KILLED）后向 Worker 发送 ExecutorStateChanged 消息。由于 Worker 退出了，所以不会有 Heartbeat 消息发送给 Master，所以无法更新 Master 最后一次接收到心跳的时间戳（lastHeartbeat）。根据 timeOutDeadWorkers（见代码清单 7-5）的实现，Master 会调用 removeWorker 方法（见代码清单 7-6）删除长期失联的 Worker 的 WorkerInfo 信息，并将此 Worker 的所有 Executor 以 LOST 状态同步更新到它们服务的 Driver Application。最后 Master 还会为 WorkerInfo 所服务的 Driver Application 重新调度，分配到其他 Worker 上。整个过程可以用图 7-19 表示。

代码清单7-50 ExecutorRunner的killProcess方法

```
private def killProcess(message: Option[String]) {
    var exitCode: Option[Int] = None
    if (process != null) {
        if (stdoutAppender != null) {
            stdoutAppender.stop()
        }
        if (stderrAppender != null) {
            stderrAppender.stop()
        }
        process.destroy()
```

```
            exitCode = Some(process.waitFor())
        }
        worker ! ExecutorStateChanged(appId, execId, state, message, exitCode)
    }
```

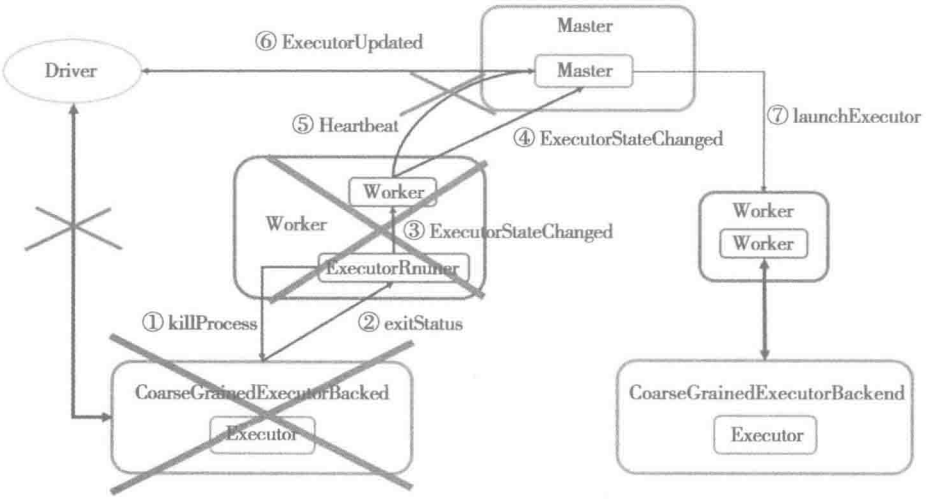


图 7-19 Worker 异常退出的容错处理

7.4.3 Master 异常退出

假如我们的 Standlone 部署集群只有一个 Master，当 Master 退出时会发生什么？

1）当 Executor 上的任务执行完毕，需要对资源回收。按照 7.3.6 节的内容，如果主动调用 SparkContext 的 top 方法，Driver Application 最终会向给自己服务的 Executor 发送 StopExecutor 消息对资源回收，Master 虽然跑路了，但是 Driver Application 依然持有这些 Executor，所以没有影响。如果不辞而别，那么 Master 将收不到 DisassociatedEvent 消息，但是 CoarseGrainedExecutorBackend 仍然可以收到 DisassociatedEvent 消息后退出进程。看来 Master 退出，如果 Worker、Executor 正常运行，则对于资源回收没有影响。

2）此时如果再发生 Executor 异常退出问题，Worker 无法通过 ExecutorStateChanged 消息促使 Master 重新给 Driver Application 调度运行 Executor，Driver Application 只能眼巴巴看着自己提交的任务无法执行。而 Executor 占用的资源也得不到回收。

3）此时如果又发生 Worker 异常退出问题，那么 Worker 和 Executor 都将停止服务，由于无法通知 Master 重新给 Driver Application 调度到其他 Worker 上，Driver Application 提交的任务也将无法执行。Worker 虽然 kill 掉了 Executor，但是 Worker 的资源将无法被其他 Driver Application 使用。

4）有新的 Driver 需要提交任务则无法成功。

基于以上分析，发现 Standlone 部署集群只有一个 Master 是万万不可以的。这也被称为

单点故障问题。解决此问题的常用方式就是采用 Master/Slaves 架构，即有多个 Master，但同时只有一个 Master 负责整个集群的调度、资源管理工作。

在代码清单 7-4 中，我们曾经提到了两个与 Master/Slaves 架构有关的组件：故障恢复的持久化引擎（persistenceEngine）和领导选举代理（leaderElectionAgent）。

Spark 目前提供的故障恢复的持久化引擎有如下几种：

- ZooKeeperPersistenceEngine：基于 ZooKeeper 实现的持久化引擎；
- FileSystemPersistenceEngine：基于文件系统的持久化引擎；
- BlackHolePersistenceEngine：默认的持久化引擎，实际上并不提供故障恢复的持久化能力。

领导选举机制（Leader Election）可以保证集群虽然存在多个 Master，但是只有一个 Master 处于激活（Active）状态，其他的 Master 处于支持（Standby）状态。当 Active 状态的 Master 出现故障时，会选举出一个 Standby 状态的 Master 作为新的 Active 状态的 Master。由于整个集群的 Worker、Driver 和 Application 的信息都已经持久化到文件系统，因此切换时只会影响新任务的提交，对于正在运行中的任务没有任何影响。Spark 目前提供的领导选举代理有两种：

- ZooKeeperLeaderElectionAgent：对 ZooKeeper 提供的选举机制的代理；
- MonarchyLeaderAgent：默认的选举机制代理。

默认情况下，Spark 不提供故障恢复，代码如下。

```
val RECOVERY_DIR = conf.get("spark.deploy.recoveryDirectory", "")
val RECOVERY_MODE = conf.get("spark.deploy.recoveryMode", "NONE")
```

当没有设置 spark.deploy.recoveryDirectory 和 spark.deploy.recoveryMode 时，RECOVERY_MODE 等于 NONE，此时匹配的持久化引擎是 BlackHolePersistenceEngine。BlackHolePersistenceEngine 虽然继承了 PersistenceEngine 的特质，但是实现方法都是空方法。

```
private[spark] class BlackHolePersistenceEngine extends PersistenceEngine {
    override def addApplication(app: ApplicationInfo) {}
    override def removeApplication(app: ApplicationInfo) {}
    override def addWorker(worker: WorkerInfo) {}
    override def removeWorker(worker: WorkerInfo) {}
    override def addDriver(driver: DriverInfo) {}
    override def removeDriver(driver: DriverInfo) {}

    override def readPersistedData() = (Nil, Nil, Nil)
}
```

这说明如果不设置 spark.deploy.recoveryDirectory 和 spark.deploy.recoveryMode，选举切换之后新的 Master 会丢失集群之前的所有信息。

1. FileSystemPersistenceEngine 搭配 MonarchyLeaderAgent 实现故障恢复

如果想用 Spark 本身实现选举和故障恢复，可以设置 spark.deploy.recoveryMode 为 FILESYSTEM。

此时匹配的持久化引擎是FileSystemPersistenceEngine。由于RECOVERY_DIR作为FileSystemPersistenceEngine的构造参数传入，而且FileSystemPersistenceEngine将会把集群信息存入RECOVERY_DIR指定的目录，见代码清单7-51，所以必须设置spark.deploy.recoveryDirectory指定目录。FileSystemPersistenceEngine最重要的方法是serializeIntoFile和deserializeFromFile，serializeIntoFile可以将任何AnyRef（如ApplicationInfo、DriverInfo、WorkerInfo）序列化写入文件，deserializeFromFile则可以将FileInputStream反序列化为任何对象（如ApplicationInfo、DriverInfo和WorkerInfo）。

代码清单7-51 FileSystemPersistenceEngine的实现

```
private[spark] class FileSystemPersistenceEngine(
    val dir: String,
    val serialization: Serialization)
extends PersistenceEngine with Logging {

new File(dir).mkdir()

override def addApplication(app: ApplicationInfo) {
    val appFile = new File(dir + File.separator + "app_" + app.id)
    serializeIntoFile(appFile, app)
}

override def removeApplication(app: ApplicationInfo) {
    new File(dir + File.separator + "app_" + app.id).delete()
}

override def addDriver(driver: DriverInfo) {
    val driverFile = new File(dir + File.separator + "driver_" + driver.id)
    serializeIntoFile(driverFile, driver)
}

override def removeDriver(driver: DriverInfo) {
    new File(dir + File.separator + "driver_" + driver.id).delete()
}

override def addWorker(worker: WorkerInfo) {
    val workerFile = new File(dir + File.separator + "worker_" + worker.id)
    serializeIntoFile(workerFile, worker)
}

override def removeWorker(worker: WorkerInfo) {
    new File(dir + File.separator + "worker_" + worker.id).delete()
}

override def readPersistedData(): (Seq[ApplicationInfo], Seq[DriverInfo],
    Seq[WorkerInfo]) = {
    val sortedFiles = new File(dir).listFiles().sortBy(_.getName)
    val appFiles = sortedFiles.filter(_.getName.startsWith("app_"))
    val apps = appFiles.map(deserializeFromFile[ApplicationInfo])
    val driverFiles = sortedFiles.filter(_.getName.startsWith("driver_"))
```

```
        val drivers = driverFiles.map(deserializeFromFile[DriverInfo])
        val workerFiles = sortedFiles.filter(_.getName.startsWith("worker_"))
        val workers = workerFiles.map(deserializeFromFile[WorkerInfo])
        (apps, drivers, workers)
    }

    private def serializeIntoFile(file: File, value: AnyRef) {
        val created = file.createNewFile()
        if (!created) { throw new IllegalStateException("Could not create file: " + file) }

        val serializer = serialization.findSerializerFor(value)
        val serialized = serializer.toBinary(value)

        val out = new FileOutputStream(file)
        try {
            out.write(serialized)
        } finally {
            out.close()
        }
    }

    def deserializeFromFile[T](file: File)(implicit m: Manifest[T]): T = {
            val fileData = new Array[Byte](file.length().asInstanceOf[Int])
            val dis = new DataInputStream(new FileInputStream(file))
            try {
                dis.readFully(fileData)
            } finally {
                dis.close()
            }

            val clazz = m.runtimeClass.asInstanceOf[Class[T]]
            val serializer = serialization.serializerFor(clazz)
            serializer.fromBinary(fileData).asInstanceOf[T]
        }
    }
```

当 spark.deploy.recoveryMode 为 FiLESYSTEM 时，领导选举代理是 MonarchyLeaderAgent。根据代码清单 7-8，选举时会从 FileSystemPersistenceEngine 中读取持久化集群信息，然后调用 beginRecovery 方法（见代码清单 7-52）恢复集群，最后设置一个向 Master 自身发送 CompleteRecovery 消息的定时调度。

beginRecovery 方法恢复集群的步骤如下。

1）将读取的集群信息中的 ApplicationInfo 重新调用 registerApplication 方法注册。

2）将读取的集群信息中的 DriverInfo 重新添加到缓存 drivers 中。

3）将读取的集群信息中的 WorkerInfo 重新调用 registerWorker 方法注册。

代码清单7-52 Master的beginRecovery方法

```
def beginRecovery(storedApps: Seq[ApplicationInfo], storedDrivers: Seq[DriverInfo],
        storedWorkers: Seq[WorkerInfo]) {
```

```
        for (app <- storedApps) {
            logInfo("Trying to recover app: " + app.id)
            try {
                registerApplication(app)
                app.state = ApplicationState.UNKNOWN
                app.driver ! MasterChanged(masterUrl, masterWebUiUrl)
            } catch {
                case e: Exception => logInfo("App " + app.id + " had exception on reconnect")
            }
        }

        for (driver <- storedDrivers) {
            drivers += driver
        }

        for (worker <- storedWorkers) {
            logInfo("Trying to recover worker: " + worker.id)
            try {
                registerWorker(worker)
                worker.state = WorkerState.UNKNOWN
                worker.actor ! MasterChanged(masterUrl, masterWebUiUrl)
            } catch {
                case e: Exception => logInfo("Worker " + worker.id + " had exception
                    on reconnect")
            }
        }
    }
```

Master 收到 CompleteRecovery 消息后，匹配执行 completeRecovery 方法。

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

completeRecovery 方法（见代码清单 7-53）用于对整个集群信息进行恢复，处理步骤如下：

1）通过同步保证对于集群恢复只发生一次。

2）将所有没有响应的 Worker 通过调用 removeWorker 方法（见代码清单 7-6）清除。

3）将所有没有响应的 Application 通过调用 finishApplication 方法清除。

4）将所有没有被调度的 Driver 重新调度。

代码清单7-53 Master的completeRecovery方法

```
def completeRecovery() {
    synchronized {
        if (state != RecoveryState.RECOVERING) { return }
        state = RecoveryState.COMPLETING_RECOVERY
    }

    workers.filter(_.state == WorkerState.UNKNOWN).foreach(removeWorker)
    apps.filter(_.state == ApplicationState.UNKNOWN).foreach(finishApplication)

    drivers.filter(_.worker.isEmpty).foreach { d =>
```

```
            logWarning(s"Driver ${d.id} was not found after master recovery")
            if (d.desc.supervise) {
                logWarning(s"Re-launching ${d.id}")
                relaunchDriver(d)
            } else {
                removeDriver(d.id, DriverState.ERROR, None)
                logWarning(s"Did not re-launch ${d.id} because it was not supervised")
            }
        }

        state = RecoveryState.ALIVE
        schedule()
        logInfo("Recovery complete - resuming operations!")
    }
```

2. 使用 ZooKeeper 提供的选举和持久化

还可以设置 spark.deploy.recoveryMode 为 ZOOKEEPER。此时匹配的持久化引擎是 ZooKeeperPersistenceEngine。ZooKeeperPersistenceEngine 也实现了 PersistenceEngine 的接口，见代码清单 7-54。

代码清单7-54 ZooKeeperPersistenceEngine的实现

```
class ZooKeeperPersistenceEngine(serialization: Serialization, conf: SparkConf)
    extends PersistenceEngine
    with Logging
{
    val WORKING_DIR = conf.get("spark.deploy.zookeeper.dir", "/spark") + "/
        master_status"
    val zk: CuratorFramework = SparkCuratorUtil.newClient(conf)

    SparkCuratorUtil.mkdir(zk, WORKING_DIR)

    override def addApplication(app: ApplicationInfo) {
        serializeIntoFile(WORKING_DIR + "/app_" + app.id, app)
    }

    override def removeApplication(app: ApplicationInfo) {
        zk.delete().forPath(WORKING_DIR + "/app_" + app.id)
    }

    override def addDriver(driver: DriverInfo) {
        serializeIntoFile(WORKING_DIR + "/driver_" + driver.id, driver)
    }

    override def removeDriver(driver: DriverInfo) {
        zk.delete().forPath(WORKING_DIR + "/driver_" + driver.id)
    }

    override def addWorker(worker: WorkerInfo) {
        serializeIntoFile(WORKING_DIR + "/worker_" + worker.id, worker)
    }
```

```
    override def removeWorker(worker: WorkerInfo) {
        zk.delete().forPath(WORKING_DIR + "/worker_" + worker.id)
    }

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

    override def readPersistedData(): (Seq[ApplicationInfo], Seq[DriverInfo],
        Seq[WorkerInfo]) = {
        val sortedFiles = zk.getChildren().forPath(WORKING_DIR).toList.sorted
        val appFiles = sortedFiles.filter(_.startsWith("app_"))
        val apps = appFiles.map(deserializeFromFile[ApplicationInfo]).flatten
        val driverFiles = sortedFiles.filter(_.startsWith("driver_"))
        val drivers = driverFiles.map(deserializeFromFile[DriverInfo]).flatten
        val workerFiles = sortedFiles.filter(_.startsWith("worker_"))
        val workers = workerFiles.map(deserializeFromFile[WorkerInfo]).flatten
        (apps, drivers, workers)
    }

    private def serializeIntoFile(path: String, value: AnyRef) {
        val serializer = serialization.findSerializerFor(value)
        val serialized = serializer.toBinary(value)
        zk.create().withMode(CreateMode.PERSISTENT).forPath(path, serialized)
    }

    def deserializeFromFile[T](filename: String)(implicit m: Manifest[T]): Option[T] = {
        val fileData = zk.getData().forPath(WORKING_DIR + "/" + filename)
        val clazz = m.runtimeClass.asInstanceOf[Class[T]]
        val serializer = serialization.serializerFor(clazz)
        try {
            Some(serializer.fromBinary(fileData).asInstanceOf[T])
        } catch {
            case e: Exception => {
                logWarning("Exception while reading persisted file, deleting", e)
                zk.delete().forPath(WORKING_DIR + "/" + filename)
                None
            }
        }
    }
}
```

当 spark.deploy.recoveryMode 为 ZOOKEEPER 时，领导选举代理是 ZooKeeperLeaderElectionAgent。ZooKeeperLeaderElectionAgent 的 preStart 方法（见代码清单 7-55）实现了基于 ZooKeeper 的选举。

代码清单7-55 ZooKeeperLeaderElectionAgent的preStart方法

```
override def preStart() {

    logInfo("Starting ZooKeeper LeaderElection agent")
```

```
        zk = SparkCuratorUtil.newClient(conf)
        leaderLatch = new LeaderLatch(zk, WORKING_DIR)
        leaderLatch.addListener(this)

        leaderLatch.start()
    }
```

注意 Curator 是 Netflix 开源的一套 ZooKeeper 客户端框架。Curator 极大地简化了 ZooKeeper 的使用，它提供了高层次的 API，并且基于 ZooKeeper 添加了很多特性。ZooKeeperPersistenceEngine 和 ZooKeeperLeaderElectionAgent 实际都是对 Curator API 的封装。

如果要以 ZooKeeper 作为热备（HA）的方案，在集群启动时需增加一些配置。

编辑 spark-env.sh，增加表 7-5 中的参数。

表 7-5 以 ZooKeeper 作为热备需要增加的配置

属 性 名
spark.deploy.recoveryMode= ZOOKEEPER
spark.deploy.zookeeper.url=cms_zk_server_1:2181, cms_zk_server_2:2181
spark.deploy.zookeeper.dir=/usr/zk/persist

使用 ZooKeeper 作为热备的选举方案可以用图 7-20 表示。

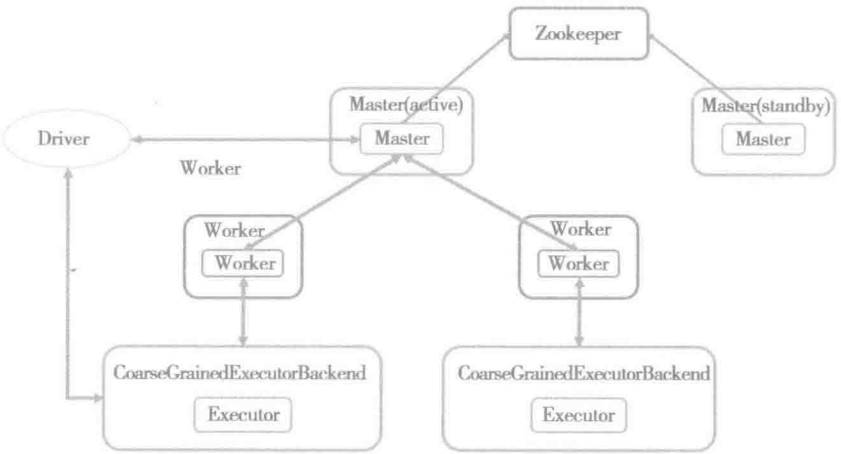


图 7-20 ZooKeeper 热备和选举示意图

7.5 其他部署方案

本节介绍 Spark 运行在第三方资源管理集群上的部署方案。Spark 目前支持运行在 YARN

和 Mesos，甚至运行在 MRv1 框架上。对于 MRv1 框架运行 Spark 的内容，有兴趣的读者可以自行研究，本文不做过多介绍。

7.5.1 YARN

在本书第 2 章，笔者曾经简单介绍了 MRv1 的缺陷以及 MRv2 的改进。MRv1 的运行时环境由 JobTracker 和 TaskTracker 两类服务组成，JobTracker 负责资源和任务的管理与调度，TaskTracker 负责单个节点的资源管理和任务执行。在 YARN 中，JobTracker 被分为两部分：ResourceManager（RM）和 ApplicationMaster（AM）。RM 负责资源管理和调度，AM 负责具体应用程序的任务划分、调度等工作。YARN 的运行时环境如图 7-21 ⊖所示。

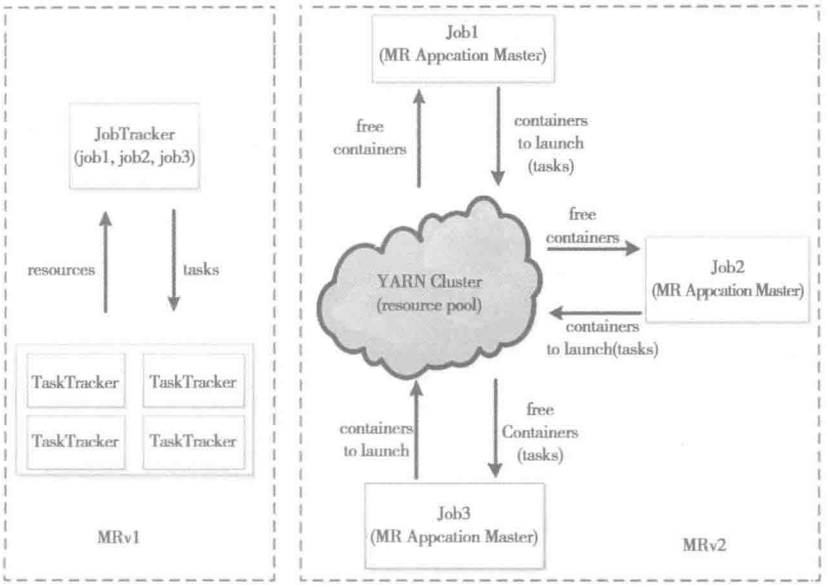


图 7-21 YARN 的运行时环境

YARN 的架构如图 7-22 所示。

YARN 的基本结构包括以下部分。

- ResourceManager（RM）：全局资源管理器，负责整个系统的资源管理与分配。RM 由调度器和应用程序管理器组成。调度器将系统资源分配给各个应用程序，资源的分配单位不再是 MRv1 中的“slot”，而是 Container。Container 是对 CPU、内存等资源的封装。应用程序管理器负责管理整个系统的应用程序，如处理程序提交，与调度器沟通后为应用程序启动 ApplicationMaster 等。
- ApplicationMaster（AM）：用户提交的每个任务都有一个 AM，它会与 RM 通信获取

⊖ 图片来源自博客 http://blog.chinaunix.net/uid-28311809-id-4383551.html。

资源，将任务划分为更细粒度的任务，与 NodeManager（NM）通信启动或停止任务，监控失败任务为其重新申请资源后重新启动等。

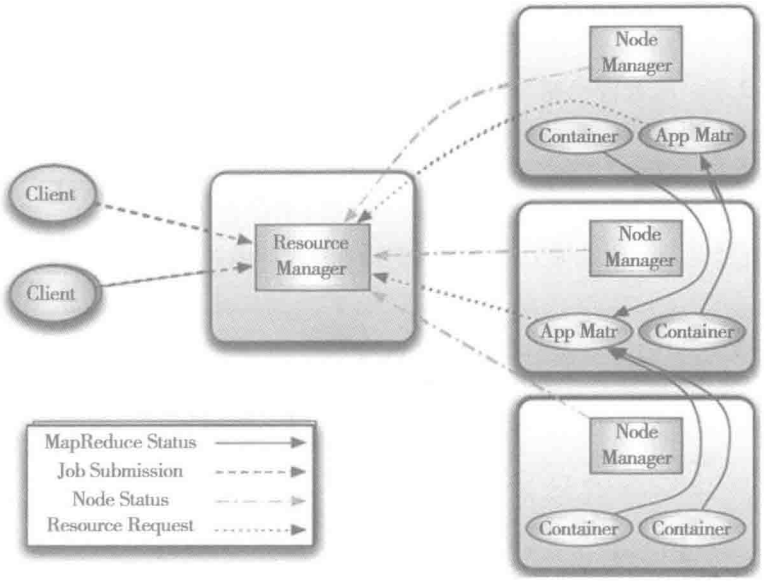


图 7-22[⊖] YARN 架构

- NodeManager（NM）：单个节点上的资源与任务管理器，它负责向 RM 定时汇报本节点的资源使用情况及各个 Container 的状态，也接受处理 AM 的启动或停止任务请求。

YARN 对于支持的 MapReduce 框架是可插拔的，Spark 对于集群管理器也支持可插拔，两者不谋而合。有关 YARN 的安装和使用，读者可自行翻阅相关书籍。假设我们已经部署好了 YARN 集群，根据图 7-22 所有应用程序需要一个 Application Master，这个 Application Master 位于 YARN 集群的一个节点上，负责管理资源申请与调度。Spark 也提供了 Application Master 的实现 ApplicationMaster。与 YARN 集成时整个 Spark 集群的启动顺序如下。

1）将 Spark 提供的 ApplicationMaster 在 YARN 集群中启动（见代码清单 7-56）。

2）ApplicationMaster 向 ResourceManager 申请 Container。

3）申请 Container 成功后，向具体的 NodeManager 发送指令启动 Container。

4）ApplicationMaster 启动对各个运行的 ContainerExecutor 进行监控。

代码清单7-56 ApplicationMaster的main方法

```
def main(args: Array[String]) = {
    SignalLogger.register(log)
    val amArgs = new ApplicationMasterArguments(args)
```

⊖ 图片源自博客 http://blog.csdn.net/iloveyin/article/details/28421009。

```
        SparkHadoopUtil.get.runAsSparkUser { () =>
            master = new ApplicationMaster(amArgs, new YarnRMClientImpl(amArgs))
            System.exit(master.run())
        }
    }
```

ApplicationMaster 会调用构造时传入的 YarnRMClientImpl 的 register 方法（见代码清单 7-57），向 YARN 注册 AM。

代码清单7-57 YarnRMClientImpl的register方法

```
override def register(
        conf: YarnConfiguration,
        sparkConf: SparkConf,
        preferredNodeLocations: Map[String, Set[SplitInfo]],
        uiAddress: String,
        uiHistoryAddress: String,
        securityMgr: SecurityManager) = {
    amClient = AMRMClient.createAMRMClient()
    amClient.init(conf)
    amClient.start()
    this.uiHistoryAddress = uiHistoryAddress

    logInfo("Registering the ApplicationMaster")
    synchronized {
        amClient.registerApplicationMaster(Utils.localHostName(), 0, uiAddress)
        registered = true
    }
    new YarnAllocationHandler(conf, sparkConf, amClient, getAttemptId(), args,
        preferredNodeLocations, securityMgr)
  }
```

下面我们设置 master 为“yarn-cluster”，那么在创建 TaskSchedulerImpl 时就会匹配 yarn-cluster 模式，见代码清单 7-58。其中 YarnClusterScheduler 继承自 TaskSchedulerImpl，因此 YarnClusterScheduler 将负责任务的提交与调度。YarnSchedulerBackend 继承自 org.apache.spark.scheduler.cluster 包中的 CoarseGrainedSchedulerBackend，YarnClusterSchedulerBackend 又继承了 YarnSchedulerBackend。于是 YarnClusterSchedulerBackend 就是 TaskSchedulerImpl 的 backend。

代码清单7-58 SparkContext匹配yarn-cluster的代码

```
case "yarn-standalone" | "yarn-cluster" =>
    if (master == "yarn-standalone") {
        logWarning(
            "\"yarn-standalone\" is deprecated as of Spark 1.0. Use \"yarn-
                cluster\" instead.")
    }
    val scheduler = try {
        val clazz = Class.forName("org.apache.spark.scheduler.cluster.YarnCluster-
             Scheduler")
```

```
        val cons = clazz.getConstructor(classOf[SparkContext])
        cons.newInstance(sc).asInstanceOf[TaskSchedulerImpl]
    } catch {
        case e: Exception => {
            throw new SparkException("YARN mode not available ?", e)
        }
    }
    val backend = try {
        val clazz =
            Class.forName("org.apache.spark.scheduler.cluster.YarnClusterScheduler-
                Backend")
        val cons = clazz.getConstructor(classOf[TaskSchedulerImpl], classOf[SparkContext])
            cons.newInstance(scheduler, sc).asInstanceOf[CoarseGrainedSchedulerB
                ackend]
        } catch {
            case e: Exception => {
                throw new SparkException("YARN mode not available ?", e)
            }
        }
        scheduler.initialize(backend)
        (backend, scheduler)
```

根据 3.10 节我们知道，最终会调用 YarnClusterSchedulerBackend 的 start 方法，其中主要确定了期望获得的 Executor 的数量，实现如下。

```
override def start() {
  super.start()
  totalExpectedExecutors = DEFAULT_NUMBER_EXECUTORS
  if (System.getenv("SPARK_EXECUTOR_INSTANCES") != null) {
    totalExpectedExecutors = IntParam.unapply(System.getenv("SPARK_EXECUTOR_
        INSTANCES"))
      .getOrElse(totalExpectedExecutors)
  }
  totalExpectedExecutors = sc.getConf.getInt("spark.executor.instances",
      totalExpectedExecutors)
}
```

Driver Application 初始化完毕会向 ApplicationMaster 进行注册，在 YARN 部署模式中，Worker 已被 NodeManager 替代，ApplicationMaster 给 Application 分配资源主要借助于代码清单 7-57 中构造的 YarnAllocationHandler。

关于 YARN 部署模式就简单介绍这些，更多内容去 http://hadoop.apache.org/index.html 了解。

7.5.2 Mesos

Mesos 是诞生于 UC Berkeley 的一个研究项目，现已成为 Apache Incubator 中的项目。Mesos 是一个集群管理器，提供了有效的、跨分布式应用或框架的资源隔离和共享，可以灵活支持 Hadoop、MPI、Hypertable、Spark 等。使用 ZooKeeper 实现容错复制，采用 Linux Container 对内存和 CPU 进行隔离。

Twitter 已经在使用 Mesos 管理集群资源。

Mesos 为了简化设计，也采用了 master/slave 结构，为了解决 master 单点故障，master 是非常轻量级的，仅保存了各种计算框架（如 Hadoop、MPI、Hypertable、Spark）和 Mesos slave 的一些状态，而这些状态很容易通过 framework 和 slave 重新注册而重构，因此很容易使用 ZooKeeper 解决 Mesos master 的单点故障问题。Mesos 的整体架构如图 7-23 ⊖所示。

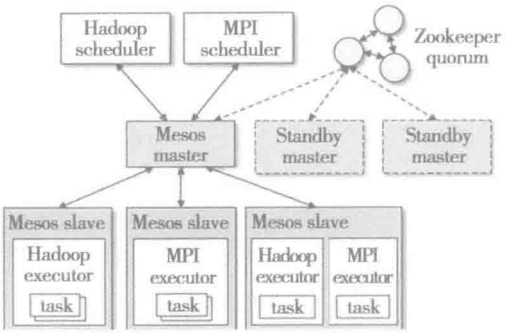


图 7-23 Mesos 整体架构

Mesos master 实际上是一个全局资源调度器，采用某种策略将某个 slave 上的空闲资源分配给各个计算框架，各种计算框架通过自己的调度器向 Mesos master 注册，以接入 Mesos 中；而 Mesos slave 主要功能是汇报任务的状态和启动各个计算框架的 Executor（比如 Spark 的 Executor）。整个 Mesos 系统采用了双层调度框架：第一层，由 Mesos 将资源分配给框架；第二层，框架自己的调度器将资源分配给自己内部的任务。

在 Mesos 中，各种计算框架是完全融入 Mesos 中的，也就是说，如果你想在 Mesos 中添加一个新的计算框架，首先需要在 Mesos 中部署一套该框架。

如果我们设置 master 为“ mesos://”，那么在创建 TaskSchedulerImpl 时就会匹配代码清单 7-59 中的实现。

代码清单7-59 SparkContext匹配mesos://的代码

```
case mesosUrl @ MESOS_REGEX(_) =>
    MesosNativeLibrary.load()
    val scheduler = new TaskSchedulerImpl(sc)
    val coarseGrained = sc.conf.getBoolean("spark.mesos.coarse", false)
    val url = mesosUrl.stripPrefix("mesos://") // strip scheme from raw Mesos URLs
    val backend = if (coarseGrained) {
        new CoarseMesosSchedulerBackend(scheduler, sc, url)
    } else {
        new MesosSchedulerBackend(scheduler, sc, url)
    }
    scheduler.initialize(backend)
    (backend, scheduler)
```

此时的 backend 可以是 CoarseMesosSchedulerBackend 或者 MesosSchedulerBackend。以 CoarseMesosSchedulerBackend 为例，它的 start 方法（见代码清单 7-60）通过调用 Mesos 的 API 完成 Driver 的注册。有关 Mesos 的 API 的细节，请访问官网 http://mesos.apache.org/。

代码清单7-60 CoarseMesosSchedulerBackend的启动

```
override def start() {
```

⊖ 部分内容引用自博文 http://www.tuicool.com/articles/buq2uqM。

```
        super.start()

        synchronized {
            new Thread("CoarseMesosSchedulerBackend driver") {
                setDaemon(true)
                override def run() {
                    val scheduler = CoarseMesosSchedulerBackend.this
                    val fwInfo = FrameworkInfo.newBuilder().setUser(sc.sparkUser).
                        setName(sc.appName).build()
                    driver = new MesosSchedulerDriver(scheduler, fwInfo, master)
                    try { {
                        val ret = driver.run()
                        logInfo("driver.run() returned with code " + ret)
                    }
                    } catch {
                        case e: Exception => logError("driver.run() failed", e)
                    }
                }
            }.start()

            waitForRegister()
        }
    }
```

7.6 小结

我们最先了解 local 部署模式，然后精读 local-cluster 部署模式，最后分析 Standalone 部署模式。这样能够由简入难，由浅入深，尽可能帮助读者降低阅读 Spark 源码与理解部署模式架构原理的难度。local 部署模式和 local-cluster 部署模式不能用于生产，为了简单，生产中可以选择使用 Standalone 部署模式。如果对 YARN 和 Mesos 也有一定了解，那么使用 YARN 和 Mesos 也非常不错。

资源调度、资源回收以及容错机制的源码剖析，除了能让我们理解 Spark 为什么能拥有很高的稳定性、可用性及容错能力外，其中各种架构设计的思想值得做架构设计的开发人员学习。

扩 展 篇

- 第 8 章　Spark SQL
- 第 9 章　流式计算
- 第 10 章　图计算
- 第 11 章　机器学习

Chapter 8 第 8 章

Spark SQL

游飏下晴空，寻芳到菊丛，带声来蕊上，连影在香中。

——耿湋

本章导读

在很多情况下，工程师不了解 Scala 语言，也不了解 Spark 的常用 API，但又希望使用强大的 Spark 所提供的数据分析能力。该怎么办？有没有什么工具或者语言能降低使用者的学习成本呢？ SQL（structured query language）语言从 20 世纪 70 年代诞生以来，如今已经走过了 40 多年。由于其常用语法简洁，学习门槛低，其流行程度和普及程度毋庸置疑。甚至可以说，一条 SQL 让世界变得简单。

正如 Hadoop 提供了 SQL 供开发人员使用，Spark 的工程师们考虑到这个问题，也开发了 Spark 自身的 SQL 处理能力。传统关系型数据库执行一条 SQL，通常都会经过分析、优化、执行这三个阶段。作为 Spark 又是如何执行 SQL 的呢？带着这个疑问，我们一起来分析 Spark SQL 吧！

8.1 Spark SQL 总体设计

Spark SQL 的前身是 Shark，Shark 是伯克利实验室 Spark 生态环境的组件之一，运行在 Spark 上。由于 Shark 对于 Hive 的太多依赖制约了 Spark 的发展，Spark SQL 由此产生。

假如我们提交了一条简单的 SQL 查询语句：

```
SELECT name FROM people WHERE age >= 13 AND age <= 19
```

这条 SQL 语句由 Projection（name）、Data Source（people）、Filter（age >= 13 AND age <= 19）组成，分别对应于 SQL 查询过程中的 Result、Data Source、Operation，即 SQL 语句是按照 Result → Data Source → Operation 的次序来描述的，如图 8-1 所示。

应该如何执行呢？我们先来看看传统关系型数据库，再来看 Spark SQL 如何处理。

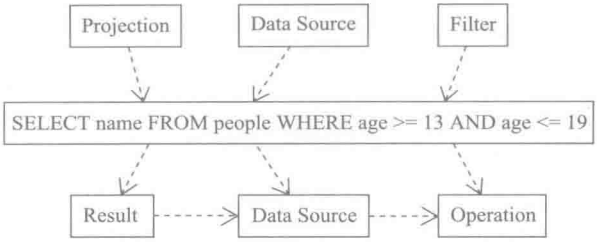


图 8-1　SQL 语义描述

8.1.1　传统关系型数据库 SQL 运行原理

在传统关系型数据库的执行过程中，先将 SQL 语句（Query）进行解析（Parse），找出其中的关键词（如 SELECT、FROM、WHERE）、表达式、Projection（a1, a2, a3）、Data Source（tableA）等。接着会对 SQL 语句校验规范，如果规范则将 SQL 语句和数据库的数据字典（列、表、视图等）进行绑定（Bind），在执行前，数据库会从多个执行计划中选择一个最优计划（Optimize），最后执行该计划（Execute），并返回结果。整个过程和 SQL 语句的次序正好相反，如图 8-2 所示。

图 8-2　SQL 执行过程

语法解析后，会构建 SQL 语句的语法树，如图 8-3 所示。

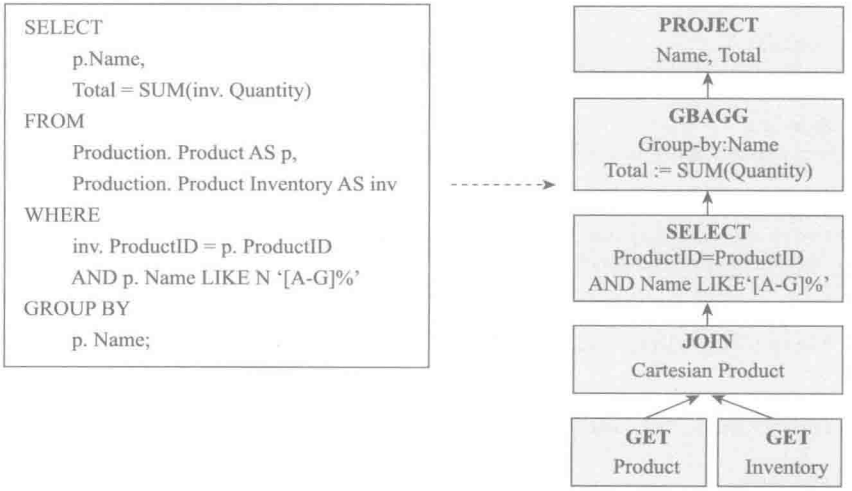


图 8-3　SQL 语句转换为语法树

8.1.2 Spark SQL 运行架构

Spark 已经无缝接入了 SQL，并且支持对多种多样数据源的查询与加载，兼容了 HiveSQL，甚至可以用 JDBC 或者 ODBC 来连接 Spark SQL。Spark 官网给出了 Spark SQL 的架构，如图 8-4 所示。从图 8-4 看出，基于 Spark SQL 可以重用 Hive 本身提供的元数据仓库（MetaStore）、HiveQL、用户自定义函数（UDF）及序列化和反序列化的工具（SerDes）。有关 Hive 中这些工具的具体使用，请读者自行阅读 Hive 相关资料。

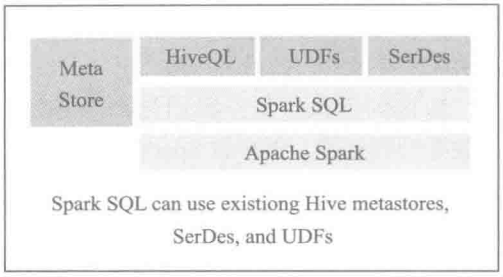


图 8-4 Spark SQL 体系架构

现在我们来看看 Spark SQL 如何处理？Spark SQL 也采用了类似的办法处理 SQL，通过了解 Spark SQL 上下文 SQLContext 的实现，我们会对 Spark SQL 运行架构有个大体的认识，其实现见代码清单 8-1。

代码清单8-1 SQLContext的实现

```
@AlphaComponent
class SQLContext(@transient val sparkContext: SparkContext)
    extends org.apache.spark.Logging
    with SQLConf
    with CacheManager
    with ExpressionConversions
    with UDFRegistration
    with Serializable {

    self =>

    @transient
    protected[sql] lazy val catalog: Catalog = new SimpleCatalog(true)

    @transient
    protected[sql] lazy val functionRegistry: FunctionRegistry = new SimpleFunctionRegistry

    @transient
    protected[sql] lazy val analyzer: Analyzer =
        new Analyzer(catalog, functionRegistry, caseSensitive = true)

    @transient
    protected[sql] lazy val optimizer: Optimizer = DefaultOptimizer

    @transient
    protected[sql] val ddlParser = new DDLParser

    @transient
    protected[sql] val sqlParser = {
    val fallback = new catalyst.SqlParser
        new catalyst.SparkSQLParser(fallback(_))
```

```
    }

    @transient
    protected[sql] val planner = new SparkPlanner

    @transient
    protected[sql] val prepareForExecution = new RuleExecutor[SparkPlan] {
        val batches =
            Batch("Add exchange", Once, AddExchange(self)) :: Nil
}
```

通过阅读 SQLContext 的源码，可以看出 SQLContext 由以下部分组成：

- Catalog：字典表，用于注册表，对表缓存后便于查询。
- DDLParser：用于解析 DDL 语句，如创建表。
- SparkSQLParser：作为 SqlParser 的代理，处理一些 SQL 中的通用关键字。
- SqlParser：用于解析 select 查询语句。
- Analyzer：对还未分析的逻辑执行计划（LogicalPlan）进行分析。
- Optimizer：对已经分析过的逻辑执行计划（LogicalPlan）进行优化。
- SparkPlanner：用于将逻辑执行计划（LogicalPlan）转换为物理执行计划（PhysicalPlan）。
- prepareForExecution：用于将物理执行计划（PhysicalPlan）转换为可执行物理计划。

这些部分共同参与到 SQL 的执行过程中，如图 8-5 所示，步骤如下：

1）SQL 语句经过 SqlParser 解析成 Unresolved LogicalPlan；
2）使用 analyzer 结合数据字典（catalog）进行绑定，生成 Resolved LogicalPlan；
3）使用 optimizer 对 Resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan；
4）使用 SparkPlan 将 LogicalPlan 转换成 PhysicalPlan；
5）使用 prepareForExecution 将 PhysicalPlan 转换成可执行物理计划；
6）使用 execute() 执行可执行物理计划，生成 SchemaRDD。

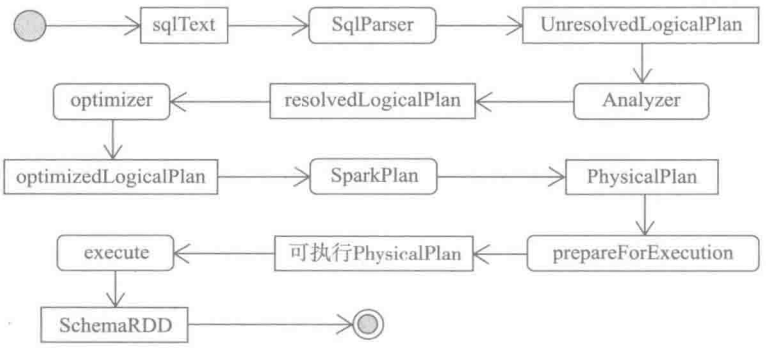


图 8-5　Spark SQL 执行过程

接下来我们将逐个介绍 Spark SQL 中的这些组件，以及这些组件是如何配合的。

8.2 字典表 Catalog

Catalog 是一个特质，定义了以下接口：

- tableExists：判断表是否存在。
- lookupRelation：使用表名查找关系。
- registerTable：注册表。
- unregisterTable：取消注册表。
- unregisterAllTables：清除所有已经注册的表。

Catalog 的接口定义见代码清单 8-2。

代码清单8-2 特质Catalog的接口定义

```
trait Catalog {

    def caseSensitive: Boolean

    def tableExists(tableIdentifier: Seq[String]): Boolean

    def lookupRelation(
        tableIdentifier: Seq[String],
        alias: Option[String] = None): LogicalPlan

    def registerTable(tableIdentifier: Seq[String], plan: LogicalPlan): Unit

    def unregisterTable(tableIdentifier: Seq[String]): Unit

    def unregisterAllTables(): Unit

    protected def processTableIdentifier(tableIdentifier: Seq[String]): Seq[String] = {
        if (!caseSensitive) {
            tableIdentifier.map(_.toLowerCase)
        } else {
          tableIdentifier
        }
    }

    protected def getDbTableName(tableIdent: Seq[String]): String = {
        val size = tableIdent.size
        if (size <= 2) {
            tableIdent.mkString(".")
        } else {
            tableIdent.slice(size - 2, size).mkString(".")
        }
    }

    protected def getDBTable(tableIdent: Seq[String]) : (Option[String], String) = {
        (tableIdent.lift(tableIdent.size - 2), tableIdent.last)
    }
}
```

SimpleCatalog 是 Catalog 的常用实现类，它实现了 Catalog 中的所有接口，见代码清单 8-3。所谓注册功能，实际就是将表名与 LogicalPlan 一起放入缓存 tables：mutable.HashMap[String, LogicalPlan] 中。

代码清单8-3 SimpleCatalog的实现

```
class SimpleCatalog(val caseSensitive: Boolean) extends Catalog {
    val tables = new mutable.HashMap[String, LogicalPlan]()

    override def registerTable(
            tableIdentifier: Seq[String],
            plan: LogicalPlan): Unit = {
        val tableIdent = processTableIdentifier(tableIdentifier)
    tables += ((getDbTableName(tableIdent), plan))
    }

    override def unregisterTable(tableIdentifier: Seq[String]) = {
        val tableIdent = processTableIdentifier(tableIdentifier)
        tables -= getDbTableName(tableIdent)
    }

    override def unregisterAllTables() = {
        tables.clear()
    }

    override def tableExists(tableIdentifier: Seq[String]): Boolean = {
        val tableIdent = processTableIdentifier(tableIdentifier)
        tables.get(getDbTableName(tableIdent)) match {
            case Some(_) => true
            case None => false
        }
    }

    override def lookupRelation(
            tableIdentifier: Seq[String],
            alias: Option[String] = None): LogicalPlan = {
        val tableIdent = processTableIdentifier(tableIdentifier)
        val tableFullName = getDbTableName(tableIdent)
        val table = tables.getOrElse(tableFullName, sys.error(s"Table Not Found:
           $tableFullName"))
        val tableWithQualifiers = Subquery(tableIdent.last, table)

        alias.map(a => Subquery(a, tableWithQualifiers)).getOrElse(tableWithQualifiers)
    }
}
```

8.3 Tree 和 TreeNode

Spark 的语法树是由 TreeNode 实现的，我们先来看看 TreeNode 的部分实现，见代码清单 8-4。

代码清单8-4 TreeNode的部分实现

```
abstract class TreeNode[BaseType <: TreeNode[BaseType]] {
    self: BaseType with Product =>

    def children: Seq[BaseType]

    def foreach(f: BaseType => Unit): Unit = {
        f(this)
        children.foreach(_.foreach(f))
    }

    def map[A](f: BaseType => A): Seq[A] = {
        val ret = new collection.mutable.ArrayBuffer[A]()
        foreach(ret += f(_))
        ret
    }

    def flatMap[A](f: BaseType => TraversableOnce[A]): Seq[A] = {
        val ret = new collection.mutable.ArrayBuffer[A]()
        foreach(ret ++= f(_))
        ret
    }

    def collect[B](pf: PartialFunction[BaseType, B]): Seq[B] = {
        val ret = new collection.mutable.ArrayBuffer[B]()
        val lifted = pf.lift
        foreach(node => lifted(node).foreach(ret.+=))
        ret
    }
```

除了 Scala 中耳熟能详的方法 foreach、map、flatMap、collect 外，TreeNode 还实现了应用 Rule 的方法，比如 transformDown、transformUp 将 Rule 应用到给定的树段，并用结果替代旧的树段；transformChildrenDown、transformChildrenUp 对一个给定的节点进行操作，通过迭代将 Rule 应用到该节点以及子节点。我们以 transformDown 为例来看看其实现，见代码清单 8-5。

代码清单8-5 TreeNode的transformDown方法

```
def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = {
    transformDown(rule)
}

def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = {
    val afterRule = rule.applyOrElse(this, identity[BaseType])
    if (this fastEquals afterRule) {
        transformChildrenDown(rule)
    } else {
        afterRule.transformChildrenDown(rule)
    }
}

def transformChildrenDown(rule: PartialFunction[BaseType, BaseType]): this.type = {
```

```
    var changed = false
    val newArgs = productIterator.map {
        case arg: TreeNode[_] if children contains arg =>
            val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
            if (!(newChild fastEquals arg)) {
            changed = true
            newChild
            } else {
            arg
          }
        case Some(arg: TreeNode[_]) if children contains arg =>
            val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
            if (!(newChild fastEquals arg)) {
                changed = true
                Some(newChild)
            } else {
               Some(arg)
            }
        case m: Map[_,_] => m
        case args: Traversable[_] => args.map {
            case arg: TreeNode[_] if children contains arg =>
                val newChild = arg.asInstanceOf[BaseType].transformDown(rule)
                if (!(newChild fastEquals arg)) {
                    changed = true
                    newChild
                } else {
                  arg
                }
            case other => other
        }
        case nonChild: AnyRef => nonChild
        case null => null
    }.toArray
    if (changed) makeCopy(newArgs) else this
}
```

TreeNode 共有 3 种。

- UnaryNode：一元节点，即只有一个子节点的节点。如 Project、Sort、Limit、Filter 等操作。
- BinaryNode：二元节点，即有左右两个子节点的节点。如 Except、Intersect 操作。
- LeafNode：叶子节点，即没有子节点的节点。如 SetCommand、DescribeCommand、ExplainCommand 等。

3 种 TreeNode 的实现见代码清单 8-6。

代码清单8-6 3种TreeNode

```
/**
 * A [[TreeNode]] that has two children, [[left]] and [[right]].
 */
```

```
trait BinaryNode[BaseType <: TreeNode[BaseType]] {
    def left: BaseType
    def right: BaseType

    def children = Seq(left, right)
}

/**
 * A [[TreeNode]] with no children.
 */
trait LeafNode[BaseType <: TreeNode[BaseType]] {
    def children = Nil
}

/**
 * A [[TreeNode]] with a single [[child]].
 */
trait UnaryNode[BaseType <: TreeNode[BaseType]] {
    def child: BaseType
    def children = child :: Nil
}
```

TreeNode 的继承体系中最重要的一些类如图 8-6 所示。

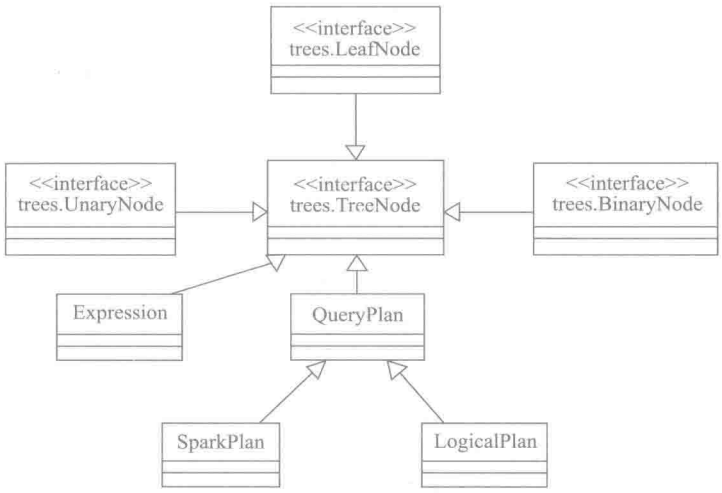


图 8-6 TreeNode 的继承体系中最重要的一些类

在 org.apache.spark.sql.catalyst.plans.logical 包中定义了大量 LogicalPlan 的子类；在 org.apache.spark.sql.catalyst.expressions 包中定义了大量 Expression 的子类；在 org.apache.spark.sql.execution 包中定义了大量 SparkPlan 的子类。

这里以 LogicalPlan 的继承体系（如图 8-7 所示）为例。其他的继承体系，有兴趣的读者可以自行查看。

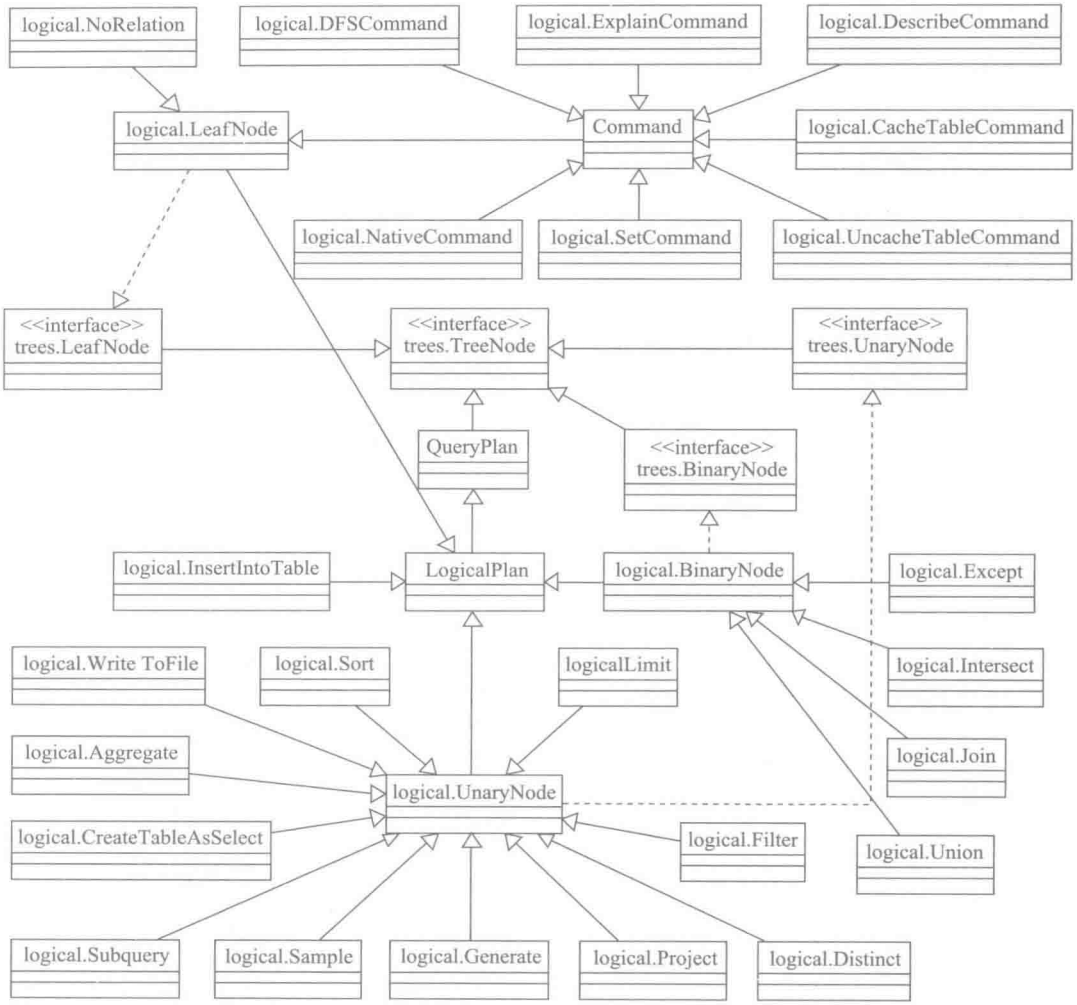


图 8-7 LogicalPlan 的继承体系

8.4 词法解析器 Parser 的设计与实现

词法解析发生在任务提交之前，在 Parser 阶段，常用的关于 SQL 解析的类有：

- DDLParser：临时表创建解析器，主要用来解析创建临时表的 DDL 语句。
- SqlParser：SQL 语句解析器，解析 select、insert 等 SQL 语句。
- SparkSQLParser：用于代理 SqlParser，对 AS、CACHE、SET 等关键字进行处理。

Parser 相关的类继承体系如图 8-8 所示。

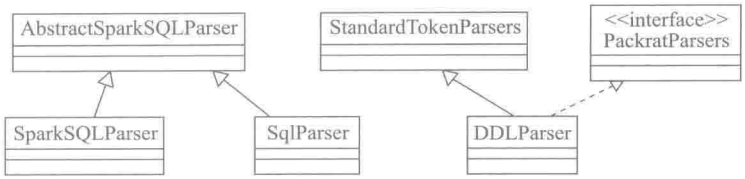


图 8-8 Parser 相关的类继承体系

8.4.1 SQL 语句解析的入口

SQLContext 的 SQL 方法是解析所有 SQL 的总入口，但实际的解析交给了 parseSql 方法，见代码清单 8-7。

代码清单8-7 SQLContext中SQL语句解析的入口

```
def sql(sqlText: String): SchemaRDD = {
  if (dialect == "sql") {
    new SchemaRDD(this, parseSql(sqlText))
  } else {
    sys.error(s"Unsupported SQL dialect: $dialect")
  }
}
```

parseSql 首先调用 DDLParser 解析 SQL，如果解析失败则调用 SparkSQLParser 解析，见代码清单 8-8。

代码清单8-8 SQLContext的parseSql方法

```
protected[sql] def parseSql(sql: String): LogicalPlan = {
  ddlParser(sql).getOrElse(sqlParser(sql))
}
```

这里看起来像是调用了 DDLParser 和 SparkSQLParser 的构造器，但是 DDLParser 却没有这样一个构造器啊！根据 Scala 语法我们知道，如果定义了一个类 A，而且定义了 apply(input: String) 方法，那么使用 A("hello") 时实际是调用其 apply 方法。那么 ddlParser(sql) 实际隐式调用了 apply 方法，见代码清单 8-9。

代码清单8-9 DDLParser的apply方法

```
def apply(input: String): Option[LogicalPlan] = {
  phrase(ddl)(new lexical.Scanner(input)) match {
    case Success(r, x) => Some(r)
    case x =>
      logDebug(s"Not recognized as DDL: $x")
      None
  }
}
```

无论 SqlParser 还是 SparkSQLParser，都继承自 AbstractSparkSQLParser。AbstractSparkSQLParser 也定义了 apply 方法，因此 sqlParser(sql) 实际隐式调用了父类的 apply 方法，见代

码清单 8-10。

代码清单8-10 AbstractSparkSQLParser的apply方法

```
def apply(input: String): LogicalPlan = phrase(start)(new lexical.Scanner(input)) match {
  case Success(plan, _) => plan
  case failureOrError => sys.error(failureOrError.toString)
}

protected case class Keyword(str: String)

protected def start: Parser[LogicalPlan]
```

以上代码中都有 phrase(query)(new lexical.Scanner(input)) 的语句，这条语句让笔者百思不得其解，最后在网络上找到了一些答案：这条语句调用了 Scala 的内置包 scala.util.parsing.combinator 中的功能，它的含义是对 input 进行解析，符合 query 的模式的就返回 Success。AbstractSparkSQLParser 中的模式为 start，但没有实现。后面会具体介绍 DDLParser 中的模式 ddl 以及 SqlParser 和 SparkSQLParser 都实现的 start 模式。

8.4.2 建表语句解析器 DDLParser

DDLParser 主要用于创建临时表，实现见代码清单 8-11。

代码清单8-11 DDLParser的实现

```
protected val CREATE = Keyword("CREATE")
protected val TEMPORARY = Keyword("TEMPORARY")
protected val TABLE = Keyword("TABLE")
protected val USING = Keyword("USING")
protected val OPTIONS = Keyword("OPTIONS")

protected val reservedWords =
    this.getClass
        .getMethods
        .filter(_.getReturnType == classOf[Keyword])
        .map(_.invoke(this).asInstanceOf[Keyword].str)

override val lexical = new SqlLexical(reservedWords)

protected lazy val ddl: Parser[LogicalPlan] = createTable

protected lazy val createTable: Parser[LogicalPlan] =
    CREATE ~ TEMPORARY ~ TABLE ~> ident ~ (USING ~> className) ~ (OPTIONS ~> options) ^^ {
        case tableName ~ provider ~ opts =>
            CreateTableUsing(tableName, provider, opts)
    }
```

从实现可以看出，DDLParser 的模式 ddl 实际是 createTable。createTable 要匹配的模式如下：

```
CREATE TEMPORARY TABLE avroTable
USING org.apache.spark.sql.avro
```

```
OPTIONS (path "../hive/src/test/resources/data/files/episodes.avro")
```

CreateTableUsing 的 run 方法实际实例化 USING 关键字后的 class，并调用它的 createRelation 方法创建 relation，见代码清单 8-12。

代码清单8-12 CreateTableUsing的实现

```
private[sql] case class CreateTableUsing(
    tableName: String,
    provider: String,
    options: Map[String, String]) extends RunnableCommand {

def run(sqlContext: SQLContext) = {
    val loader = Utils.getContextOrSparkClassLoader
    val clazz: Class[_] = try loader.loadClass(provider) catch {
      case cnf: java.lang.ClassNotFoundException =>
        try loader.loadClass(provider + ".DefaultSource") catch {
          case cnf: java.lang.ClassNotFoundException =>
            sys.error(s"Failed to load class for data source: $provider")
        }
    }
    val dataSource = clazz.newInstance().asInstanceOf[org.apache.spark.sql.sources.
       RelationProvider]
    val relation = dataSource.createRelation(sqlContext, options)

    sqlContext.baseRelationToSchemaRDD(relation).registerTempTable(tableName)
    Seq.empty
  }
}
```

8.4.3 SQL 语句解析器 SqlParser

SqlParser 用于解析 SQL 语句，其基本实现见代码清单 8-13。

代码清单8-13 SqlParser的实现

```
protected val SELECT = Keyword("SELECT")
protected val SEMI = Keyword("SEMI")
protected val SQRT = Keyword("SQRT")
protected val STRING = Keyword("STRING")
protected val SUBSTR = Keyword("SUBSTR")
protected val SUBSTRING = Keyword("SUBSTRING")
protected val SUM = Keyword("SUM")
protected val TABLE = Keyword("TABLE")
protected val THEN = Keyword("THEN")
protected val TIMESTAMP = Keyword("TIMESTAMP")
protected val TRUE = Keyword("TRUE")
protected val UNION = Keyword("UNION")
protected val UPPER = Keyword("UPPER")
protected val WHEN = Keyword("WHEN")
protected val WHERE = Keyword("WHERE")
```

```
protected val reservedWords =
    this
        .getClass
        .getMethods
        .filter(_.getReturnType == classOf[Keyword])
        .map(_.invoke(this).asInstanceOf[Keyword].str)

override val lexical = new SqlLexical(reservedWords)
```

SqlParser 支持对 select、insert、ordering、projection 等多种 SQL 的解析，由于其实现的语法晦涩难懂，笔者查阅 Scala 自带的 API 文档，先罗列几个操作符（函数）的含义，如表 8-1 所示。

表 8-1 操作符（函数）含义

操作符（函数）	含 义
~>	按顺序组合的解析器，匹配后只保留右边。如：A~>B 则保留 B
~	按顺序组合的解析器，用于匹配。如：A~>B 则保证 A 必须在 B 左边
^^^	组合的解析器，用于将成功的结果替换原值
^^	组合的解析器，匹配后将右边返回。如：A^^B 则返回 B

我们首先看看 SqlParser 的 start 定义，它可以将 UNION ALL 语句替换为 Union；将 INTERSECT 关键词替换为 Intersect；将 EXCEPT 替换为 Except；将 UNION DISTINCT 替换为 Distinct。其实现见代码清单 8-14。

代码清单8-14 SqlParser的start定义

```
protected lazy val start: Parser[LogicalPlan] =
    ( select *
        ( UNION ~ ALL       ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Union(q1, q2) }
        | INTERSECT         ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Intersect(q1, q2) }
        | EXCEPT            ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Except(q1, q2)}
        | UNION ~ DISTINCT.? ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Distinct(Union(q1,
            q2)) }
        )
    | insert
    )
```

以 SqlParser 的 select 方法为例，理解 SqlParser 的实现，见代码清单 8-15。

代码清单8-15 SqlParser的select代码

```
protected lazy val select: Parser[LogicalPlan] =
    SELECT ~> DISTINCT.? ~
        repsep(projection, ",") ~
        (FROM   ~> relations).? ~
        (WHERE  ~> expression).? ~
        (GROUP  ~  BY ~> replsep(expression, ",")).? ~
        (HAVING ~> expression).? ~
```

```
    (ORDER ~ BY ~> ordering).? ~
    (LIMIT ~> expression).? ^^ {
        case d ~ p ~ r ~ f ~ g ~ h ~ o ~ l  =>
            val base = r.getOrElse(NoRelation)
            val withFilter = f.map(Filter(_, base)).getOrElse(base)
            val withProjection = g
                .map(Aggregate(_, assignAliases(p), withFilter))
                .getOrElse(Project(assignAliases(p), withFilter))
            val withDistinct = d.map(_ => Distinct(withProjection)).getOrElse
               (withProjection)
            val withHaving = h.map(Filter(_, withDistinct)).getOrElse
               (withDistinct)
            val withOrder = o.map(Sort(_, withHaving)).getOrElse(withHaving)
            val withLimit = l.map(Limit(_, withOrder)).getOrElse(withOrder)
            withLimit
    }
```

select 的代码实现：

1）匹配 SELECT 语句，获取 DISTINCT 语句、投影字段 projection、表 relations、WHERE 后的表达式、GROUP BY 后的表达式、HAVING 后的表达式、排序字段 ordering、LIMIT 后的表达式。

2）依次将匹配的字符串层层封装为 Filter、Aggregate、Project、Distinct、Sort、Limit（这里说的 Filter、Aggregate、Project、Distinct、Sort、Limit 正是图 8-7 展示的继承体系中的各种 TreeNode），最终形成一棵 LogicalPlan 的 Tree。

经过对 SqlParser 源码的阅读，我们知道 Spark 目前不支持 delete、update 等操作。

我们抽几个具体的 TreeNode 来看看其实现。Join 是最常用的 TreeNode 之一，包括左外连接、右外连接、全外连接、笛卡尔积等，见代码清单 8-16。

代码清单8-16 basicOperators.scala中实现Join操作的代码

```
case class Join(
    left: LogicalPlan,
    right: LogicalPlan,
    joinType: JoinType,
    condition: Option[Expression]) extends BinaryNode {

    override def output = {
        joinType match {
            case LeftSemi =>
                left.output
            case LeftOuter =>
                left.output ++ right.output.map(_.withNullability(true))
            case RightOuter =>
                left.output.map(_.withNullability(true)) ++ right.output
            case FullOuter =>
                left.output.map(_.withNullability(true)) ++ right.output.map
                    (_.withNullability(true))
            case _ =>
```

```
                left.output ++ right.output
        }
    }
```

8.4.4 Spark 代理解析器 SparkSQLParser

SparkSQLParser 用于代理 SqlParser 对 AS、CACHE、SET、LAZY、TABLE、UNCACHE 这些关键字的处理，见代码清单 8-17。

代码清单8-17 SparkSQLParser的实现

```
private[sql] class SparkSQLParser(fallback: String => LogicalPlan) extends
    AbstractSparkSQLParser {

    protected val AS      = Keyword("AS")
    protected val CACHE   = Keyword("CACHE")
    protected val LAZY    = Keyword("LAZY")
    protected val SET     = Keyword("SET")
    protected val TABLE   = Keyword("TABLE")
    protected val UNCACHE = Keyword("UNCACHE")

    protected implicit def asParser(k: Keyword): Parser[String] =
        lexical.allCaseVersions(k.str).map(x => x : Parser[String]).reduce(_ | _)

    private val reservedWords: Seq[String] =
        this
          .getClass
          .getMethods
          .filter(_.getReturnType == classOf[Keyword])
          .map(_.invoke(this).asInstanceOf[Keyword].str)

    override val lexical = new SqlLexical(reservedWords)

    override protected lazy val start: Parser[LogicalPlan] = cache | uncache | set | others
```

SparkSQLParser 的 start 定义实际是分别匹配 cache、uncache、set 或者 others。

以 SparkSQLParser 的 set 方法为例，看看其实现。set 是通过内部类 SetCommandParser 来实现将 set 语句后的 k = v 匹配封装为 Some(k ~ v)，之后再封装为图 8-7 中 TreeNode 的子类 SetCommand，见代码清单 8-18。

代码清单8-18 SparkSQLParser中SetCommandParser的实现

```
private object SetCommandParser extends RegexParsers {
    private val key: Parser[String] = "(?m)[^=]+".r

    private val value: Parser[String] = "(?m).*$".r

    private val pair: Parser[LogicalPlan] =
        (key ~ ("=".r ~> value).?).? ^^ {
          case None => SetCommand(None)
          case Some(k ~ v) => SetCommand(Some(k.trim -> v.map(_.trim)))
```

```
        }

    def apply(input: String): LogicalPlan = parseAll(pair, input) match {
        case Success(plan, _) => plan
        case x => sys.error(x.toString)
    }
}

private lazy val set: Parser[LogicalPlan] =
    SET ~> restInput ^^ {
        case input => SetCommandParser(input)
    }
```

8.5 Rule 和 RuleExecutor

根据前面内容的介绍，我们知道 SQL 语句经过 Parser 阶段的处理，转换为 Unresolved LogicalPlan 的一棵树，Analyzer 和 optimizer 将会对 LogicalPlan 的这棵树施加各种分析与优化操作，这些分析与优化操作实际就是一系列的 Rule。Rule 是一个抽象类，让我们看看它的实现，见代码清单 8-19。

代码清单8-19 Rule的抽象定义

```
abstract class Rule[TreeType <: TreeNode[_]] extends Logging {

    val ruleName: String = {
        val className = getClass.getName
        if (className endsWith "$") className.dropRight(1) else className
    }

    def apply(plan: TreeType): TreeType
}
```

RuleExecutor 用于执行 Rule，见代码清单 8-20。RuleExecutor 中的一些概念：

- Strategy：执行策略，即执行 Rule 的最大次数。
- Once：只执行一次 Rule 的策略。
- FixedPoint：达到 FixedPoint 指定次数或前后两次的树结构没变化时停止操作的策略。
- Batch：一批 Rule 及其相应的执行策略。

代码清单8-20 RuleExecutor的抽象定义

```
abstract class RuleExecutor[TreeType <: TreeNode[_]] extends Logging {

    abstract class Strategy { def maxIterations: Int }

    case object Once extends Strategy { val maxIterations = 1 }

    case class FixedPoint(maxIterations: Int) extends Strategy
```

```
protected case class Batch(name: String, strategy: Strategy, rules: Rule[TreeType]*)

protected val batches: Seq[Batch]

def apply(plan: TreeType): TreeType = {
    var curPlan = plan

    batches.foreach { batch =>
        val batchStartPlan = curPlan
        var iteration = 1
        var lastPlan = curPlan
        var continue = true

        while (continue) {
            curPlan = batch.rules.foldLeft(curPlan) {
                case (plan, rule) =>
                    val result = rule(plan)
                    if (!result.fastEquals(plan)) {
                        logTrace(
                            s"""
                                |=== Applying Rule ${rule.ruleName} ===
                                |${sideBySide(plan.treeString, result.treeString).
                                    mkString("\n")}
                            """.stripMargin)
                    }

                    result
            }
            iteration += 1
            if (iteration > batch.strategy.maxIterations) {
                if (iteration != 2) {
                    logInfo(s"Max iterations (${iteration - 1}) reached for batch
                        ${batch.name}")
                }
                continue = false
            }

        if (curPlan.fastEquals(lastPlan)) {
            logTrace(
                s"Fixed point reached for batch ${batch.name} after ${iteration - 1}
                    iterations.")
            continue = false
        }
        lastPlan = curPlan
    }

    if (!batchStartPlan.fastEquals(curPlan)) {
        logDebug(
            s"""
            |=== Result of Batch ${batch.name} ===
            |${sideBySide(plan.treeString, curPlan.treeString).mkString("\n")}
        """.stripMargin)
    } else {
```

```
                logTrace(s"Batch ${batch.name} has no effect.")
                }
            }

            curPlan
        }
    }
```

RuleExecutor 的 apply 方法的功能描述：通过遍历，取出 batches 里缓存的每个 Batch；再遍历每个 Batch 中的 Rule，按照 Strategy 指定的次数应用到 TreeNode。如果给 TreeNode 应用 Rule 后的 TreeNode 与之前相同，会退出当前 Batch。

RuleExecutor 的继承体系，如图 8-9 所示。

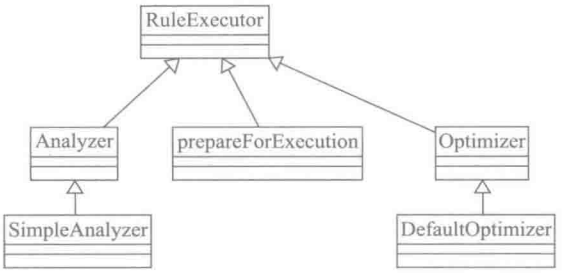


图 8-9 RuleExecutor 的继承体系

8.6 Analyzer 与 Optimizer 的设计与实现

Analyzer 将 Unresolved LogicalPlan 与数据字典（catalog）进行绑定，生成 Resolved LogicalPlan。

Optimizer 对 Resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan。

在 SQLContext 中使用 Analyzer 与 Optimizer 的唯一入口是如下代码中的 executePlan。

```
protected[sql] def executePlan(plan: LogicalPlan): this.QueryExecution =
  new this.QueryExecution { val logical = plan }
```

executePlan 方法创建 QueryExecution 的匿名实现类实例，QueryExecution 中通过懒执行的方式使用 Analyzer、Optimizer、SparkPlanner、prepareForExecution，见代码清单 8-21。

代码清单8-21 SQLContext中QueryExecution的实现

```
@DeveloperApi
protected abstract class QueryExecution {
    def logical: LogicalPlan

    lazy val analyzed = ExtractPythonUdfs(analyzer(logical))
    lazy val withCachedData = useCachedData(analyzed)
    lazy val optimizedPlan = optimizer(withCachedData)

    lazy val sparkPlan = {
        SparkPlan.currentContext.set(self)
        planner(optimizedPlan).next()
    }
    lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)
```

```
    lazy val toRdd: RDD[Row] = executedPlan.execute()

    protected def stringOrError[A](f: => A): String =
        try f.toString catch { case e: Throwable => e.toString }

    def simpleString: String =
        s"""== Physical Plan ==
           |${stringOrError(executedPlan)}
        """.stripMargin.trim

    override def toString: String =
        s"""== Parsed Logical Plan ==
            |${stringOrError(logical)}
            |== Analyzed Logical Plan ==
            |${stringOrError(analyzed)}
            |== Optimized Logical Plan ==
            |${stringOrError(optimizedPlan)}
            |== Physical Plan ==
            |${stringOrError(executedPlan)}
            |Code Generation: ${stringOrError(executedPlan.codegenEnabled)}
            |== RDD ==
        """.stripMargin.trim
}
```

toRdd 也是懒执行的，真正促使它发生转换的地方是在任务提交阶段执行代码清单 5-19 中的 runJob 方法时，首先调用 RDD 的 partitions 触发的。根据代码清单 5-11、代码清单 5-12 及代码清单 5-13，我们知道最终会调用 SchemaRDD 的 dependencies 方法。实际是代码清单 5-28 中 RDD 的 dependencies 方法，那么最终调用 SchemaRDD 的 getDependencies 方法，代码如下。

```
override protected def getDependencies: Seq[Dependency[_]] = {
    schema // Force reification of the schema so it is available on executors.
    List(new OneToOneDependency(queryExecution.toRdd))
}
```

调用 QueryExecution 的 toRdd 方法最终引发了 Analyzer、Optimizer 将一系列 Rule 应用到 LogicalPlan。

QueryExecution 中调用的 useCachedData 方法实际是与 SQLContext 同在包 org.apache.spark.sql 下的 CacheManager 的 useCachedData，用于将 LogicalPlan 的树段替换为缓存中的，见代码清单 8-22。树段替换就是用 TreeNode 的 transformDown 方法来完成的。

代码清单8-22 CacheManager的useCachedData方法

```
private[sql] def useCachedData(plan: LogicalPlan): LogicalPlan = {
    plan transformDown {
        case currentFragment =>
            lookupCachedData(currentFragment)
                .map(_.cachedRepresentation.withOutput(currentFragment.output))
                .getOrElse(currentFragment)
    }
}
```

```
private[sql] def lookupCachedData(plan: LogicalPlan): Option[CachedData] = readLock {
    cachedData.find(cd => plan.sameResult(cd.plan))
}
```

8.6.1 语法分析器 Analyzer

QueryExecution 中的 analyzer(logical) 语句实际调用了 Analyzer 的父类 RuleExecutor 的 apply 方法来应用自己的 batches，见代码清单 8-23。Analyzer 的 FixedPoint 目前是固定的 100，从其注释看出将来会用参数传递值。通过继承 Analyzer 并且覆盖 extendedRules 用于提供额外的 Rule。

代码清单8-23 Analyzer的实现

```
val fixedPoint = FixedPoint(100)

val extendedRules: Seq[Rule[LogicalPlan]] = Nil

lazy val batches: Seq[Batch] = Seq(
    Batch("MultiInstanceRelations", Once,
        NewRelationInstances),
    Batch("Resolution", fixedPoint,
        ResolveReferences ::
        ResolveRelations ::
        ResolveSortReferences ::
        NewRelationInstances ::
        ImplicitGenerate ::
        StarExpansion ::
        ResolveFunctions ::
        GlobalAggregates ::
        UnresolvedHavingClauseAttributes ::
        TrimGroupingAliases ::
        typeCoercionRules ++
        extendedRules : _*),
    Batch("Check Analysis", Once,
        CheckResolution,
        CheckAggregation),
    Batch("AnalysisOperators", fixedPoint,
        EliminateAnalysisOperators)
)
```

Analyzer 中已经内置了很多 Rule，包括：ResolveReferences、ResolveRelations、StarExpansion 等。经过 Analyzer 的加工，Unresolved LogicalPlan 已经成为 Resolved LogicalPlan。

以 ResolveRelations 为例来大致了解 Analyzer。ResolveRelations 用来把 LogicalPlan 中匹配 UnresolvedRelation 的部分，替换为字典表 Catalog 中注册的 LogicalPlan，见代码清单 8-24。

代码清单8-24 Analyzer中ResolveRelations的apply方法

```
object ResolveRelations extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
```

```
        case i @ InsertIntoTable(UnresolvedRelation(tableIdentifier, alias), _, _, _) =>
            i.copy(
                table = EliminateAnalysisOperators(catalog.lookupRelation(table-
                    Identifier, alias)))
        case UnresolvedRelation(tableIdentifier, alias) =>
            catalog.lookupRelation(tableIdentifier, alias)
    }
}
```

8.6.2 优化器 Optimizer

Optimizer 与 Analyzer 一样，也是通过父类 RuleExecutor 的 apply 方法来应用自己的 batches，Optimizer 的默认实现是 DefaultOptimizer，见代码清单 8-25。DefaultOptimizer 也内置了很多的 Rule，比如 NullPropagation、ConstantFolding 等。经过 Optimizer 对 Resolved LogicalPlan 的优化，生成 Optimized LogicalPlan。

代码清单8-25 Optimizer的实现

```
abstract class Optimizer extends RuleExecutor[LogicalPlan]

object DefaultOptimizer extends Optimizer {
    val batches =
        Batch("Combine Limits", FixedPoint(100),
            CombineLimits) ::
        Batch("ConstantFolding", FixedPoint(100),
            NullPropagation,
            ConstantFolding,
            LikeSimplification,
            BooleanSimplification,
            SimplifyFilters,
            SimplifyCasts,
            SimplifyCaseConversionExpressions,
            OptimizeIn) ::
        Batch("Decimal Optimizations", FixedPoint(100),
            DecimalAggregates) ::
        Batch("Filter Pushdown", FixedPoint(100),
            UnionPushdown,
            CombineFilters,
            PushPredicateThroughProject,
            PushPredicateThroughJoin,
            ColumnPruning) :: Nil
}
```

无论是 Analyzer 中内置的 Rule，还是 DefaultOptimizer 内置的 Rule，将 Rule 应用到 LogicalPlan 都是通过 TreeNode 里的 transform 系列函数。以 SimplifyFilters 为例，它所做的优化包括：

- 如果过滤条件总是等于 true，则删除它，即此过滤条件不起作用。
- 如果过滤条件总是等于 null 或者 false，将输入替换为空的 relation，即将输入全部滤除。

从 SimplifyFilters 的实现不难看出，它正是将自身规则作为参数传递给 transform 函数的，见代码清单 8-26。

代码清单8-26 Optimizer中SimplifyFilters的实现

```
object SimplifyFilters extends Rule[LogicalPlan] {
    def apply(plan: LogicalPlan): LogicalPlan = plan transform {
        case Filter(Literal(true, BooleanType), child) => child
        case Filter(Literal(null, _), child) => LocalRelation(child.output, data = Seq.
            empty)
        case Filter(Literal(false, BooleanType), child) => LocalRelation(child.output,
            data = Seq.empty)
    }
}
```

8.7 生成物理执行计划

经过了 SqlParser、Analyzer、Optimizer 的处理，生成的逻辑执行计划无法被当做一般的 Job 来处理，为了能够将逻辑执行计划按照其他 Job 一样对待，需要将逻辑执行计划转变为物理执行计划了。

物理执行计划 SparkPlan

首先使用 SparkPlanner，实际使用 QueryPlanner 的 apply 方法，见代码清单 8-27。

代码清单8-27 QueryPlanner的apply方法

```
def apply(plan: LogicalPlan): Iterator[PhysicalPlan] = {
    val iter = strategies.view.flatMap(_(plan)).toIterator
    assert(iter.hasNext, s"No plan for $plan")
    iter
}
```

SparkPlanner 中 strategies 的定义见代码清单 8-28。

代码清单8-28 SparkPlanner中strategies的定义

```
def strategies: Seq[Strategy] =
    extraStrategies ++ (
    CommandStrategy(self) ::
    DataSourceStrategy ::
    TakeOrdered ::
    HashAggregation ::
    LeftSemiJoin ::
    HashJoin ::
    InMemoryScans ::
    ParquetOperations ::
    BasicOperators ::
    CartesianProduct ::
    BroadcastNestedLoopJoin :: Nil)
```

每个 Strategy 都实现了 apply 方法。这些 Strategy 中，最常用的要算 BasicOperators 了，其实现见代码清单 8-29。可以看到它对最常用的 SQL 关键字都做了处理。每个处理的分支，都会先调用 planLater 方法，planLater 方法给 child 节点的 LogicalPlan 应用 SparkPlanner，见代码清单 8-30。于是就形成了迭代处理的过程，最终实现将整棵 LogicalPlan 树使用 SparkPlanner 来完成转换。

代码清单8-29 SparkStrategies中BasicOperators的实现

```
object BasicOperators extends Strategy {
    def numPartitions = self.numPartitions

    def apply(plan: LogicalPlan): Seq[SparkPlan] = plan match {
        case logical.Distinct(child) =>
            execution.Distinct(partial = false,
                execution.Distinct(partial = true, planLater(child))) :: Nil

        case logical.Sort(sortExprs, child) if sqlContext.externalSortEnabled =>
            execution.ExternalSort(sortExprs, global = true, planLater(child)):: Nil
        case logical.Sort(sortExprs, child) =>
            execution.Sort(sortExprs, global = true, planLater(child)):: Nil

        case logical.SortPartitions(sortExprs, child) =>
            execution.Sort(sortExprs, global = false, planLater(child)) :: Nil
        case logical.Project(projectList, child) =>
            execution.Project(projectList, planLater(child)) :: Nil
        case logical.Filter(condition, child) =>
            execution.Filter(condition, planLater(child)) :: Nil
        case logical.Aggregate(group, agg, child) =>
            execution.Aggregate(partial = false, group, agg, planLater(child)) :: Nil
        case logical.Sample(fraction, withReplacement, seed, child) =>
            execution.Sample(fraction, withReplacement, seed, planLater(child)) :: Nil
        case SparkLogicalPlan(alreadyPlanned) => alreadyPlanned :: Nil
        case logical.LocalRelation(output, data) =>
            val nPartitions = if (data.isEmpty) 1 else numPartitions
            PhysicalRDD(
                output,
                RDDConversions.productToRowRdd(sparkContext.parallelize(data,
                    nPartitions),
                    StructType.fromAttributes(output))) :: Nil
        case logical.Limit(IntegerLiteral(limit), child) =>
            execution.Limit(limit, planLater(child)) :: Nil
        case Unions(unionChildren) =>
            execution.Union(unionChildren.map(planLater)) :: Nil
        case logical.Except(left, right) =>
            execution.Except(planLater(left), planLater(right)) :: Nil
        case logical.Intersect(left, right) =>
            execution.Intersect(planLater(left), planLater(right)) :: Nil
        case logical.Generate(generator, join, outer, _, child) =>
            execution.Generate(generator, join = join, outer = outer, planLater(child))
                :: Nil
        case logical.NoRelation =>
```

```
            execution.PhysicalRDD(Nil, singleRowRdd) :: Nil
        case logical.Repartition(expressions, child) =>
            execution.Exchange(HashPartitioning(expressions, numPartitions),
                planLater(child)) :: Nil
        case e @ EvaluatePython(udf, child, _) =>
            BatchPythonEvaluation(udf, e.output, planLater(child)) :: Nil
        case LogicalRDD(output, rdd) => PhysicalRDD(output, rdd) :: Nil
        case _ => Nil
    }
}
```

代码清单8-30 QueryPlanner的planLater方法

```
protected def planLater(plan: LogicalPlan) = apply(plan).next()
```

BasicOperators 方法实际就是将 logical.XXX 转换为 execution.XXX，将 LogicalRDD 转换为 PhysicalRDD。

prepareForExecution 在执行前做准备工作，它的原理与 Analyzer 和 Optimizer 一样，不再赘述。

8.8 执行物理执行计划

经过分析、优化、逻辑计划转换为物理计划的懒执行，最终调用 SparkPlan 的 execute 方法执行物理计划。以 execution.Project 为例，其 execute 方法见代码清单 8-31。

代码清单8-31 Project及其excute方法

```
@DeveloperApi
case class Project(projectList: Seq[NamedExpression], child: SparkPlan) extends
    UnaryNode {
    override def output = projectList.map(_.toAttribute)

    @transient lazy val buildProjection = newMutableProjection(projectList, child.output)

    def execute() = child.execute().mapPartitions { iter =>
        val resuableProjection = buildProjection()
        iter.map(resuableProjection)
    }
}
```

Project 的 execute 方法的执行步骤：

1）调用 child 的 execute 方法，以保证将要投影的输入数据已经经过处理。

2）调用 SparkPlan 的 newMutableProjection 来处理其投影操作，newMutableProjection 的实现见代码清单 8-32。

代码清单8-32 SparkPlan的newMutableProjection方法

```
protected def newMutableProjection(
        expressions: Seq[Expression],
        inputSchema: Seq[Attribute]): () => MutableProjection = {
```

```
        log.debug(
            s"Creating MutableProj: $expressions, inputSchema: $inputSchema, codegen:
                $codegenEnabled")
        if(codegenEnabled) {
            GenerateMutableProjection(expressions, inputSchema)
        } else {
            () => new InterpretedMutableProjection(expressions, inputSchema)
        }
    }
```

newMutableProjection 默认情况下使用 InterpretedMutableProjection 处理投影，其实现见代码清单 8-33。BindReferences.bindReference 再次使用了 transform 方法，用于给表达式绑定引用，比如将 List(name#1) 替换为 List(input[1])。最终的投影由 InterpretedMutableProjection 的 apply 方法来完成。BindReferences.bindReference 的实现见代码清单 8-34。

代码清单8-33 Projection.scala中的InterpretedMutableProjection实现

```
case class InterpretedMutableProjection(expressions: Seq[Expression]) extends
MutableProjection {
    def this(expressions: Seq[Expression], inputSchema: Seq[Attribute]) =
        this(expressions.map(BindReferences.bindReference(_, inputSchema)))

    private[this] val exprArray = expressions.toArray
    private[this] var mutableRow: MutableRow = new GenericMutableRow(exprArray.size)
    def currentValue: Row = mutableRow

    override def target(row: MutableRow): MutableProjection = {
        mutableRow = row
        this
    }

    override def apply(input: Row): Row = {
        var i = 0
        while (i < exprArray.length) {
            mutableRow(i) = exprArray(i).eval(input)
            i += 1
        }
        mutableRow
    }
}
```

代码清单8-34 BindReferences bindReference的实现

```
object BindReferences extends Logging {

    def bindReference[A <: Expression](
        expression: A,
        input: Seq[Attribute],
        allowFailures: Boolean = false): A = {
    expression.transform { case a: AttributeReference =>
        attachTree(a, "Binding attribute") {
```

```
            val ordinal = input.indexWhere(_.exprId == a.exprId)
            if (ordinal == -1) {
                if (allowFailures) {
                    a
                } else {
                    sys.error(s"Couldn't find $a in ${input.mkString("[", ",", "]")}")
                }
            } else {
                BoundReference(ordinal, a.dataType, a.nullable)
            }
        }
    }.asInstanceOf[A] // Kind of a hack, but safe.  TODO: Tighten return type when possible.
}
```

再以 execution.Filter 为例，其 execute 方法见代码清单 8-35。

Filter 的 execute 方法的执行步骤如下：

1）调用 child 的 execute 方法，以保证将要过滤的输入数据已经经过处理。

2）调用 SparkPlan 的 newPredicate 来处理其过滤操作，newPredicate 的实现见代码清单 8-36。

代码清单8-35 Filter及其excute方法

```
@DeveloperApi
case class Filter(condition: Expression, child: SparkPlan) extends UnaryNode {
    override def output = child.output

    @transient lazy val conditionEvaluator = newPredicate(condition, child.output)

    def execute() = child.execute().mapPartitions { iter =>
        iter.filter(conditionEvaluator)
    }
}
```

newPredicate 默认使用 InterpretedPredicate 处理过滤，其实现见代码清单 8-37。BindReferences.bindReference 方法在此处将 ((age#0 >= 13) && (age#0 <= 19)) 转换为 [(input[0] >= 13), (input[0] <= 19)]。最终的过滤由 InterpretedPredicate 的第二个 apply 方法来完成。

代码清单8-36 SparkPlan的newPredicate方法

```
protected def newPredicate(
        expression: Expression, inputSchema: Seq[Attribute]): (Row) => Boolean = {
    if (codegenEnabled) {
        GeneratePredicate(expression, inputSchema)
    } else {
        InterpretedPredicate(expression, inputSchema)
    }
}
```

代码清单8-37 InterpretedPredicate的实现

```
object InterpretedPredicate {
    def apply(expression: Expression, inputSchema: Seq[Attribute]): (Row => Boolean) =
        apply(BindReferences.bindReference(expression, inputSchema))

    def apply(expression: Expression): (Row => Boolean) = {
        (r: Row) => expression.eval(r).asInstanceOf[Boolean]
    }
}
```

execution.Project 和 execution.Filter 都有 child，是不是所有的 SparkPlan 的子类都有 child 呢？

当然也有例外，比如 execution.PhysicalRDD 是没有 child 的，因为 execution.PhysicalRDD 一般是作为最底层的 LogicalPlan 节点，其代码实现如下。

```
case class PhysicalRDD(output: Seq[Attribute], rdd: RDD[Row]) extends LeafNode {
    override def execute() = rdd
}
```

基于整个 SparkPlan 的 execute 体系，就可以保证先执行低层（孩子）的 SparkPlan 的转换动作，然后才执行当前 SparkPlan 的转换动作，最终完成 SQL 的执行。

8.9 Hive

Hive 最初是基于 Hadoop 开发的一个对 HDFS 上的数据进行分析与查询的工具，它可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，也可以将 SQL 语句转换为 MapReduce 任务运行。Hive 的学习成本低，大大降低了分析人员对大规模数据集合的分析门槛。关于 Hive 技术的具体内容，请读者自行查阅相关资料。

Spark 中提供的 Hive 会被转换为 Spark 中的任务，而不是 MapReduce 任务。

HiveContext 是处理 Hive 的上下文环境，HiveContext 继承了 SQLContext，可见 Hive 在架构上是基于 Spark SQL 的。HiveContext 覆盖了 SQLContext 中的一些实现，包括：

- Catalog：由 HiveMetastoreCatalog 替换了 SimpleCatalog，主要是针对 Hive 的元数据缓存做了相应的字典表实现。
- Analyzer：被匿名实现类覆盖。
- SparkPlanner：被匿名实现类覆盖。
- ExtendedHiveQlParser：代替 SqlParser。

Hive SQL 的执行过程与 Spark SQL 的执行过程类似，可以用图 8-10 表示。

8.9.1 Hive SQL 语法解析器

HiveContext 覆盖了 SQLContext 对 SQL 解析的入口，DDL 的解析依然交给 DDLParser，但是 Hive SQL 则交由 HiveQl 的 parseSql 方法处理，见代码清单 8-38。

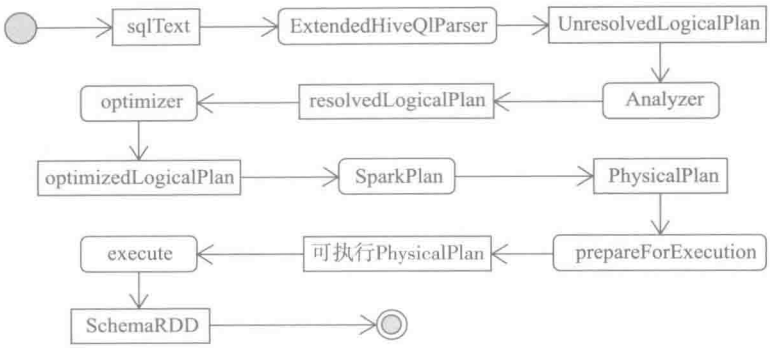


图 8-10 Hive SQL 的执行过程

代码清单8-38 HiveContext解析sql的入口

```
override def sql(sqlText: String): SchemaRDD = {
    if (dialect == "sql") {
        super.sql(sqlText)
    } else if (dialect == "hiveql") {
        new SchemaRDD(this, ddlParser(sqlText).getOrElse(HiveQl.parseSql(sqlText)))
    }  else {
        sys.error(s"Unsupported SQL dialect: $dialect.  Try 'sql' or 'hiveql'")
    }
}
```

HiveQl 的 parseSql 方法使用 ExtendedHiveQlParser 解析 Hive SQL，依然由 SparkSQLParser 作为代理，见代码清单 8-39。

代码清单8-39 HiveQl的parseSql方法

```
protected val hqlParser = {
    val fallback = new ExtendedHiveQlParser
    new SparkSQLParser(fallback(_))
}

def parseSql(sql: String): LogicalPlan = hqlParser(sql)
```

ExtendedHiveQlParser 用于扩展对 Hive SQL 的解析，其实现见代码清单 8-40。

代码清单8-40 ExtendedHiveQlParser的实现

```
private[hive] class ExtendedHiveQlParser extends AbstractSparkSQLParser {
    protected implicit def asParser(k: Keyword): Parser[String] =
        lexical.allCaseVersions(k.str).map(x => x : Parser[String]).reduce(_ | _)

    protected val ADD  = Keyword("ADD")
    protected val DFS  = Keyword("DFS")
    protected val FILE = Keyword("FILE")
    protected val JAR  = Keyword("JAR")

    private val reservedWords =
```

```
        this
            .getClass
            .getMethods
            .filter(_.getReturnType == classOf[Keyword])
            .map(_.invoke(this).asInstanceOf[Keyword].str)

    override val lexical = new SqlLexical(reservedWords)
```

我们首先看看 ExtendedHiveQlParser 的 start 定义，可见解析的 SQL 匹配需要 ADD、DFS、FILE、JAR 等语法，见代码清单 8-41。

代码清单8-41 ExtendedHiveQlParser的start定义

```
protected lazy val start: Parser[LogicalPlan] = dfs | addJar | addFile | hiveQl

protected lazy val hiveQl: Parser[LogicalPlan] =
    restInput ^^ {
        case statement => HiveQl.createPlan(statement.trim)
    }

protected lazy val dfs: Parser[LogicalPlan] =
    DFS ~> wholeInput ^^ {
        case command => NativeCommand(command.trim)
    }

private lazy val addFile: Parser[LogicalPlan] =
    ADD ~ FILE ~> restInput ^^ {
        case input => AddFile(input.trim)
    }

private lazy val addJar: Parser[LogicalPlan] =
    ADD ~ JAR ~> restInput ^^ {
        case input => AddJar(input.trim)
    }
```

8.9.2 Hive SQL 元数据分析

为了分析 Hive SQL 的元数据，匿名实现的 Analyzer 覆盖了 extendedRules，提供额外的 Rule，见代码清单 8-42。

代码清单8-42 HiveContext的analyzer定义

```
@transient
override protected[sql] lazy val analyzer =
    new Analyzer(catalog, functionRegistry, caseSensitive = false) {
        override val extendedRules =
            catalog.CreateTables ::
            catalog.PreInsertionCasts ::
            ExtractPythonUdfs ::
            Nil
    }
```

8.9.3 Hive SQL 物理执行计划

为生成 Hive SQL 的物理执行计划，也通过匿名实现的方式，扩展 SparkPlanner 中的执行策略，见代码清单 8-43。

代码清单8-43 HiveContext的hivePlanner定义

```
@transient
val hivePlanner = new SparkPlanner with HiveStrategies {
    val hiveContext = self

    override def strategies: Seq[Strategy] = extraStrategies ++ Seq(
        DataSourceStrategy,
        CommandStrategy(self),
        HiveCommandStrategy(self),
        TakeOrdered,
        ParquetOperations,
        InMemoryScans,
        ParquetConversion, // Must be before HiveTableScans
        HiveTableScans,
        DataSinks,
        Scripts,
        HashAggregation,
        LeftSemiJoin,
        HashJoin,
        BasicOperators,
        CartesianProduct,
        BroadcastNestedLoopJoin
      )
}
```

8.10 应用举例：JavaSparkSQL

Spark 的源码自带了使用 Spark SQL 的例子 JavaSparkSQL，我们就用它来了解 Spark SQL 功能的使用。JavaSparkSQL 中使用构建好的 JavaSparkContext 作为参数构建 JavaSQLContext，代码实现如下。

```
SparkConf sparkConf = new SparkConf().setAppName("JavaSparkSQL").setMaster("local");
JavaSparkContext ctx = new JavaSparkContext(sparkConf);
JavaSQLContext sqlCtx = new JavaSQLContext(ctx);
```

其实质是使用 SparkContext 作为参数构建 SQLContext。

```
class JavaSQLContext(val sqlContext: SQLContext) extends UDFRegistration {
def this(sparkContext: JavaSparkContext) = this(new SQLContext(spark-
   Context.sc))
```

JavaSparkSQL 接着调用 SparkContext.textFile 加载文本文件生成 HadoopRDD，然后调用 map 方法生成 MappedRDD，见代码清单 8-44。

代码清单8-44 JavaSparkSQL例子

```
JavaRDD<Person> people = ctx.textFile("src/main/resources/people.txt").map(
    new Function<String, Person>() {
        @Override
        public Person call(String line) {
            String[] parts = line.split(",");

            Person person = new Person();
            person.setName(parts[0]);
            person.setAge(Integer.parseInt(parts[1].trim()));

            return person;
        }
    });
```

people.txt 文本文件中的内容如下：

```
Michael, 29
Andy, 30
Justin, 19
```

JavaSparkSQL 接着将 schema 应用到 RDD 和 Person.class，代码如下。

```
JavaSchemaRDD schemaPeople = sqlCtx.applySchema(people, Person.class);
```

applySchema 方法（见代码清单 8-45）主要处理以下工作：

1）调用 getSchema 获取 beanClass 的 schema 信息，即通过 Java 内省机制获得 JavaBean，遍历 JavaBean，返回 AttributeReference 的序列，见代码清单 8-46。AttributeReference 是属性名、Spark 数据类型、是否允许空的三元组。

2）调用 MappedRDD 的 mapPartitions 构造 MapPartitionsRDD。mapPartitions 的实现见代码清单 8-47。

3）创建 LogicalRDD 并封装为 JavaSchemaRDD，JavaSchemaRDD 实际持有了 SchemaRDD，并将 SchemaRDD 封装为 MappedRDD，见代码清单 8-48 和代码清单 8-49。

代码清单8-45 JavaSQLContext的applySchema方法

```
def applySchema(rdd: JavaRDD[_], beanClass: Class[_]): JavaSchemaRDD = {
    val attributeSeq = getSchema(beanClass)
    val className = beanClass.getName
    val rowRdd = rdd.rdd.mapPartitions { iter =>
        // BeanInfo is not serializable so we must rediscover it remotely for each partition.
        val localBeanInfo = Introspector.getBeanInfo(
            Class.forName(className, true, Utils.getContextOrSparkClassLoader))
        val extractors =
            localBeanInfo.getPropertyDescriptors.filterNot(_.getName == "class").map(_.
                getReadMethod)

        iter.map { row =>
            new GenericRow(
            extractors.zip(attributeSeq).map { case (e, attr) =>
```

```
                DataTypeConversions.convertJavaToCatalyst(e.invoke(row), attr.
                    dataType)
            }.toArray[Any]
        ): ScalaRow
    }
  }
  new JavaSchemaRDD(sqlContext, LogicalRDD(attributeSeq, rowRdd)(sqlContext))
}
```

代码清单8-46 JavaSQLContext的getSchema方法

```
protected def getSchema(beanClass: Class[_]): Seq[AttributeReference] = {
  val beanInfo = Introspector.getBeanInfo(beanClass)

  val fields = beanInfo.getPropertyDescriptors.filterNot(_.getName == "class")
  fields.map { property =>
    val (dataType, nullable) = property.getPropertyType match {
      case c: Class[_] if c.isAnnotationPresent(classOf[SQLUserDefinedType]) =>
        (c.getAnnotation(classOf[SQLUserDefinedType]).udt().newInstance(),
          true)
      case c: Class[_] if c == classOf[java.lang.String] =>
        (org.apache.spark.sql.StringType, true)
      case c: Class[_] if c == java.lang.Short.TYPE =>
        (org.apache.spark.sql.ShortType, false)
      case c: Class[_] if c == java.lang.Integer.TYPE =>
        (org.apache.spark.sql.IntegerType, false)
      case c: Class[_] if c == java.lang.Long.TYPE =>
        (org.apache.spark.sql.LongType, false)
      case c: Class[_] if c == java.lang.Double.TYPE =>
        (org.apache.spark.sql.DoubleType, false)
      case c: Class[_] if c == java.lang.Byte.TYPE =>
        (org.apache.spark.sql.ByteType, false)
      case c: Class[_] if c == java.lang.Float.TYPE =>
        (org.apache.spark.sql.FloatType, false)
      case c: Class[_] if c == java.lang.Boolean.TYPE =>
        (org.apache.spark.sql.BooleanType, false)
      // 忽略部分代码
    }
    AttributeReference(property.getName, dataType, nullable)()
  }
}
```

代码清单8-47 RDD的mapPartitions方法

```
def mapPartitions[U: ClassTag](
    f: Iterator[T] => Iterator[U], preservesPartitioning: Boolean = false): RDD[U] = {
  val func = (context: TaskContext, index: Int, iter: Iterator[T]) => f(iter)
  new MapPartitionsRDD(this, sc.clean(func), preservesPartitioning)
}
```

代码清单8-48 LogicalRDD的实现

```
case class LogicalRDD(output: Seq[Attribute], rdd: RDD[Row])(sqlContext: SQLContext)
```

```
    extends LogicalPlan with MultiInstanceRelation {

    def children = Nil

    def newInstance() =
        LogicalRDD(output.map(_.newInstance()), rdd)(sqlContext).asInstanceOf[this.
            type]

    override def sameResult(plan: LogicalPlan) = plan match {
        case LogicalRDD(_, otherRDD) => rdd.id == otherRDD.id
        case _ => false
    }
}
```

代码清单8-49 JavaSchemaRDD的实现

```
class JavaSchemaRDD(
        @transient val sqlContext: SQLContext,
        @transient val baseLogicalPlan: LogicalPlan)
    extends JavaRDDLike[Row, JavaRDD[Row]]
    with SchemaRDDLike {

    private[sql] val baseSchemaRDD = new SchemaRDD(sqlContext, logicalPlan)

    val schemaRDD: SchemaRDD = baseSchemaRDD

    override val classTag = scala.reflect.classTag[Row]

    override def wrapRDD(rdd: RDD[Row]): JavaRDD[Row] = JavaRDD.fromRDD(rdd)

    val rdd = baseSchemaRDD.map(new Row(_))
```


LogicalRDD 继承自 LogicalPlan。

JavaSparkSQL 接着注册临时表 people，代码如下。

```
schemaPeople.registerTempTable("people");
```

registerTempTable 方法，见代码清单 8-50。

代码清单8-50 SchemaRDDLike的registerTempTable方法

```
def registerTempTable(tableName: String): Unit = {
    sqlContext.registerRDDAsTable(baseSchemaRDD, tableName)
}
```

这里用 Catalog 来注册表，给 rdd.queryExecution.logical 注册临时表，见代码清单 8-51。

代码清单8-51 SQLContext的registerRDDAsTable方法

```
def registerRDDAsTable(rdd: SchemaRDD, tableName: String): Unit = {
    catalog.registerTable(Seq(tableName), rdd.queryExecution.logical)
}
```

queryExecution 是通过懒执行方式创建的 QueryExecution，代码如下。

```
@transient
@DeveloperApi
lazy val queryExecution = sqlContext.executePlan(baseLogicalPlan)
```

sqlContext.executePlan 已在 8.6 节详细介绍，由此可知注册到 catalog 的实际是 applySchema 中创建的 LogicalRDD。

回到 JavaSparkSQL，应用程序接着构造 SQL，代码如下。

```
JavaSchemaRDD teenagers = sqlCtx.sql("SELECT name FROM people WHERE age >= 13 AND
    age <= 19");
```

JavaSQLContext 的 sql 方法的实现见代码清单 8-52。

代码清单8-52 JavaSQLContext的sql方法

```
def sql(sqlText: String): JavaSchemaRDD = {
    if (sqlContext.dialect == "sql") {
        new JavaSchemaRDD(sqlContext, sqlContext.parseSql(sqlText))
    } else {
        sys.error(s"Unsupported SQL dialect: $sqlContext.dialect")
    }
}
```

此处调用了 8.4.1 节代码清单 8-8 中的 parseSql 方法，根据之前的源码分析，我们知道 DDLParser 进行解析时不会匹配，只会用 SqlParser 解析，解析完成，返回的 LogicalPlan 的实际类型是 Project，最后将 sqlContext 和 Project 封装为 JavaSchemaRDD。此时，Project 的 projectList 只包括 name，Project 的儿子节点是 Filter；Filter 的 condition 等于 (('age >= 13) && ('age <= 19))，Filter 的儿子节点是 UnresolvedRelation，UnresolvedRelation 的 tableIdentifier 只包括 people，这说明 SQL 查询的 relation 还没有找到，只知道 relation 的表名是 people。此时，debug 视图如图 8-11 所示。

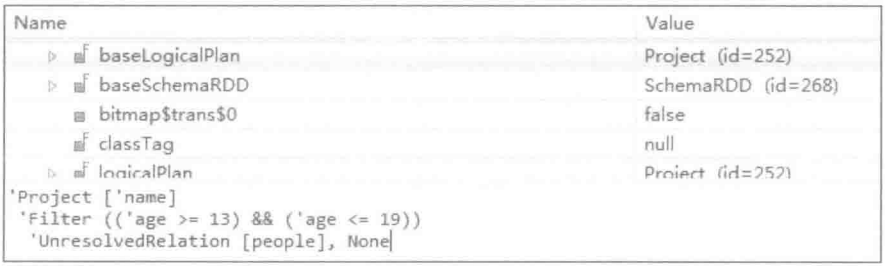


图 8-11 Unresolved LogicalPlan 的 debug 视图

根据 8.6 节的分析，我们知道在 RDD 转换的过程中，通过 Analyzer 中的规则 ResolveRelations 将 UnresolvedRelation 替换为 LogicalRDD。

此时 Project 的样子如图 8-12 所示。

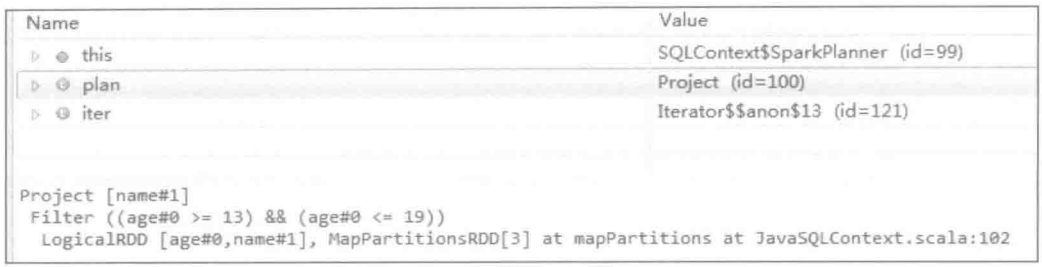


图 8-12 Resolved LogicalPlan 的 debug 视图

根据 8.7.1 节的分析，我们知道 LogicalRDD(output, rdd) 会被转换为 PhysicalRDD(output, rdd)；logical.Filter 被转换为 execution.Filter；logical.Project 被转换为 execution.Project。

此时 Project 的样子如图 8-13 所示。

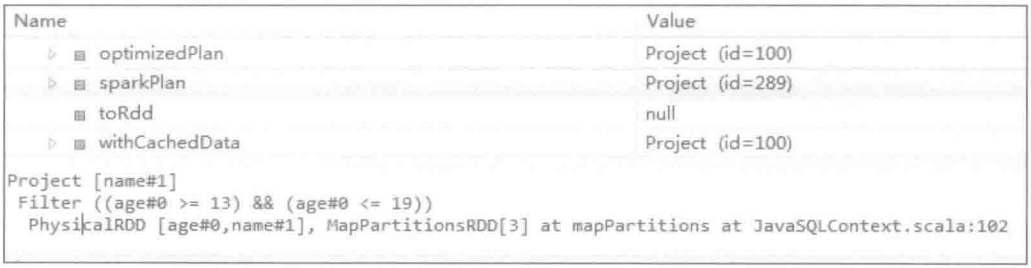


图 8-13 PhysicalPlan 的 debug 视图

回到 JavaSparkSQL，将 JavaSchemaRDD 转换为 MappedRDD，调用 collect 方法运行任务，见代码清单 8-53。

代码清单8-53 JavaSparkSQL调用collect的代码

```
List<String> teenagerNames = teenagers.map(new Function<Row, String>() {
    @Override
    public String call(Row row) {
    return "Name: " + row.getString(0);
    }
}).collect();
```

根据第 6 章的知识，我们知道在迭代计算过程中会调用到 SchemaRDD 的 compute 方法，见代码清单 8-54。

代码清单8-54 SchemaRDD的compute方法

```
override def compute(split: Partition, context: TaskContext): Iterator[Row] =
    firstParent[Row].compute(split, context).map(ScalaReflection.
        convertRowToScala(_, this.schema))
```

在 MapPartitionsRDD 的 compute 方法中，执行了函数 f，见代码清单 8-55。

代码清单8-55 MapPartitionsRDD的compute方法

```
override def compute(split: Partition, context: TaskContext) =
    f(context, split.index, firstParent[T].iterator(split, context))
```

整个 DAG 中一共有 3 个 MapPartitionsRDD，似乎只感觉到代码清单 8-45 中调用 mapPartitions 产生的 MapPartitionsRDD。怎么还会多出两个呢？这两个分别是 8.8 节中介绍的 Project 中的 execute 方法调用 mapPartitions 以及 Filter 的 execute 方法调用 mapPartitions 产生的，见代码清单 8-31 和代码清单 8-35。这两个偏函数分别用于投影和条件筛选操作。而代码清单 8-42 中的偏函数用于将记录转变为 GenericRow。

根据第 5 章的内容知道会回调代码清单 8-44 中的函数 Function，以及回调代码清单 8-53 中的 Function。最终输出结果如下：

```
Name: Justin
```

8.11 小结

Spark 增加对 SQL 的支持是市场决定的，这必然吸引一大批熟悉 SQL 的用户使用 Spark。关系型数据库历史悠久，对于 SQL 的处理机制相当成熟，因此 Spark SQL 对于 SQL 的解析处理过程并没有什么新奇之处。此外，Spark 还用类似的方式实现了对 Hive 的支持，也吸引了很多熟练操作 Hadoop 的工程师。就笔者日常所用的大数据而言，SQL 无疑是最常用的场景。本章最后还以一个例子带读者熟悉 Spark SQL 从编码到提交的整个过程。

第 9 章 Chapter 9

流式计算

流水淘沙不暂停，前波未灭后波生。令人忽忆潇湘渚，回暗迎神三两声。

——刘禹锡

本章导读

传统互联网开发中，经常会有这样一种需求：根据过去一周内的网站浏览数据，统计出 pv、uv 最高的页面用报表或图表展示。在过去，这样一种基本又常见的需求总是先将数据存储到日志或者数据库中，然后每天由一个定时任务进行统计计算后再存储到数据库中，供报表模块查询。用户想要看到过去一周的数据，必须要等到第二天。现在很多业务场景中，越来越多的用户希望实时看到过去 1 小时、10 分钟甚至 10 秒内的统计数据，传统的互联网计算方式显然无法满足。Hadoop 虽然也拥有强大的离线计算能力，但由于其实时性差，因此也无法满足这类需求。

于是 Storm、Spark 等一系列适用于流式数据的实时处理系统诞生了。本章将围绕 Spark Streaming，给大家介绍大数据最有魅力的应用之一——流式计算。

9.1 Spark Streaming 总体设计

Spark Streaming 类似于 Apache Storm。根据 Spark 官方文档介绍，Spark Streaming 具有高吞吐量和容错能力强这两个特点。Spark Streaming 支持的数据输入源很多，例如 Kafka、Flume、Twitter、MQTT、ZeroMQ、Kinesis 和简单的 TCP 套接字等。数据输入后可以用 Spark 的高度抽象原语，如 map、reduce、join、window 等进行运算。而结果也能保存在很多地方，如 HDFS、数据库等。另外 Spark Streaming 也能和 MLlib（机器学习）以及 Graphx 完美融合。

Spark Streaming 的输入与输出可以用图 9-1 来表示。

1. 流式计算的总体思路

按照传统统计工具的做法，应当先将数据存储至数据库中，然后再进行计算。我们知道如果按照这种老办法是无法实时处理数据的，那么是不是数据不存储到磁盘，直接就拿来计算呢？这样的确能节省磁盘 I/O 操作，并且节省时间，应该可以解决实时计算的问题吧。假如数据在内存，还未来得及计算就因为宕机、断电等原因丢失了，该怎么办？ Spark 在流式计算中引入了检查点 CheckPoint 和日志，以便能从 CheckPoint 和日志中恢复中间计算结果。

图 9-1 Spark Streaming 的输入与输出

本质上，它的工作原理如下：Spark Streaming 接收实时输入数据流并将它们按批次划分，然后交给 Spark 引擎处理生成按照批次划分的结果流。它的工作原理如图 9-2 所示。

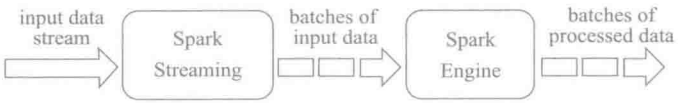


图 9-2 Spark Streaming 工作原理

Spark Streaming 提供了表示连续数据流的、高度抽象的被称为离散流的 Dstream。可以使用 Kafka、Flume 和 Kinesis 这些数据源产生的输入数据流创建 Dstream，也可以在其他 Dstream 上使用 map、reduce、join、window 等操作创建 Dstream。Dstream 本质上表示 RDD 的序列。

2. 数据源支持

目前，Spark 工程中已经提供了很多数据源的支持，如表 9-1 所示。

表 9-1 Spark Streaming 目前支持的数据源

数 据 源	Maven Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

3. Spark 与 Storm 对比

Storm 是一个流处理系统，那么 Spark 与它有什么区别？

- 适用范围比较。Storm 目前只适用于流式数据的处理，而 Spark 除了可以用于流式计算，其应用范围要宽广得多，比如 Spark 还可以用于批处理、SQL 查询、Hive SQL、图计算及机器学习领域。
- 吞吐量比较。Storm 以数据记录为最小单位进行处理和容错。由于单条记录处理的成本较高，Spark Streaming 首先将数据切分为一定时间范围（Duration）的数据集，然后累积一批（Batch）Duration 数据集后单独启动一个任务线程处理。这种方式大大提高了 Spark Streaming 流式计算处理的吞吐量。
- 容错比较。由于 Storm 用与传统关系型数据库相类似的以数据记录为单位容错，所以一条条数据恢复显然很慢。而 Spark Streaming 借助于 Spark 核心提供的从 DAG 重新调度任务和并行执行，能快速地完成数据从故障中恢复的工作。

9.2 StreamingContext 初始化

StreamingContext 是 Spark Streaming 的主要入口，StreamingContext 的主构造器见代码清单 9-1，有三个参数，分别是：

- SparkContext：Spark Streaming 的最终处理实际是交给 SparkContext 的。
- Checkpoint：检查点。
- Duration：设定 streaming 每个批次的积累时间。

代码清单9-1 StreamingContext的实现

```
class StreamingContext private[streaming] (
        sc_ : SparkContext,
        cp_ : Checkpoint,
        batchDur_ : Duration
    ) extends Logging {

    private[streaming] val graph: DStreamGraph = {
        if (isCheckpointPresent) {
            cp_.graph.setContext(this)
            cp_.graph.restoreCheckpointData()
            cp_.graph
        } else {
          assert(batchDur_ != null, "Batch duration for streaming context cannot be null")
          val newGraph = new DStreamGraph()
          newGraph.setBatchDuration(batchDur_)
          newGraph
        }
    }

    private val nextReceiverInputStreamId = new AtomicInteger(0)

    private[streaming] val scheduler = new JobScheduler(this)
```

```
private[streaming] val waiter = new ContextWaiter

private[streaming] val progressListener = new StreamingJobProgressListener(this)

private[streaming] val uiTab: Option[StreamingTab] =
    if (conf.getBoolean("spark.ui.enabled", true)) {
        Some(new StreamingTab(this))
    } else {
        None
    }

private val streamingSource = new StreamingSource(this)
SparkEnv.get.metricsSystem.registerSource(streamingSource)
```

从 StreamingContext 的实现，可以了解 StreamingContext 由以下内容构成：

- DstreamGraph：处理 Dstream 之间的依赖关系。
- JobScheduler：定时生成 Spark Job。
- ContextWaiter：用于等待任务执行结束。
- StreamingJobProgressListener：监听 Streaming Job，用以更新 StreamingTab 的显示。
- StreamingTab：流式计算的标签页，由 Spark UI 负责展示。
- StreamingSource：流式计算的测量数据源。

9.3 输入流接收器规范 Receiver

Receiver 定义了输入流接收器的接口，每种输入流只需要定义实现了 onStart 和 onStop 两个方法的实现类即可。在 onStart 方法中通常需要设置成员（如启动线程、打开 socket 等）开始接收数据并且调用 store(...) 方法将收到的数据存储到 Spark 内存；在 onStop 方法中通常需要清除成员（如停止线程、关闭 socket 等）停止接收数据。一个自定义的 Receiver 大致如下。

```
class MyReceiver extends Receiver<String> {
    public MyReceiver(StorageLevel storageLevel) {
        super(storageLevel);
    }

    public void onStart() {
    }

    public void onStop() {
    }
}
```

Spark 已经实现了很多 Receiver，其继承关系如图 9-3 所示。

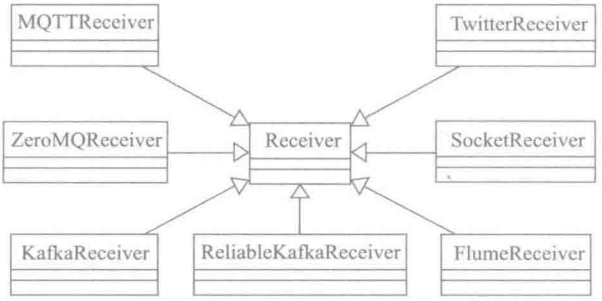


图 9-3 Receiver 继承关系

我们以 MQTTReceiver 为例，来看

看如何自定义一个 Receiver，见代码清单 9-2。其中 MemoryPersistence 负责对消息的持久化以保证重启后不会丢失；构造 MqttClient 需要三个参数，分别是：MQTT 服务器提供的消息消费地址 brokerUrl、clientID 和 MemoryPersistence。MqttCallback 用于消息到达时自动将消息存储到 Spark 内存。connect 方法用于连接 MQTT 服务器，subscribe 方法用于订阅消息主题。

代码清单9-2 MQTTInputDStream.scala中MQTTReceiver的实现

```
private[streaming] class MQTTReceiver(
        brokerUrl: String,
        topic: String,
        storageLevel: StorageLevel
    ) extends Receiver[String](storageLevel) {

    def onStop() {}

    def onStart() {

        val persistence = new MemoryPersistence()

        val client = new MqttClient(brokerUrl, MqttClient.generateClientId(),
           persistence)

        val callback: MqttCallback = new MqttCallback() {

            override def messageArrived(arg0: String, arg1: MqttMessage) {
                store(new String(arg1.getPayload(),"utf-8"))
            }

            override def deliveryComplete(arg0: IMqttDeliveryToken) {
            }

            override def connectionLost(arg0: Throwable) {
                restart("Connection lost ", arg0)
            }
        }

        client.setCallback(callback)
        client.connect()
        client.subscribe(topic)

    }
}
```

9.4 数据流抽象 DStream

Dstream 是 Spark Streaming 中所有数据流的抽象，这里对抽象类 Dstream 中定义的一些主要方法进行介绍：

- dependencies：Dstream 依赖的父级 Dstream 列表。

- compute (validTime: Time)：在指定时间生成一个 RDD。
- isInitialized：Dstream 是否已经初始化。
- persist(level: StorageLevel)：使用指定的存储级别持久化 Dstream 的 RDD。
- persist：存储到内存。
- cache：缓存到内存，与 persist 方法一样。
- checkpoint(interval: Duration)：在 interval 周期后给生成的 RDD 设置检查点。
- setGraph(g: DStreamGraph)：设置 Dstream 及祖宗 Dstream 的 DstreamGraph。
- getOrCompute(time: Time)：从缓存 generatedRDDs = new HashMap[Time, RDD[T]] 中获取 RDD，如果缓存中不存在，则生成 RDD 并持久化、设置检查点并放入缓存。
- generateJob(time: Time)：给指定的 Time 对象生成 Job。
- print：打印 Dstream 生成的每个 RDD 的头 10 个元素。
- window(windowDuration: Duration)：基于原有的 Dstream，返回一个包含了所有在时间滑动窗口中可见元素的新的 Dstream。
- reduceByWindow：基于原有 Dstream，对所有元素进行 reduce 操作后，每个 RDD 都只包含一个元素的新的 Dstream。
- countByWindow：基于原有 Dstream，对所有元素进行 count 操作后，每个 RDD 都只包含一个元素的新的 Dstream。
- union(that: DStream[T])：返回包含了当前 Dstream 与另一个 Dstream 的所有数据的新的 Dstream。
- slice(fromTime: Time, toTime: Time)：返回从 fromTime 到 toTime 之间的所有 RDD 的序列。
- saveAsObjectFiles：将每个 RDD 作为序列化对象存储。
- saveAsTextFiles：将每个 RDD 存储为文本文件。
- register：将当前 Dstream 注册到 DstreamGraph 的输出流中。

9.4.1 Dstream 的离散化

Dstream 本质上表示连续的一系列的 RDD。Dstream 中的每个 RDD 包含了一定间隔的数据，如图 9-4。

任何对 Dstream 的操作都会转化为对底层 RDD 的操作。如果 lines 是一个 Dstream，在其上执行 flatMap 操作，将 lines 转化为命名为 words 的 Dstream 的过程，实际就是对 lines 底层 RDD 执行 flatMap 操作，并将它们转化为 words 的底层 RDD 的过程，如图 9-5 所示。

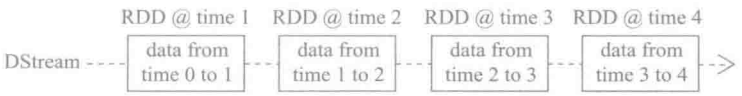


图 9-4 Dstream 的离散化示意图

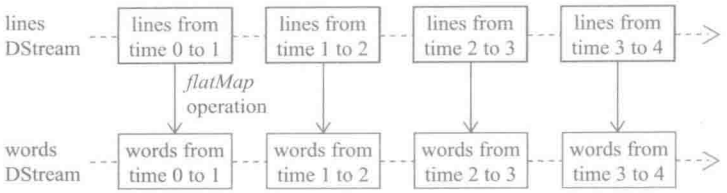


图 9-5 Dstream 的转换

9.4.2 数据源输入流 InputDStream

InputDStream 作为数据源输入流的超类，其继承体系如图 9-6 所示。

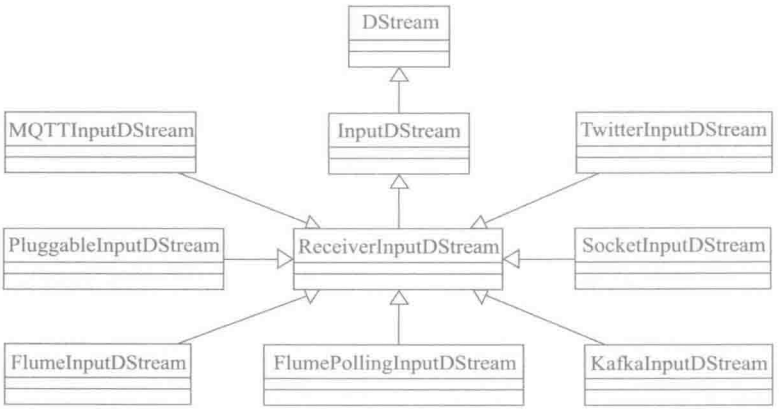


图 9-6 Input Dstream 继承体系

ReceiverInputDStream 定义了获取输入流接收器的接口方法 getReceiver，每个子类都需要实现 getReceiver 方法。SocketInputDStream 是 Spark Streaming 内置的以 Socket 作为输入的 ReceiverInputDStream。此外，在 flume、kafka、mqtt、twitter 包中定义了多种多样的 ReceiverInputDStream，它们都内置了相应的 Receiver，这样就可以接入这些输入流了。以 MQTTInputDStream 为例，MQTTInputDStream 内置了 MQTTReceiver，见代码清单 9-3。

代码清单9-3 MQTTInputDStream的实现

```
private[streaming]
class MQTTInputDStream(
    @transient ssc_ : StreamingContext,
    brokerUrl: String,
    topic: String,
    storageLevel: StorageLevel
) extends ReceiverInputDStream[String](ssc_) {

def getReceiver(): Receiver[String] = {
```

```
        new MQTTReceiver(brokerUrl, topic, storageLevel)
    }
    }
```

Spark 目前不是也支持接入 ZeroMQ 的输入流吗？为什么只见有 ZeroMQReceiver，而不见其相应的 ReceiverInputDStream 实现呢？ Spark 为了可扩展任何输入流，提供插件式的 ReceiverInputDStream，它就是 PluggableInputDStream。PluggableInputDStream 的 getReceiver 方法实际返回的是参数传入的 Receiver，见代码清单 9-4。用户想要接入新的输入流时，先看看 Spark 的最新版本是否已经提供了相应的 ReceiverInputDStream 实现。如果是 Spark 目前还不支持的输入流，可以自己实现其 Receiver，然后作为参数传递给 PluggableInputDStream 即可。

代码清单9-4 PluggableInputDStream的实现

```
private[streaming]
class PluggableInputDStream[T: ClassTag](
    @transient ssc_ : StreamingContext,
    receiver: Receiver[T]) extends ReceiverInputDStream[T](ssc_) {

    def getReceiver(): Receiver[T] = {
       receiver
    }
}
```

由于 ReceiverInputDStream 继承自 InputDStream，InputDStream 在初始化时会执行下面的语句。

```
ssc.graph.addInputStream(this)
```

所以 ReceiverInputDStream 及其子类在初始化时都会被加入当前 StreamingContext 的 DstreamGraph 输入流中。

DstreamGraph 的 addInputStream 方法的实现如下。

```
def addInputStream(inputStream: InputDStream[_]) {
    this.synchronized {
        inputStream.setGraph(this)
        inputStreams += inputStream
    }
}
```

addInputStream 在将 InputDStream 加入 DstreamGraph 的输入流的同时，InputDStream 还调用 setGraph 方法（见代码清单 9-5）持有 DstreamGraph。

代码清单9-5 Dstream的setGraph方法

```
private[streaming] def setGraph(g: DStreamGraph) {
    if (graph != null && graph != g) {
        throw new SparkException("Graph is already set in " + this + ", cannot set it
            again")
    }
```

```
        graph = g
        dependencies.foreach(_.setGraph(graph))
    }
```

9.4.3 Dstream 转换及构建 DStream Graph

还记得在 5.3 节介绍的 word count 例子中，通过操作 RDD（resilient distributed datasets，弹性分布式数据集）提供的接口，如 map、reduce、filter 等，实现 RDD 的转换过程吗？在 Spark Streaming 中，Dstream 提供的接口与 RDD 提供的接口非常相似。构建完 ReceiverInputDStream 后，会调用各种 Dstream 的接口方法，如 map、reduceByKey、flatMap、filter 等，对 Dstream 进行转换。有关 Dstream 转换的类继承体系如图 9-7 所示。

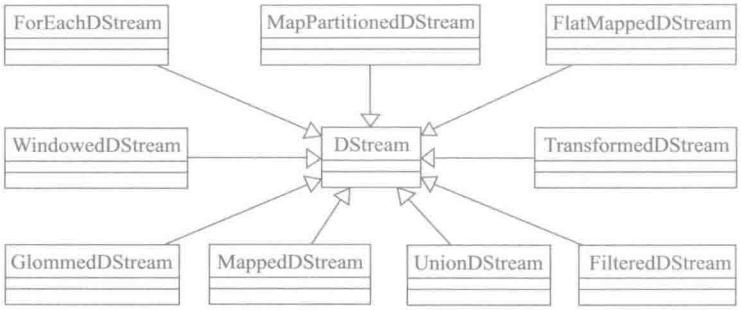


图 9-7 Dstream 转换的类继承体系

Spark Streaming 程序中创建 Dstream，并对 Dstream 进行各种转换，最后各个 Dstream 之间的依赖关系就形成了一张 DStream Graph，如图 9-8[⊖]所示。

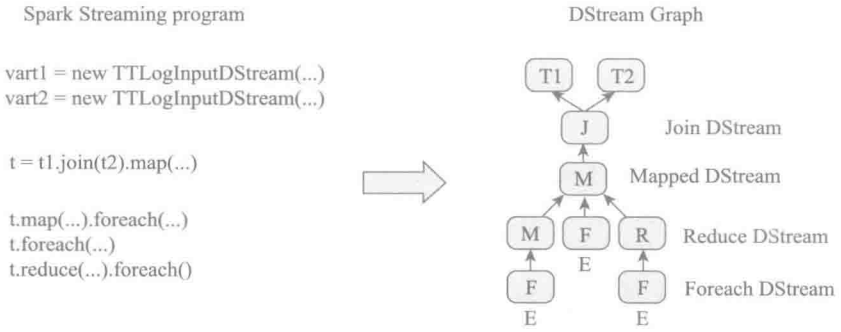


图 9-8 Streaming 程序转换为 DStream Graph 的过程

在 JobGenerator 的启动过程中，会执行 Dstream 与 RDD 的转换，可以用图 9-9[⊖]来表示。

⊖ 图片来源自博文 http://www.tuicool.com/articles/iuMryu。

⊖ 图片来源自博文 http://www.tuicool.com/articles/iuMryu。

Dstream 转换的源码分析过程将在 9.5 节详细阐述。

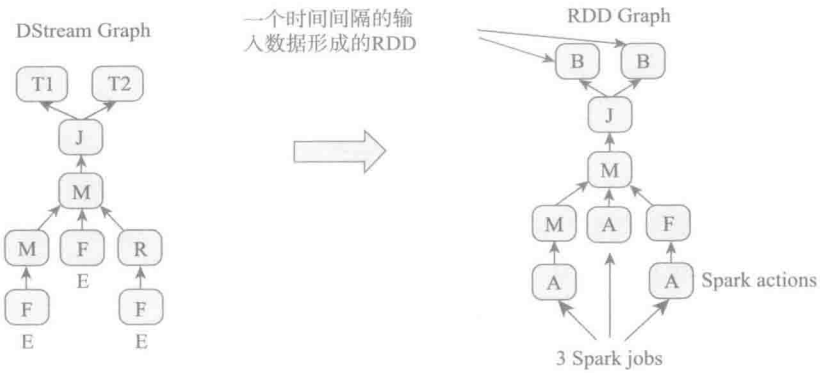


图 9-9 Dstream 与 RDD 的转换

9.5 流式计算执行过程分析

首先，我们用图 9-10 来展示整个 Spark Streaming 执行的流程。

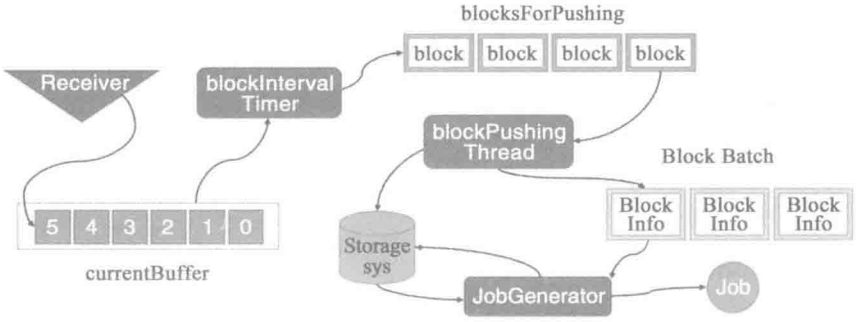


图 9-10 Spark Streaming 执行的流程

这里对图 9-10 中的各个组件进行简单介绍：

- Receiver：Spark Streaming 内置的输入流接收器或者用户自定义的接收器，用于从数据源接收源源不断的数据流。
- currentBuffer：用于缓存输入流接收器接收的数据流。
- blockIntervalTimer：一个定时器，用于将 CurrentBuffer 中缓存的数据流封装为 Block 后放入 blocksForPushing。
- blocksForPushing：用于缓存将要使用的 Block。
- blockPushingThread：此线程每隔 100 毫秒从 blocksForPushing 中取出一个 Block 存入存储体系，并缓存到 ReceivedBlockQueue。

- Block Batch：Block 批次，按照批次时间间隔，从 ReceivedBlockQueue 中获取一批 Block。
- JobGenerator：Job 生成器，用于给每一批 Block 生成一个 Job。

下面通过流式计算的例子 CustomReceiver，来详细分析 Spark Streaming 执行的流程。

9.5.1 流式计算例子 CustomReceiver

Spark 1.2 的源码自带了使用 Spark Streaming 的例子 CustomReceiver，我们就用它来了解 Spark Streaming 功能的使用。根据 Spark 官网描述，如果用 local 模式至少需要指定 2 个 CPU 内核数目，即使用 local[2] 的部署模式。好奇的读者不免要问为什么，这个问题留到后面回答，先来看看 CustomReceiver 的实现，见代码清单 9-6。

注意 Spark Streaming 需要一个 streaming 源，CustomReceiver 的源码注释中建议使用 nc -lk 9999 命令来模拟。nc(Netcat) 是一个流行多年的 TCP/UDP 的监听小工具，也是黑客常用的工具，有兴趣的读者可以自行研究。

代码清单9-6 CustomReceiver及其main方法

```
object CustomReceiver {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: CustomReceiver <hostname> <port>")
            System.exit(1)
        }

        StreamingExamples.setStreamingLogLevels()

        val sparkConf = new SparkConf().setAppName("CustomReceiver").
           setMaster("local[2]");
        val ssc = new StreamingContext(sparkConf, Seconds(1))

        val lines = ssc.receiverStream(new CustomReceiver(args(0), args(1).
           toInt))
        val words = lines.flatMap(_.split(" "))
        val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

从代码清单 9-6 中看到 CustomReceiver 的执行步骤如下。

1）初始化 StreamingContext，并指定批次的时间间隔是 1 秒。

2）创建自定义的 CustomReceiver。CustomReceiver 通过创建 socket 连接 nc 提供的 streaming 服务，见代码清单 9-7。

代码清单9-7 CustomReceiver的实现

```
class CustomReceiver(host: String, port: Int)
    extends Receiver[String](StorageLevel.MEMORY_AND_DISK_2) with Logging {

    def onStart() {
        new Thread("Socket Receiver") {
            override def run() { receive() }
        }.start()
    }

    def onStop() {
    }

    private def receive() {
        var socket: Socket = null
        var userInput: String = null
        try {
            logInfo("Connecting to " + host + ":" + port)
            socket = new Socket(host, port)
            logInfo("Connected to " + host + ":" + port)
            val reader = new BufferedReader(new InputStreamReader(socket.
               getInputStream(), "UTF-8"))
            userInput = reader.readLine()
            while(!isStopped && userInput != null) {
                store(userInput)
                userInput = reader.readLine()
            }
            reader.close()
            socket.close()
            logInfo("Stopped receiving")
            restart("Trying to connect again")
        } catch {
          case e: java.net.ConnectException =>
              restart("Error connecting to " + host + ":" + port, e)
          case t: Throwable =>
              restart("Error receiving data", t)
        }
    }
}
```

3）以 CustomReceiver 为参数，调用 StreamingContext 的 receiverStream 方法（见代码清单 9-8）创建 PluggableInputDStream。

代码清单9-8 StreamingContext的receiverStream方法

```
def receiverStream[T: ClassTag](
    receiver: Receiver[T]): ReceiverInputDStream[T] = {
    new PluggableInputDStream[T](this, receiver)
}
```

4）调用 PluggableInputDStream 的父类 Dstream 的 flatMap 方法，将 PluggableInputDStream 封装为 FlatMappedDStream，见代码清单 9-9。这里正是 9.4.2 节所描述内容的开始。

代码清单9-9 DStream的flatMap方法

```
def flatMap[U: ClassTag](flatMapFunc: T => Traversable[U]): DStream[U] = {
    new FlatMappedDStream(this, context.sparkContext.clean(flatMapFunc))
}
```

5）调用 FlatMappedDStream 的父类 Dstream 的 map 方法，将 FlatMappedDStream 封装为 MappedDStream，见代码清单 9-10。

代码清单9-10 DStream的map方法

```
def map[U: ClassTag](mapFunc: T => U): DStream[U] = {
    new MappedDStream(this, context.sparkContext.clean(mapFunc))
}
```

6）调用 MappedDStream 的 reduceByKey 方法？ MappedDStream 和它的父类中都没有定义 reduceByKey，这里与我们介绍 RDD 的 reduceByKey 方法时类似，也发生了隐式转换，见代码清单 9-11。

代码清单9-11 StreamingContext中的隐式转换

```
implicit def toPairDStreamFunctions[K, V](stream: DStream[(K, V)])
        (implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null) = {
    new PairDStreamFunctions[K, V](stream)
}
```

转换为 PairDStreamFunctions 后，调用它的 reduceByKey 方法，见代码清单 9-12。

代码清单9-12 PairDStreamFunctions的reduceByKey方法

```
def reduceByKey(reduceFunc: (V, V) => V): DStream[(K, V)] = {
    reduceByKey(reduceFunc, defaultPartitioner())
}
```

首先获取默认的 Partitioner，见代码清单 9-13。

代码清单9-13 PairDStreamFunctions的defaultPartitioner方法

```
private[streaming] def defaultPartitioner(numPartitions: Int = self.ssc.sc.defaultParallelism) = {
    new HashPartitioner(numPartitions)
}
```

然后调用重载的 reduceByKey 方法，其中调用了 combineByKey 方法，见代码清单 9-14。

代码清单9-14 PairDStreamFunctions中重载的reduceByKey方法

```
def reduceByKey(reduceFunc: (V, V) => V, partitioner: Partitioner): DStream[(K, V)] = {
    val cleanedReduceFunc = ssc.sc.clean(reduceFunc)
    combineByKey((v: V) => v, cleanedReduceFunc, cleanedReduceFunc, partitioner)
}
```

PairDStreamFunctions 的 combineByKey 方法将 MappedDStream 转换为 ShuffledDStream，见代码清单 9-15。

代码清单9-15 PairDStreamFunctions的 combineByKey方法

```
def combineByKey[C: ClassTag](
    createCombiner: V => C,
    mergeValue: (C, V) => C,
    mergeCombiner: (C, C) => C,
    partitioner: Partitioner,
    mapSideCombine: Boolean = true): DStream[(K, C)] = {
    new ShuffledDStream[K, V, C](self, createCombiner, mergeValue, mergeCombiner,
        partitioner,
        mapSideCombine)
}
```

7）print 方法又将 ShuffledDStream 封装为 ForEachDStream，见代码清单 9-16。

代码清单9-16 Dstream的print方法

```
def print() {
    def foreachFunc = (rdd: RDD[T], time: Time) => {
        val first11 = rdd.take(11)
        println ("-------------------------------------------")
        println ("Time: " + time)
        println ("-------------------------------------------")
        first11.take(10).foreach(println)
        if (first11.size > 10) println("...")
        println()
    }
    new ForEachDStream(this, context.sparkContext.clean(foreachFunc)).register()
}
```

调用 register 方法将 ForEachDStream 注册到 DstreamGraph 的输出流中，见代码清单 9-17。

代码清单9-17 Dstream的register方法

```
private[streaming] def register(): DStream[T] = {
    ssc.graph.addOutputStream(this)
    this
}
```

DStreamGraph 的 addOutputStream 方法，见代码清单 9-18。

代码清单9-18 DStreamGraph的addOutputStream方法

```
def addOutputStream(outputStream: DStream[_]) {
    this.synchronized {
        outputStream.setGraph(this)
        outputStreams += outputStream
    }
}
```

8）CustomReceiver 最后调用 StreamingContext 的 start 方法来启动流式计算的过程，其实现见代码清单 9-19。

代码清单9-19 StreamingContext的start方法

```
def start(): Unit = synchronized {
```

```
        if (state == Started) {
            throw new SparkException("StreamingContext has already been started")
        }
        if (state == Stopped) {
            throw new SparkException("StreamingContext has already been stopped")
        }
        validate()
        sparkContext.setCallSite(DStream.getCreationSite())
        scheduler.start()
        state = Started
    }
```

StreamingContext 的 start 方法中主要调用了 JobScheduler 的 start 方法（见代码清单 9-20）。JobScheduler 的启动由以下步骤组成：

1）创建 eventActor 的匿名类实现，主要用于处理各类 JobScheduler 的事件。

2）启动 StreamingListenerBus，实现原理与 LiveListenerBus 相同，主要用于更新 Spark UI 中 StreamTab 的内容。

3）创建并启动 ReceiverTracker。ReceiverTracker 用于处理数据接收、数据缓存、Block 生成等工作。

4）启动 JobGenerator。JobGenerator 负责对 DstreamGraph 的初始化、Dstream 与 RDD 的转换、生成 Job、提交执行等工作。

我们主要关注 ReceiverTracker 和 JobGenerator 的启动过程。

代码清单9-20 JobScheduler的start方法

```
def start(): Unit = synchronized {
    if (eventActor != null) return // scheduler has already been started

    logDebug("Starting JobScheduler")
    eventActor = ssc.env.actorSystem.actorOf(Props(new Actor {
        def receive = {
            case event: JobSchedulerEvent => processEvent(event)
        }
    }), "JobScheduler")

    listenerBus.start()
    receiverTracker = new ReceiverTracker(ssc)
    receiverTracker.start()
    jobGenerator.start()
    logInfo("Started JobScheduler")
}
```

9.5.2 Spark Streaming 执行环境构建

ReceiverTracker 的启动过程，实际是对 Spark Streaming 执行环境构建的过程。ReceiverTracker 的 start 方法中，如果 receiverInputStreams 中存在 ReceiverInputDStream，则向 ActorSystem 注册 ReceiverTrackerActor，并且启动 receiverExecutor，见代码清单 9-21。

代码清单9-21 ReceiverTracker的start方法

```
def start() = synchronized {
    if (actor != null) {
        throw new SparkException("ReceiverTracker already started")
    }

    if (!receiverInputStreams.isEmpty) {
        actor = ssc.env.actorSystem.actorOf(Props(new ReceiverTrackerActor),
            "ReceiverTracker")
        if (!skipReceiverLaunch) receiverExecutor.start()
        logInfo("ReceiverTracker started")
    }
}
```

ReceiverTracker 中的 receiverInputStreams 实际调用了 DStreamGraph 的 getReceiverInputStreams 方法，代码如下。

```
private val receiverInputStreams = ssc.graph.getReceiverInputStreams()
```

getReceiverInputStreams 方法用于过滤出 DStreamGraph 的 inputStreams 中的 ReceiverInputDStream，实现如下。

```
def getReceiverInputStreams() = this.synchronized {
    inputStreams.filter(_.isInstanceOf[ReceiverInputDStream[_]])
        .map(_.asInstanceOf[ReceiverInputDStream[_]])
        .toArray
}
```

ReceiverTrackerActor 用于接收来自 Receiver 的消息，包括 RegisterReceiver、AddBlock、ReportError、DeregisterReceiver 等，见代码清单 9-22。

代码清单9-22 ReceiverTrackerActor接收来自Receiver的消息代码

```
private class ReceiverTrackerActor extends Actor {
    def receive = {
        case RegisterReceiver(streamId, typ, host, receiverActor) =>
            registerReceiver(streamId, typ, host, receiverActor, sender)
            sender ! true
        case AddBlock(receivedBlockInfo) =>
            sender ! addBlock(receivedBlockInfo)
        case ReportError(streamId, message, error) =>
            reportError(streamId, message, error)
        case DeregisterReceiver(streamId, message, error) =>
            deregisterReceiver(streamId, message, error)
            sender ! true
    }
}
```

receiverExecutor 的类型是 ReceiverLauncher。ReceiverLauncher 的 start 方法用于启动匿名线程 thread，见代码清单 9-23。

代码清单9-23 ReceiverLauncher启动的匿名线程thread的实现

```
@transient val thread  = new Thread() {
    override def run() {
        try {
            SparkEnv.set(env)
            startReceivers()
        } catch {
          case ie: InterruptedException => logInfo("ReceiverLauncher
              interrupted")
        }
    }
}

def start() {
    thread.start()
}
```

Thread线程的主要作用在于调用startReceivers方法（见代码清单9-24）。其处理步骤如下：

1）获取每个ReceiverInputDStream的Receiver，本例中即为CustomReceiver。

2）获取每个ReceiverInputDStream的优先选择机器。

3）调用SparkContext的makeRDD方法，将所有Receiver封装为ParallelCollectionRDD，并行度是receivers的数量。makeRDD方法实际调用parallelize，见代码清单9-25。parallelize中构造了ParallelCollectionRDD，见代码清单9-26。

4）定义偏函数startReceiver，此函数用于接收每个Receiver的数据。

5）以startReceiver为参数，提交任务。

注意 ssc.sparkContext.makeRDD(1 to 50, 50).map(x => (x, 1)).reduceByKey(_ + _, 20).collect()这条语句通过运行重复的作业来确保所有的slave都已经注册了，避免所有的receivers都到一个节点上。

代码清单9-24 ReceiverTracker的startReceivers方法

```
private def startReceivers() {
    val receivers = receiverInputStreams.map(nis => {
        val rcvr = nis.getReceiver()
        rcvr.setReceiverId(nis.id)
        rcvr
    })

    val hasLocationPreferences = receivers.map(_.preferredLocation.isDefined).reduce(_ && _)
    val tempRDD =
        if (hasLocationPreferences) {
            val receiversWithPreferences = receivers.map(r => (r, Seq(r.preferred
                Location.get)))
            ssc.sc.makeRDD[Receiver[_]](receiversWithPreferences)
```

```
            } else {
                ssc.sc.makeRDD(receivers, receivers.size)
            }

        val checkpointDirOption = Option(ssc.checkpointDir)
        val serializableHadoopConf = new SerializableWritable(ssc.sparkContext.
           hadoopConfiguration)

        val startReceiver = (iterator: Iterator[Receiver[_]]) => {
            if (!iterator.hasNext) {
                throw new SparkException(
                    "Could not start receiver as object not found.")
            }
            val receiver = iterator.next()
            val supervisor = new ReceiverSupervisorImpl(
                receiver, SparkEnv.get, serializableHadoopConf.value, checkpointDirOption)
            supervisor.start()
            supervisor.awaitTermination()
        }
        if (!ssc.sparkContext.isLocal) {
            ssc.sparkContext.makeRDD(1 to 50, 50).map(x => (x, 1)).reduceByKey(_ + _,
                20).collect()
        }

        running = true
        ssc.sparkContext.runJob(tempRDD, ssc.sparkContext.clean(startReceiver))
        running = false
    }
```

代码清单9-25 SparkContext的makeRDD方法

```
def makeRDD[T: ClassTag](seq: Seq[T], numSlices: Int = defaultParallelism):
   RDD[T] = {
    parallelize(seq, numSlices)
}
```

代码清单9-26 SparkContext的parallelize方法

```
def parallelize[T: ClassTag](seq: Seq[T], numSlices: Int = defaultParallelism):
   RDD[T] = {
    assertNotStopped()
    new ParallelCollectionRDD[T](this, seq, numSlices, Map[Int, Seq[String]]())
}
```

任务提交阶段执行代码清单 5-19 中的 runJob 方法时，首先调用 RDD 的 partitions。根据 5.3 节的代码清单 5-10 和代码清单 5-11，我们知道最终会调用 ParallelCollectionRDD 的 getPartitions 方法。ParallelCollectionRDD 的 getPartitions 方法，见代码清单 9-27。

代码清单9-27 ParallelCollectionRDD的getPartitions方法

```
override def getPartitions: Array[Partition] = {
    val slices = ParallelCollectionRDD.slice(data, numSlices).toArray
```

```
        slices.indices.map(i => new ParallelCollectionPartition(id, i, slices(i))).toArray
    }
```

ParallelCollectionRDD 的 slice 方法（见代码清单 9-28）用于将一个集合切分为 numSlices 个子集合，这样做可以让多个 Receiver 有效地运行在 Spark 上。此处实际是将 Receiver 划分为多个序列，然后给每个序列创建一个 ParallelCollectionPartition。ParallelCollectionPartition 的数目实际由 numSlices 决定。

代码清单9-28 ParallelCollectionRDD的slice方法

```
def slice[T: ClassTag](seq: Seq[T], numSlices: Int): Seq[Seq[T]] = {
    if (numSlices < 1) {
        throw new IllegalArgumentException("Positive number of slices required")
    }
    def positions(length: Long, numSlices: Int): Iterator[(Int, Int)] = {
        (0 until numSlices).iterator.map(i => {
            val start = ((i * length) / numSlices).toInt
            val end = (((i + 1) * length) / numSlices).toInt
            (start, end)
        })
    }
    seq match {
        case r: Range => {
            positions(r.length, numSlices).zipWithIndex.map({ case ((start, end),
                index) =>
                // If the range is inclusive, use inclusive range for the last slice
                if (r.isInclusive && index == numSlices - 1) {
                   new Range.Inclusive(r.start + start * r.step, r.end, r.step)
                }
                else {
                    new Range(r.start + start * r.step, r.start + end * r.step,
                       r.step)
                }
            }).toSeq.asInstanceOf[Seq[Seq[T]]]
        }
        case nr: NumericRange[_] => {
            val slices = new ArrayBuffer[Seq[T]](numSlices)
            var r = nr
            for ((start, end) <- positions(nr.length, numSlices)) {
                val sliceSize = end - start
                slices += r.take(sliceSize).asInstanceOf[Seq[T]]
                r = r.drop(sliceSize)
            }
            slices
        }
        case _ => {
            val array = seq.toArray // To prevent O(n^2) operations for List etc
            positions(array.length, numSlices).map({
                case (start, end) =>
                    array.slice(start, end).toSeq
           }).toSeq
        }
    }
```

任务执行阶段，根据 6.1 节的分析，我们清楚最终会调用 ParallelCollectionRDD 的 compute 方法。compute 方法（见代码清单 9-29）会构建 InterruptibleIterator。

代码清单9-29 ParallelCollectionRDD的compute方法

```
override def compute(s: Partition, context: TaskContext) = {
    new InterruptibleIterator(context, s.asInstanceOf[ParallelCollectionPartition
      [T]].iterator)
}
```

根据代码清单 5-62 中对 ResultTask 的 runTask 方法的介绍，知道最后会回调代码清单 9-24 中定义的偏函数 startReceiver。startReceiver 首先新建一个 ReceiverSupervisorImpl，然后调用它的 start 方法（见代码清单 9-30）启动 Supervisor。

代码清单9-30 ReceiverSupervisor的start方法

```
def start() {
    onStart()
    startReceiver()
}
```

start 方法中先后调用了 onStart 与 startReceiver 方法。我们先来分析 onStart 方法。onStart 实际调用了 blockGenerator 的 start 方法。

```
override protected def onStart() {
    blockGenerator.start()
}
```

blockGenerator 实际是 BlockGenerator 的匿名实现类，blockGenerator 的参数是 BlockGeneratorListener 的匿名实现类，重写了 onError 和 onPushBlock 方法，见代码清单 9-31。

代码清单9-31 ReceiverSupervisorImpl.scala中BlockGenerator的匿名实现类

```
private val blockGenerator = new BlockGenerator(new BlockGeneratorListener {
    def onAddData(data: Any, metadata: Any): Unit = { }

    def onGenerateBlock(blockId: StreamBlockId): Unit = { }

    def onError(message: String, throwable: Throwable) {
        reportError(message, throwable)
    }

    def onPushBlock(blockId: StreamBlockId, arrayBuffer: ArrayBuffer[_]) {
        pushArrayBuffer(arrayBuffer, None, Some(blockId))
    }
}, streamId, env.conf)
```

BlockGenerator 的 start 方法见代码清单 9-32，其处理步骤如下：

1）启动 blockIntervalTimer。blockIntervalTimer 的类型是 RecurringTimer，RecurringTimer 的 start 方法实际启动了内置的线程 thread，这个 thread 按照指定的周期 blockInterval 回调 callback 方法，即回调 updateCurrentBuffer 方法，见代码清单 9-33 和代码清单 9-34。

代码清单9-32 BlockGenerator的start方法

```
def start() {
    blockIntervalTimer.start()
    blockPushingThread.start()
    logInfo("Started BlockGenerator")
}
```

代码清单9-33 BlockGenerator的部分实现

```
private val clock = new SystemClock()
private val blockInterval = conf.getLong("spark.streaming.blockInterval", 200)
private val blockIntervalTimer =
    new RecurringTimer(clock, blockInterval, updateCurrentBuffer, "BlockGenerator")
private val blockQueueSize = conf.getInt("spark.streaming.blockQueueSize", 10)
private val blocksForPushing = new ArrayBlockingQueue[Block](blockQueueSize)
private val blockPushingThread = new Thread() { override def run() {
    keepPushingBlocks() } }

@volatile private var currentBuffer = new ArrayBuffer[Any]
```

代码清单9-34 RecurringTimer的实现

```
private[streaming]
class RecurringTimer(clock: Clock, period: Long, callback: (Long) => Unit, name: String)
    extends Logging {

    private val thread = new Thread("RecurringTimer - " + name) {
        setDaemon(true)
        override def run() { loop }
    }

    def start(startTime: Long): Long = synchronized {
        nextTime = startTime
        thread.start()
        logInfo("Started timer for " + name + " at time " + nextTime)
        nextTime
    }

    private def loop() {
        try {
            while (!stopped) {
                clock.waitTillTime(nextTime)
                callback(nextTime)
                prevTime = nextTime
                nextTime += period
                logDebug("Callback for " + name + " called at time " + prevTime)
            }
        } catch {
            case e: InterruptedException =>
        }
    }
```

updateCurrentBuffer 方法（见代码清单 9-35）的作用如下：

① 生成新的 StreamBlockId，生成规则："input-" + streamId + "-" + (调用 time – blockInterval)，blockInterval 为生成 block 的间隔时长。

② 将当前 currentBuffer 与 StreamBlockId 封装为 Block。

③ 调用 BlockGeneratorListener 的 onGenerateBlock。本例中，BlockGeneratorListener 的匿名实现类并未实现 onGenerateBlock 方法。

④ 将新建的 Block 放入 blocksForPushing 中。

代码清单9-35 BlockGenerator的updateCurrentBuffer方法

```
private def updateCurrentBuffer(time: Long): Unit = synchronized {
    try {
        val newBlockBuffer = currentBuffer
        currentBuffer = new ArrayBuffer[Any]
        if (newBlockBuffer.size > 0) {
            val blockId = StreamBlockId(receiverId, time - blockInterval)
            val newBlock = new Block(blockId, newBlockBuffer)
            listener.onGenerateBlock(blockId)
            blocksForPushing.put(newBlock)  // put is blocking when queue is full
            logDebug("Last element in " + blockId + " is " + newBlockBuffer.last)
        }
    } catch {
        case ie: InterruptedException =>
            logInfo("Block updating timer thread was interrupted")
        case e: Exception =>
            reportError("Error in block updating thread", e)
    }
}
```

2）启动线程 blockPushingThread，此线程主要调用 keepPushingBlocks，见代码清单 9-33。keepPushingBlocks 方法（见代码清单 9-36）的作用如下：

① 当 BlockGenerator 没有停止时，每隔 100 毫秒从 blocksForPushing 中取出一个 Block，然后调用 pushBlock 方法。

② 一旦 BlockGenerator 停止了，将 blocksForPushing 中所有的 Block 取出，然后调用 pushBlock 方法。

代码清单9-36 BlockGenerator的keepPushingBlocks方法

```
private def keepPushingBlocks() {
    while(!stopped) {
        Option(blocksForPushing.poll(100, TimeUnit.MILLISECONDS)) match {
            case Some(block) => pushBlock(block)
            case None =>
        }
    }
    logInfo("Pushing out the last " + blocksForPushing.size() + " blocks")
    while (!blocksForPushing.isEmpty) {
        logDebug("Getting block ")
        val block = blocksForPushing.take()
```

```
            pushBlock(block)
            logInfo("Blocks left to push " + blocksForPushing.size())
        }
        logInfo("Stopped block pushing thread")
    }
```

pushBlock 调用 BlockGeneratorListener 的 onPushBlock 方法，见代码清单 9-37。

代码清单9-37 BlockGenerator的pushBlock方法

```
private def pushBlock(block: Block) {
    listener.onPushBlock(block.id, block.buffer)
    logInfo("Pushed block " + block.id)
}
```

onPushBlock 又调用了代码清单 9-31 中 BlockGeneratorListener 的匿名实现类的 onPushBlock 方法，onPushBlock 方法然后调用了 ReceiverSupervisorImpl 的 pushArrayBuffer。pushArrayBuffer 实际代理了 pushAndReportBlock，见代码清单 9-38。

代码清单9-38 ReceiverSupervisorImpl的pushArrayBuffer方法

```
def pushArrayBuffer(
        arrayBuffer: ArrayBuffer[_],
        metadataOption: Option[Any],
        blockIdOption: Option[StreamBlockId]
    ) {
    pushAndReportBlock(ArrayBufferBlock(arrayBuffer), metadataOption, blockIdOption)
}
```

pushAndReportBlock 方法见代码清单 9-39。

代码清单9-39 ReceiverSupervisorImpl的pushAndReportBlock方法

```
def pushAndReportBlock(
        receivedBlock: ReceivedBlock,
        metadataOption: Option[Any],
        blockIdOption: Option[StreamBlockId]
    ) {
    val blockId = blockIdOption.getOrElse(nextBlockId)
    val numRecords = receivedBlock match {
        case ArrayBufferBlock(arrayBuffer) => arrayBuffer.size
        case _ => -1
    }

    val time = System.currentTimeMillis
    val blockStoreResult = receivedBlockHandler.storeBlock(blockId, receivedBlock)
    logDebug(s"Pushed block $blockId in ${(System.currentTimeMillis - time)} ms")

    val blockInfo = ReceivedBlockInfo(streamId, numRecords, blockStoreResult)
    val future = trackerActor.ask(AddBlock(blockInfo))(askTimeout)
    Await.result(future, askTimeout)
    logDebug(s"Reported block $blockId")
}
```

pushAndReportBlock 方法的处理步骤如下：

1）调用 receivedBlockHandler 的 storeBlock 方法将 block 存储到 Spark 的存储体系中。从代码清单 9-40 可以看出默认情况下 receivedBlockHandler 为 BlockManagerBasedBlockHandler。BlockManagerBasedBlockHandler 的 storeBlock 正是使用第 4 章介绍过的 BlockManager 的 putBytes 或者 putIterator 来实现的，见代码清单 9-41。

代码清单9-40 ReceiverSupervisorImpl.scala中receivedBlockHandler的定义

```
private val receivedBlockHandler: ReceivedBlockHandler = {
    if (env.conf.getBoolean("spark.streaming.receiver.writeAheadLog.enable", false)) {
      if (checkpointDirOption.isEmpty) {
          throw new SparkException(
             "Cannot enable receiver write-ahead log without checkpoint directory
                set. " +
              "Please use streamingContext.checkpoint() to set the checkpoint directory. " +
              "See documentation for more details.")
        }
        new WriteAheadLogBasedBlockHandler(env.blockManager, receiver.streamId,
            receiver.storageLevel, env.conf, hadoopConf, checkpointDirOption.get)
    } else {
        new BlockManagerBasedBlockHandler(env.blockManager, receiver.storageLevel)
    }
}
```

代码清单9-41 BlockManagerBasedBlockHandler的storeBlock方法

```
def storeBlock(blockId: StreamBlockId, block: ReceivedBlock): ReceivedBlockStoreResult
   = {
    val putResult: Seq[(BlockId, BlockStatus)] = block match {
        case ArrayBufferBlock(arrayBuffer) =>
            blockManager.putIterator(blockId, arrayBuffer.iterator, storageLevel,
                tellMaster = true)
        case IteratorBlock(iterator) =>
            blockManager.putIterator(blockId, iterator, storageLevel, tellMaster = true)
        case ByteBufferBlock(byteBuffer) =>
            blockManager.putBytes(blockId, byteBuffer, storageLevel, tellMaster = true)
        case o =>
            throw new SparkException(
                s"Could not store $blockId to block manager, unexpected block
                    type ${o.getClass.getName}")
    }
    if (!putResult.map { _._1 }.contains(blockId)) {
        throw new SparkException(
            s"Could not store $blockId to block manager with storage level
                $storageLevel")
    }
    BlockManagerBasedStoreResult(blockId)
}
```

2）将上一步返回的结果等信息封装为样例类 ReceivedBlockInfo 的实例，再封装为样例

类AddBlock的实例，最后向trackerActor发送AddBlock消息。从代码清单9-22中知道，ReceiverTrackerActor收到AddBlock消息后，会调用addBlock方法，根据代码清单9-42和代码清单9-43，我们知道最终会将ReceivedBlockInfo更新至当前Receiver在streamIdToUnallocatedBlockQueues = new mutable.HashMap[Int, ReceivedBlockQueue]中对应的ReceivedBlockQueue中。

代码清单9-42 ReceiverTracker的addBlock方法

```
private def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = {
    receivedBlockTracker.addBlock(receivedBlockInfo)
}
```

代码清单9-43 ReceivedBlockTracker的addBlock方法

```
def addBlock(receivedBlockInfo: ReceivedBlockInfo): Boolean = synchronized {
    writeToLog(BlockAdditionEvent(receivedBlockInfo))
    getReceivedBlockQueue(receivedBlockInfo.streamId) += receivedBlockInfo
    logDebug(s"Stream ${receivedBlockInfo.streamId} received " +
        s"block ${receivedBlockInfo.blockStoreResult.blockId}")
    true
}

private def getReceivedBlockQueue(streamId: Int): ReceivedBlockQueue = {
    streamIdToUnallocatedBlockQueues.getOrElseUpdate(streamId, new ReceivedBlockQueue)
}
```

到这里，我们知道线程blockPushingThread将blocksForPushing中的Block不断存储到存储体系中，blocksForPushing中的Block是由blockIntervalTimer内置的线程源源不断从currentBuffer中供给的。那么，currentBuffer中的数据来自哪里？让我们接着往下看。

介绍完ReceiverSupervisorImpl的onStart，我们现在开始分析startReceiver方法，startReceiver方法见代码清单9-44，其执行步骤如下。

1）调用receiver的onStart方法。本例中，receiver即为CustomReceiver，从代码清单9-7中我们知道，CustomReceiver的onStart方法启动了命名为Socket Receiver的线程，这个线程主要用于使用套接字获取数据源的数据流，然后调用store方法，见代码清单9-45。这里的executor是什么？还记得之前构造的ReceiverSupervisorImpl吗？它的父类在初始化时，会执行代码清单9-46中的代码，通过调用Receiver的attachExecutor方法，将ReceiverSupervisorImpl设置为executor，见代码清单9-47。

代码清单9-44 ReceiverSupervisor的startReceiver方法

```
def startReceiver(): Unit = synchronized {
    try {
        logInfo("Starting receiver")
        receiver.onStart()
        logInfo("Called receiver onStart")
        onReceiverStart()
        receiverState = Started
    } catch {
      case t: Throwable =>
```

```
            stop("Error starting receiver " + streamId, Some(t))
        }
    }
```

代码清单9-45 Receiver的store方法

```
def store(dataItem: T) {
    executor.pushSingle(dataItem)
}

private def executor = {
    assert(executor_ != null, "Executor has not been attached to this receiver")
    executor_
}
```

代码清单9-46 ReceiverSupervisor初始化时执行的代码

```
receiver.attachExecutor(this)
```

代码清单9-47 Receiver的attachExecutor方法

```
private[streaming] def attachExecutor(exec: ReceiverSupervisor) {
    assert(executor_ == null)
    executor_ = exec
}
```

ReceiverSupervisorImpl 的 pushSingle 方法，见代码清单 9-48。BlockGenerator 的 addData 方法首先调用 waitToPush 限制 Receiver 消费数据流的速率，然后将数据放入 currentBuffer 中，见代码清单 9-49。

代码清单9-48 ReceiverSupervisorImpl的pushSingle方法

```
def pushSingle(data: Any) {
    blockGenerator.addData(data)
}
```

代码清单9-49 BlockGenerator的addData方法

```
def addData (data: Any): Unit = synchronized {
    waitToPush()
    currentBuffer += data
}
```

2）调用 ReceiverSupervisorImpl 的 onReceiverStart 方法，用于给 ReceiverTrackerActor 发送 RegisterReceiver 消息，见代码清单 9-50。ReceiverTrackerActor 接收 RegisterReceiver 后调用 registerReceiver 方法，见代码清单 9-22。registerReceiver 方法创建 ReceiverInfo 后注册到 receiverInfo 中，见代码清单 9-51。

代码清单9-50 ReceiverSupervisorImpl的onReceiverStart方法

```
override protected def onReceiverStart() {
    val msg = RegisterReceiver(
        streamId, receiver.getClass.getSimpleName, Utils.localHostName(), actor)
```

```
    val future = trackerActor.ask(msg)(askTimeout)
    Await.result(future, askTimeout)
}
```

代码清单9-51 RegisterReceiver的registerReceiver方法

```
private def registerReceiver(
        streamId: Int,
        typ: String,
        host: String,
        receiverActor: ActorRef,
        sender: ActorRef
    ) {
    if (!receiverInputStreamIds.contains(streamId)) {
        throw new SparkException("Register received for unexpected id " + streamId)
    }
    receiverInfo(streamId) = ReceiverInfo(
        streamId, s"${typ}-${streamId}", receiverActor, true, host)
    listenerBus.post(StreamingListenerReceiverStarted(receiverInfo(streamId)))
    logInfo("Registered receiver for stream " + streamId + " from " + sender.path.address)
}
```

有关创建并启动 ReceiverTracker 的过程已经叙述完毕，根据第 7 章的内容我们知道，blockIntervalTimer、blockPushingThread 及 Receiver 都运行在 Executor 上。

9.5.3 任务生成过程

介绍完 ReceiverTracker 的创建和启动，现在回头看看启动 JobGenerator 的 start 方法，见代码清单 9-52。

代码清单9-52 JobGenerator的start方法

```
def start(): Unit = synchronized {
    if (eventActor != null) return // generator has already been started

    eventActor = ssc.env.actorSystem.actorOf(Props(new Actor {
        def receive = {
            case event: JobGeneratorEvent =>  processEvent(event)
        }
    }), "JobGenerator")
    if (ssc.isCheckpointPresent) {
        restart()
    } else {
        startFirstTime()
    }
}
```

JobGenerator 的 start 方法的处理过程如下：

1）创建接收 JobGeneratorEvent 消息的 Actor，即 eventActor。

2）调用 startFirstTime 方法，对 DStreamGraph 中的 outputStreams 和 inputStreams 进行一些初始化动作，见代码清单 9-53 和代码清单 9-54。

代码清单9-53 JobGenerator的startFirstTime方法

```
private def startFirstTime() {
    val startTime = new Time(timer.getStartTime())
    graph.start(startTime - graph.batchDuration)
    timer.start(startTime.milliseconds)
    logInfo("Started JobGenerator at " + startTime)
}
```

代码清单9-54 DStreamGraph的start方法

```
def start(time: Time) {
    this.synchronized {
        if (zeroTime != null) {
            throw new Exception("DStream graph computation already started")
        }
        zeroTime = time
        startTime = time
        outputStreams.foreach(_.initialize(zeroTime))
        outputStreams.foreach(_.remember(rememberDuration))
        outputStreams.foreach(_.validate)
        inputStreams.par.foreach(_.start())
    }
}
```

代码清单 9-53 中的 timer 也是一个 RecurringTimer 对象，根据 RecurringTimer 的原理我们知道，timer 会按照 batchDuration 指定的时间间隔，向 eventActor 发送 GenerateJobs 消息，见代码清单 9-55。GenerateJobs 继承自 JobGeneratorEvent，所以 eventActor 会调用 processEvent 方法处理。processEvent 则将 GenerateJobs 消息的处理委托给 generateJobs 处理，见代码清单 9-56。

代码清单9-55 JobGenerator.scala中timer的定义

```
private val timer = new RecurringTimer(clock, ssc.graph.batchDuration.milliseconds,
    longTime => eventActor ! GenerateJobs(new Time(longTime)), "JobGenerator")
```

代码清单9-56 JobGenerator的processEvent方法

```
private def processEvent(event: JobGeneratorEvent) {
    logDebug("Got event " + event)
    event match {
        case GenerateJobs(time) => generateJobs(time)
        case ClearMetadata(time) => clearMetadata(time)
        case DoCheckpoint(time) => doCheckpoint(time)
        case ClearCheckpointData(time) => clearCheckpointData(time)
    }
}
```

generateJobs 方法的实现见代码清单 9-57。

代码清单9-57 JobGenerator的generateJobs方法

```
private def generateJobs(time: Time) {
    SparkEnv.set(ssc.env)
```

```
        Try {
            jobScheduler.receiverTracker.allocateBlocksToBatch(time) // allocate
                received blocks to batch
            graph.generateJobs(time) // generate jobs using allocated block
        } match {
            case Success(jobs) =>
                val receivedBlockInfos =
                jobScheduler.receiverTracker.getBlocksOfBatch(time).mapValues { _.to-
                    Array }
                jobScheduler.submitJobSet(JobSet(time, jobs, receivedBlockInfos))
            case Failure(e) =>
                jobScheduler.reportError("Error generating jobs for time " + time, e)
        }
        eventActor ! DoCheckpoint(time)
    }
```

generateJobs 方法的处理步骤如下。

1）调用 ReceiverTracker 的 allocateBlocksToBatch 方法，实际是调用 ReceivedBlockTracker 的 allocateBlocksToBatch 方法，见代码清单 9-58 和代码清单 9-59。

代码清单9-58 ReceiverTracker的allocateBlocksToBatch方法

```
def allocateBlocksToBatch(batchTime: Time): Unit = {
    if (receiverInputStreams.nonEmpty) {
        receivedBlockTracker.allocateBlocksToBatch(batchTime)
    }
}
```

代码清单9-59 ReceivedBlockTracker的allocateBlocksToBatch方法

```
def allocateBlocksToBatch(batchTime: Time): Unit = synchronized {
    if (lastAllocatedBatchTime == null || batchTime > lastAllocatedBatchTime) {
        val streamIdToBlocks = streamIds.map { streamId =>
            (streamId, getReceivedBlockQueue(streamId).dequeueAll(x => true))
        }.toMap
        val allocatedBlocks = AllocatedBlocks(streamIdToBlocks)
        writeToLog(BatchAllocationEvent(batchTime, allocatedBlocks))
        timeToAllocatedBlocks(batchTime) = allocatedBlocks
        lastAllocatedBatchTime = batchTime
        allocatedBlocks
    } else {
        logInfo(s"Possibly processed batch $batchTime need to be processed again
            in WAL recovery")
    }
}
```

调用 getReceivedBlockQueue 方法获取 ReceivedBlockQueue，并将 ReceivedBlockQueue 中的 ReceivedBlockInfo 全部出列后封装为 AllocatedBlocks，见代码清单 9-60。最后将 AllocatedBlocks 注册到 timeToAllocatedBlocks = new mutable.HashMap[Time, AllocatedBlocks] 中并更新最后分配批次的时间 lastAllocatedBatchTime 后返回 AllocatedBlocks。

代码清单9-60 ReceivedBlockTracker.scala中AllocatedBlocks的实现

```
private[streaming]
case class AllocatedBlocks(streamIdToAllocatedBlocks: Map[Int, Seq[ReceivedBlockInfo]]) {
    def getBlocksOfStream(streamId: Int): Seq[ReceivedBlockInfo] = {
        streamIdToAllocatedBlocks.get(streamId).getOrElse(Seq.empty)
    }
}
```

2）调用 DStreamGraph 的 generateJobs 方法生成 Job，本例中，实际调用的是 ForEachDStream 的 generateJob 方法，见代码清单 9-61 和代码清单 9-62。

代码清单9-61 DStreamGraph的generateJobs方法

```
def generateJobs(time: Time): Seq[Job] = {
    logDebug("Generating jobs for time " + time)
    val jobs = this.synchronized {
        outputStreams.flatMap(outputStream => outputStream.generateJob(time))
    }
    logDebug("Generated " + jobs.length + " jobs for time " + time)
    jobs
}
```

代码清单9-62 ForEachDStream的generateJob方法

```
override def generateJob(time: Time): Option[Job] = {
    parent.getOrCompute(time) match {
        case Some(rdd) =>
            val jobFunc = () => {
                ssc.sparkContext.setCallSite(creationSite)
                foreachFunc(rdd, time)
            }
            Some(new Job(time, jobFunc))
        case None => None
    }
}
```

generateJob 方法中通过 Dstream 的 getOrCompute 方法与各个 Dstream 的 compute 方法组成职责链模式，因而依次调用顺序如下：

ForEachDStream → Dstream. getOrCompute → ShuffledDStream. Compute → Dstream. getOrCompute → MappedDStream. compute → Dstream. getOrCompute → FlatMappedDStream. compute → Dstream. getOrCompute → ReceiverInputDStream. compute。

以上过程见代码清单 9-62 ～代码清单 9-67。

代码清单9-63 Dstream的getOrCompute方法

```
private[streaming] def getOrCompute(time: Time): Option[RDD[T]] = {
    generatedRDDs.get(time).orElse {
        if (isTimeValid(time)) {
            val rddOption = PairRDDFunctions.disableOutputSpecValidation.
                withValue(true) {
                compute(time)
```

```
            }
            ssc.sparkContext.setCallSite(prevCallSite)

            rddOption.foreach { case newRDD =>
                if (storageLevel != StorageLevel.NONE) {
                    newRDD.persist(storageLevel)
                    logDebug(s"Persisting RDD ${newRDD.id} for time $time to
                        $storageLevel")
                }
                if (checkpointDuration != null && (time - zeroTime).isMultipleOf(ch
                    eckpointDuration)) {
                    newRDD.checkpoint()
                    logInfo(s"Marking RDD ${newRDD.id} for time $time for
                        checkpointing")
                }
                generatedRDDs.put(time, newRDD)
            }
            rddOption
        } else {
            None
        }
    }
}
```

代码清单9-64 ShuffledDStream的compute方法

```
override def compute(validTime: Time): Option[RDD[(K,C)]] = {
    parent.getOrCompute(validTime) match {
        case Some(rdd) => Some(rdd.combineByKey[C](
            createCombiner, mergeValue, mergeCombiner, partitioner, mapSideCombine))
        case None => None
    }
}
```

代码清单9-65 MappedDStream的compute方法

```
override def compute(validTime: Time): Option[RDD[U]] = {
    parent.getOrCompute(validTime).map(_.map[U](mapFunc))
}
```

代码清单9-66 FlatMappedDStream的compute方法

```
override def compute(validTime: Time): Option[RDD[U]] = {
    parent.getOrCompute(validTime).map(_.flatMap(flatMapFunc))
}
```

代码清单9-67 ReceiverInputDStream的compute方法

```
override def compute(validTime: Time): Option[RDD[T]] = {
    val blockRDD = {
        if (validTime < graph.startTime) {
            new BlockRDD[T](ssc.sc, Array.empty)
        } else {
            val blockInfos =
```

```
                ssc.scheduler.receiverTracker.getBlocksOfBatch(validTime).get(id).
                    getOrElse(Seq.empty)
            val blockStoreResults = blockInfos.map { _.blockStoreResult }
            val blockIds = blockStoreResults.map { _.blockId.asInstanceOf[BlockId]
                }.toArray

            val resultTypes = blockStoreResults.map { _.getClass }.distinct
            if (resultTypes.size > 1) {
                logWarning("Multiple result types in block information, WAL
                    information will be ignored.")
        }

        if (resultTypes.size == 1 && resultTypes.head == classOf[WriteAheadLogBase
            dStoreResult]) {
            val logSegments = blockStoreResults.map {
                _.asInstanceOf[WriteAheadLogBasedStoreResult].segment
            }.toArray
                new WriteAheadLogBackedBlockRDD[T](ssc.sparkContext,
                    blockIds, logSegments, storeInBlockManager = true, StorageLevel.
                        MEMORY_ONLY_SER)
                } else {
                new BlockRDD[T](ssc.sc, blockIds)
            }
        }
    }
    Some(blockRDD)
}
```

ReceiverInputDStream 的 compute 方法调用 ReceiverTracker 的 getBlocksOfBatch 方法获取 AllocatedBlocks 中的 streamIdToAllocatedBlocks: Map[Int, Seq[ReceivedBlockInfo]]，见代码清单 9-68 和代码清单 9-69。然后取出每个 ReceivedBlockInfo 的 blockStoreResult，再取出每个 blockStoreResult 的 BlockId 及其类型信息，最终生成 BlockRDD。回调 FlatMappedDStream 的 flatMap 方法时会将 BlockRDD 转换为 FlatMappedRDD，回调 MappedDStream 的 map 方法时将 FlatMappedRDD 转换为 MappedRDD，回调 ShuffledDStream 的 combineByKey 方法时将 MappedRDD 转换为 ShuffledMapRDD，最终回调 ForEachDStream 将 RDD 与 jobFunc 封装为 Job。最后将所有返回的 job 封装为 JobSet 后调用 JobScheduler 的 submitJobSet 方法。

代码清单9-68 ReceiverTracker的getBlocksOfBatch方法

```
def getBlocksOfBatch(batchTime: Time): Map[Int, Seq[ReceivedBlockInfo]] = {
    receivedBlockTracker.getBlocksOfBatch(batchTime)
}
```

代码清单9-69 ReceivedBlockTracker的getBlocksOfBatch方法

```
def getBlocksOfBatch(batchTime: Time): Map[Int, Seq[ReceivedBlockInfo]] = synchronized {
    timeToAllocatedBlocks.get(batchTime).map { _.streamIdToAllocatedBlocks
        }.getOrElse(Map.empty)
}
```

submitJobSet方法中将Job封装为JobHandler，见代码清单9-70，然后通过线程执行JobHandler中Job的run方法，见代码清单9-71。

代码清单9-70 JobScheduler的submitJobSet方法

```
def submitJobSet(jobSet: JobSet) {
    if (jobSet.jobs.isEmpty) {
        logInfo("No jobs added for time " + jobSet.time)
    } else {
        jobSets.put(jobSet.time, jobSet)
        jobSet.jobs.foreach(job => jobExecutor.execute(new JobHandler(job)))
        logInfo("Added jobs for time " + jobSet.time)
    }
}
```

代码清单9-71 JobScheduler.scala中JobHandler的run方法

```
private class JobHandler(job: Job) extends Runnable {
    def run() {
        eventActor ! JobStarted(job)
        PairRDDFunctions.disableOutputSpecValidation.withValue(true) {
            job.run()
        }
        eventActor ! JobCompleted(job)
    }
}
```

Job的run方法（见代码清单9-72）实际调用了创建Job时使用的jobFunc函数，进而调用foreachFunc。本例中foreachFunc即为代码清单9-16中的foreachFunc函数。foreachFunc函数中调用了RDD的take方法，而take方法不断调用SparkContext的runJob方法提交任务，见代码清单9-73。

代码清单9-72 Job的run方法

```
def run() {
    result = Try(func())
}
```

代码清单9-73 RDD的take方法

```
def take(num: Int): Array[T] = {
    if (num == 0) {
        return new Array[T](0)
    }

    val buf = new ArrayBuffer[T]
    val totalParts = this.partitions.length
    var partsScanned = 0
    while (buf.size < num && partsScanned < totalParts) {
        var numPartsToTry = 1
        if (partsScanned > 0) {
            if (buf.size == 0) {
```

```
                numPartsToTry = partsScanned * 4
            } else {
              numPartsToTry = Math.max((1.5 * num * partsScanned / buf.size).toInt -
                  partsScanned, 1)
              numPartsToTry = Math.min(numPartsToTry, partsScanned * 4)
            }
          }

          val left = num - buf.size
          val p = partsScanned until math.min(partsScanned + numPartsToTry,
            totalParts)
          val res = sc.runJob(this, (it: Iterator[T]) => it.take(left).toArray, p,
            allowLocal = true)

          res.foreach(buf ++= _.take(num - buf.size))
          partsScanned += numPartsToTry
        }

        buf.toArray
    }
```


注意 虽然每个 job 只会调用一次 take 方法，但是如果传递给 take 的参数值较大，可能导致一次运行 runJob 返回的记录数不足，那么 take 方法中的循环体将会运行多次 runJob，直到 buf 的 size 达到 take 参数的要求。此外，take 方法还会尝试从多个 partition 上运行 runJob。

JobScheduler 产生并提交执行的任务要加工的数据在哪里？在生成 Job 的过程中，调用 ReceiverInputDStream 的 compute 方法会生成 WriteAheadLogBackedBlockRDD 或者 BlockRDD。take 方法中执行 runJob 也会触发 6.1 节的过程，迭代执行各个 RDD 的 compute 方法。以默认生成的 BlockRDD 为例，它的 compute 方法通过调用 BlockManager 的 get 方法获取 blockPushingThread 生成的 Block，这样就有了要计算的数据了，见代码清单 9-74。

BlockManager 的 get 方法已在 4.8.11 节介绍过，此处不再赘述。

代码清单9-74 BlockRDD的compute方法

```
override def compute(split: Partition, context: TaskContext): Iterator[T] = {
    assertValid()
    val blockManager = SparkEnv.get.blockManager
    val blockId = split.asInstanceOf[BlockRDDPartition].blockId
    blockManager.get(blockId) match {
        case Some(block) => block.data.asInstanceOf[Iterator[T]]
        case None =>
          throw new Exception("Could not compute split, block " + blockId + " not found")
    }
}
```

流式计算的整个过程远比一般的执行流程要长，所以源码分析过程需要读者仔细认真。

为了让读者看得明白，笔者尽可能保留 Spark 中的源码。这样一来势必增加了一些篇幅，但降低了读者阅读的门槛，笔者认为还是值得的。

我们在 nc 命令下输入图 9-11 中的字符。

Spark Streaming 的输出片段如图 9-12 所示。

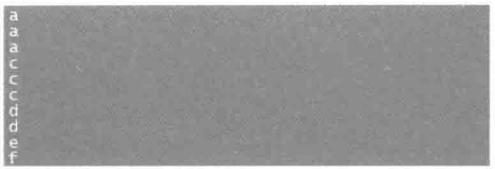


图 9-11 nc 命令下输入的字符

```
-------------------------------------------
(a,3)

-------------------------------------------
Time: 1437476660000 ms
-------------------------------------------
(d,2)
(f,1)
(e,1)
(c,3)

-------------------------------------------
Time: 1437476670000 ms
-------------------------------------------
```

图 9-12 Spark Streaming 的输出片段

这里笔者需要回答欠下读者的一个问题，还记得本机一开始的 local[2] 吗？在回答这个问题前，先回顾下 5.4.5 节介绍 LocalActor 接收到 reviveOffers 消息的处理——使用代码清单 3-36 中的 reviveOffers 创建 WorkerOffer。WorkerOffer 的创建依赖于变量 freeCores。Local[2] 部署模式将设置 freeCores 的初始值为 2。

如果你细心查看本节的目录设置，就不难发现答案。9.5.2 节详解了 startReceivers 方法中执行的 runJob 方法，这个任务将驻留在 executor 中，构成 Spark Streaming 的执行环境。此时，freeCores 减小为 1。9.5.3 节的 JobGenerator 不断创建 Job，并执行 runJob，不过这些 Job 是顺序提交的，所以还有一个任务占用一个 CPU 内核。因此 local 模式下设置的内核数不得小于 2。

9.6 窗口操作

Spark Streaming 还提供了窗口计算，即可以转换滑动窗口内的数据。图 9-13 展示了滑动窗口模型。

每次窗口都在源 Dstream 上滑动，窗口内的 RDD 将被合并生成窗口 Dstream 内的 RDD。图中展示的例子以三个最近的数据单元作为窗口操作，并且窗口滑动两个数据单元。这说明任何窗口操作都需要指定两个参数：

- 窗口长度：窗口的周期长度。
- 滑动间隔：窗口转换的间隔。

根据 9.5.3 节的描述，在生成任

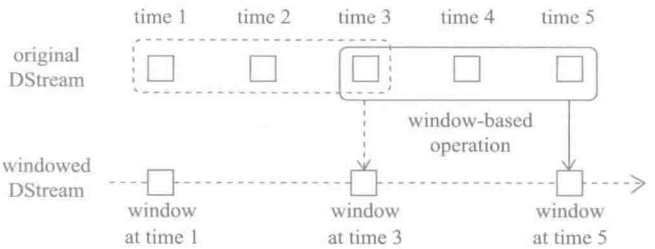


图 9-13 滑动窗口模型

务阶段发生了 Dstream 的迭代转换 RDD 的过程。我们只需要查看 WindowedDStream 是如何在这一阶段转换的就能明白窗口滑动的实现，见代码清单 9-75。

代码清单9-75 WindowedDStream的compute方法

```
override def compute(validTime: Time): Option[RDD[T]] = {
    val currentWindow = new Interval(validTime - windowDuration + parent.
       slideDuration, validTime)
    val rddsInWindow = parent.slice(currentWindow)
    val windowRDD = if (rddsInWindow.flatMap(_.partitioner).distinct.length == 1) {
        logDebug("Using partition aware union for windowing at " + validTime)
        new PartitionerAwareUnionRDD(ssc.sc, rddsInWindow)
    } else {
        logDebug("Using normal union for windowing at " + validTime)
            new UnionRDD(ssc.sc,rddsInWindow)
    }
    Some(windowRDD)
}
```

WindowedDStream 的 compute 方法的执行步骤如下。

1）计算滑动间隔，滑动时间由起始时间和结束时间组成，见代码清单 9-76。

起始时间 = 当前有效时间 – 窗口周期时长 + 父级 Dstream 的滑动周期时长

结束时间 = 当前有效时间

2）滑动窗口，其实现见代码清单 9-77 和代码清单 9-78。

3）生成 windowRDD，如果滑动窗口的数量是 1，则创建 PartitionerAwareUnionRDD，否则创建 UnionRDD。

代码清单9-76 Interval的实现

```
private[streaming]
class Interval(val beginTime: Time, val endTime: Time) {
    def this(beginMs: Long, endMs: Long) = this(new Time(beginMs), new Time(endMs))

    def duration(): Duration = endTime - beginTime

    def + (time: Duration): Interval = {
        new Interval(beginTime + time, endTime + time)
    }

    def - (time: Duration): Interval = {
        new Interval(beginTime - time, endTime - time)
    }

    def < (that: Interval): Boolean = {
        if (this.duration != that.duration) {
            throw new Exception("Comparing two intervals with different durations ["
                + this + ", "
            + that + "]")
        }
```

```
        this.endTime < that.endTime
    }

    def <= (that: Interval) = (this < that || this == that)

    def > (that: Interval) = !(this <= that)

    def >= (that: Interval) = !(this < that)

    override def toString = "[" + beginTime + ", " + endTime + "]"
}
```

代码清单9-77 Dstream的slice方法

```
def slice(interval: Interval): Seq[RDD[T]] = {
    slice(interval.beginTime, interval.endTime)
}
```

代码清单9-78 Dstream重载的slice方法

```
def slice(fromTime: Time, toTime: Time): Seq[RDD[T]] = {
    if (!isInitialized) {
        throw new SparkException(this + " has not been initialized")
    }
    if (!(fromTime - zeroTime).isMultipleOf(slideDuration)) {
       logWarning("fromTime (" + fromTime + ") is not a multiple of slideDuration ("
       + slideDuration + ")")
    }
    if (!(toTime - zeroTime).isMultipleOf(slideDuration)) {
        logWarning("toTime (" + fromTime + ") is not a multiple of slideDuration ("
        + slideDuration + ")")
    }
    val alignedToTime = toTime.floor(slideDuration)
    val alignedFromTime = fromTime.floor(slideDuration)

    logInfo("Slicing from " + fromTime + " to " + toTime +
        " (aligned to " + alignedFromTime + " and " + alignedToTime + ")")

    alignedFromTime.to(alignedToTime, slideDuration).flatMap(time => {
        if (time >= zeroTime) getOrCompute(time) else None
    })
}
```

9.7 应用举例

本章一开始就列举过 Spark Streaming 支持多种数据源，其中包括 MQTT（message queuing telemetry transport，消息队列遥测传输）。MQTT 是 IBM 开发的一个即时通信协议，有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有联网物品和外部连接起来，

被用来当做传感器和致动器（比如通过 Twitter 让房屋联网）的通信协议。mosquitto 是一款实现了 MQTT v3.1 的开源消息代理软件，它提供轻量级的，支持可发布 / 可订阅的消息推送模式，使设备对设备之间的短消息通信变得简单。

本节就以 mosquitto 作为 MQTT 数据源，使用 Spark Streaming 对 mosquitto 推送的数据执行 word count 计算。

9.7.1 安装 mosquitto

mosquitto 的下载地址：http://mosquitto.org/download/

笔者从 mosquitto 官网提供的下载地址选择 mosquitto-1.4.2.tar.gz 的源码包进行下载。下载方法如下：

```
wget http://mosquitto.org/files/source/mosquitto-1.4.2.tar.gz
```

移动到选好的安装目录，如：

```
mv mosquitto-1.4.2.tar.gz ../install/
```

进入安装目录并解压缩：

```
cd ../install/
tar zxvf mosquitto-1.4.2.tar.gz
```

进入 mosquitto 根目录：

```
cd mosquitto-1.4.2
```

编译、安装：

```
make
sudo make install
```

9.7.2 启动 mosquitto

使用 mosquitto 命令启动 mosquitto，如图 9-14 所示。

```
[        a@v218142118 ~/install/mosquitto-1.4.2/src]$ ./mosquitto
1437554392: mosquitto version 1.4.2 (build date 2015-07-22 16:37:44+0800) starting
1437554392: Using default config.
1437554392: Opening ipv4 listen socket on port 1883.
1437554392: Opening ipv6 listen socket on port 1883.
1437554392: Warning: Address family not supported by protocol
1437556172: New connection from 10.62.63.18 on port 1883.
1437556172: New client connected from 10.62.63.18 as          1437556172139 (c1, k60).
1437556187: Socket error on client          .1437556172139, disconnecting.
```

图 9-14 使用 mosquitto 命令启动 mosquitto

可以看到 mosquitto 默认的监听端口是 1883，使用 netstat –nat 命令查看 mosquitto 的监听协议，如图 9-15 所示。

所以 mosquitto 的 broker 地址是 tcp://10.218.142.118:1883。

```
[          @v218142118 ~]$ netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0      0 0.0.0.0:12201               0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:873                 0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:47435               0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:111                 0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:80                  0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:8080                0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:17776             0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:17777             0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:17779             0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:8182              0.0.0.0:*                   LISTEN
tcp        0      0 10.218.142.118:8182         0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:22                  0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:7001                0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:1883                0.0.0.0:*                   LISTEN
tcp        0      0 0.0.0.0:8000                0.0.0.0:*                   LISTEN
tcp        0      0 127.0.0.1:15776             0.0.0.0:*                   LISTEN
```

图 9-15 使用 netstat –nat 命令查看 mosquitto 的监听协议

9.7.3 MQTTWordCount

Spark 源码中已有现成的使用 MQTT 的例子 MQTTWordCount，见代码清单 9-79。

代码清单9-79 MQTT的例子MQTTWordCount

```
object MQTTWordCount {

    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println(
                "Usage: MQTTWordCount <MqttbrokerUrl> <topic>")
            System.exit(1)
        }

        val Seq(brokerUrl, topic) = args.toSeq
        val sparkConf = new SparkConf().setAppName("MQTTWordCount").setMaster
           ("local[2]");
        val ssc = new StreamingContext(sparkConf, Seconds(2))
        val lines = MQTTUtils.createStream(ssc, brokerUrl, topic, StorageLevel.
           MEMORY_ONLY_SER_2)

        val words = lines.flatMap(x => x.toString.split(" "))
        val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
        wordCounts.print()
        ssc.start()
        ssc.awaitTermination()
    }
}
```

MQTTWordCount 实际是 mosquitto 中消息或者说是主题的订阅者，所以我们需要给 MQTTWordCount 这个 Scala 应用程序增加启动参数设置 mosquitto 的 broker 地址和主题，我们以 hello 作为一个简单的主题，如图 9-16 所示。

mosquitto 本身不会产生消息，需要一个发布者，不断向主题 hello 发送消息。Spark 源码中已有现成的发布者 MQTTPublisher，但是由于 MQTTPublisher 发送消息频率过高，mosquitto 会拒绝接收，所以笔者在代码中多加了“Thread.sleep(10);”，见代码清单 9-80。

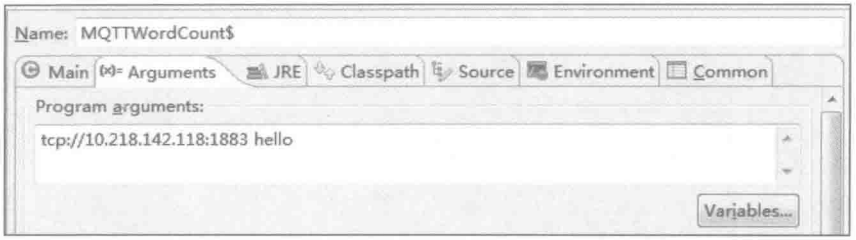


图 9-16 增加启动参数设置 mosquitto 的 broker 地址和主题

代码清单9-80 MQTTWordCount.scala中MQTTPublisher的实现

```
object MQTTPublisher {

    var client: MqttClient = _

    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println("Usage: MQTTPublisher <MqttBrokerUrl> <topic>")
            System.exit(1)
        }

        StreamingExamples.setStreamingLogLevels()

        val Seq(brokerUrl, topic) = args.toSeq

        try {
           var peristance:MqttClientPersistence =new MqttDefaultFilePersistence("/
              tmp")
           client = new MqttClient(brokerUrl, MqttClient.generateClientId(),
               peristance)
        } catch {
          case e: MqttException => println("Exception Caught: " + e)
        }

        client.connect()

        val msgtopic: MqttTopic = client.getTopic(topic)
        val msg: String = "hello mqtt demo for spark streaming"

        while (true) {
            val message: MqttMessage = new MqttMessage(String.valueOf(msg).
               getBytes("utf-8"))
            msgtopic.publish(message)
            println("Published data. topic: " + msgtopic.getName() + " Message: " +
                message)
            Thread.sleep(10);
        }
        client.disconnect()
    }
}
```

MQTTPublisher 也需要设置 mosquitto 的 broker 地址和主题，设置过程与 MQTTWordCount 一样。

我们首先启动 MQTTWordCount，然后启动 MQTTPublisher，MQTTPublisher 会不断发送消息：

```
hello mqtt demo for spark streaming
```

MQTTWordCount 输出结果的片段如图 9-17 所示。

```
-------------------------------------------
Time: 1437558414000 ms
-------------------------------------------
(mqtt,66)
(hello,66)
(streaming,66)
(spark,66)
(demo,66)
(for,66)

-------------------------------------------
Time: 1437558416000 ms
-------------------------------------------
(mqtt,78)
(hello,78)
(streaming,78)
(spark,78)
(demo,78)
(for,78)
```

图 9-17 MQTTWordCount 输出结果的片段

9.8 小结

Spark Streaming 支持了大数据中常用的业务场景——流式计算。已经内置了对多种数据源的支持，同时提供 PluggableInputDStream 方便用户进行更多数据源的扩展，提升了 Spark Streaming 的可扩展性。本章之前的内容介绍的 Spark 运行任务都只会在 Executor 中驻留一段时间。Spark Streaming 为了解决数据源的持续性，首先创建一个驻留在 Executor 内的任务提供数据流的持续接入，并不断生成任务去拉取最新的数据并计算。这些都是 Spark Streaming 与其他类型的任务所不同的。

Chapter 10 第 10 章

图 计 算

法自画生，画自法立，无法非也，终于有法亦非也。

——潘天寿

本章导读

2010 年，Google 提出了适合复杂机器学习的分布式图计算 Pregel 框架。同年，CMU 的 Select 实验室提出了 GraphLab 框架，GraphLab 是面向机器学习的流处理并行框架。GraphLab 基于最初的并行概念实现了 1.0 版本，在机器学习的流处理并行性能方面得到很大的提升，并引起业界的广泛关注。2012 年 GraphLab 升级到 2.1 版本，进一步优化了其并行模型，尤其是自然图的并行性能得到显著改进。

早在 0.5 版本，Spark 就带了一个小型的 Bagel 模块，提供了类似 Pregel 的功能。随着对图计算需求的增大，Spark 开始设计自己的分布式图计算框架 GraphX。通过扩展 RDD，实现了图的高层次抽象——由顶点和边构成的属性图。为了支持图计算，GraphX 暴露了一个包含基础操作（例如 subgraph、joinVertices 和 aggregateMessages）的集合以及对 Pregel API 的优化版本。现在，GraphX 还在不断增加图的算法集合并简化图的分析任务。

10.1 Spark GraphX 总体设计

Spark 中目前与图计算有关的部分包括：Bagel 和 GraphX。本章主要介绍 GraphX，在正式开始之前，我们先从计算模型角度介绍一些概念。

10.1.1　图计算模型

目前的图计算框架基本上都遵循 BSP（bulk synchronous parallell，整体同步并行）计算模式。BSP 模式如图 10-1 所示。

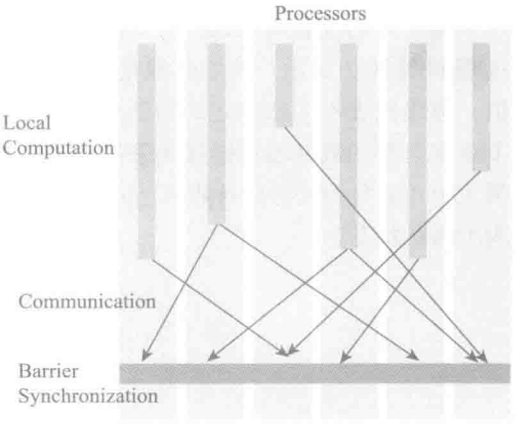


图 10-1　BSP 模式

BSP 模式中有以下概念：

- Processors：并行计算进程，它对应到集群中的多个节点，每个节点可以有多个 Processor；
- LocalComputation：单个 Processor 的计算，每个 Processor 都会切分一些节点作计算；
- Communication：Processor 之间的通信。在 BSP 模型中，对图节点的访问分布到了不同的 Processor 中，有时哪怕是关系紧密具有局部聚类特点的节点也未必会分布到同一个 Processor 或同一个集群节点上，所有需要用到的数据都需要通过 Processor 之间的消息传递来实现同步；
- BarrierSynchronization：栅栏同步，每一次同步标志着上一个超步的完成和下一个超步的开始；
- Superstep：超步，对应于 BSP 的一次计算迭代。

基于 BSP 模式，目前有两种比较成熟的图计算模型：Pregel 模型和 GAS 模型。

由于 GraphX 主要基于 Pregel 实现，所以我们重点理解下 Pregel 模型的原理。在 Pregel 计算模型中，输入是一个有向图，该有向图的每一个顶点都有一个标识符以及与之对应的值。每一条有向边都和其源顶点关联，并且记录自身的值和目标顶点的标识符。

一个典型的 Pregel 计算过程如下：

1）读取图数据并对图初始化；

2）当图被初始化完毕，执行一系列的超步直到整个计算结束，这些超步之间通过一些全局的同步点分隔；

3）输出计算结果。

在每个超步中，顶点的计算都是并行的，每个顶点执行相同的用于表达给定算法逻辑的用户自定义函数。每个顶点都可以修改自身及出边的状态，接收前一个超步（S–1）中发送给它的消息，并发送消息给其他顶点（这些消息将会在下一个超步中被接收），甚至是修改整个图的拓扑结构。边在这种计算模式中并不是核心对象，没有相应的计算运行在其上。

算法结束的时机取决于所有的顶点是否都已经投票（vote）标识自身已经达到 halt 状态。在第 0 个超步，所有顶点都处于 active 状态，所有的 active 顶点都会参与对应超步中的计算。顶点通过将其自身的 status 设置成 halt 来表示它已经不再 active。这就表示该顶点没有进一步的计算需要执行，除非该顶点收到其他顶点传送的消息，否则 Pregel 框架将不会在接下来的

超步中执行该顶点。如果顶点在接收到消息后进入 active 状态，那么在随后的计算中该顶点必须显式的 deactive。整个计算在所有顶点都达到 inactive 状态，并且没有消息要传送时结束。这种简单的状态机如图 10-2 所示。

为了帮助读者对 Pregel 有更深入的了解，以 Pregel 的最大值计算过程为例，其计算执行过程如图 10-3 所示，其中实线箭头表示图的链接关系，而节点内的数字代表节点的当前数值，虚线代表不同超步之间的消息传递关系，灰色节点是不活跃节点。在每个超步中，每个节点将自身的数值通过链接关系发送给其他节点，接收到消息的节点将接收到的所有消息的最大值（记为 M）与自身数值（记为 C）比较，如果 $M > C$，那么 $C = M$；否则当前节点变为不活跃状态。

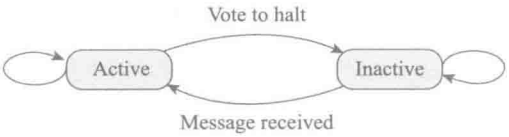


图 10-2 BSP 模式的简单状态机

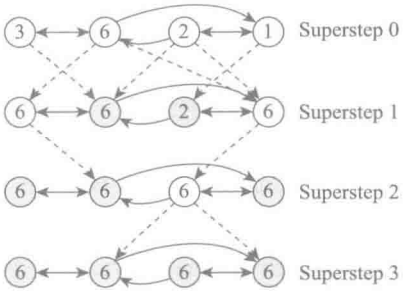


图 10-3 Pregel 的最大值计算

我们从左到右将 4 个节点依次记为 A、B、C、D，并对图 10-3 中 Pregel 的最大值计算作个简单介绍。

1）第 0 步超步，所有节点都是活跃的。从左到右，A 节点收到 B 节点的消息 6，发现 6 比 A 节点自身数值 3 大，A 节点将会在进入下一个超步前将自身数值更新为 6；B 节点同时收到 A 节点和 C 节点的消息，这两条消息的最大值是 3，3 比 B 节点自身数值 6 小，B 节点将会在进入下一个超步前将自身设置为不活跃状态；C 节点收到 D 节点的消息 1，发现 1 比 C 节点自身数值 2 小，C 节点将会在进入下一个超步前将自身设置为不活跃状态；D 节点同时收到 B 节点和 C 节点的消息，这两条消息的最大值是 6,6 比 D 节点自身数值 1 大，D 节点将会在进入下一个超步前将自身数值更新为 6。

2）第 1 步超步，A、D 节点是活跃的。A 节点收到 B 节点的消息 6，发现 6 与 A 节点自身数值 6 一样大，A 节点将会在进入下一个超步前将自身设置为不活跃状态；B 节点同时收到 A 节点和 C 节点的消息，这两条消息的最大值是 6，6 与 B 节点自身数值 6 一样大，B 节点在下一个超步自身仍将处于不活跃状态；C 节点收到 D 节点的消息 6，发现 6 比 C 节点自身数值 2 大，C 节点将会在进入下一个超步前将自身数值更新为 6，并处于活跃状态；D 节点同时收到 B 节点和 C 节点的消息，这两条消息的最大值是 6，6 与 D 节点自身数值 6 一样大，D 节点将会在进入下一个超步前将自身设置为不活跃状态。

3）第 2 步超步，C 节点是活跃的。A 节点收到 B 节点的消息 6，发现 6 与 A 节点自身数

值6一样大，A节点在下一个超步自身仍将处于不活跃状态；B节点同时收到A节点和C节点的消息，这两条消息的最大值是6，6与B节点自身数值6一样大，B节点在下一个超步自身仍将处于不活跃状态；C节点收到D节点的消息6，发现6与C节点自身数值6一样大，C节点将会在进入下一个超步前将自身设置为不活跃状态；D节点同时收到B节点和C节点的消息，这两条消息的最大值是6，6与D节点自身数值6一样大，D节点在下一个超步自身仍将处于不活跃状态。

4）第3步超步，所有节点都是不活跃的。由于所有节点都到达不活跃状态，任务结束。

10.1.2 属性图

什么是属性图？顶点和边都带有属性信息的图。多个边有可能共享同一个源或者顶点。每个顶点的VertexID由一个64位长度的标识符表示。GraphX不会对顶点标识符作任何排序约束。同样，每个边具有相应的源和目的顶点的标识符。当顶点和边是原生数据类型（比如long、double等）时，将它们存储在专门的数组中，这样可减小内存占用。

在某些情况下，用户希望一张图中的顶点拥有不同的属性类型。这可以通过继承来实现。例如，以用户和产品为二分图为例，我们可以像代码清单10-1这样做。

代码清单10-1 自定义顶点属性

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends
    VertexProperty
// 使用自定义属性生成的graph可能会是下面的类型:
var graph: Graph[VertexProperty, String] = null
```

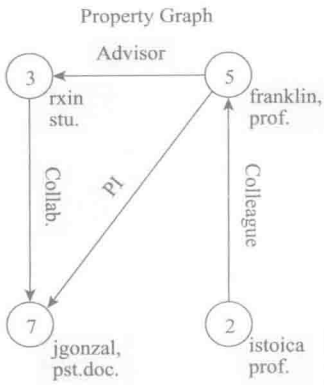
属性图本身是不可改变的，只能通过创建新的、期望改变的图来达到改变图中的值或者结构。属性图还支持分布式和容错。原始图的主要部分在新图中会被重用，以降低固有功能数据结构的成本。executor使用范围划分顶点的方式分割图。图中的每个partition都可以在不同的机器上在失败后重建。

我们来看看Graph的基本结构，见代码清单10-2。

代码清单10-2 Graph的基本结构

```
abstract class Graph[VD: ClassTag, ED: ClassTag] protected () extends Serializable {
    @transient val vertices: VertexRDD[VD]
    @transient val edges: EdgeRDD[ED]
    @transient val triplets: RDD[EdgeTriplet[VD, ED]]
```

VertexRDD[VD]和EdgeRDD[ED]分别继承了RDD[(VertexID, VD)]和RDD[Edge[ED]]，并围绕图形计算和存储，提供了额外的功能的内部优化。假设要构建一个属性图，其中包括对GraphX项目的各种合作者。顶点属性可能包含用户名和职业，可以在边上添加注释描述合作者间的关系，如图10-4所示。



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

图 10-4 属性图的构建

最终的图将会是如下的类型签名。

```
val userGraph: Graph[(sString, String), String]
```

可以使用文件、RDD 等多种方式构成属性图。最通用的方法可能是使用 Graph 对象。例如，可以使用代码清单 10-3 中的 RDD 集合构造一个图。

代码清单10-3 构造属性图示例

```
// 假设SparkContext已经构建完成
val sc: SparkContext
// 为所有顶点创建RDD
val users: RDD[(VertexId, (String, String))] =
    sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                         (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// 为所有边创建RDD
val relationships: RDD[Edge[String]] =
    sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                         Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// 定义一个默认用户与未知的用户的关系
val defaultUser = ("John Doe", "Missing")
// 构建初始化的Graph
val graph = Graph(users, relationships, defaultUser)
```

在上边的例子中，我们看到边同时拥有源顶点的 VertexID 和目的顶点的 VertexID，还拥有自身的属性。还能直接通过 Graph 对象对其中的顶点和边进行条件过滤和计数，见代码清单 10-4。

代码清单10-4 过滤与计数

```
val graph: Graph[(String, String), String] // 上边例子中构建的Graph
// 对所有职位是 postdocs的用户计数
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// 统计出源顶点的VertexId大于目标顶点的VertexId的边数
graph.edges.filter(e => e.srcId > e.dstId).count
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

还可以使用 SQL 表达式，如代码清单 10-5。

代码清单10-5 SQL表达式

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

EdgeTriplet 继承了 Edge，并且添加了 srcAttr 和 dstAttr 用来保护源顶点和目的顶点的属性。Vertices、Edges 和 Triplets 可以用图 10-5 表示。

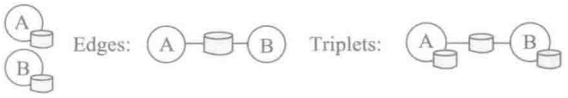


图 10-5 Vertices、Edges 和 Triplets 的比较

代码清单 10-6 演示了如何使用 EdgeTriplet。

代码清单10-6 EdgeTriplet使用示例

```
val graph: Graph[(String, String), String] //上边例子中构建的Graph
// 使用triplets创建RDD
val facts: RDD[String] =
    graph.triplets.map(triplet =>
        triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

10.1.3 GraphX 的类继承体系

前一节简单介绍了 GraphX 中一些 API 的使用，为了从宏观上对 GraphX 的 API 有个把握，图 10-6 列出了 GraphX 的类继承体系，从中看出 Graph 是最核心的类，其具体实现为 GraphImpl，GraphOps 是符合 Scala 语言风格的实现。Pregel 是 Spark 对 Pregel 模型的具体实现，其中已经包括了最短路径（ShortestPaths）、网页排名（PageRank）、关联组件（ConnectedComponents）等。Graph 还实现了三角形计数（TriangleCount）。

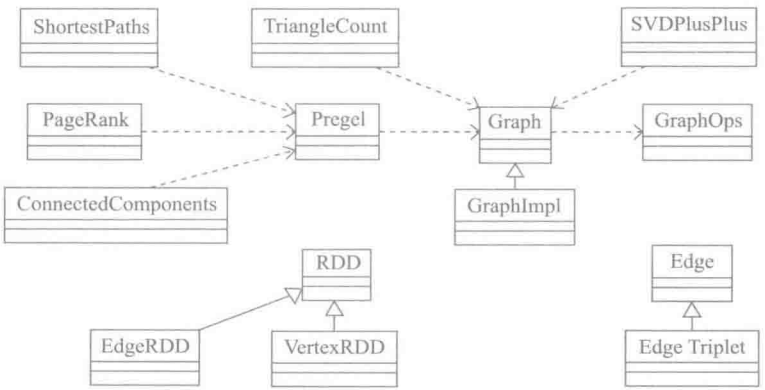


图 10-6 GraphX 的类继承体系

10.2 图操作

图的操作包括两部分：

- Graph 中定义的经过优化的核心操作；
- GraphOps 中定义的为了方便用组合方式表示的操作。

Graph 中的部分操作实际隐式使用了 GraphOps 中的操作。

10.2.1 属性操作

类似于 RDD 的 map 操作，属性图含有代码清单 10-7 中列出的操作。

代码清单10-7 属性操作

```
class Graph[VD, ED] {
    def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
    def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

上面的每个函数都会对原图操作后产生一个新图。这些函数都有一个关键的特性，那就是最终产生的 Graph 可以重用原生 Graph 的结构。下面的代码片段在逻辑上是一样的，但是代码清单 10-8 中的代码片段不会保留结构化指数并且不会从 GraphX 系统的优化中受益。

代码清单10-8 不重用原来Graph的结构

```
val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }
val newGraph = Graph(newVertices, graph.edges)
```

相反，代码清单 10-9 中使用 mapVertices 函数会保留结构化指数。

代码清单10-9 重用原来Graph的结构

```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

10.2.2 结构操作

目前 GraphX 仅支持一些简单的结构操作，Spark 官网说以后会增加更多。这些基本的结构操作见代码清单 10-10。

代码清单10-10 结构操作

```
class Graph[VD, ED] {
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
    def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

这里简要介绍下这些结构操作的功能：

- reverse：返回包含了所有边反向的新图。这个操作很有用，例如可以计算反向的 PageRank。
- subgraph：返回顶点满足函数 vpred 的过滤条件和边满足函数 epred 的过滤条件的新图。
- mask：构建一个包含了当前 Graph 和输入 Graph 都拥有的顶点和边的子图。
- groupEdges：合并连接了同一对顶点的边后产生的新图。被合并的边的权重也会合并。此功能常常用于减小 Graph 的尺寸。

10.2.3 连接操作

很多情况下都需要与其他 RDD 集合的连接操作，可以使用代码清单 10-11 中的这些操作。

代码清单10-11 连接操作

```
class Graph[VD, ED] {
    def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
       : Graph[VD, ED]
    def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD,
       Option[U]) => VD2)
      : Graph[VD2, ED]s
}
```

这里简要介绍下这些连接操作的功能：

- joinVertices：对顶点和输入的 RDD 做连接操作，返回包含了对连接结果应用 map 函数后的顶点组成的 Graph。连接时没有匹配的顶点，保留原来的值。
- outerJoinVertices：与 joinVertices 方法类似，不同的是 map 函数会应用到所有的顶点。

joinVertices 的使用示例见代码清单 10-12。

代码清单10-12 joinVertices的使用示例

```
val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
    graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
    (id, oldCost, extraCost) => oldCost + extraCost)
```

outerJoinVertices 的使用示例见代码清单 10-13。

代码清单10-13 outerJoinVertices的使用示例

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
    outDegOpt match {
        case Some(outDeg) => outDeg
        case None => 0 // 出度为none意味着出度为0
    }
}
```

10.2.4 聚合操作

在很多 Graph 的分析任务中，聚合兄弟顶点的信息是关键步骤。例如，想要知道每个用户的粉丝数以及这些粉丝的平均年龄，这就会用到聚合操作。很多迭代图的算法（例如，PageRank、Shortest Path、Connected component）都会多次聚合相邻顶点的属性。

1. 聚合消息

GraphX 的核心聚合操作是 aggregateMessages，这个操作应用用户定义的 sendMsg 函数到 Graph 的每个 EdgeTriplet 上，然后使用 mergeMsg 函数在目的顶点上聚合。aggregateMessages 的接口定义见代码清单 10-14。

代码清单10-14 aggregateMessages的接口定义

```
class Graph[VD, ED] {
    def aggregateMessages[Msg: ClassTag](
        sendMsg: EdgeContext[VD, ED, Msg] => Unit,
        mergeMsg: (Msg, Msg) => Msg,
        tripletFields: TripletFields = TripletFields.All)
      : VertexRDD[Msg]
}
```

GraphX 的早期版本提供了 mapReduceTriplets 来处理聚合操作。aggregateMessages 相比于 mapReduceTriplets 有很大的性能提升，所以 Spark 官网建议将 mapReduceTriplets 迁移到 aggregateMessages。

现在我们来计算每个用户的粉丝的平均年龄，见代码清单 10-15。

代码清单10-15 aggregateMessages使用示例

```
// 导入随机graph生成库
import org.apache.spark.graphx.util.GraphGenerators
// 创建一个graph，使用age作为顶点属性。为简单起见，使用随机的graph
val graph: Graph[Double, Int] =
    GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) =>
        id.toDouble )
// 计算年龄比用户自己大的粉丝的数量和所有粉丝的年龄总和
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
    triplet => { // Map函数
        if (triplet.srcAttr > triplet.dstAttr) {
            // 每个用户计数为1，将计数与年龄一起发送给目的顶点
            triplet.sendToDst(1, triplet.srcAttr)
        }
    },
    // 累加计数与年龄
    (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce函数
)
// 每个顶点收到的消息，用粉丝总年龄除以粉丝总数，得到粉丝平均年龄
val avgAgeOfOlderFollowers: VertexRDD[Double] =
    olderFollowers.mapValues( (id, value) => value match { case (count, totalAge)
```

```
        => totalAge / count } )
// 显示计算结果
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

2. 计算度数

一种常见的聚合任务是计算每个顶点的度，即每个顶点相邻的边数。人们常常需要知道每个顶点的出度、入度以及度数。GraphOps 中包含了计算每个顶点的度数的操作集合。以计算图中所有顶点的最大出度、最大入度以及最大度数为例，见代码清单 10-16。

代码清单10-16 计算度数示例

```
// 首先定义一个reduce函数用于计算最大度数
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
    if (a._2 > b._2) a else b
}
// 计算每种度数的最大值的顶点
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

3. 收集相邻顶点

在有些情况下需要收集每个顶点的相邻顶点以及相邻顶点的属性，可以使用 collectNeighborIds 和 collectNeighbors，见代码清单 10-17。

代码清单10-17 收集相邻顶点示例

```
class GraphOps[VD, ED] {
    def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array
       [VertexId]]
    def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array
       [(VertexId, VD)] ]
}
```

由于这些聚合操作都通过复制信息，并且需要大量的通信，所以十分消耗资源。尽可能将这些操作转换为使用 aggregateMessages。

10.3 Pregel API

Graph 由于顶点的属性都依赖于相邻顶点的属性，而这些相邻顶点的属性又依赖于它们的邻居的属性，所以 Graph 是一种迭代的数据结构。结果很多重要的算法都迭代地重复计算每个顶点的属性，直到到达一个定点条件为止。图的并行抽象已经用来表达这些迭代算法。GraphX 提供了多种多样的 Pregel API，下面展示 Pregel API 的实现，见代码清单 10-18。

代码清单10-18 GraphOps的apply方法

```
def apply[VD: ClassTag, ED: ClassTag, A: ClassTag]
    (graph: Graph[VD, ED],
```

```
      initialMsg: A,
      maxIterations: Int = Int.MaxValue,
      activeDirection: EdgeDirection = EdgeDirection.Either)
     (vprog: (VertexId, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED] =
{
      var g = graph.mapVertices((vid, vdata) => vprog(vid, vdata, initialMsg)).cache()
      // compute the messages
      var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
      var activeMessages = messages.count()
      // Loop
      var prevG: Graph[VD, ED] = null
      var i = 0
      while (activeMessages > 0 && i < maxIterations) {
          // Receive the messages. Vertices that didn't get any messages do not
              appear in newVerts.
          val newVerts = g.vertices.innerJoin(messages)(vprog).cache()
          // Update the graph with the new vertices.
          prevG = g
          g = g.outerJoinVertices(newVerts) { (vid, old, newOpt) => newOpt.getOrElse
             (old) }
          g.cache()
          val oldMessages = messages
          messages = g.mapReduceTriplets(sendMsg, mergeMsg, Some((newVerts,
              activeDirection))).cache()
          activeMessages = messages.count()
          logInfo("Pregel finished iteration " + i)
          // Unpersist the RDDs hidden by newly-materialized RDDs
          oldMessages.unpersist(blocking=false)
          newVerts.unpersist(blocking=false)
          prevG.unpersistVertices(blocking=false)
          prevG.edges.unpersist(blocking=false)
          // count the iteration
          i += 1
      }
      g
    } // end of apply
} // end of class Pregel
```

Pregel 有两个参数列表（形如 graph.pregel(list1)(list2)）。第一个参数列表包含的配置参数包括：

- initialMsg：初始化消息；
- maxIterations：最大迭代次数；
- activeDir：发送消息的方向，即沿着出边的方向还是入边的方向。

第二个参数列表包含用户定义的函数，例如：

- vprog：顶点处理函数；
- sendMsg：计算消息；

❑ mergeMsg：合并消息。

本节首先介绍 Dijkstra 算法，然后给出 Pregel API 如何实现此算法。

10.3.1　Dijkstra 算法

Dijkstra 算法又称为单源最短路径，所谓单源是在一个有向图中，从一个顶点出发，求该顶点至所有可到达顶点的最短路径问题。先用来介绍 Dijkstra 算法的处理步骤[⊖]：

1）假设有向图是一个拥有 6 个顶点的图，边的连接如图 10-7 所示。1 号顶点作为我们的出发顶点，5 号顶点作为我们的目的顶点。初始时，除了出发顶点的距离等于 0，其他顶点的距离都设置为 ∞（无穷）。

2）计算路径 1 → 2 的长度，如图 10-8 所示。算得 1 → 2 的长度是 7，如图 10-9 所示。

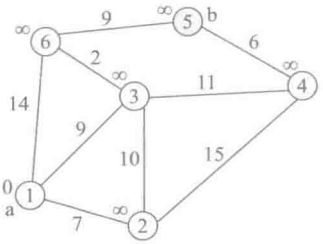


图 10-7　有 6 个顶点的有向图

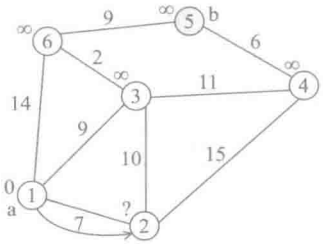


图 10-8　计算路径 1 → 2 的长度

3）计算路径 1 → 3 的长度，如图 10-10 所示。算得 1 → 3 的长度是 9，如图 10-11 所示。

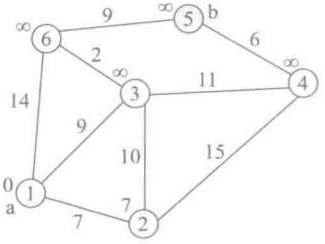


图 10-9　路径 1 → 2 的长度是 7

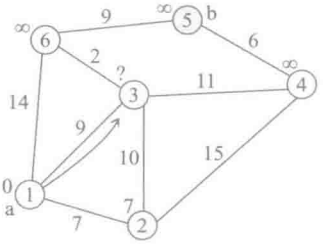


图 10-10　计算路径 1 → 3 的长度

4）计算路径 1 → 6 的长度，如图 10-12 所示。算得路径 1 → 6 的长度是 14，如图 10-13 所示。

5）关于顶点 1 的计算已经结束，所以标记为 Out，如图 10-14 所示。根据之前的计算结果比较出最小路径应当是路径 1 → 2。

⊖ 部分内容参考 http://baike.baidu.com/link?url=PgYJqT_KnlLaAhrgxuV79cPDM00q8fQhTlM9eVAkUCBAlg7q8nLG13ZWUbrKZYicgg33Ft7tgnBDmcNjJl4lPSZb0tqCTVlfmydFbg_pamaN1URcO_dpMOEMaBmVvsGVlhkzctycwWYH0Kx4hj92gO08M8wr7kZNrdwaH18J69K。

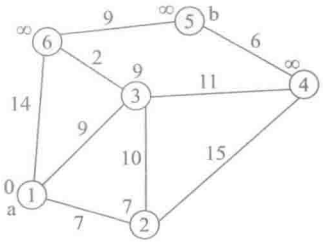


图 10-11 路径 1 → 3 的长度是 9

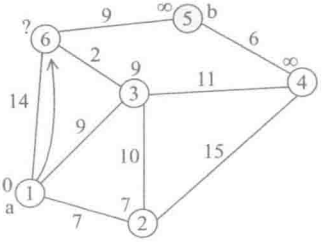


图 10-12 计算路径 1 → 6 的长度

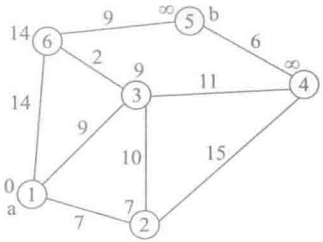


图 10-13 路径 1 → 6 的长度是 14

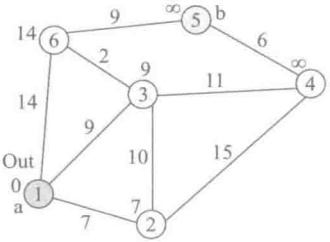


图 10-14 标记顶点 1 为 out

6）计算路径 1 → 2 → 3 的长度，如图 10-15 所示。算得路径 1 → 2 → 3 的长度是 10 + 7，此时从顶点 1 到顶点 3 有两条路径，分别是 1 → 3 和 1 → 2 → 3。从这两条路径中选择最短路径，就需要比较它们的距离，即 9 < 10 + 7，如图 10-16 所示。

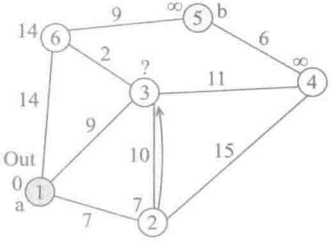


图 10-15 计算路径 1 → 2 → 3 的长度

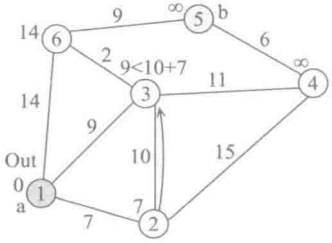


图 10-16 比较 1 → 3 和 1 → 2 → 3 的距离

7）上一步算得顶点 1 到顶点 3 的最短路径是 1 → 3，最短长度是 9。计算路径 1 → 2 → 4 的长度，如图 10-17 所示。算得路径 1 → 2 → 4 的长度是 15 + 7 = 22，如图 10-18 所示。

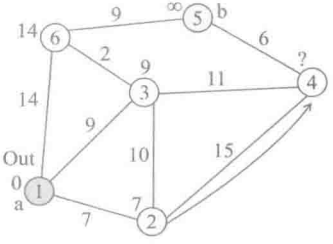


图 10-17 计算路径 1 → 2 → 4 的长度

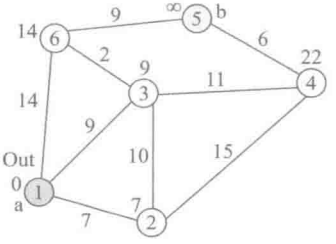


图 10-18 1 → 2 → 4 的长度是 15 + 7 = 22

8）关于顶点 2 的计算已经结束，所以标记为 Out，如图 10-19 所示。由于 1 → 2 → 4 的距离是 22，计算路径 1 → 3 → 4 的长度，需要先计算路径 3 → 4 的长度，如图 10-20 所示。

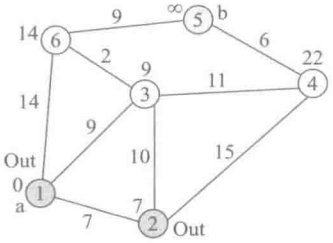


图 10-19　标记顶点 2 为 out

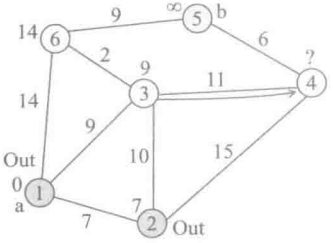


图 10-20　计算路径 3 → 4 的长度

9）计算得到路径 1 → 3 → 4 的长度是 11 + 9 = 20，小于路径 1 → 2 → 4 的长度，所以顶点 1 到顶点 4 的最小路径是 1 → 3 → 4，如图 10-21 所示。

10）计算路径 1 → 3 → 6 的长度，如图 10-22 所示。算得路径 1 → 3 → 6 的长度是 9 + 2，此时从顶点 1 到顶点 6 有两条路径，分别是 1 → 6 和 1 → 3 → 6。从这两条路径中选择最短路径，就需要比较它们的距离，即 14 > 9 + 2，如图 10-23 所示。算得顶点 1 到顶点 6 的最短路径是 1 → 3 → 6，最小长度是 11，如图 10-24 所示。

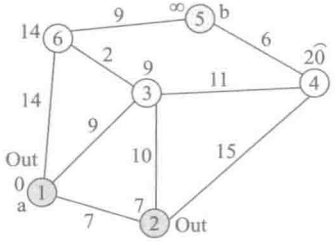


图 10-21　比较 1 → 3 → 4 和 1 → 2 → 4 的距离

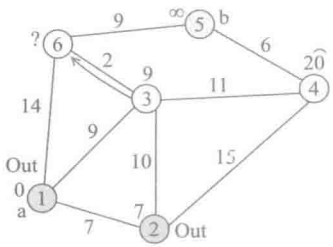


图 10-22　计算路径 1 → 3 → 6 的长度

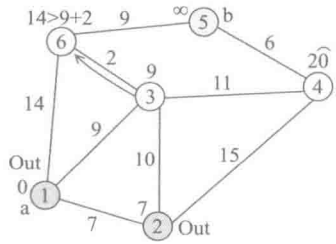


图 10-23　比较 1 → 6 和 1 → 3 → 6 的距离

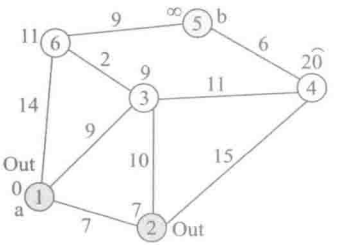


图 10-24　1 → 3 → 6 的长度是 11

11）关于顶点 3 的计算已经结束，所以标记为 Out，如图 10-25 所示。计算路径 1 → 6 → 5 的长度，需要先计算路径 6 → 5 的长度，如图 10-26 所示。

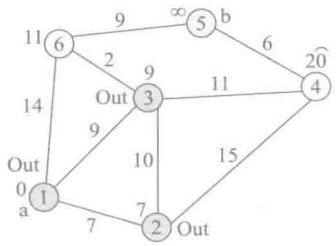


图 10-25 标记顶点 3 为 out

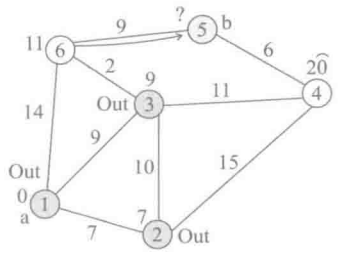


图 10-26 计算路径 6→5 的长度

12）计算得到路径 1→3→6→5 的长度是 11+ 9 = 20，如图 10-27 所示。关于顶点 6 的计算已经完成，所以也标记为 Out。由于路径 1→3→6→5 的长度等于路径 1→3→4 的长度，所以不用再计算，如图 10-28 所示。

最终计算的结果如图 10-29 所示。

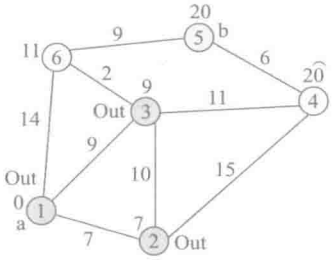


图 10-27 路径 1→3→6→5 的长度是 20

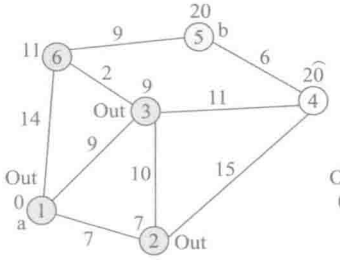


图 10-28 标记顶点 6 为 out

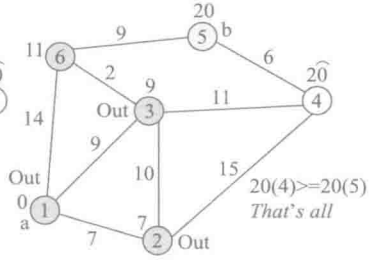


图 10-29 最终计算的结果

10.3.2 Dijkstra 的实现

我们现在使用 Pregel API 来实现 Dijkstra 算法，见代码清单 10-19。

代码清单10-19 Dijkstra算法实现

```
import org.apache.spark.graphx._
// 导入随机graph生成库
import org.apache.spark.graphx.util.GraphGenerators
// 一个边的属性包含距离的graph
val graph: Graph[Int, Double] =
    GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.
        toDouble)
val sourceId: VertexId = 42 // 最终的source
// 初始化graph，除了source的距离是无限的外，其它顶点都是0
val initialGraph = graph.mapVertices((id, _) => if (id == sourceId) 0.0 else Double.
   PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
    (id, dist, newDist) => math.min(dist, newDist), // 计算顶点距离
    triplet => {  // 发送消息，选择最短路径
        if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
            Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
```

```
        } else {
            Iterator.empty
        }
    },
    (a,b) => math.min(a,b) // 合并消息
    )
println(sssp.vertices.collect.mkString("\n"))
```

10.4 Graph 的构建

GraphX 提供了多种从 RDD 或者磁盘上的顶点和边的集合构建 Graph 的方法。默认情况下，这些构建方式没有一个对 Graph 的边重新分区，所有的边都会在默认的分区里。Graph.groupEdges 需要 Graph 重新分区，因为它假设相同的边将会位于同一个分区，所以你必须在调用 groupEdges 之前调用 Graph.partitionBy。

10.4.1 从边的列表加载 Graph

GraphLoader.edgeListFile 提供了一种从磁盘上的边的列表加载 Graph 的方法，见代码清单 10-20。

代码清单10-20 GraphLoader的edgeListFile方法定义

```
object GraphLoader {
    def edgeListFile(
        sc: SparkContext,
        path: String,
        canonicalOrientation: Boolean = false,
        minEdgePartitions: Int = 1)
      : Graph[Int, Int]
}
```

它会解析下面文件内容中邻接的源顶点的 vertexId 和目的顶点的 vertexId，并且忽略 # 号开头的行。

```
# This is a comment
2 1
4 1
1 2
```

它从指定的边创建了一个 Graph，并且自动创建边中提到的顶点。所有顶点和边的属性默认为 1。参数 canonicalOrientation 允许在正向重新调整边。参数 minEdgePartitions 指定生成的边的分区的最小数量。

10.4.2 在 Graph 中创建图的方法

Graph 中创建图的方法见代码清单 10-21。

代码清单10-21 Graph中创建图的方法

```
object Graph {
    def apply[VD, ED](
        vertices: RDD[(VertexId, VD)],
        edges: RDD[Edge[ED]],
        defaultVertexAttr: VD = null)
      : Graph[VD, ED]
    def fromEdges[VD, ED](
        edges: RDD[Edge[ED]],
        defaultValue: VD): Graph[VD, ED]
    def fromEdgeTuples[VD](
        rawEdges: RDD[(VertexId, VertexId)],
        defaultValue: VD,
        uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]
}
```

这里对 Graph 中创建图的方法简要介绍如下：

- Graph.apply 允许从边和顶点的 RDD 集合创建一个 Graph。
- Graph.fromEdges 允许仅从边的 RDD 集合创建一个 Graph。它会自动创建边中提到的顶点，并且赋予默认值。
- Graph.fromEdgeTuples 允许仅从边的元组创建一个 Graph。它会给边分配默认值 1，并且自动创建边中提到的顶点，并且赋予默认值。

10.5 顶点集合抽象 VertexRDD

VertexRDD[VD] 继承自 RDD[(VertexID, VD)]，并且添加了 VertexID 的唯一性约束。此外，VertexRDD[VD] 代表一组顶点属性为 VD 的集合。VertexRDD[VD] 内部实际将顶点属性都存入一个可以重复使用的 hashmap。如果两个 VertexRDD 都源自相同的基础 VertexRDD，可以将它们在固定的时间之后合并。VertexRDD 中定义的接口见代码清单 10-22。

代码清单10-22 VertexRDD中定义的接口

```
class VertexRDD[VD] extends RDD[(VertexID, VD)] {
    def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]
    def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]
    def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]
    def minus(other: RDD[(VertexId, VD)])
    def diff(other: VertexRDD[VD]): VertexRDD[VD]
    def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD,
      Option[VD2]) => VD3): VertexRDD[VD3]
    def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2):
      VertexRDD[VD2]
    def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)], reduceFunc: (VD2,
      VD2) => VD2): VertexRDD[VD2]
}
```

对这些接口做些简要说明：

- filter：滤出满足条件的 VertexRDD，并且保留之前的索引结构。
- mapValues：对 VertexRDD 进行转换，但保留之前的索引结构。
- minus：按照 VertexId 过滤出唯一的 VertexRDD 的集合。
- diff：求 VertexRDD 的顶点集合与其他顶点集合的差集。
- leftJoin：类似于数据库的左连接，实质上利用内部索引加速连接的连接操作。
- innerJoin：类似于数据库的内连接，实质上利用内部索引加速连接的连接操作。
- aggregateUsingIndex：使用 RDD 的内部索引来加速 reduceByKey 操作。

Spark 官网给出了使用 aggregateUsingIndex 的例子，见代码清单 10-23。setA 是 VertexId 从 1 到 100 的顶点构成的 VertexRDD；rddB 是 VertexId 从 1 到 100，每个 VertexId 都会有 2 个对偶的 RDD 数组；rddB.count 此时等于 200。使用 aggregateUsingIndex 将 setA 与 rddB 按照 VertexId 进行聚合后生成的 setB 的大小 setB.count 等于 100。

代码清单10-23 aggregateUsingIndex使用示例

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id => (id, 1)))
val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id =>
    List((id, 1.0), (id, 2.0)))
rddB.count
val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)
setB.count
val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

10.6 边集合抽象 EdgeRDD

EdgeRDD[ED] 继承自 RDD[Edge[ED]]，使用 PartitionStrategy 定义的多种分区策略中的一种来组织边在块中的分区。每个分区，边的属性和相邻的结构都单独存储，这样当改变属性值时能最大程度的复用。

EdgeRDD 提供了代码清单 10-24 中所示的三个函数。

代码清单10-24 EdgeRDD中的三个函数定义

```
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
def reverse: EdgeRDD[ED]
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) =>
    ED3): EdgeRDD[ED3]
```

简要介绍下这三个函数：

- mapValues：在保留结构的同时转换边的属性；
- reverse：在重用属性和结构的前提下，反转所有边；
- innerJoin：采用相同的分区策略连接两个 EdgeRDD 的分区。

10.7 图分割

一些高层次的理解有助于我们对可扩展性算法的设计和 API 的优化使用。当一个 Graph 很大或者需要分布式运行时，需要对 Graph 进行划分。常用的划分方法有边分割与顶点分割，如图 10-30 所示。

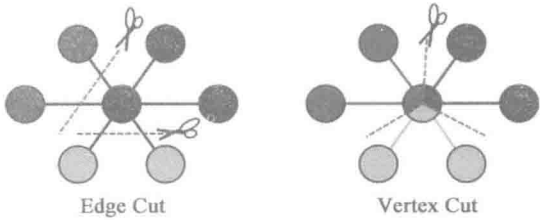


图 10-30 边分割与顶点分割示意图

GraphX 采用了顶点切分的方法来分布图的划分，而不是使用边的切分。GraphX 按照图中顶点的方式划分，能同时减少交互与存储的开销。逻辑上，将边划分到不同机器，并允许顶点跨越多台机器。边的具体分割方法取决于 PartitionStrategy。通过使用 Graph.partitionBy 方法，用户可以选择不同的划分策略对 Graph 重新划分。默认的分区策略是当 Graph 构造时提供的边的初始划分。用户可以方便地切换到 2D 分区，如图 10-31 所示。

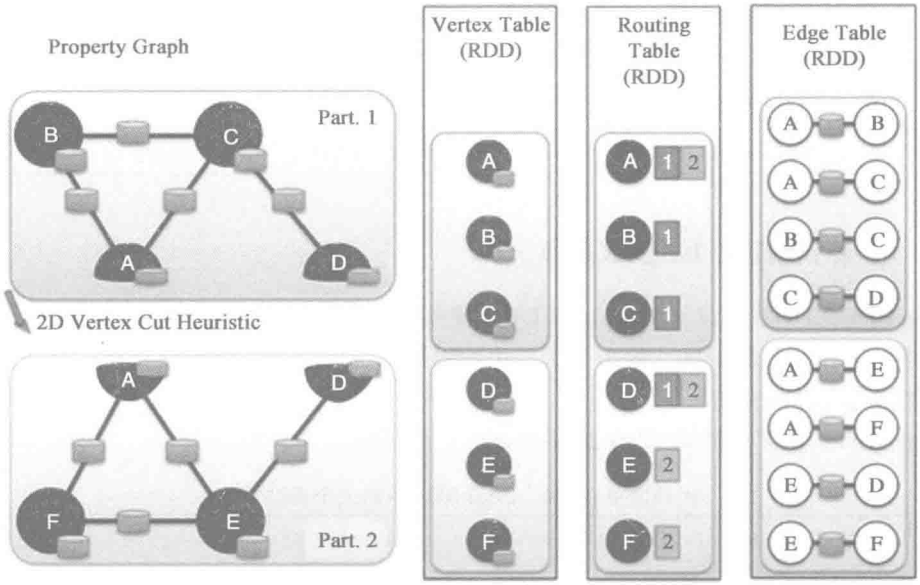


图 10-31 2D 分区

分区策略

PartitionStrategy 中已经定义了一些分区策略，我们逐个来看：

（1）EdgePartition1D

EdgePartition1D 仅使用源顶点的 VertexId 分配边到分区，相同源顶点的边会被划分到一个分区。其算法实现是将源顶点的 VertexId 乘以 mixingPrime 这个很大的数，然后除以分区

数，求余数，见代码清单 10-25。

代码清单10-25 PartitionStrategy.scala中EdgePartition1D的实现

```
case object EdgePartition1D extends PartitionStrategy {
    override def getPartition(src: VertexId, dst: VertexId, numParts:
        PartitionID): PartitionID = {
        val mixingPrime: VertexId = 1125899906842597L
        (math.abs(src * mixingPrime) % numParts).toInt
    }
}
```

（2）EdgePartition2D

EdgePartition2D 是一种二维分区策略，见代码清单 10-26。其算法的步骤如下：

1）求取小于分区数 numParts 的平方根的最小整数 ceilSqrtNumParts；

2）计算列号 col：计算源顶点 VertexId 和 mixingPrime 的乘积除以 ceilSqrtNumParts 的余数；

3）计算行号 row：计算目标顶点 VertexId 和 mixingPrime 的乘积除以 ceilSqrtNumParts 的余数；

4）分区：由表达式 (col * ceilSqrtNumParts + row) % numParts 决定。

代码清单10-26 PartitionStrategy.scala中EdgePartition2D的实现

```
case object EdgePartition2D extends PartitionStrategy {
    override def getPartition(src: VertexId, dst: VertexId, numParts:
        PartitionID): PartitionID = {
        val ceilSqrtNumParts: PartitionID = math.ceil(math.sqrt(numParts)).toInt
        val mixingPrime: VertexId = 1125899906842597L
        val col: PartitionID = (math.abs(src * mixingPrime) % ceilSqrtNumParts).toInt
        val row: PartitionID = (math.abs(dst * mixingPrime) % ceilSqrtNumParts).toInt
        (col * ceilSqrtNumParts + row) % numParts
    }
}
```

（3）RandomVertexCut

RandomVertexCut 通过对源顶点和目标顶点的 VertexId 求取哈希值，然后除以分区数 numParts 取余数，见代码清单 10-27。

代码清单10-27 PartitionStrategy.scala中RandomVertexCut的实现

```
case object RandomVertexCut extends PartitionStrategy {
    override def getPartition(src: VertexId, dst: VertexId, numParts:
        PartitionID): PartitionID = {
        math.abs((src, dst).hashCode()) % numParts
    }
}
```

（4）CanonicalRandomVertexCut

CanonicalRandomVertexCut 是 RandomVertexCut 的升级版，需要判断源顶点和目标顶点的

大小，以不同的对偶顺序求取哈希值，最后除以分区数 numParts 取余数，见代码清单 10-28。

代码清单10-28 PartitionStrategy.scala中CanonicalRandomVertexCut的实现

```
case object CanonicalRandomVertexCut extends PartitionStrategy {
    override def getPartition(src: VertexId, dst: VertexId, numParts:
        PartitionID): PartitionID = {
        if (src < dst) {
            math.abs((src, dst).hashCode()) % numParts
        } else {
            math.abs((dst, src).hashCode()) % numParts
        }
    }
}
```

10.8 常用算法

为了简化分析任务，GraphX 包含了 Graph 算法的集合。这些算法主要都集成在 org.apache.spark.graphx.lib 包中，并且可以通过在 Graph 中隐式调用 GraphOps 的方法来直接访问。在 10.3.1 和 10.3.2 这两节已经详细介绍过 Shortest Path（最短路径）的 Dijkstra 算法，本节还将介绍其他图算法。

10.8.1 网页排名

Google 是如今全球最成功的互联网搜索引擎，但在 Google 出现之前，也曾出现过很多通用或专业领域的搜索引擎。Google 最终能击败所有竞争对手，有一部分原因是因为它解决了搜素引擎的最大难题：对搜索结果按重要性排序。解决这个问题的算法就是 PageRank。PageRank 算法计算每一个网页的 PageRank 值，然后根据这个值的大小对网页的重要性进行排序。它的思想是模拟一个悠闲的上网者，上网者首先随机选择一个网页打开，然后在这个网页上待了几分钟后，跳转到该网页所指向的链接，这样无所事事、漫无目的地在网页上跳来跳去，PageRank 就是估计这个悠闲的上网者分布在各个网页上的概率。

1. PageRank 算法

要理解 PageRank 算法[⊖]，需要有基本的高等代数基础。为了降低理解门槛，先从简单的 PageRank 问题入手。假设当前网页中共有 A、B、C、D 四个网页，每个网页都可以作为一个顶点，如图 10-32 所示。如果网页 A 有链接可以跳转到 B，那么存在一条有向边 A → B。

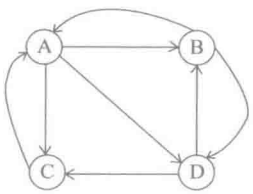


图 10-32 A、B、C、D 四个网页的 PageRank

定义 1：如果一个顶点有 k 条出边，那么跳转到任意一个出边的目的顶点的概率为 $1/k$。

⊖ 部分内容参考自 http://blog.jobbole.com/71431/。

根据定义 1，从图 10-32 可以看出顶点 A 的出度是 3，分别有 3 条有向边到达 B、C、D，所以从顶点 A 跳转到 B、C、D 的概率都是 1/3。同理，顶点 B 跳转到 A、D 的概率都是 1/2，顶点 C 跳转到 A 的概率是 1，顶点 D 跳转到 B、C 的概率都是 1/2。

定义 2：*如果网页数目是 N，它们之间的跳转概率可以用 $N*N$ 的二维矩阵 M 表示。其中 $M[i][j]$ 表示从第 j 个顶点到第 i 个顶点的跳转概率。*

根据定义 2，图 10-32 可以表示的二维矩阵 M 如下：

$$M=\begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

定义 3：*如果网页数目是 N，则跳转至其中任一网页的概率是 $1/N$。*

根据定义 3，则跳转至图 10-32 中任一网页的概率是 1/4，可以用如下的 4×1 的矩阵表示：

$$V=\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

现在计算跳转到各个页面的概率，用矩阵 M 乘以矩阵 V_0 可以得到新的 4×1 矩阵：

$$V_1=MV_0=\begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}=\begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

得到 V_1 后，接着迭代运算 $V_2=MV_1$、$V_3=MV_2$，最终会收敛为 $V=[3/9, 2/9, 2/9, 2/9]$：

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}\begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}\begin{bmatrix} 15/48 \\ 11/48 \\ 11/48 \\ 11/48 \end{bmatrix}\begin{bmatrix} 11/32 \\ 7/32 \\ 7/32 \\ 7/32 \end{bmatrix}\cdots\begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

（1）终止点问题

上述例子中的顶点间都是强连通的，即从任意顶点可以到达其他顶点。真实情况下的很多网页是不满足强连通的，当用户访问一个没有任何外链的网页时，他可能放弃继续浏览，也可能重新输入一个网址访问。我们将图 10-32 中 C 到 A 的有向边去掉，那么整个图就变为不是强连通的，如图 10-33 所示。

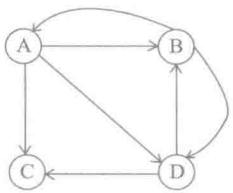


图 10-33　A、B、C、D 四个网页的终止点问题

图 10-33 的二维矩阵如下：

$$M=\begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

按照之前的迭代过程，最终所有元素都会是 0：

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}\begin{bmatrix}3/24\\5/24\\5/24\\5/24\end{bmatrix}\begin{bmatrix}5/48\\7/48\\7/48\\7/48\end{bmatrix}\begin{bmatrix}21/288\\31/288\\31/288\\31/288\end{bmatrix}\cdots\begin{bmatrix}0\\0\\0\\0\end{bmatrix}$$

（2）陷阱问题

有些情况下，某些网页不存在指向其他网页的链接，但却存在指向自己的链接。给图 10-33 中的顶点 C 增加指向自己的有向边，如图 10-34 所示。

图 10-34 的二维矩阵如下：

$$\begin{bmatrix}0 & 1/2 & 0 & 0\\1/3 & 0 & 0 & 1/2\\1/3 & 0 & 1 & 1/2\\1/3 & 1/2 & 0 & 0\end{bmatrix}$$

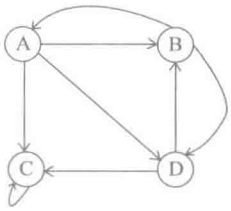


图 10-34 A、B、C、D 四个网页的陷阱问题

按照之前的迭代过程，会变得很不均衡：

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}\begin{bmatrix}3/24\\5/24\\11/24\\5/24\end{bmatrix}\begin{bmatrix}5/48\\7/48\\29/48\\7/48\end{bmatrix}\begin{bmatrix}21/288\\31/288\\205/288\\31/288\end{bmatrix}\cdots\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$$

为了克服以上出现的这些影响结果不正确的问题，需要考虑这样一种情况：每个用户无论在任何页面，随时都有可能以一个较小概率跳转至其他页面。如果用户访问 A、B、C、D 页面的概率为 $1-\beta$，那么跳出这些页面访问其他页面的平均概率是 $\beta/4$，N 个顶点跳出的概率分别为 β/N。假设 e 代表 N 各顶点的单位访问概率：

$$e=\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$$

那么所有页面跳出的总概率是 $e\frac{\beta}{N}$，最终得出如下公式：

$$V'=(1-\beta)MV+e\frac{\beta}{N}$$

假设 β 等于 0.2，则图 10-33 的二维矩阵使用公式计算得到的结果如下：

$$V'=\begin{bmatrix}0 & 2/5 & 0 & 0\\4/15 & 0 & 0 & 2/5\\4/15 & 0 & 0 & 2/5\\4/15 & 2/5 & 0 & 0\end{bmatrix}V+\begin{bmatrix}1/20\\1/20\\1/20\\1/20\end{bmatrix}$$

β 等于 0.2，则图 10-34 的二维矩阵使用公式计算得到的结果如下：

$$V'=\begin{bmatrix}0 & 2/5 & 0 & 0\\4/15 & 0 & 0 & 2/5\\4/15 & 0 & 4/15 & 2/5\\4/15 & 2/5 & 0 & 0\end{bmatrix}V+\begin{bmatrix}1/20\\1/20\\1/20\\1/20\end{bmatrix}$$

按照最终的这两个公式持续迭代会发现终止点问题和陷阱问题都不存在了。

2. PageRank 的实现

PageRank 用来测量 Graph 中每个顶点的重要性。假设一条边从 u 到 v 代表 u 认可 v 的重要性。例如，一个 Twitter 的用户被很多其他人关注，这个用户的排位将会很高。

GraphX 自带的 PageRank 中已经实现了静态或者动态 PageRank 的方法。静态 PageRank 方法运行固定的迭代次数，而动态 PageRank 将一直运行直到排名不再变化。Graph.pageRank 方法实际隐式调用了 GraphOps 的 pageRank 方法，见代码清单 10-29。

代码清单10-29 GraphOps的pageRank方法

```
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double] = {
    PageRank.runUntilConvergence(graph, tol, resetProb)
}
```

GraphOps 的 pageRank 方法直接调用了 PageRank 的 runUntilConvergence 方法，见代码清单 10-30。

代码清单10-30 PageRank的runUntilConvergence方法

```
def runUntilConvergence[VD: ClassTag, ED: ClassTag](
    graph: Graph[VD, ED], tol: Double, resetProb: Double = 0.15): Graph[Double, Double] =
{
    // 使用顶点和边构造PageRank的Graph，其中每个边都有1/出度的权重，每个顶点的属性都是1
    val pagerankGraph: Graph[(Double, Double), Double] = graph
        // 将度与每个顶点关联
        .outerJoinVertices(graph.outDegrees) {
        (vid, vdata, deg) => deg.getOrElse(0)
        }
        // 设置基于度的边的权重
        .mapTriplets( e => 1.0 / e.srcAttr )
        // 设置顶点的属性等于0.0
        .mapVertices( (id, attr) => (0.0, 0.0) )
        .cache()
       // 下面定义三个函数在GraphX中实现PageRank
    def vertexProgram(id: VertexId, attr: (Double, Double), msgSum: Double):
       (Double, Double) = {
        val (oldPR, lastDelta) = attr
        val newPR = oldPR + (1.0 - resetProb) * msgSum
        (newPR, newPR - oldPR)
    }
    def sendMessage(edge: EdgeTriplet[(Double, Double), Double]) = {
        if (edge.srcAttr._2 > tol) {
            Iterator((edge.dstId, edge.srcAttr._2 * edge.attr))
        } else {
            Iterator.empty
        }
    }
    def messageCombiner(a: Double, b: Double): Double = a + b
    // PageRank中所有顶点收到的初始化消息
    val initialMessage = resetProb / (1.0 - resetProb)
```

```
    // 执行动态的Pregel版本
    Pregel(pagerankGraph, initialMessage, activeDirection = EdgeDirection.Out)(
        vertexProgram, sendMessage, messageCombiner)
        .mapVertices((vid, attr) => attr._1)
}
```

GraphX 还提供了一个运行 PageRank 在社会网络数据集上的例子。文件 graphx/data/users.txt 提供了用户的集合，文件 graphx/data/followers.txt 提供了这些用户间的关系的集合，我们计算这些用户的 PageRank，见代码清单 10-31。

代码清单10-31 PageRank例子

```
//将边加载为graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// 允许PageRank
val ranks = graph.pageRank(0.0001).vertices
// 使用用户名连接排名
val users = sc.textFile("graphx/data/users.txt").map { line =>
    val fields = line.split(",")
    (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
    case (id, (username, rank)) => (username, rank)
}
// 打印结果
println(ranksByUsername.collect().mkString("\n"))
```

10.8.2 Connected Components 的应用

图论中的 Connected Components 是指无向图的子图中，任意两个顶点之间有边相连，并且不会与超图中的任意顶点相连。图 10-35 展示了一个 Connected Components 的例子。

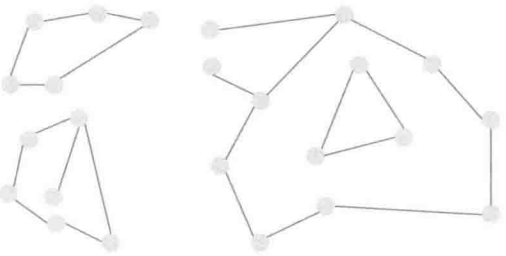


图 10-35 Connected Components 示意图

实现 Connected Components 通常使用广度优先搜索和深度优先搜寻两种算法。我们以广度优先搜索算法为例讲解。

1. 广度优先搜索算法

breadth-first search（广度优先搜索算法）是从某一顶点出发，优先拜访相邻的顶点。再访问刚才拜访的顶点的还未访问过的相邻顶点，直到所有顶点都被访问过。在访问过程中需要队列记录顶点访问的顺序。

1）假设我们有 a、b、d、e、f、g 这 6 个顶点组成的 Graph，如图 10-36 所示。假设我们记录顶点访问顺序的队列是 queue。

2）我们从 a 点开始访问，需要将 a 放入 queue，此时 queue：a，如图 10-37 所示。

3）从 queue 中取出 a，a 有 2 个相邻顶点 d、g 需要访问。此时 queue：空。

4）我们先访问 g，需要将 g 放入 queue，此时 queue：g，如图 10-38 所示。

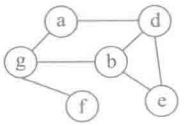


图 10-36 6 个顶点组成的 Graph

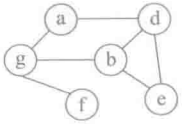


图 10-37 将 a 放入 queue

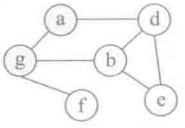


图 10-38 将 g 放入 queue

5）我们再访问 d，需要将 d 放入 queue，此时 queue：g、d，如图 10-39 所示。

6）由于已经访问过 a 的所有相邻顶点，现在从 queue 中取出下一元素 g，g 有 2 个相邻顶点 b、f 需要访问。此时 queue：d。

7）我们先访问 f，需要将 f 放入 queue，此时 queue：d、f，如图 10-40 所示。

8）我们再访问 b，需要将 b 放入 queue，此时 queue：d、f、b，如图 10-41 所示。

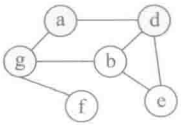


图 10-39 将 d 放入 queue

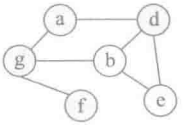


图 10-40 将 f 放入 queue

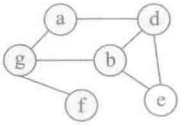


图 10-41 将 b 放入 queue

9）现在从 queue 中取出下一元素 d，d 有 2 个相邻顶点 b、e 需要访问，此时 queue：f、b。

10）由于 b 已经访问过，所以只需要访问 e，将 e 放入 queue，此时 queue：f、b、e，如图 10-42 所示。

11）之后陆续取出 f、b、e，由于这些节点没有相邻顶点，不再有进入 queue 的元素，访问结束。

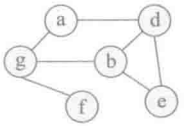


图 10-42 将 e 放入 queue

2. ConnectedComponents 的实现

GraphX 中已经实现了 connectedComponents 方法，Graph 实际隐式调用了 GraphOps 的 connectedComponents 方法，见代码清单 10-32。

代码清单10-32 GraphOps的connectedComponents方法

```
def connectedComponents(): Graph[VertexId, ED] = {
    ConnectedComponents.run(graph)
}
```

connectedComponents 方法中调用了 ConnectedComponents，其实现见代码清单 10-33。

代码清单10-33 ConnectedComponents的run方法

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED]): Graph[VertexId, ED] = {
    val ccGraph = graph.mapVertices { case (vid, _) => vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]) = {
        if (edge.srcAttr < edge.dstAttr) {
```

```
            Iterator((edge.dstId, edge.srcAttr))
        } else if (edge.srcAttr > edge.dstAttr) {
            Iterator((edge.srcId, edge.dstAttr))
        } else {
            Iterator.empty
        }
    }
    val initialMessage = Long.MaxValue
    Pregel(ccGraph, initialMessage, activeDirection = EdgeDirection.Either)(
        vprog = (id, attr, msg) => math.min(attr, msg),
        sendMsg = sendMessage,
        mergeMsg = (a, b) => math.min(a, b))
} // end of connectedComponents
```

GraphX 提供了一个运行 connectedComponents 在社会网络数据集上的例子。采用的样例数据还是采用 PageRank 例子中用到的，见代码清单 10-34。

代码清单10-34 connectedComponents例子

```
// 加载PageRank例子需要的数据生成Graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// 查找connected components
val cc = graph.connectedComponents().vertices
// 使用username连接connected components
val users = sc.textFile("graphx/data/users.txt").map { line =>
    val fields = line.split(",")
    (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
    case (id, (username, cc)) => (username, cc)
}
println(ccByUsername.collect().mkString("\n"))
```

10.8.3 三角关系统计

相信很多人都曾遇到数一个复杂图形中的三角形个数的问题，三角关系统计（TriangleCount）的原理与之类似，也需要在一个 Graph 中算出三角形数目。此功能经常用于社区，对关注关系的统计能够有效证明社区的稳定性。社区中的三角关系出现越多，则说明社区越稳定。

GraphX 中已经实现了 triangleCount 方法，Graph 实际隐式调用了 GraphOps 的 triangleCount 方法（Scala 的特性），见代码清单 10-35。

代码清单10-35 GraphOps的triangleCount方法

```
def triangleCount(): Graph[Int, ED] = {
    TriangleCount.run(graph)
}
```

其中调用了 TriangleCount 的 run 方法，其实现见代码清单 10-36。

代码清单10-36 TriangleCount的run方法

```
def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD,ED]): Graph[Int, ED] = {
    // 删除冗余的边
    val g = graph.groupEdges((a, b) => a).cache()
    // 构建社区关系
    val nbrSets: VertexRDD[VertexSet] =
        g.collectNeighborIds(EdgeDirection.Either).mapValues { (vid, nbrs) =>
            val set = new VertexSet(4)
            var i = 0
            while (i < nbrs.size) {
                // 防止自循环
                if(nbrs(i) != vid) {
                    set.add(nbrs(i))
                }
                i += 1
            }
            set
        }
    // 使用Graph连接顶点集合
    val setGraph: Graph[VertexSet, ED] = g.outerJoinVertices(nbrSets) {
        (vid, _, optSet) => optSet.getOrElse(null)
    }
    // 此函数用于计算小顶点与大顶点之间的交集
    def edgeFunc(ctx: EdgeContext[VertexSet, ED, Int]) {
        assert(ctx.srcAttr != null)
        assert(ctx.dstAttr != null)
        val (smallSet, largeSet) = if (ctx.srcAttr.size < ctx.dstAttr.size) {
            (ctx.srcAttr, ctx.dstAttr)
        } else {
            (ctx.dstAttr, ctx.srcAttr)
        }
        val iter = smallSet.iterator
        var counter: Int = 0
        while (iter.hasNext) {
            val vid = iter.next()
            if (vid != ctx.srcId && vid != ctx.dstId && largeSet.contains(vid)) {
                counter += 1
            }
        }
        ctx.sendToSrc(counter)
        ctx.sendToDst(counter)
    }
    // 计算边的交集
    val counters: VertexRDD[Int] = setGraph.aggregateMessages(edgeFunc, _ + _)
    // 由于每个三角形会被计算两次，所以合并计数后除以2
    g.outerJoinVertices(counters) {
        (vid, _, optCounter: Option[Int]) =>
            val dblCount = optCounter.getOrElse(0)
            assert((dblCount & 1) == 0)
            dblCount / 2
    }
}
```

10.9 应用举例

Spark 自带的例子 LiveJournalPageRank 演示了 PageRank。根据其代码注释知道，需要去网址：http://snap.stanford.edu/data/soc-LiveJournal1.html 下载构建图需要的数据集。

LiveJournalPageRank 需要以下参数：

- 数据集文件：下载后的文件 soc-LiveJournal1.txt.gz 解压后的文件路径为 D:\soc-LiveJournal1.txt；
- 输出文件：--output=<output_file> 选项指定；
- 分区数：通过 --numEPart=<num_edge_partitions> 选项指定；
- 分区策略：--partStrategy 选项指定，可以选择 RandomVertexCut、EdgePartition1D、EdgePartition2D 和 CanonicalRandomVertexCut 中的任意一个。

LiveJournalPageRank 的实现，见代码清单 10-37。

代码清单10-37 LiveJournalPageRank的实现

```
object LiveJournalPageRank {
    def main(args: Array[String]) {
        if (args.length < 1) {
           System.exit(-1)
        }
        Analytics.main(args.patch(0, List("pagerank"), 0))
    }
```

其中实际调用了 Analytics 的 main 函数，其中根据 taskType 分别执行 PageRank、Connected Components、Triangle Count 的例子。我们只列出其中 PageRank 相关的代码，见代码清单 10-38。

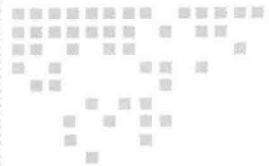
代码清单10-38 Analytics.scala中与PageRank相关的代码

```
case "pagerank" =>
    val tol = options.remove("tol").map(_.toFloat).getOrElse(0.001F)
    val outFname = options.remove("output").getOrElse("")
    val numIterOpt = options.remove("numIter").map(_.toInt)
    options.foreach {
        case (opt, _) => throw new IllegalArgumentException("Invalid option: " + opt)
    }
        println("======================================")
        println("|             PageRank               |")
        println("======================================")
        val sc = new SparkContext(conf.setAppName("PageRank(" + fname + ")").
           setMaster("local[2]"));
        val unpartitionedGraph = GraphLoader.edgeListFile(sc, fname,
            numEdgePartitions = numEPart,
            edgeStorageLevel = edgeStorageLevel,
            vertexStorageLevel = vertexStorageLevel).cache()
        val graph = partitionStrategy.foldLeft(unpartitionedGraph)(_.partitionBy(_))
        println("GRAPHX: Number of vertices " + graph.vertices.count)
        println("GRAPHX: Number of edges " + graph.edges.count)
        val pr = (numIterOpt match {
```

```
    case Some(numIter) => PageRank.run(graph, numIter)
    case None => PageRank.runUntilConvergence(graph, tol)
}).vertices.cache()
println("GRAPHX: Total rank: " + pr.map(_._2).reduce(_ + _))
if (!outFname.isEmpty) {
    logWarning("Saving pageranks of pages to " + outFname)
    pr.map { case (id, r) => id + "\t" + r }.saveAsTextFile(outFname)
}
sc.stop()
```

10.10 小结

GraphX 遵循 BSP 模式，因此拥有整体同步并行计算的能力。GraphX 中实现的图由顶点和边的集合组成。属性图包含属性、结构、连接、聚合等操作。创建 Graph 包括两种方式：使用 GraphLoader.edgeListFile 从磁盘文件加载和使用 Graph 对象构造。为适应分布式图计算，GraphX 还提供了图分割的能力。通过介绍 Dijkstra、PageRank、Connected Components 等图论中的算法，让读者对 Pregel API 能有更深层次的理解。



Chapter 11 第 11 章

机器学习

不愤不启，不悱不发。举一隅不以三隅反，则不复也。

——《论语 · 述而》

本章导读

机器学习（machine learning，ML）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。ML 专注于研究计算机模拟或实现人类的学习行为，以获取新知识、新技能，并重组已学习的知识结构使之不断改善自身。

MLlib 是 Spark 提供的可扩展的机器学习库。MLlib 已经集成了大量机器学习的算法，由于 MLlib 涉及的算法众多，笔者只对部分算法进行了分析，其余算法只是简单列出公式，读者如果想要对公式进行推理，需要自己寻找有关概率论、数理统计、数理分析等方面的专门著作。本章更侧重于机器学习 API 的使用，基本能够满足大多数读者的需要。

11.1 机器学习概论[⊖]

机器学习也属于人工智能的范畴，该领域主要研究的对象是人工智能，尤其是如何在经验学习中改善具体算法。机器学习是人工智能研究较为年轻的分支，它的发展过程大致可分为如下 4 个阶段：

第一阶段：20 世纪 50 年代中叶至 60 年代中叶，属于热烈时期。

第二阶段：20 世纪 60 年代中叶至 70 年代中叶，称为冷静时期。

⊖ 此节部分内容参考自 http://baike.baidu.com/link?url=p26UJX578vOjDX6TRh6R0hBX0pJIdApbOccex87A7By3-xhzvCFzp-f58ZWhGu87Qe_v1Ny3iN6RDhlLU3vdK_。

第三阶段：20 世纪 70 年代中叶至 80 年代中叶，称为复兴时期。

第四阶段：从 1986 年开始至今。

（1）机器学习的组成

机器学习的基本结构由环境、知识库和执行部分三部分组成。环境向学习部分（属于知识库的一部分）提供某些信息，学习部分利用这些信息修改知识库，以增进执行部分完成任务的效能，执行部分根据知识库完成任务，同时把获得的信息反馈给学习部分。

（2）学习策略

学习策略是指机器学习过程中所采用的推理策略。学习系统一般由学习和环境两部分组成。环境（如书本或教师）提供信息，学习部分则实现信息转换、存储，并从中获取有用的信息。学习过程中，学生（学习部分）使用的推理越少，他对教师（环境）的依赖就越大，教师的负担也就越重。根据学生实现信息转换所需推理的多少和难易程度，以从简单到复杂，从少到多的次序可以将学习策略分为以下 6 种基本类型：

- 机械学习（rote learning）：学习者不需要任何推理或转换，直接获取环境所提供的信息。属于此类的如塞缪尔的跳棋程序。
- 示教学习（learning from instruction）：学习者从环境获取信息，把知识转换成内部可使用的表示形式，并将新知识和原有知识有机地合为一体。此种学习策略需要学生有一定程度的推理能力，但环境仍要做大量的工作。典型应用是 FOO 程序。
- 演绎学习（learning by deduction）：学习者通过推理获取有用的知识。典型应用是宏操作（macro-operation）学习。
- 类比学习（learning by analogy）：学习者根据两个不同领域（源域、目标域）中的知识相似性，通过类比，从源域的知识推导出目标域的相应知识。此类应用如卢瑟福类比。
- 基于解释的学习 (explanation-based learning, EBL)：学习者根据教师提供的目标概念和此概念的例子、领域理论及可操作准则，首先给出解释来说明为什么该例子满足目标概念，然后将解释推广为目标概念的一个满足可操作准则的充分条件。著名的 EBL 系统有迪乔恩（G.DeJong）的 GENESIS 等。
- 归纳学习（learning from induction）：由环境提供某概念的一些实例或反例，让学习者通过归纳推理得出该概念的一般描述。归纳学习是最基本的，发展也较为成熟的学习方法，在人工智能领域中已得到广泛的研究和应用。

学习策略还可以从所获取知识的表示形式、应用领域等维度分类，有兴趣的读者可以自行研究，此处不再赘述。

（3）应用领域

目前，机器学习已经广泛应用于数据挖掘、计算机视觉、自然语言处理、生物特征识别、搜索引擎、医学诊断、检测信用卡欺诈、证券市场分析、DNA 序列测序、语音和手写识别、战略游戏和机器人等领域。

11.2 Spark MLlib 总体设计

MLlib（machine learning library）是 Spark 提供的可扩展的机器学习库。Mllib 中已经包含了一些通用的学习算法和工具，如：分类、回归、聚类、协同过滤、降维以及底层的优化原语等算法和工具。

MLlib 提供的 API 主要分为以下两类：

- spark.mllib 包中提供的主要 API。
- spark.ml 包中提供的构建机器学习工作流的高层次的 API。

11.3 数据类型

MLlib 支持存储在一台机器上的局部向量和矩阵以及由一个或多个 RDD 支持的分布式矩阵。局部向量和局部矩阵是提供公共接口的简单数据模型。Breeze 和 jblas 提供了底层的线性代数运算。Breeze 提供了一组线性代数和数字计算的库，具体信息访问 http://www.scalanlp.org/。jblas 提供了使用 Java 开发的线性代数库，具体信息访问 http://jblas.org/。

11.3.1 局部向量

MLlib 支持两种局部向量类型：密集向量（dense）和稀疏向量（sparse）。密集向量由 double 类型的数组支持，而稀疏向量则由两个平行数组支持。例如，向量 (1.0, 0.0, 3.0) 由密集向量表示的格式为 [1.0, 0.0, 3.0]，由稀疏向量表示的格式为（3, [0, 2], [1.0, 3.0]）。

这里对稀疏向量再做些解释。3 是向量 (1.0, 0.0, 3.0) 的长度，除去 0 值外，其他两个值的索引和值分别构成了数组 [0, 2] 和数组 [1.0, 3.0]。

有关向量的类如图 11-1 所示。

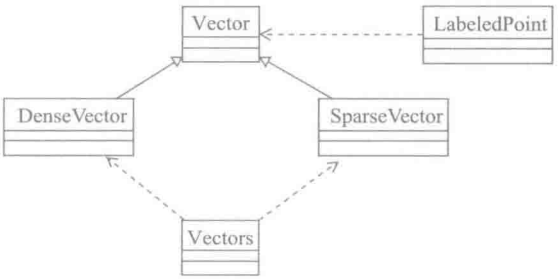


图 11-1 MLlib 中的向量类

Vector 是所有局部向量的基类，Dense-Vector 和 SparseVector 都是 Vector 的具体实现。Spark 官方推荐使用 Vectors 中实现的工厂方法创建局部向量，就像下面这样。

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
// 创建密集向量(1.0, 0.0, 3.0).
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
// 给向量(1.0, 0.0, 3.0)创建疏向量
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
// 通过指定非0的项目，创建稀疏向量 (1.0, 0.0, 3.0)
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

注意 Scala 默认会导入 scala.collection.immutable.Vector，所以必须显式导入 org.apache.spark.mllib.linalg.Vector 才能使用 Mllib 提供的 Vector。

上面例子中以数组为参数，调用 Vectors 的 sparse 接口，见代码清单 11-1。使用 Seq 创建稀疏向量，其本质依然是使用数组，见代码清单 11-2。

代码清单11-1 Vectors的sparse接口

```
def sparse(size: Int, indices: Array[Int], values: Array[Double]): Vector =
    new SparseVector(size, indices, values)
```

代码清单11-2 Vectors重载的sparse接口

```
def sparse(size: Int, elements: Seq[(Int, Double)]): Vector = {
    require(size > 0)
    val (indices, values) = elements.sortBy(_._1).unzip
    var prev = -1
    indices.foreach { i =>
        require(prev < i, s"Found duplicate indices: $i.")
        prev = i
    }
    require(prev < size)
    new SparseVector(size, indices.toArray, values.toArray)
}
```

11.3.2 标记点

标记点是将密集向量或者稀疏向量与应答标签相关联。在 MLlib 中，标记点用于监督学习算法。MLlib 使用 double 类型存储标签，所以我们能在回归和分类中使用标记点。如果只有两种分类，可以使用二分法，一个标签要么是 1.0，要么是 0.0。如果有很多分类，标签应该从零开始：0、1、2……

标记点由样例类 LabeledPoint 来表示，其使用方式如下。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
//使用标签1.0和一个密集向量创建一个标记点
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
//使用标签0.0和一个疏向量创建一个标记点
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0)))
```

用稀疏的训练数据做练习是很常见的，好在 MLlib 支持读取存储在 LIBSVM 格式中的训

练例子。LIBSVM 格式是一种每一行表示一个标签稀疏特征向量的文本格式，其格式如下：

```
label index1:value1 index2:value2 ...
```

LIBSVM 是林智仁（Lin Chih-Jen）教授等开发设计的一个简单、易用和快速有效的 SVM 模式识别与回归的软件包。具体信息请阅读 http://www.csie.ntu.edu.tw/~cjlin/libsvm/。MLlib 已经提供了 MLUtils.loadLibSVMFile 方法读取存储在 LIBSVM 格式文本文件中的训练数据，见代码清单 11-3。

代码清单11-3 读取LIBSVM文件内容

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.rdd.RDD
val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_
   libsvm_data.txt")
```

11.3.3 局部矩阵

MLlib 支持数据存储在单个 double 类型数组的密矩阵。先来看这样一个矩阵：

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

这个矩阵是如何存储的？它只是存储到一维数组 [1.0, 3.0, 5.0, 2.0, 4.0, 6.0]，这个矩阵的尺寸是 3*2，即 3 行 2 列。

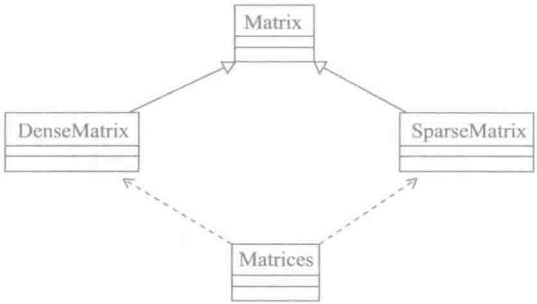


图 11-2 MLlib 中局部矩阵相关的类

有关局部矩阵的类如图 11-2 所示。

局部矩阵的基类是 Matrix，目前有一个实现类 DenseMatrix。Spark 官方推荐使用 Matrices 中实现的工厂方法创建局部矩阵，例如：

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices}
// 创建密矩阵((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

11.3.4 分布式矩阵

分布式矩阵分布式地存储在一个或者多个 RDD 中。如何存储数据量很大的分布式矩阵？最重要的在于选择一个正确的格式。如果将分布式矩阵转换为不同格式，可能需要全局的 shuffle，成本非常昂贵。

有关分布式矩阵的类如图 11-3 所示。

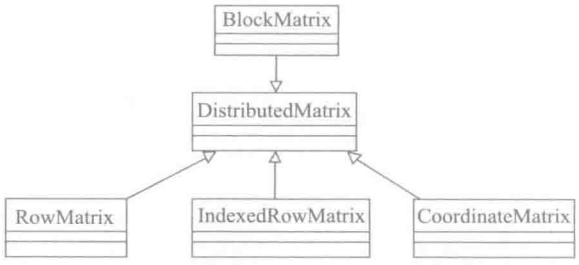


图 11-3 MLlib 中分布式矩阵相关的类

迄今为止，MLlib 已经实现了 4 种类型的分布式矩阵：

- RowMatrix：最基本的分布式矩阵类型，是面向行且行索引无意义的分布式矩阵。RowMatrix 的行实际是多个局部向量的 RDD，列受限于 integer 的范围大小。RowMatrix 适用于列数不大以便于单个局部向量可以合理地传递给 Driver，也能在单个节点上存储和操作的情况。

下面展示了可以使用 RDD[Vector] 实例来构建 RowMatrix 的例子。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val rows: RDD[Vector] = ...
val mat: RowMatrix = new RowMatrix(rows)
val m = mat.numRows()
val n = mat.numCols()
```

- IndexedRowMatrix：与 RowMatrix 类似，但却面向索引的分布式矩阵。IndexedRowMatrix 常用于识别行或者用于执行连接操作。可以使用 RDD[IndexedRow] 实例创建 IndexedRowMatrix。IndexedRow 的实现如下。

```
@Experimental
case class IndexedRow(index: Long, vector: Vector)
```

通过删除 IndexedRowMatrix 的行索引，可以将 IndexedRowMatrix 转换为 RowMatrix。下面的例子演示了如何使用 IndexedRowMatrix。

```
import org.apache.spark.mllib.linalg.distributed.{IndexedRow, IndexedRowMatrix,
    RowMatrix}
val rows: RDD[IndexedRow] = ...
val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)
val m = mat.numRows()
val n = mat.numCols()
val rowMat: RowMatrix = mat.toRowMatrix()
```

- CoordinateMatrix：使用坐标列表（COO）格式存储的分布式矩阵。支持 CoordinateMatrix 的 RDD 实际是 (i: Long, j: Long, value: Double) 这样的三元组，i 是行索引，j 是列索引，value 是实际存储的值。CoordinateMatrix 适用于行和列都很大且矩阵很稀疏的情况。

可以使用 RDD[MatrixEntry] 实例创建 CoordinateMatrix。MatrixEntry 的实现如下。

```
@Experimental
case class MatrixEntry(i: Long, j: Long, value: Double)
```

通过调用 CoordinateMatrix 的 toIndexedRowMatrix 方法，可以将 CoordinateMatrix 转换为 IndexedRowMatrix。下面的例子演示了 CoordinateMatrix 的使用。

```
import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix, MatrixEntry}
val entries: RDD[MatrixEntry] = ...
val mat: CoordinateMatrix = new CoordinateMatrix(entries)
val m = mat.numRows()
val n = mat.numCols()
```

```
val indexedRowMatrix = mat.toIndexedRowMatrix()
```

- BlockMatrix：由 RDD[MatrixBlock] 支持的分布式矩阵。MatrixBlock 实际是 ((Int, Int), Matrix) 这样的二元组，(Int, Int) 是 Block 的索引，Matrix 是记录块大小的子矩阵。BlockMatrix 支持与其他 BlockMatrix 的 add 和 multiply，还提供 validate 方法用于校验当前 BlockMatrix 是否恰当构建。

通过调用 IndexedRowMatrix 或者 CoordinateMatrix 的 toBlockMatrix 方法，可以方便转换为 BlockMatrix。toBlockMatrix 方法创建的 Block 的默认大小是 1024×1024。可以使用 toBlockMatrix(rowsPerBlock, colsPerBlock) 方法改变 Block 的大小。下面的例子演示了 BlockMatrix 的使用。

```
import org.apache.spark.mllib.linalg.distributed.{BlockMatrix, CoordinateMatrix,
    MatrixEntry}
val entries: RDD[MatrixEntry] = ...
val coordMat: CoordinateMatrix = new CoordinateMatrix(entries)
val matA: BlockMatrix = coordMat.toBlockMatrix().cache()
matA.validate()
val ata = matA.transpose.multiply(matA)
```

由于 MLlib 会缓存矩阵的大小，所以支持分布式矩阵的 RDD 必须要有明确的类型，否则会导致出错。

11.4 基础统计

MLlib 提供了很多统计方法，包括摘要统计、相关统计、分层抽样、假设检验、随机数生成等。这些都涉及统计学、概率论的专业知识，笔者只会在随机数生成中简单介绍其数学原理，其余内容请读者自行研究。

11.4.1 摘要统计

调用 Statistics 类的 colStats 方法，可以获得 RDD[Vector] 的列的摘要统计。colStats 方法返回了 MultivariateStatisticalSummary 对象，MultivariateStatisticalSummary 对象包含了列的最大值、最小值、平均值、方差、非零元素的数量以及总数。下面的例子演示了如何使用 colStats。

```
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
val observations: RDD[Vector] = ...
val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
println(summary.mean) // 每个列值组成的密集向量
println(summary.variance) // 列向量方差
println(summary.numNonzeros) // 每个列的非零值个数
```

colStats 实际使用了 RowMatrix 的 computeColumnSummaryStatistics 方法，见代码清单 11-4。

代码清单11-4　Statistics的colStats方法

```
@Experimental
def colStats(X: RDD[Vector]): MultivariateStatisticalSummary = {
    new RowMatrix(X).computeColumnSummaryStatistics()
}
```

11.4.2　相关统计

计算两个序列之间的相关性是统计中通用的操作。MLlib 提供了计算多个序列之间相关统计的灵活性。目前支持的关联方法运用了皮尔森相关系数（Pearson correlation coefficient）和斯皮尔森秩相关系数（Spearman's rank correlation coefficient）。

1. 皮尔森相关系数

皮尔森相关系数也称皮尔森积矩相关系数（Pearson product-moment correlation coefficient），是一种线性相关系数。皮尔森相关系数是用来反映两个变量线性相关程度的统计量。

$$r = \frac{1}{n-1}\sum_{i=0}^{n}\left(\frac{x_i - x}{s_x}\right)\left(\frac{y_i - y}{s_y}\right)$$

相关系数用 r 表示，其中 n 为样本量，x_i，y_i，s_x，s_y 分别为两个变量的观测值和均值。r 描述的是两个变量间线性相关强弱的程度。r 的取值在 -1 与＋ 1 之间，若 $r > 0$，表明两个变量是正相关，即一个变量的值越大，另一个变量的值也会越大；若 $r < 0$，表明两个变量是负相关，即一个变量的值越大另一个变量的值反而会越小。r 的绝对值越大表明相关性越强，要注意的是这里并不存在因果关系。若 $r = 0$，表明两个变量间不是线性相关，但有可能是其他方式的相关（比如曲线方式）。

2. 斯皮尔森秩相关系数

斯皮尔森秩相关系数也称为 Spearman 的 ρ，是由 Charles Spearman 命名的，一般用希腊字母 ρ_s（rho）或 r_s 表示。Spearman 秩相关系数是一种无参数（与分布无关）的检验方法，用于度量变量之间联系的强弱。在没有重复数据的情况下，如果一个变量是另外一个变量的严格单调函数，则 Spearman 秩相关系数就是＋ 1 或 -1，称变量完全 Spearman 秩相关。注意和 Pearson 完全相关的区别，只有当两变量存在线性关系时，Pearson 相关系数才为＋ 1 或 -1。Spearman 秩相关系数为：

$$\rho_s = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

Statistics 提供了计算序列之间相关性的方法，默认情况下使用皮尔森相关系数，使用方法如下。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics
val sc: SparkContext = ...
```

```
val seriesX: RDD[Double] = ... // a series
val seriesY: RDD[Double] = ... // 和seriesX必须有相同的分区数和基数
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
val data: RDD[Vector] = ... // 每个向量必须是行，不能是列
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

Statistics 中相关性的实现，见代码清单 11-5。

代码清单11-5 Statistics中相关性的实现

```
@Experimental
def corr(X: RDD[Vector]): Matrix = Correlations.corrMatrix(X)
@Experimental
def corr(X: RDD[Vector], method: String): Matrix = Correlations.corrMatrix(X, method)
@Experimental
def corr(x: RDD[Double], y: RDD[Double]): Double = Correlations.corr(x, y)
@Experimental
def corr(x: RDD[Double], y: RDD[Double], method: String): Double = Correlations.corr(x, y,
   method)
```

其实质是代理了 Correlations，Correlations 中相关性的实现见代码清单 11-6。

代码清单11-6 Correlations中相关性的实现

```
def corr(x: RDD[Double],
       y: RDD[Double],
       method: String = CorrelationNames.defaultCorrName): Double = {
    val correlation = getCorrelationFromName(method)
    correlation.computeCorrelation(x, y)
}
def corrMatrix(X: RDD[Vector],
        method: String = CorrelationNames.defaultCorrName): Matrix = {
    val correlation = getCorrelationFromName(method)
    correlation.computeCorrelationMatrix(X)
}
def getCorrelationFromName(method: String): Correlation = {
    try {
       CorrelationNames.nameToObjectMap(method)
    } catch {
      case nse: NoSuchElementException =>
         throw new IllegalArgumentException("Unrecognized method name. Supported
            correlations: "
         + CorrelationNames.nameToObjectMap.keys.mkString(", "))
    }
}
private[mllib] object CorrelationNames {
    // Note: after new types of correlations are implemented, please update this map.
    val nameToObjectMap = Map(("pearson", PearsonCorrelation), ("spearman",
       SpearmanCorrelation))
    val defaultCorrName: String = "pearson"
}
```

11.4.3　分层抽样

分层抽样（Stratified sampling）是先将总体按某种特征分为若干次级（层），然后再从每一层内进行独立取样，组成一个样本的统计学计算方法。为了对分层抽样有更直观的感受，请看下面的例子：

某市现有机动车共 1 万辆，其中大巴车 500 辆，小轿车 6000 辆，中巴车 1000 辆，越野车 2000 辆，工程车 500 辆。现在要了解这些车辆的使用年限，决定采用分层抽样方式抽取 100 个样本。按照车辆占比，各类车辆的抽样数量分别为 5，60，10，20，5。

摘要统计和相关统计都集成在 Statistics 中，而分层抽样只需要调用 RDD[(K, V)] 的 sampleByKey 和 sampleByKeyExact 即可。为了分层抽样，其中的键可以被认为是标签，值是具体的属性。sampleByKey 方法采用掷硬币的方式来决定是否将一个观测值作为采样，因此需要一个预期大小的样本数据。sampleByKeyExact 则需要更多更有效的资源，但是样本数据的大小是确定的。sampleByKeyExact 方法允许用户采样符合 $[f_k \cdot n_k] \forall k \in K$，其中 f_k 是键 k 的函数，n_k 是 RDD[(K, V)] 中键为 k 的 (K, V) 对，K 是键的集合。下例演示了如何使用分层抽样。

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.rdd.PairRDDFunctions
val sc: SparkContext = ...
val data = ... // an RDD[(K, V)] of any key value pairs
val fractions: Map[K, Double] = ... // specify the exact fraction desired from each key
val approxSample = data.sampleByKey(withReplacement = false, fractions)
val exactSample = data.sampleByKeyExact(withReplacement = false, fractions)
```

RDD 中本身没有实现 sampleByKey 和 sampleByKeyExact，实际是将 RDD 隐式转换为 PairRDDFunctions 后，调用 PairRDDFunctions 的 sampleByKey 和 sampleByKeyExact。

11.4.4　假设检验

假设检验（hypothesis testing）是数理统计学中根据一定假设条件由样本推断总体的一种方法。

如果对总体的某种假设是真实的，那么不利于或不能支持这一假设的事件 A（小概率事件）在一次试验中几乎不可能发生；要是在一次试验中 A 竟然发生了，就有理由怀疑该假设的真实性，拒绝这一假设。小概率原理可以用图 11-4 表示。

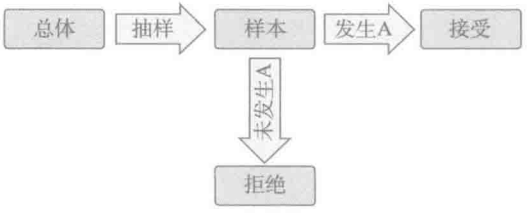


图 11-4　小概率原理

H_0 表示原假设，H_1 表示备选假设。常见的假设检验有如下几种：

- 双边检验：$H_0 : \mu = \mu_0, H_1 : \mu \neq \mu_0$
- 右侧单边检验：$H_0 : \mu \leqslant \mu_0, H_1 : \mu > \mu_0$
- 左侧单边检验：$H_0 : \mu \geqslant \mu_0, H_1 : \mu < \mu_0$

假设检验是一个强大的工具，无论结果是否是偶然的，都可以决定结果是否具有统计特征。MLlib 目前支持皮尔森卡方测试（Pearson's chi-squared test）。输入数据的类型决定了是做卡方适合度检测还是独立性检测。卡方适合度检测的输入数据类型应当是向量，而卡方独立性检测需要的数据类型是矩阵。RDD[LabeledPoint] 可以作为卡方检测的输入类型。下例演示了如何使用假设检验。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.stat.Statistics._
val sc: SparkContext = ...
val vec: Vector = ... // 事件的频率组成的vector
val goodnessOfFitTestResult = Statistics.chiSqTest(vec)
println(goodnessOfFitTestResult)
val mat: Matrix = ... // 偶然性matrix
val independenceTestResult = Statistics.chiSqTest(mat)
println(independenceTestResult)
val obs: RDD[LabeledPoint] = ... // (feature, label) pairs
val featureTestResults: Array[ChiSqTestResult] = Statistics.chiSqTest(obs)
var i = 1
featureTestResults.foreach { result =>
    println(s"Column $i:\n$result")
    i += 1
} // summary of the test
```

11.4.5 随机数生成

随机数可以看做随机变量，什么是随机变量？将一枚质地均匀的硬币抛掷 3 次，记录它的结果，有

$$\Omega = \{\text{uuu, uud, udd, udu, ddd, ddu, duu, dud}\}$$

其中 u 代表正面朝上，d 代表反面朝上，整个集合 Ω 是投掷 3 次硬币的样本空间。正面朝上的次数可能是 0,1,2,3。由于样本空间 Ω 中的结果都是随机发生的，所以出现正面的次数 X 就是随机的，X 即为随机变量。如果投掷硬币，直到出现正面的投掷次数为 Y，那么 Y 的取值可能是 0，1，2，3，…。如果随机变量的取值是有限的（比如 X）或者是可列的（比如 Y），那么就称为离散随机变量。

刚才说的投掷 3 次硬币的情况下，使用 P（X = 取值）的方式表达每种取值的概率，我们不难得出：

$$P(X=0)=\frac{1}{8}$$
$$P(X=1)=\frac{3}{8}$$
$$P(X=2)=\frac{3}{8}$$
$$P(X=3)=\frac{1}{8}$$

如果样本空间上随机变量的取值用 x_1，x_2，x_3，…表示，那么存在满足 $p(x_i) = P(X = x_i)$ 和 $\Sigma_i p(x_i) = 1$ 的函数 p。这个函数 p 称为随机变量 X 的概率质量函数（probability mass function）或者频率函数（frequency function）。

如果 X 取值累积某个范围的值，那么其累积分布函数（cumulative distribution function,cdf）定义如下：

$$F(x) = P(X \leqslant x), -\infty < x < \infty$$

累积分布函数满足：

$$\lim_{x \to -\infty} F(x) = 0 \text{ 和 } \lim_{x \to \infty} F(x) = 1$$

同一样本空间上两个离散随机变量 X 和 Y 的可能取值分别为 x_1，x_2，…和 y_1，y_2，…，如果对所有的 i 和 j，满足：

$$P(X = x_i, Y = y_j) = P(X = x_i)P(Y = y_j)$$

则 X 和 Y 是独立的。将此定义推广到两个以上离散随机变量的情形，如果对所有 i、j 和 k，满足：

$$P(X = x_i, Y = y_j, Z = z_k) = P(X = x_i)P(Y = y_j)P(Z = z_k)$$

则 X、Y 和 Z 是相互独立的。

刚才所说的 X、Y 的取值都是离散的，还有一种情况下取值是连续的。以人的寿命为例，可以是任意的正实数值。与频率函数相对的是密度函数（density function）$f(x)$。$f(x)$ 有这些性质：$f(x) \geqslant 0$，f 分段连续且 $\int_{-\infty}^{\infty} f(x)\mathrm{d}x = 1$。如果 X 是具有密度函数 f 的随机变量，那么对于任意的 $a < b$，X 落在区间 (a, b) 上的概率是密度函数从 a 到 b 的下方面积：

$$P(a < X < b) = \int_a^b f(x)\mathrm{d}x$$

随机数生成对于随机算法、随机协议和随机性能测试都很有用。MLlib 支持均匀分布（uniform distribution）、标准正态分布（standard normal distribution）、泊松分布（poisson distribution）等生成随机 RDD。

MLlib 有关随机数的类如图 11-5 所示。

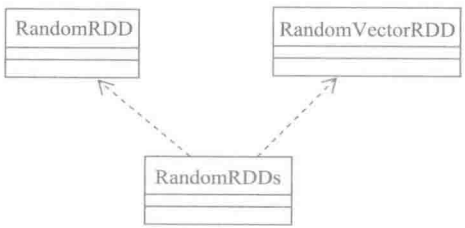


图 11-5　MLlib 有关随机数的类

以泊松分布为例，先看看它的数学定义。参数为 $\lambda(\lambda > 0)$ 的泊松频率函数（Poisson frequency function）是

$$P(X = k) = \frac{\lambda^k}{k!}\mathrm{e}^{-\lambda},\ k = 0,1,2,\cdots$$

当 $\lambda = 0.1$、1、5、10 时的泊松分布如图 11-6 所示。

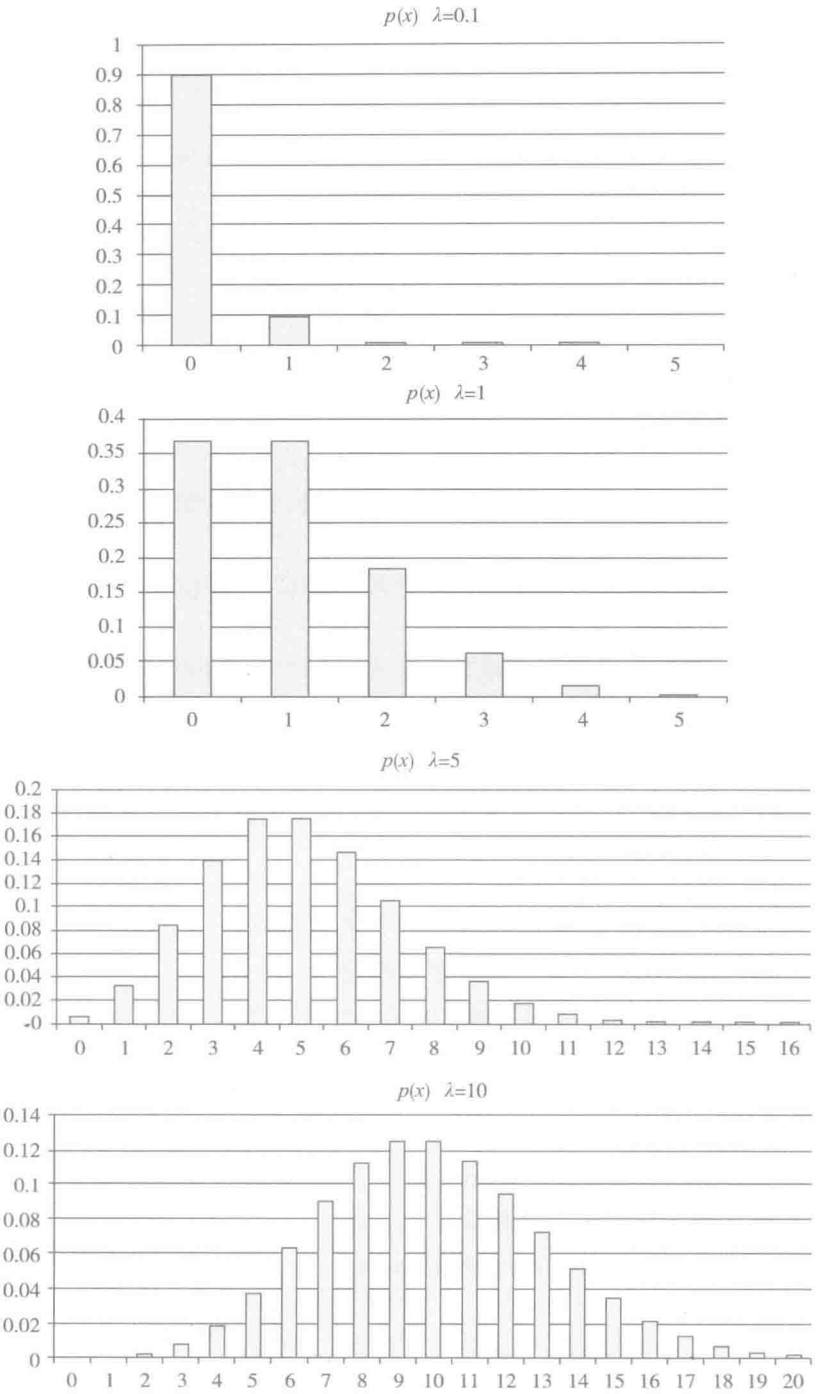

p(x) λ=0.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0 1 2 3 4 5
p(x) λ=1
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0 1 2 3 4 5
p(x) λ=5
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
-0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
p(x) λ=10
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

图 11-6 泊松分布

RandomRDDs 提供了工厂方法创建 RandomRDD 和 RandomVectorRDD。下面的例子中生成了一个包含 100 万个 double 类型随机数的 RDD[double]，其值符合标准正态分布 $N(0, 1)$，分布于 10 个分区，然后将其映射到 $N(1, 4)$。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._
val sc: SparkContext = ...
val u = normalRDD(sc, 1000000L, 10)
val v = u.map(x => 1.0 + 2.0 * x)
```

11.5　分类和回归

MLlib 支持多种多样的分析方法，例如，二元分类（binary classification）、多元分类（multiclass classification）和回归（regression）。表 11-1 列出了各类问题的支持算法。

表 11-1　分类和回归相关算法

问题类型	支持方法
二元分类	线性支持向量机（linear SVMs），逻辑回归（logistic regression），决策树（decision trees），随机森林（random forests），梯度提升树（gradient-boosted trees），朴素贝叶斯（naive Bayes）
多元分类	逻辑回归，决策树，随机森林，朴素贝叶斯
回归	线性最小二乘（linear least squares），套索（Lasso），岭回归（ridge regression），决策树，随机森林，梯度提升树，保序回归（isotonic regression）

11.5.1　数学公式

许多标准的机器学习方法都可以配制成凸优化问题，即找到一个极小的凸函数 f 依赖于一个 d 项的可变向量 w。形式上，我们可以写为优化问题 $\min_{w \in R} df(w)$，其中所述目标函数的形式为：

$$f(w) := \lambda R(w) + \frac{1}{n} \sum_{i=1}^{n} L(w; x_i, y_i)$$

这里的向量 $x_i \in R^d$ 是训练数据，$1 \leqslant i \leqslant n$ 并且 $y_i \in R^d$ 是想要预测数据的相应的标签。如果 $L(w; x_i, y_i)$ 能表示为 $w^T X$ 和 y 的函数，我们就说这个方法是线性的。几个 MLlib 的分类和回归算法都属于这一类，并在这里讨论。

目标函数 f 有两个部分：控制该模型的复杂的正则化部分和用于在训练数据上测量模型的误差的损失部分。损失函数 $L(w)$ 是典型的基于 w 的凸函数。固定的正则化参数 $\lambda \geqslant 0$（regParam 代码）定义了最小损失（即训练误差）和最小化模型的复杂性（即避免过度拟合）这两个目标之间的权衡。

（1）损失函数

在统计学，统计决策理论和经济学中，损失函数是指一种将一个事件（在一个样本空间

中的一个元素）映射到一个表达与其事件相关的经济成本或机会成本的实数上的一种函数。通常而言，损失函数由损失项（loss term）和正则项（regularization term）组成。表 11-2 列出了常用的损失函数。

表 11-2 常用的损失函数

	loss function $L(w, x, y)$	gradient 或 sub-gradient
hinge loss	$\max\{0, 1-yw^Tx\}, y\in\{-1, +1\}$	$\begin{cases}-y\cdot x & \text{if } yw^Tx<1\\ 0 & \text{otherwise}\end{cases}$
logistic loss	$\log\{1+\exp(-yw^Tx)\}, y\in\{-1, +1\}$	$-y\left(1-\frac{1}{1+\exp(-yw^Tx)}\right)\cdot x$
squared loss	$\frac{1}{2}(w^Tx-y)^2, y\in\mathbb{R}$	$(w^Tx-y)\cdot x$

这里对表 11-2 中的一些内容做些说明：

- Hinge loss：常用于软间隔支持向量机（Soft-margin SVM）的损失函数；
- Logistic loss：常用于逻辑回归（logistic regression）的损失函数；
- Squared loss：常用于最小二乘的损失函数；
- Gradient or sub gradient：梯度与次梯度。

（2）正规化

正规化的目的是鼓励简单的模型，并避免过度拟合。MLlib 支持以下正规化，如表 11-3 所示：

表 11-3 MLlib 支持的正规化

	regularizer $R(w)$	gradient 或 sub-gradient
zero (unregularized)	0	0
L2	$\frac{1}{2}\|w\|_2^2$	w
L1	$\|w\|_1$	sign(w)

这里的 sign(w) 是由向量 w 中所有项的符号（±1）组成的向量。平滑度 L2 正规化问题一般比 L1 正规化容易解决。然而 L1 正规化能帮助促进稀疏权重，导致更小、更可解释的模型，其中后者于特征选择是有用的。没有任何正规化，特别是当训练实例的数目是小的，不建议训练模型。

（3）优化

线性方法使用凸优化来优化目标函数。MLlib 使用两种方法：新元和 L-BFGS 来描述优化部分。目前，大多数算法的 API 支持随机梯度下降（SGD），并有一些支持 L-BFGS。请参阅优化选择，在优化方法中选择。

11.5.2　线性回归

线性回归（linear regression）是一类简单的指导学习方法。线性回归是预测定量响应变量的有用工具。很多统计学习方法都是从线性回归推广和扩展得到的，所以我们有必要重点理解它。

1. 简单线性回归

简单线性回归非常简单，只根据单一的预测变量 X 预测定量响应变量 Y。它假定 X 与 Y 之间存在线性关系。其数学关系如下：

$$Y \approx \beta_0 + \beta_1 X$$

≈表示近似。这种线性关系可以描述为 Y 对 X 的回归。β_0 和 β_1 是两个未知的常量，被称为线性模型的系数，它们分别表示线性模型中的截距和斜率。

β_0 和 β_1 怎么得到呢？通过大量样本数据估算出估计值。假如样本数据如下：

$$(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$$

此时问题转换为在坐标中寻找一条与所有点的距离最大程度接近的直线问题，如图 11-7[⊖]所示。

使用最小二乘方法最终求得的估计值 (β'_0, β'_1)。

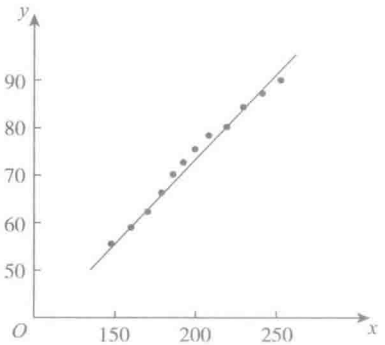


图 11-7　简单线性回归

实际情况，所有的样本或者真实数据不可能真的都在一条直线上，每个坐标都会有误差，所以可以表示为如下关系：

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

上式也称为总体回归直线（population regression line），是对 X 和 Y 之间真实关系的最佳线性近似。

2. 多元线性回归

相比简单线性回归，实践中常常不止一个预测变量，这就要求对简单线性回归进行扩展。虽然可以给每个预测变量单独建立一个简单线性回归模型，但无法做出单一的预测。更好的方法是扩展简单线性回归模型，使它可以直接包含多个预测变量。一般情况下，假设有 p 个不同的预测变量，多元线性回归模型为：

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$

其中 X_j 代表第 j 个预测变量，β_j 代表第 j 个预测变量和响应变量之间的关联。

11.5.3　分类

11.5.2 节的线性回归模型中假设响应变量 Y 是定量的，但很多时候，Y 却是定性的。比如

⊖　参考 http://baike.baidu.com/link?url=E7XoEHDT-uWtFaF4AXArXoFMfc8QT2uwSnF9kBD8FD8EHqaW9Su6HYbQRpkOKaEZojszYuT40pJZD0cht024OK。

杯子的材质是定性变量，可以是玻璃、塑料或不锈钢等。定性变量也叫分类变量。预测定性响应值是指对观测分类。

分类的目标是划分项目分类。最常见的分类类型是二元分类，二元分类有两种分类，通常命名为正和负。如果有两个以上的分类，它被称为多元分类。MLlib 支持两种线性方法分类：线性支持向量机和逻辑回归。线性支持向量机仅支持二元分类，而逻辑回归对二元分类和多元分类都支持。对于这两种方法，MLlib 支持 L1 和 L2 正规化变体。MLlib 中使用 RDD[LabeledPoint] 代表训练数据集，其中标签索引从 0 开始，如 0，1，2，…。对于二元标签 y 在 MLlib 中使用 0 表示负，使用 +1 表示正。

1. 线性支持向量机

线性支持向量机（SVM）是用于大规模分类任务的标准方法。正是在介绍损失函数时提到的：

$$L(w;x,y):=\max\{0,1-yw^{\mathrm{T}}x\}$$

默认情况下，线性支持向量机使用 L2 正规化训练。MLlib 也支持选择 L1 正规化，在这种情况下，问题就变成了线性问题。线性支持向量机算法输出 SVM 模型。给定一个新的数据点，记为 X，该模型基于 $w^{\mathrm{T}}x$ 的值做预测。默认情况下，如果 $w^{\mathrm{T}}x \geqslant 0$ 则结果是正的，否则为负。

下例展示了如何加载样本数据集，执行训练算法。

```
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
// 加载LIBSVM格式的训练数据
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 将数据切分为训练数据 (60%) 和测试数据 (40%)
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// 运行训练算法构建模型
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// 清除默认阈值
model.clearThreshold()
// 在测试数据上计算原始分数
val scoreAndLabels = test.map { point =>
    val score = model.predict(point.features)
    (score, point.label)
}
// 获取评估指标
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)
// 保存和加载模型
model.save(sc, "myModelPath")
val sameModel = SVMModel.load(sc, "myModelPath")
```

SVMWithSGD.train 默认执行 L2 正规化，可以设置正则化参数为 1.0 来执行 L1 正规化。配置及优化 SVMWithSGD 的代码如下：

```
import org.apache.spark.mllib.optimization.L1Updater
val svmAlg = new SVMWithSGD()
svmAlg.optimizer.
    setNumIterations(200).
    setRegParam(0.1).
    setUpdater(new L1Updater)
val modelL1 = svmAlg.run(training)
```

2. 逻辑回归

逻辑回归（logistic regression，LR）被广泛用于预测二元响应。它正是在介绍损失函数时提到的：

$$L(w; x, y) := \log(1+\exp(-yw^{\mathrm{T}}x))$$

对于二元分类问题，该算法输出二元逻辑回归模型。给定一个新的数据点，记为 X，该模型基于应用逻辑函数

$$f(z)=\frac{1}{1+e^{-z}}$$

做预测，其中 $z = w^{\mathrm{T}}x$。默认情况下，如果 $f(w^{\mathrm{T}}x) > 0.5$，输出为正，否则为负。虽然不像线性支持向量机，逻辑回归模型中，$f(z)$ 的原始输出具有一概率解释（即 X 是正的概率）。

二元逻辑回归可以推广到多元逻辑回归来训练和预测多元分类问题。对于多元分类问题，该算法将输出一个多元逻辑回归模型，其中包含 K–1 个二元逻辑回归模型。MLlib 实现了两种算法来解决逻辑回归分析：小批量梯度下降和 L-BFGS。Spark 官方推荐 L-BFGS，因为它比小批梯度下降的收敛更快。

下例演示了如何使用逻辑回归。

```
//运行训练算法构建模型
val model = new LogisticRegressionWithLBFGS()
    .setNumClasses(10)
    .run(training)
//在测试数据上计算原始分数
val predictionAndLabels = test.map { case LabeledPoint(label, features) =>
    val prediction = model.predict(features)
    (prediction, label)
}
//获取评估指标.
val metrics = new MulticlassMetrics(predictionAndLabels)
val precision = metrics.precision
println("Precision = " + precision)
//保存和加载模型
model.save(sc, "myModelPath")
val samleModel = LogisticRegressionModel.load(sc, "myModelPath")
```

11.5.4 回归

1. 线性最小二乘、套索和岭回归

线性最小二乘公式是回归问题最常见的公式。在介绍损失函数时也提到过它的公式：

$$L(w; x, y) := \frac{1}{2}(w^{\mathrm{T}}x - y)^2$$

多种多样的回归方法通过使用不同的正规化类型，都派生自线性最小二乘。例如，普通最小二乘或线性最小二乘使用非正规化；岭回归使用 L2 正规化；套索使用 L1 正规化。对于所有这些模型的损失和训练误差：

$$\frac{1}{n}\sum_{i=1}^{n}(w^{\mathrm{T}}x_i - y_i)^2$$

就是均方误差。

下面的例子演示了如何使用线性回归。

```
// 加载解析数据
val data = sc.textFile("data/mllib/ridge-data/lpsa.data")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble)))
}.cache()
// 构建模型
val numIterations = 100
val model = LinearRegressionWithSGD.train(parsedData, numIterations)
// 使用训练样本计算模型并且计算训练误差
val valuesAndPreds = parsedData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val MSE = valuesAndPreds.map{case(v, p) => math.pow((v - p), 2)}.mean()
println("training Mean Squared Error = " + MSE)
// 保存与加载模型
model.save(sc, "myModelPath")
val sameModel = LinearRegressionModel.load(sc, "myModelPath")
```

2. 流线性回归

流式数据可以适用于线上的回归模型，每当有新数据到达时，更新模型的参数。MLlib 目前使用普通最小二乘法支持流线性回归。除了每批数据到达时，模型更新最新的数据外，实际与线下的执行是类似的。

下面的例子，假设已经初始化好了 StreamingContext ssc 来演示流线性回归（streaming linear regression）。

```
val numFeatures = 3
val model = new StreamingLinearRegressionWithSGD()
.setInitialWeights(Vectors.zeros(numFeatures))
model.trainOn(trainingData)
model.predictOnValues(testData.map(lp => (lp.label, lp.features))).print()
```

```
ssc.start()
ssc.awaitTermination()
```

11.6　决策树

决策树是分类和回归的机器学习任务中常用的方法。决策树广泛使用，因为它们很容易解释，处理分类的功能，延伸到多元分类设置，不需要缩放功能，并能捕捉到非线性和功能的交互。

MLlib 使用连续和分类功能支持决策树的二元和多元的分类和回归。通过行实现分区数据，允许分布式训练数以百万计的实例。

11.6.1　基本算法

决策树是一个贪心算法，即在特性空间上执行递归的二元分割。决策树为每个最底部（叶）分区预测相同的标签。为了在每个树节点上获得最大的信息，每个分区是从一组可能的划分中选择的最佳分裂。

1. 节点不纯度和信息增益

节点不纯度是节点上标签的均匀性的量度。当前实现提供了两种分类不纯度测量的方法（基尼不纯度和熵）和一种回归不纯度测量的方法（方差），如表 11-4 所示。

表 11-4　不纯度测量方法

不 纯 度	任　务	公　式	描　述
基尼不纯度（Gini impurity）	分类	$\sum_{i=1}^{c} f_i(1-f_i)$	f_i 是节点上标签 i 的频率函数，C 是唯一标签的数量
熵（Entropy）	分类	$\sum_{i=1}^{c} -f_i\log(f_i)$	同上
方差（Variance）	回归	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i 是实例的标签，N 是实例的数量，μ 是 $\frac{1}{N}\sum_{i=1}^{N} y_i$ 给出的均值

信息增益是父节点不纯度与两个子节点不纯度的加权总和之间的差。假设将有 s 个分区，大小为 N 的数据集 D 划分为两个数据集 D_{left} 和 D_{right}，那么信息增益为：

$$\text{IG}(D,s) = \text{Impurity}(D) - \frac{N_{\text{left}}}{N}\text{Impurity}(D_{\text{left}}) - \frac{N_{\text{right}}}{N}\text{Impurity}(D_{\text{right}})$$

2. 划分候选人

（1）连续特征

对于单机上实现的小数据集，给每个连续特征划分的候选人在此特征上有唯一值。有些实现了对特征值排序，为了加速计算，使用这些有序的唯一值划分候选人。对于大的分布式

数据集，排序是很昂贵的。通过在样本数据分数上执行位计算，实现了计算近似的划分候选人集合。有序划分创建了“箱”，可使用 maxBins 参数指定这样的容器的最大数量。


注意 箱的数量不能大于实例的数目 N（因为默认的 maxBins 值在罕见情况下为 32）。如果条件不满足，生成树算法自动降低垃圾箱的数量。

（2）分类特征

对于有 M 种可能值的分类特征，将会有 $2^{M-1}-1$ 个划分候选人。对于二元（0/1）分类和回归，我们可以通过平均标签排序的分类特征值，减少划分候选人至 $M-1$ 的数量。例如，对于一个有 A、B 和 C 三个分类的分类特征的二元分类问题，其相应的标签 1 的比例是 0.2、0.6 和 0.4 时，分类特征是有序的 A、C、B。两个划分候选人分别是 A|C, B 和 A, C|B，其中 | 标记划分。

在多元分类中共有 $2^{M-1}-1$ 种可能的划分，无论何时都可能被使用。当 $2^{M-1}-1$ 比参数 maxBins 大时，我们使用与二元分类和回归相类似的方法。M 种分类特征用不纯度排序，最终得到需要考虑的 $M-1$ 个划分候选人。

3. 停止规则

递归树的构建当满足下面三个条件之一时会停在一个节点。

- 节点的深度与 maxBins 相等；
- 没有划分候选人导致信息增益大于 minInfoGain；
- 没有划分候选人产生的子节点都至少有 minInstancesPerNode 个训练实例。

11.6.2 使用例子

下面的例子演示了使用基尼不纯度作为不纯度算法且树深为 5 的决策树执行分类。

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// 训练决策树模型
//  空categoricalFeaturesInfo说明所有的特征是连续的
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32
//  trainingData是训练数据
val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
    impurity, maxDepth, maxBins)
// 在测试实例上计算
val labelAndPreds = testData.map { point =>
```

```
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification tree model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = DecisionTreeModel.load(sc, "myModelPath")
```

下面的例子演示了使用方差作为不纯度算法且树深为 5 的决策树执行分类。

```
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "variance"
val maxDepth = 5
val maxBins = 32
val model = DecisionTree.trainRegressor(trainingData, categoricalFeaturesInfo,
   impurity,maxDepth, maxBins)
val labelsAndPredictions = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression tree model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = DecisionTreeModel.load(sc, "myModelPath")
```

11.7 随机森林

合奏是一个创建由其他模型的集合组合而成的模型的学习算法。MLlib 支持两个主要的合奏算法：梯度提升决策树（GradientBoostedTrees）和随机森林（RandomForest），它们都使用决策树作为其基础模型。梯度提升决策树（GBT）和随机森林虽然都是决策树合奏的学习算法，但是训练过程是不同的。关于合奏有以下几个权衡点：

- GBT 每次都要训练一棵树，所以它们比随机森林需要更长的时间来训练。随机森林可以平行地训练多棵树。另一方面，GBT 往往比随机森林更合理地使用更小（浅）的树并且训练小树会花费更少的时间。
- 随机森林更不易发生过度拟合。随机森林训练更多的树会减少多半的过度拟合，而 GBT 训练更多的树会增加过度拟合。（在统计语言中，随机森林通过使用更多的树木减少方差，而 GBT 通过使用更多的树木减少偏差。）
- 随机森林可以更容易调整，因为性能与树木的数量是单调增加的。但如果 GBT 树木的数量增长过大，性能可能开始下降。

总之，两种算法都是有效的，具体选择应取决于特定的数据集。

随机森林是分类与回归中最成功的机器学习模型之一。为了减少过度拟合的风险，随机森林将很多决策树结合起来。和决策树相似，随机森林处理分类的功能，延伸到多元分类设

置，不需要缩放功能，并能捕捉到非线性和功能的交互。

MLlib 使用连续和分类功能支持随机森林的二元和多元的分类和回归。

11.7.1 基本算法

随机森林训练一个决策树的集合，所以训练可以并行。该算法随机性注入训练过程，使每个决策树会有一点不同。结合每棵树的预测降低了预测的方差，改进了测试数据的性能。

1. 随机注入

算法随机性注入训练的过程包括：

1）每次迭代对原始数据集进行二次采样获得不同的训练集，即引导。

2）考虑在树的每个节点上将特征的不同随机子集分割。

除了这些随机性，每个决策树个体都以同样的方法训练。

2. 预测

对随机森林做预测，就必须聚合它的决策树集合的预测。分类和回归的聚合是不同的：

- 分类采用多数表决。每棵树的预测作为对分类的一次投票，收到最多投票的分类就是预测结果。
- 回归采用平均值。每棵树都有一个预测值，这些树的预测值的平均值就是预测结果。

11.7.2 使用例子

下面例子演示了使用随机森林执行分类。

```
// 训练随机森林模型
// 空categoricalFeaturesInfo 说明所有特征是连续的
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "gini"
val maxDepth = 4
val maxBins = 32
val model = RandomForest.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
    numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)
val labelAndPreds = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification forest model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = RandomForestModel.load(sc, "myModelPath")
```

下面例子演示了使用随机森林执行回归。

```
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "variance"
val maxDepth = 4
val maxBins = 32
val model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo,
    numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)
val labelsAndPredictions = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression forest model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = RandomForestModel.load(sc, "myModelPath")
```

11.8 梯度提升决策树

GBT 迭代训练决策树，以便最小化损失函数。和决策树相似，随机森林处理分类的功能，延伸到多元分类设置，不需要缩放功能，并能捕捉到非线性和功能的交互。

MLlib 使用连续和分类功能支持梯度提升决策树的二元和多元的分类和回归。

11.8.1 基本算法

GBT 迭代训练一个决策树的序列。在每次迭代中，算法使用当前合奏来预测每个训练实例的标签，然后将预测与真实的标签进行比较。数据集被重新贴上标签，将重点放在预测不佳的训练实例上。因此，在下一迭代中，决策树将帮助纠正先前的错误。重贴标签的具体机制是由损失函数定义的。随着每次迭代，GBT 进一步减少训练数据上的损失函数。表 11-5 列出了 MLlib 中 GBT 支持的损失函数。请注意，每个损失只适用于分类或回归之一。其中 N 表示实例数量，y_i 表示实例 i 的标签，x_i 表示实例 i 的特征，$F(X_i)$ 表示实例 i 的模型预测标签。

表 11-5 MLlib 中 GBT 支持的损失函数

损　失	任　务	公　式	描　述
Log Loss	分类	$2\sum_{i=1}^{N}\log(1+\exp(-2y_iF(X_i)))$	二次二项式负对数似然
Squared Error	回归	$\sum_{i=1}^{N}(y_i-F(X_i))^2$	也称为 L2 损失。回归任务的默认损失
Absolute Error	回归	$\sum_{i=1}^{N}\lvert y_i-F(X_i)\rvert$	也称为 L1 损失。比均方误差更稳定

11.8.2 使用例子

下例演示了用 Log Loss 作为损失函数，使用 GBT 执行分类的例子。

```
// 训练GradientBoostedTrees模型
//  默认使用LogLoss
val boostingStrategy = BoostingStrategy.defaultParams("Classification")
boostingStrategy.numIterations = 3 // Note: Use more iterations in practice.
boostingStrategy.treeStrategy.numClasses = 2
boostingStrategy.treeStrategy.maxDepth = 5
//  空categoricalFeaturesInfo说明所有特征是连续的
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()
val model = GradientBoostedTrees.train(trainingData, boostingStrategy)
val labelAndPreds = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification GBT model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = GradientBoostedTreesModel.load(sc, "myModelPath")
```

下例演示了用 Squared Error 作为损失函数，使用 GBT 执行回归的例子。

```
//训练GradientBoostedTrees模型
//  defaultParams指定了Regression，默认使用SquaredError
val boostingStrategy = BoostingStrategy.defaultParams("Regression")
boostingStrategy.numIterations = 3 // Note: Use more iterations in practice.
boostingStrategy.treeStrategy.maxDepth = 5
//  空categoricalFeaturesInfo说明所有特征是连续的
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]()
val model = GradientBoostedTrees.train(trainingData, boostingStrategy)
val labelsAndPredictions = testData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
}
val testMSE = labelsAndPredictions.map{ case(v, p) => math.pow((v - p), 2)}.mean()
println("Test Mean Squared Error = " + testMSE)
println("Learned regression GBT model:\n" + model.toDebugString)
model.save(sc, "myModelPath")
val sameModel = GradientBoostedTreesModel.load(sc, "myModelPath")
```

11.9 朴素贝叶斯

11.9.1 算法原理

我们首先来介绍一些数学中的理论，然后来看朴素贝叶斯。

条件概率：A 和 B 表示两个事件，且 $P(B) \neq 0$（B 事件发生的概率不等于 0），则给定事件 B 发生的条件下事件 A 发生的条件概率定义为：

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

使用条件概率推导出乘法定律：A 和 B 表示两个事件，且 $P(B) \neq 0$（B 事件发生的概率不等于0）。那么：

$$P(A \cap B) = P(A \mid B)P(B)$$

将乘法定律扩展为全概率定律：事件 B_1，B_2，…，B_n 满足 $\mathrm{U}_{i=1}^{n} B_i = \Omega$，$B_i \cap B_j = \varnothing$，$i \neq j$，且对所有的 i，$P(B_i) > 0$。那么，对于任意的 A，满足：

$$P(A) = \sum_{i=1}^{n} P(A \mid B_i)P(B_i)$$

贝叶斯公式：事件 A，B_1，B_2，…，B_n，其中 B_i 不相交，$\mathrm{U}_{i=1}^{n} B_i = \Omega$，且对所有的 i，$P(B_i) > 0$。那么：

$$P(B_j \mid A) = \frac{P(A \mid B_j)P(B_j)}{\sum_{i=1}^{n} P(A \mid B_i)P(B_i)}$$

朴素贝叶斯分类算法是一种基于每对特征之间独立性的假设的简单的多元分类算法。朴素贝叶斯的思想：对于给出的待分类项，求解在此项出现的条件下各个类别出现的概率，哪个最大，就认为此待分类项属于哪个类别。朴素贝叶斯分类的定义如下：

1）设 $x = \{a_1, a_2, \cdots, a_m\}$ 为待分类项，每个 a_i 为 x 的一个特征属性；

2）类别集合 $C = \{y_1, y_2, \cdots, y_n\}$；

3）计算 $P(y_1|x)$，$P(y_2|x)$，…，$P(y_n|x)$；

4）如果 $P(y_k|x) = \max\{(y_1|x), P(y_2|x), \cdots, P(y_n|x)\}$，则 $x \in y_k$。

关键在第三步：

1）找到一个已知分类的待分类项集合，这个集合叫做训练样本集。

2）统计得到在各类别下各个特征属性的条件概率估计。即 $P(a_1|y_1)$，$P(a_2|y_1)$，…，$P(a_m|y_1)$；$(a_1|y_2)$，$P(a_2|y_2)$，…，$P(a_m|y_2)$；…；$(a_1|y_n)$，$P(a_2|y_n)$，…，$P(a_m|y_n)$。

3）如果各个特征属性都是独立的，则根据贝叶斯公式可以得到以下推导：

$$P(y_t|x) = \frac{P(x|y_i)P(y_i)}{P(x)}$$

$$P(x|y_i)P(y_i) = P(a_1|y_i), P(a_2|y_i), \cdots, P(a_m|y_i) = P(y_i)\prod_{j=1}^{m} P(a_j|y_i)$$

朴素贝叶斯能够被非常有效的训练。它被单独传给训练数据，计算给定标签特征的条件概率分布并给出观察结果用于预测。

MLlib 支持多项朴素贝叶斯和伯努利朴素贝叶斯。这些模型典型的应用是文档分类。在这方面，每个观察是一个文档，每个特征代表一个条件，其值是条件的频率（在多项朴素贝叶斯中）或一个由零个或一个指示该条件是否在文档中找到（在伯努利朴素贝叶斯中）。特征值

必须是非负的。模型类型选择使用可选的参数“多项”或“伯努利”。“多项”作为默认模型。通过设置参数 λ（默认为 1.0）添加剂平滑。为文档分类，输入特征向量通常是稀疏的，因为稀疏向量能利用稀疏性的优势。因为训练数据只使用一次，所以没有必要缓存它。

11.9.2 使用例子

下面的例子演示了如何使用多项朴素贝叶斯。

```
import org.apache.spark.mllib.classification.{NaiveBayes, NaiveBayesModel}
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
val data = sc.textFile("data/mllib/sample_naive_bayes_data.txt")
val parsedData = data.map { line =>
    val parts = line.split(',')
    LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.
        toDouble)))
}
val splits = parsedData.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0)
val test = splits(1)
val model = NaiveBayes.train(training, lambda = 1.0, modelType = "multinomial")
val predictionAndLabel = test.map(p => (model.predict(p.features), p.label))
val accuracy = 1.0 * predictionAndLabel.filter(x => x._1 == x._2).count() / test.
   count()
model.save(sc, "myModelPath")
val sameModel = NaiveBayesModel.load(sc, "myModelPath")
```

11.10 保序回归

11.10.1 算法原理

保序回归（isotonic regression）属于回归算法，其定义为：给定一个有限的实数集合 $Y=\{y_1, y_2, \cdots, y_n\}$ 表示观测响应，$X=\{x_1, x_2, \cdots, x_n\}$ 表示未知的响应值，进行拟合找到一个最小化函数：

$$f(x)=\sum_{i=1}^{n}\omega_i(y_i-x_i)^2$$

并使用 $x_1 \leqslant x_2 \cdots \leqslant \cdots \leqslant x_n$ 对目标排序，其中 ω_i 是大于 0 的权重。最终的函数被称为保序回归，并且它是唯一的。它可以看做是排序限制下的最小二乘问题。基本上保序回归是拟合原始数据点最佳的单调函数。

MLlib 支持 PAVA（pool adjacent violators algorithm），此算法使用一种办法来平行化保序回归。保序回归有一个可选参数 isotonic，默认值是 true。此参数指定保序回归是保序的（单调增加）还是不保序的（单调减少）。

保序回归的结果被视为分段线性函数。因此，预测的规则是：

1）如果预测输入能准确匹配训练特征，那么返回相关预测。如果有多个预测匹配训练特征，那么会返回其中之一。

2）如果预测输入比所有的训练特征低或者高，那么最低和最高的训练特征各自返回。如果有多个预测比所有的训练特征低或者高，那么都会返回。

3）如果预测输入介于两个训练特征，那么预测会被视为分段线性函数和从最接近的训练特征中计算得到的插值。

11.10.2　使用例子

下面的例子演示了如何使用保序回归。

```
import org.apache.spark.mllib.regression.{IsotonicRegression, IsotonicRegres-
    sionModel}
// 省略数据加载及样本划分的代码
val model = new IsotonicRegression().setIsotonic(true).run(training)
val predictionAndLabel = test.map { point =>
    val predictedLabel = model.predict(point._2)
    (predictedLabel, point._1)
}
val meanSquaredError = predictionAndLabel.map{case(p, l) => math.pow((p - l), 2)}.
mean()
println("Mean Squared Error = " + meanSquaredError)
model.save(sc, "myModelPath")
val sameModel = IsotonicRegressionModel.load(sc, "myModelPath")
```

11.11　协同过滤

协同过滤通常用于推荐系统。这些技术旨在填补用户关联矩阵的缺失项。MLlib 支持基于模型的协同过滤，用户和产品由可以预测缺失项的潜在因素的小集合来描述。MLlib 采用交替最小二乘（ALS）算法来学习这些潜在的因素。

矩阵分解的标准方法基于协同过滤处理用户项矩阵的条目是明确的。现实世界的用例只能访问隐式反馈是更常见的（例如浏览、点击、购买、喜欢、股份等）。

下面的例子演示了如何使用协同过滤。

```
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel
import org.apache.spark.mllib.recommendation.Rating
val data = sc.textFile("data/mllib/als/test.data")
val ratings = data.map(_.split(',') match { case Array(user, item, rate) =>
        Rating(user.toInt, item.toInt, rate.toDouble)
    })
// 使用ALS构建推荐模型
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, 0.01)
// 模型计算
```

```
val usersProducts = ratings.map { case Rating(user, product, rate) =>
    (user, product)
}
val predictions =
    model.predict(usersProducts).map { case Rating(user, product, rate) =>
        ((user, product), rate)
    }
val ratesAndPreds = ratings.map { case Rating(user, product, rate) =>
    ((user, product), rate)
}.join(predictions)
val MSE = ratesAndPreds.map { case ((user, product), (r1, r2)) =>
    val err = (r1 - r2)
    err * err
}.mean()
println("Mean Squared Error = " + MSE)
model.save(sc, "myModelPath")
val sameModel = MatrixFactorizationModel.load(sc, "myModelPath")
```

11.12 聚类

聚类分析又称群分析，它是研究（样品或指标）分类问题的一种统计分析方法。聚类分析以相似性为基础，在一个聚类中的模式之间比不在同一聚类中的模式之间具有更多的相似性。MLlib 支持的聚类算法如下：

- K-means；
- 高斯混合（Gaussian mixture）；
- power iteration clustering（PIC）；
- latent Dirichlet allocation（LDA）；
- 流式 K-means。

11.12.1 K-means

K-means 算法是硬聚类算法，是典型的基于原型的目标函数聚类方法的代表，它是数据点到原型的某种距离作为优化的目标函数，利用函数求极值的方法得到迭代运算的调整规则。

聚类属于无监督学习，以往的回归、朴素贝叶斯、SVM 等都是有类别标签 y 的，也就是说，样本中已经给出了样本的分类。而聚类的样本中却没有给定 y，只有特征 x，比如假设宇宙中的星星可以表示成三维空间中的点集 (x, y, z)。聚类的目的是找到每个样本 x 潜在的类别 y，并将同类别 y 的样本 x 放在一起。

在聚类问题中，训练样本 $X = \{x_1, x_2, \cdots, x_m\}$，每个 $x_i \in R^n$，K-means 算法是将样本聚类成 k 个簇（cluster），具体算法描述如下：

1）随机选取 k 个聚类质心点（cluster centroids）为 $\mu_1, \mu_2, \cdots, u_k \in R^n$；

2）重复下面过程直到收敛。

对每一个样本 i 计算它应该属于的类

$$c_i := \text{argmin}_j \|x_i - \mu_j\|^2$$

对于每一个类重新计算该类的质心

$$u_i := \frac{\sum_{i=1}^{m} \{c_i = j\} x_i}{\sum_{i=1}^{m} \{c_i = j\}}$$

k 是我们事先给定的聚类数，c_i 代表样本 i 与 k 个类中距离最近的那个类，c_i 的值是 1 到 k 中的一个。质心 u_j 代表我们对属于同一个类的样本中心点的猜测，拿星团模型来解释就是要将所有的星星聚成 k 个星团，首先随机选取 k 个宇宙中的点（或者 k 个星星）作为 k 个星团的质心，然后第一步对于每一个星星计算其到 k 个质心中每一个的距离，接着选取距离最近的那个星团作为 c_i，这样经过第一步每一个星星都有了所属的星团；第二步对于每一个星团，重新计算它的质心 μ_j（对里面所有的星星坐标求平均）。重复迭代第一步和第二步直到质心不变或者变化很小。图 11-8 演示了以上过程[⊖]。

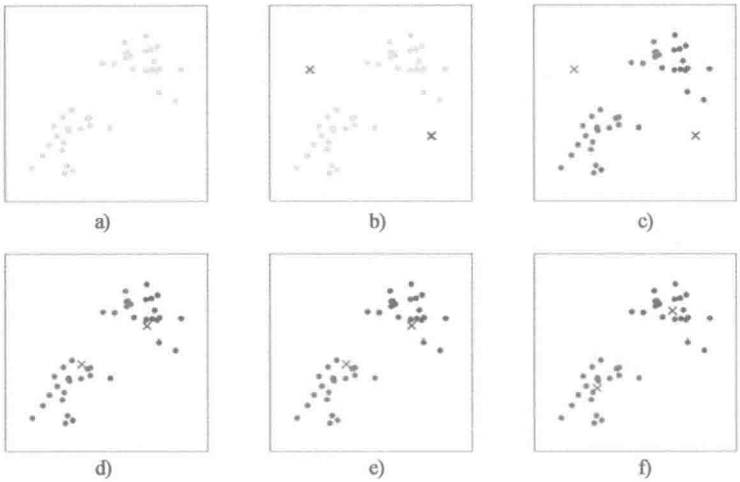


图 11-8　K-means 算法示例

下面的例子演示了 K-means 算法的使用。

```
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.linalg.Vectors
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)
val WSSSE = clusters.computeCost(parsedData)
```

⊖　部分内容引用自博文 http://www.cnblogs.com/jerrylead/archive/2011/04/06/2006910.html。

```
println("Within Set Sum of Squared Errors = " + WSSSE)
clusters.save(sc, "myModelPath")
val sameModel = KMeansModel.load(sc, "myModelPath")
```

11.12.2 高斯混合

K-means 的结果是每个数据点被分配到其中某一个 cluster 了，而高斯混合则给出这些数据点被分配到每个 cluster 的概率。高斯混合的算法与 K-means 算法类似，有兴趣的读者可以自行研究，这里只给出 MLlib 中高斯混合的使用例子。

```
import org.apache.spark.mllib.clustering.GaussianMixture
import org.apache.spark.mllib.clustering.GaussianMixtureModel
import org.apache.spark.mllib.linalg.Vectors
val data = sc.textFile("data/mllib/gmm_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').map(_.toDouble))).cache()
val gmm = new GaussianMixture().setK(2).run(parsedData)
gmm.save(sc, "myGMMModel")
val sameModel = GaussianMixtureModel.load(sc, "myGMMModel")
for (i <- 0 until gmm.k) {
    println("weight=%f\nmu=%s\nsigma=\n%s\n" format
        (gmm.weights(i), gmm.gaussians(i).mu, gmm.gaussians(i).sigma))
}
```

11.12.3 快速迭代聚类

快速迭代聚类（power iteration clustering，PIC）是一种简单可扩展的图聚类方法。其使用例子如下：

```
import org.apache.spark.mllib.clustering.{PowerIterationClustering, PowerIteration-
    ClusteringModel}
import org.apache.spark.mllib.linalg.Vectors
val similarities: RDD[(Long, Long, Double)] = ...
val pic = new PowerIterationClustering()
    .setK(3)
    .setMaxIterations(20)
val model = pic.run(similarities)
model.assignments.foreach { a =>
    println(s"${a.id} -> ${a.cluster}")
}
model.save(sc, "myModelPath")
val sameModel = PowerIterationClusteringModel.load(sc, "myModelPath")
```

11.12.4 latent Dirichlet allocation

latent Dirichlet allocation（LDA）是一个三层贝叶斯概率模型，包含词、主题和文档三层结构。文档到主题服从 Dirichlet 分布，主题到词服从多项式分布。[⊖]

LDA 是一种非监督机器学习技术，可以用来识别大规模文档集（document collection）或

⊖ 参考文档 http://blog.sina.com.cn/s/blog_50d4c97b0100n9ee.html。

语料库（corpus）中潜藏的主题信息。它采用了词袋（bag of words）的方法，这种方法将每一篇文档视为一个词频向量，从而将文本信息转化为了易于建模的数字信息。但是词袋方法没有考虑词与词之间的顺序，这简化了问题的复杂性，同时也为模型的改进提供了契机。每一篇文档代表了一些主题所构成的一个概率分布，而每一个主题又代表了很多单词所构成的一个概率分布。由于 Dirichlet 分布随机向量各分量间的弱相关性（之所以还有点“相关”，是因为各分量之和必须为 1），使得我们假想的潜在主题之间也几乎是不相关的，这与很多实际问题并不相符，从而造成了 LDA 的又一个遗留问题。

对于语料库中的每篇文档，LDA 定义了如下生成过程（generative process）：

1）对每一篇文档，从主题分布中抽取一个主题；

2）从上述被抽到的主题所对应的单词分布中抽取一个单词；

3）重复上述过程直至遍历文档中的每一个单词。

下例演示了 LDA 的使用。

```
import org.apache.spark.mllib.clustering.LDA
import org.apache.spark.mllib.linalg.Vectors
val data = sc.textFile("data/mllib/sample_lda_data.txt")
val parsedData = data.map(s => Vectors.dense(s.trim.split(' ').map(_.toDouble)))
// Index documents with unique IDs
val corpus = parsedData.zipWithIndex.map(_.swap).cache()
val ldaModel = new LDA().setK(3).run(corpus)
println("Learned topics (as distributions over vocab of " + ldaModel.vocabSize +
    " words):")
val topics = ldaModel.topicsMatrix
for (topic <- Range(0, 3)) {
    print("Topic " + topic + ":")
    for (word <- Range(0, ldaModel.vocabSize)) { print(" " + topics(word, topic)); }
    println()
}
```

11.12.5　流式 K-means

当数据流到达，我们可能想要动态地估算 cluster，并更新它们。该算法采用了小批量的 K-means 更新规则。对每一批数据，将所有的点分配到最近的 cluster，并计算最新的 cluster 中心，然后更新每个 cluster 的公式为：

$$c_{t+1} = \frac{c_t n_t a + x_t m_t}{n_t a + m_t}$$

$$n_{t+1} = n_t + m_t$$

c_t 是前一次计算得到的 cluster 中心，n_t 是已经分配到 cluster 的点数，x_t 是从当前批次得到的 cluster 的新中心，m_t 是当前批次加入 cluster 的点数。衰减因子 a 可被用于忽略过去的数据：$a = 1$ 时所有数据都从一开始就被使用；$a = 0$ 时只有最近的数据将被使用。这类似于一个指数加权移动平均值。

下面的例子演示了流式 K-means 的使用。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.clustering.StreamingKMeans
val trainingData = ssc.textFileStream("/training/data/dir").map(Vectors.parse)
val testData = ssc.textFileStream("/testing/data/dir").map(LabeledPoint.parse)
val numDimensions = 3
val numClusters = 2
val model = new StreamingKMeans()
    .setK(numClusters)
    .setDecayFactor(1.0)
    .setRandomCenters(numDimensions, 0.0)
model.trainOn(trainingData)
model.predictOnValues(testData.map(lp => (lp.label, lp.features))).print()

ssc.start()
ssc.awaitTermination()
```

11.13 维数减缩

维数减缩是减少所考虑变量的数量的过程。维数减缩有两种方式：

- 奇异值分解；
- 主成分分析。

11.13.1 奇异值分解

奇异值分解（singular value decomposition，SVD）将一个矩阵因子分解为三个矩阵 U、Σ 和 V：

$$A = U\Sigma V^{\mathrm{T}}$$

其中 U 是正交矩阵，其列被称为左奇异向量；Σ 是对角矩阵，其对角线是非负的且以降序排列，因此被称为奇异值；V 也是正交矩阵，其列被称为右奇异向量。

对于大的矩阵，除了顶部奇异值和它的关联奇异值，我们不需要完全分解。这样可以节省存储、去噪声和恢复矩阵的低秩结构。如果我们保持前 7 个顶部奇异值，那么最终的低秩矩阵为：

$$U\text{：}\ m\times k$$
$$\Sigma\text{：}\ k\times k$$
$$V\text{：}\ n\times k$$

假设 n 小于 m。奇异值和右奇异向量来源于特征值和 Gramian 矩阵 $A^{\mathrm{T}}A$ 的特征向量。矩阵存储左奇异向量 U，通过矩阵乘法 $U = A(VS^{-1})$ 计算。使用的实际方法基于计算成本自动被定义：

1）如果 ($n < 100$) 或者 ($k > n/2$)，我们首先计算 Gramian 矩阵，然后再计算其顶部特征向量并将特征向量本地化到 Driver。这需要单次传递，在每个 Executor 和 Driver 上使用 $O(n^2)$

的存储，并花费 Driver 上 $O(n^2k)$ 的时间。

2）否则，将使用分布式的方式计算 A^TA 的值，并且发送给 ARPACK 计算顶部特征向量。这需要 $O(k)$ 次传递，每个 Executor 上使用 $O(n)$ 的存储，在 Driver 上使用 $O(nk)$ 的存储。


注意 ARPACK 是一个解决大规模特征问题的软件包，它实际是一个 Fortran77 子程序的集合。

下面的例子演示了 SVD 的使用。

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
val mat: RowMatrix = ...
// 计算顶部的20个奇异值并响应奇异向量
val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(20,
computeU = true)
val U: RowMatrix = svd.U // The U factor is a RowMatrix.
val s: Vector = svd.s // The singular values are stored in a local dense vector.
val V: Matrix = svd.V // The V factor is a local dense matrix.
```

11.13.2 主成分分析

主成分分析（Principal compoment analysis，PCA）是一种统计方法，此方法找到一个旋转，使得第一坐标具有可能的最大方差，并且每个随后的坐标都具有可能的最大方差。旋转矩阵的列被称为主成分。下面的例子演示了使用 RowMatrix 计算主成分。

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val mat: RowMatrix = ...
// 计算顶部的10个主成分
val pc: Matrix = mat.computePrincipalComponents(10) // 主成分存储在密矩阵中
val projected: RowMatrix = mat.multiply(pc)
```

下面的例子演示了 PCA 的使用。

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.PCA
val data: RDD[LabeledPoint] = ...
// 计算顶部的10个主成分
val pca = new PCA(10).fit(data.map(_.features))
val projected = data.map(p => p.copy(features = pca.transform(p.features)))
```

11.14 特征提取与转型

11.14.1 术语频率反转

术语频率反转（term frequency-inverse document frequency，TF-IDF）是一个反映文集的文档中的术语的重要性，广泛应用于文本挖掘的特征矢量化方法。术语表示为 t，文档表示为 d，

文集表示为 D。术语频率 $TF(t, d)$ 表示术语 t 在文档 d 中出现的频率，文档频率 $DF(t, D)$ 表示包含术语 t 的文档数量。如果我们仅使用术语频率来测量重要性，则很容易过度强调术语出现的很频繁，而携带的关于文档的信息很少，例如 a、the 和 of 等。如果术语非常频繁地跨文集出现，这意味着它并没有携带文档的特定信息。反转文档频率是一个术语提供了多少信息的数值度量：

$$IDF(t,D) = \log \frac{|D|+1}{DF(t,D)+1}$$

$|D|$ 表示文集中的文档总数。因为使用了对数，如果一个术语出现于所有的文档中，它的 *IDF* 值变为 0。需要注意的是，应用一个平滑项，以避免被零除。*TF-IDF* 方法基于 *TF* 与 *IDF*，它的公式如下：

$$TFIDF(t,d,D) = TF(t,d) \cdot IDF(t,D)$$

这里有一些术语频率和文档频率定义的变种。在 MLlib 中，为了使 TF 和 IDF 更灵活，将它们分开了。MLlib 实现术语频率时使用了哈希。应用哈希函数将原始特征映射到了索引（术语），术语频率因此依赖于 map 的索引来计算。这种方法避免了需要计算一个全局术语到索引图，这对于大型语料库开销会很大。但由于不同的原始特征在哈希后可能变为同样的术语，所以存在潜在的哈希冲突。为了降低碰撞的机会，我们可以增加目标特征的维度（即哈希表中的桶数）。默认的特征维度是 $2^{20} = 1\,048\,576$。


注意 MLlib 还没有给文本片段提供工具，读者可以参考 http://nlp.stanford.edu/ 和 https://github.com/scalanlp/chalk。

下面的例子演示了 HashingTF 的使用。

```
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.linalg.Vector
val sc: SparkContext = ...
val documents: RDD[Seq[String]] = sc.textFile("...").map(_.split(" ").toSeq)
val hashingTF = new HashingTF()
val tf: RDD[Vector] = hashingTF.transform(documents)
```

下面的例子演示了 IDF 的使用。

```
import org.apache.spark.mllib.feature.IDF
// ... 跟着前面的例子继续
tf.cache()
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
val idf2 = new IDF(minDocFreq = 2).fit(tf)
val tfidf2: RDD[Vector] = idf2.transform(tf)
```

11.14.2 单词向量转换

Word2Vec 计算由单词表示的分布式向量。分布式表征的主要优点是，类似的单词在矢量

空间是接近的，这使得泛化小说模式更容易和模型估计更稳健。分布式向量表示被证实在许多自然语言处理应用中有用，例如，命名实体识别、消歧、解析、标记和机器翻译。

MLlib 的 Word2Vec 实现采用了 skip-gram 模型。skip-gram 的训练目标是学习单词的向量表示，其善于在同一个句子预测其上下文。给定了单词序列 ω_1，ω_2，…，ω_T，skip-gram 模型的目标是最大化平均对数似然，公式如下：

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j=-k}^{j=k}\log p(\omega_{t+j}\mid\omega_t)$$

其中 k 是训练窗口的大小。每个单词 ω 与两个向量 u_ω 和 v_ω 关联，并且由单独的向量来表示。通过给定的单词 ω_j 正确预测 ω_i 的概率是由 softmax 模型决定的。softmax 模型如下：

$$p(\omega_i\mid\omega_j)=\frac{\exp(u_{\omega_i}^{T}v_{\omega_j})}{\sum_{i=1}^{V}\exp(u_i^{T}v_{\omega_j})}$$

V 表示词汇量。

由于计算 $\log p(\omega_i|\omega_j)$ 的成本与 V 成正比（V 很容易就达到百万以上），所以 softmax 这种 skip-gram 模型是很昂贵的。为了加速 Word2Vec 计算，MLlib 使用分级 softmax，它可以减少计算 $\log p(\omega_i|\omega_j)$ 的复杂度到 $O(\log(V))$。

下面的例子演示了 Word2Vec 的使用。

```
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.{Word2Vec, Word2VecModel}
val input = sc.textFile("text8").map(line => line.split(" ").toSeq)
val word2vec = new Word2Vec()
val model = word2vec.fit(input)
val synonyms = model.findSynonyms("china", 40)
for((synonym, cosineSimilarity) <- synonyms) {
    println(s"$synonym $cosineSimilarity")
}
model.save(sc, "myModelPath")
val sameModel = Word2VecModel.load(sc, "myModelPath")
```

11.14.3　标准尺度

通过缩放到单位方差和 / 或通过在训练集的样本上使用列摘要统计移除均值使特征标准化，这是常见的预处理步骤。例如，当所有的特征都有单位方差和 / 或零均值时，支持向量机的 RBF 核或者 L1 和 L2 正规化线性模型通常能更好地工作。标准化可以提高在优化过程中的收敛速度，并且还可以防止在模型训练期间，非常大的差异会对特征发挥过大的影响。

StandardScaler 的构造器有两个参数：

- withMean：默认 false。用于缩放前求均值，这将建立一个密集的输出，所以不能在稀疏输入上正常工作，并将引发异常。

❑ withStd：默认 true。缩放数据到标准单位误差。

以下例子演示了 StandardScaler 的使用。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.StandardScaler
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
val scaler1 = new StandardScaler().fit(data.map(x => x.features))
val scaler2 = new StandardScaler(withMean = true, withStd = true).fit(data.map(x =>
   x.features))
// scaler3是与scaler2相同的模型，并且产生相同的转换
val scaler3 = new StandardScalerModel(scaler2.std, scaler2.mean)
// data1将是单位方差
val data1 = data.map(x => (x.label, scaler1.transform(x.features)))
// 不转换到密集向量，用0均值转换将在稀疏向量引起异常
// data2将是单位方差和0均值
val data2 = data.map(x => (x.label, scaler2.transform(Vectors.dense(x.features.toArray))))
```

11.14.4 正规化尺度

正规化尺度把样本划分为单位 L^p 范式，即维度。这是一种常见的对文本分类或集群化的操作。例如，两个 L^2 正规化 *TF-IDF* 向量的点积是这些向量的余弦近似值。

设二维空间内有两个向量 $\vec{a}$ 和 $\vec{b}$，它们的夹角为 $\theta(0 \leqslant \theta \leqslant \pi)$，则点积定义为以下实数：

$$\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos\theta$$

MLlib 提供 Normalizer 支持正规化，Normalizer 有以下构造参数：

p：正规化到 L^p 空间，默认为 2。

下面的例子演示了 Normalizer 的使用。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.Normalizer
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.util.MLUtils
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
val normalizer1 = new Normalizer()
val normalizer2 = new Normalizer(p = Double.PositiveInfinity)
// data1中的每个样本将被使用$L^2$范式正规化
val data1 = data.map(x => (x.label, normalizer1.transform(x.features)))
// data2中的每个样本将被使用$L^\infty$范式正规化
val data2 = data.map(x => (x.label, normalizer2.transform(x.features)))
```

11.14.5 卡方特征选择器

ChiSqSelector 用于卡方特征选择。它运转在具有分类特征的标签数据上。ChiSqSelector 对基于分类进行独立卡方测试的特征排序，并且过滤（选择）最接近标签的顶部特征。

ChiSqSelector 有以下构造器参数：

numTopFeatures：选择器将要过滤（选择）的顶部特征数量。

下边的例子演示了 ChiSqSelector 的使用。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.feature.ChiSqSelector
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// 因为ChiSqSelector需要分类特征，所以把数据离散到16个箱子
val discretizedData = data.map { lp =>
    LabeledPoint(lp.label, Vectors.dense(lp.features.toArray.map { x => x / 16 } ) )
}
// 创建一个选择50个特征的ChiSqSelector
val selector = new ChiSqSelector(50)
// 创建ChiSqSelector模型
val transformer = selector.fit(discretizedData)
// 从每个特征向量过滤出顶部的50 个特征
val filteredData = discretizedData.map { lp =>
    LabeledPoint(lp.label, transformer.transform(lp.features))
}
```

11.14.6 Hadamard 积

ElementwiseProduct 采用逐个相乘的方式，使用给定的权重与每个输入向量相乘。换言之，它采用一个标量乘法器扩展数据集的每一列。这表示 Hadamard 积对输入向量 v，使用转换向量 w，最终生成一个结果向量。Hadamard 积可由以下公式表示：

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \circ \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_N w_N \end{pmatrix}$$

ElementwiseProduct 的构造器参数为：

w：转换向量。

下面代码演示了 ElementwiseProduct 的使用。

```
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.feature.ElementwiseProduct
import org.apache.spark.mllib.linalg.Vectors
//创建一些矢量数据;也适用于稀疏向量
val data = sc.parallelize(Array(Vectors.dense(1.0, 2.0, 3.0), Vectors.dense(4.0,
   5.0, 6.0)))
val transformingVector = Vectors.dense(0.0, 1.0, 2.0)
val transformer = new ElementwiseProduct(transformingVector)
//批量变换和每行变换，得到相同的结果
val transformedData = transformer.transform(data)
val transformedData2 = data.map(x => transformer.transform(x))
```

11.15 频繁模式挖掘

分析大规模数据集的第一个步骤通常是挖掘频繁项目、项目集、亚序列或其他子结构，

这在数据挖掘中作为一个活跃的研究主题已多年了。其数学原理读者可以去维基百科[⊖]了解。MLlib 提供了频繁模式挖掘（frequent pattern mining）的并行实现——FP-growth 算法。

FP-growth

给定一个交易数据集，FP-growth 的第一步骤是计算项目的频率，并确定频繁项目。FP-growth 虽然与 Apriori 类算法有相同的设计目的，但是 FP-growth 的第二步使用后缀树（FP 树）结构对交易数据编码且不会显式生成候选集（生成候选集通常开销很大）。第二步之后，就可以从 FP 树中抽取频繁项目集。MLlib 中实现了 FP-growth 的平行版本，叫做 PFP。PFP 可以将 FP-growth 的工作分发到其他机器，比单机运行有更好的扩展性。

FPGrowth 有以下参数：

- minSupport：项目集被确定为频繁的最小数量。
- numPartitions：分发任务的数量。

下面的例子演示了 FPGrowth 的使用。

```
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.fpm.{FPGrowth, FPGrowthModel}
val transactions: RDD[Array[String]] = ...
val fpg = new FPGrowth()
    .setMinSupport(0.2)
    .setNumPartitions(10)
val model = fpg.run(transactions)
model.freqItemsets.collect().foreach { itemset =>
    println(itemset.items.mkString("[", ",", "]") + ", " + itemset.freq)
}
```

11.16 预言模型标记语言

预言模型标记语言（predictive model markup language，PMML）是一种基于 XML 的语言，它能够定义和共享应用程序之间的预测模型（predictive model）。

MLlib 支持将模型导出为预言模型标记语言，表 11-6 列出了 MLlib 模型导出为 PMML 的相应模型。

表 11-6 MLlib 模型导出为 PMML 的相应模型

MLlib 模型	PMML 模型
KMeansModel	ClusteringModel
LinearRegressionModel	RegressionModel (functionName="regression")
RidgeRegressionModel	RegressionModel (functionName="regression")

⊖ 关联规则的维基百科 https://en.wikipedia.org/wiki/Association_rule_learning。

（续）

MLlib 模型	PMML 模型
LassoModel	RegressionModel (functionName="regression")
SVMModel	RegressionModel (functionName="classification" normalizationMethod="none")
Binary LogisticRegressionModel	RegressionModel (functionName="classification" normalizationMethod="logit")

下面的例子演示了将 KMeansModel 导出为 PMML 格式。

```
import org.apache.spark.mllib.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors
val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)
// 导出为PMML
println("PMML Model:\n" + clusters.toPMML)
// 导出为PMML格式的字符串
clusters.toPMML
// 导出为PMML格式的本地文件
clusters.toPMML("/tmp/kmeans.xml")
// 导出为PMML格式的数据到分布式文件系统目录
clusters.toPMML(sc,"/tmp/kmeans")
// 导出为PMML格式到OutputStream
clusters.toPMML(System.out)
```

11.17　管道

Spark 1.2 增加了一个新包 spark.ml，目的是提供一套高层次的 API，帮助用户创建、调试机器学习的管道。spark.ml 的标准化 API 用于将多种机器学习算法组合到一个管道或工作流中。下面列出了 Spark ML API 的主要概念：

- ML Dataset：由 Hive table 或者数据源的数据构成的可容纳各种数据类型的 DataFrame 作为数据集。例如，数据集可以由不同的列分别存储文本、特征向量、标签和预测值。
- Transformer：是一种将 DataFrame 转换为另一个 DataFrame 的算法。例如，ML 模型是一个将特征 RDD 转换为预测值 RDD 的 Transformer。
- Estimator：适用于 DataFrame，并生成一个 Transformer。例如，学习算法是一个在数据集上训练并生成一个模型的 Estimator。
- Pipeline：链接多个 Transformer 和 Estimator，一起构成 ML 的工作流。
- Param：所有 Transformer 和 Estimator 用于指定参数的通用 API。

11.17.1 管道工作原理

机器学习中，运行一系列的算法去处理数据或者从数据学习的场景是很常见的。例如，一个简单的文本文档处理工作流可能包含以下阶段：

1）将文档文本切分成单词；

2）将文档的单词转换为数字化的特征向量；

3）使用特征向量和标签学习一个预测模型。

Spark ML 以一系列按序运行的 PipelineStage 组成的管道来表示这样的工作流。这一系列的 Stage 要么是 Transformer，要么是 Estimator。数据集通过管道中的每个 Stage 都会被修改。比如 Transformer 的 transform() 方法将在数据集上被调用，Estimator 的 fit() 方法被调用生成一个 Transformer，然后此 Transformer 的 transform() 方法也将在数据集上被调用。图 11-9 展示了简单文本文档工作流例子使用管道的处理流程。

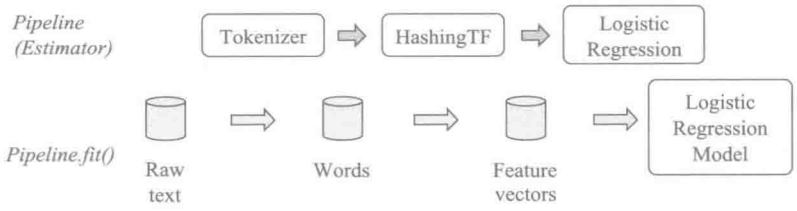


图 11-9 Estimator.fit 方法的管道处理流程

图 11-9 说明整个管道由 3 个 Stage 组成。Tokenizer 和 HashingTF 都是 Transformer，LogisticRegression 是 Estimator。每个圆柱体都说明它本身是一个 DataFrame。整个处理流程如下：

1）在由原生文本文档构成的原始数据集上应用 Pipeline.fit() 方法。

2）Tokenizer.transform() 将原生文本文档切分为单词，并向数据集增加单词列。

3）HashingTF.transform() 将单词列转换为特征向量，并向数据集增加向量列。

4）因为 LogisticRegression 是 Estimator，所以管道第一次调用 LogisticRegression.fit() 生成了 LogisticRegressionModel。如果管道中还有更多的 Stage，将会在传递数据集到下一个 Stage 之前在数据集上调用 LogisticRegressionModel 的 transform()。

管道本身是一个 Estimator。因此调用 Pipeline 的 fit() 方法最后生成了 PipelineModel，PipelineModel 也是一个 Transformer。这个 PipelineModel 会在测试时间使用，测试过程如图 11-10 所示。

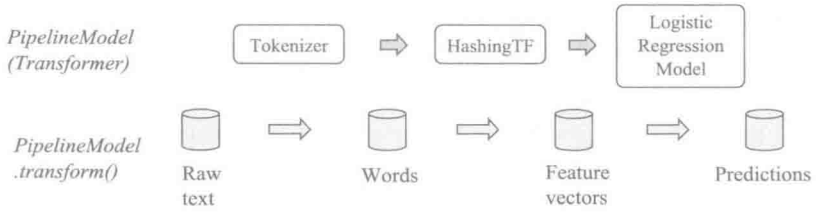


图 11-10 PipelineModel.transform 方法的管道处理流程

图 11-10 说明 PipelineModel 的测试过程与图 11-9 的管道有相同的 Stage 数量。但是图 11-9 的管道中的所有 Estimator 在此时都已经变为 Transformer。当在测试数据集上调用 PipelineModel 的 transform() 方法时，数据在管道中按序通过。每个 Stage 的 transform() 方法都会更新数据集，并将数据集传递给下一个 Stage。

刚才介绍的例子中，管道是线性的，即每个 stage 都使用由上一个 stage 生产的数据。只要数据流图构成了 DAG，它就有可能不是线性的。如果管道构成了 DAG，那么这些 Stage 就必须指定拓扑顺序。

11.17.2　管道 API 介绍

Spark ML 的 Transformer 和 Estimator 指定参数具有统一的 API。有两种方式指定参数：

- 给实例设置参数。例如，lr 是 LogisticRegression 的实例，可以调用 lr.setMaxIter(10) 使得调用 lr.fit() 时最多迭代 10 次。
- 传递 ParamMap 给 fit() 或者 transform() 方法。通过这种方式指定的参数值将会覆盖所有由 set 方式指定的参数值。

下面的例子演示了 Estimator、Transformer 和 Param 的使用。

```
val conf = new SparkConf().setAppName("SimpleParamsExample")
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
// 准备训练数据，Spark SQL可以将RDD[LabeledPoint]转换为DataFrame
val training = sc.parallelize(Seq(
    LabeledPoint(1.0, Vectors.dense(0.0, 1.1, 0.1)),
    LabeledPoint(0.0, Vectors.dense(2.0, 1.0, -1.0)),
    LabeledPoint(0.0, Vectors.dense(2.0, 1.3, 1.0)),
    LabeledPoint(1.0, Vectors.dense(0.0, 1.2, -0.5))))
// 创建一个LogisticRegression实例，此实例是一个Estimator
val lr = new LogisticRegression()
// 打印输出参数，文件和任何默认值
println("LogisticRegression parameters:\n" + lr.explainParams() + "\n")
// 使用setter方法设置参数
lr.setMaxIter(10).setRegParam(0.01)
// 训练LogisticRegression模型
val model1 = lr.fit(training.toDF)
println("Model 1 was fit using parameters: " + model1.parent.extractParamMap)
// 使用一个ParamMap指定参数
val paramMap = ParamMap(lr.maxIter -> 20)
paramMap.put(lr.maxIter, 30) // 覆盖原始的maxIter
paramMap.put(lr.regParam -> 0.1, lr.threshold -> 0.55) // 指定多个参数
// 合并ParamMap
val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability") // 改变输出列名
val paramMapCombined = paramMap ++ paramMap2
// 使用paramMapCombined训练模型
val model2 = lr.fit(training.toDF, paramMapCombined)
println("Model 2 was fit using parameters: " + model2.parent.extractParamMap)
// 准备测试数据
```

```
val test = sc.parallelize(Seq(
    LabeledPoint(1.0, Vectors.dense(-1.0, 1.5, 1.3)),
    LabeledPoint(0.0, Vectors.dense(3.0, 2.0, -0.1)),
    LabeledPoint(1.0, Vectors.dense(0.0, 2.2, -1.5))))
case class LabeledDocument(id: Long, text: String, label: Double)
case class Document(id: Long, text: String)
// 导入implicit可以将sqlContext 转换为DataFrame
val conf = new SparkConf().setAppName("SimpleTextClassificationPipeline")
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._
// 准备训练文档
val training = sc.parallelize(Seq(
    LabeledDocument(0L, "a b c d e spark", 1.0),
    LabeledDocument(1L, "b d", 0.0),
    LabeledDocument(2L, "spark f g h", 1.0),
    LabeledDocument(3L, "hadoop mapreduce", 0.0)))
// 配置ML pipeline, 它由三个stage组成: tokenizer, hashingTF和lr
val tokenizer = new Tokenizer()
    .setInputCol("text")
    .setOutputCol("words")
val hashingTF = new HashingTF()
    .setNumFeatures(1000)
    .setInputCol(tokenizer.getOutputCol)
    .setOutputCol("features")
val lr = new LogisticRegression()
    .setMaxIter(10)
    .setRegParam(0.01)
val pipeline = new Pipeline()
    .setStages(Array(tokenizer, hashingTF, lr))
```

下面的例子演示了管道的使用。

```
// LogisticRegression.transform仅使用'features'列，由于之前使用lr.probabilityCol对列
重命名，
//所以最后输出'myProbability'列，而不是'probability'
model2.transform(test.toDF)
    .select("features", "label", "myProbability", "prediction")
    .collect()
    .foreach { case Row(features: Vector, label: Double, prob: Vector,
        prediction: Double) =>println(s"($features, $label) -> prob=$prob,
        prediction=$prediction")
    }
sc.stop()
val model = pipeline.fit(training.toDF)
val test = sc.parallelize(Seq(
    Document(4L, "spark i j k"),
    Document(5L, "l m n"),
    Document(6L, "mapreduce spark"),
    Document(7L, "apache hadoop")))
// 在测试文档上做预测
model.transform(test.toDF)
    .select("id", "text", "probability", "prediction")
```

```
    .collect()
    .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double) =>
        println(s"($id, $text) --> prob=$prob, prediction=$prediction")
    }
sc.stop()
```

11.17.3 交叉验证

模型选择是 Spark ML 中很重要的课题。通过对整个管道的调整，而不是对管道中的每个元素的调整，促成对管道模型的选择。当前 spark.ml 使用 CrossValidator 支持模型选择。CrossValidator 本身携带一个 Estimator、一组 ParamMap 以及一个 Evaluator。CrossValidator 开始先将数据集划分为多组，每组都由训练数据集和测试数据集组成。例如，需要划分 3 组，那么 CrossValidator 将生成三个数据集对（训练，测试）。每一对都使用 2/3 的数据用于训练，1/3 的数据用于测试。CrossValidator 会迭代 ParamMap 的集合。对于每个 ParamMap，它都会训练给定的 Estimator 并使用给定的 Evaluator 计算。ParamMap 将会产出最佳的计算模型（对多个数据集对求平均），CrossValidator 最终使用这个最佳的 ParamMap 和整个数据集拟合 Estimator。下边的例子演示了 CrossValidator 的使用。使用 ParamGridBuilder 构造网格参数：hashingTF.numFeatures 有 3 个值，r.regParam 有 2 个值。这个网格将会有 3×2 = 6 个参数设置供 CrossValidator 选择。使用了 2 组数据集对，那么一共有 (3×2)×2 = 12 种不同的模型被训练。

使用 ParamGridBuilder 构建网格参数，会使 CrossValidator 的开销很大，要慎重使用。

下面的代码演示了交叉验证的使用。

```
case class LabeledDocument(id: Long, text: String, label: Double)
case class Document(id: Long, text: String)
val conf = new SparkConf().setAppName("CrossValidatorExample")
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._
// 准备LabeledDocument
val training = sc.parallelize(Seq(
    LabeledDocument(0L, "a b c d e spark", 1.0),
    LabeledDocument(1L, "b d", 0.0),
    LabeledDocument(2L, "spark f g h", 1.0),
    LabeledDocument(3L, "hadoop mapreduce", 0.0),
    LabeledDocument(4L, "b spark who", 1.0),
    LabeledDocument(5L, "g d a y", 0.0),
    LabeledDocument(6L, "spark fly", 1.0),
    LabeledDocument(7L, "was mapreduce", 0.0),
    LabeledDocument(8L, "e spark program", 1.0),
    LabeledDocument(9L, "a e c l", 0.0),
    LabeledDocument(10L, "spark compile", 1.0),
    LabeledDocument(11L, "hadoop software", 0.0)))
val tokenizer = new Tokenizer()
```

```
    .setInputCol("text")
    .setOutputCol("words")
val hashingTF = new HashingTF()
    .setInputCol(tokenizer.getOutputCol)
    .setOutputCol("features")
val lr = new LogisticRegression()
    .setMaxIter(10)
val pipeline = new Pipeline()
    .setStages(Array(tokenizer, hashingTF, lr))
val crossval = new CrossValidator()
    .setEstimator(pipeline)
    .setEvaluator(new BinaryClassificationEvaluator)
// 使用ParamGridBuilder构造网格参数：hashingTF.numFeatures有3个值，r.regParam有2个值
// 这个网格将会有3 x 2 = 6 个参数设置供CrossValidator 选择
val paramGrid = new ParamGridBuilder()
    .addGrid(hashingTF.numFeatures, Array(10, 100, 1000))
    .addGrid(lr.regParam, Array(0.1, 0.01))
    .build()
crossval.setEstimatorParamMaps(paramGrid)
crossval.setNumFolds(2) // 实际会使用3+个
// 运行交叉验证并选择最佳的参数集合
val cvModel = crossval.fit(training.toDF)

val test = sc.parallelize(Seq(
    Document(4L, "spark i j k"),
    Document(5L, "l m n"),
    Document(6L, "mapreduce spark"),
    Document(7L, "apache hadoop")))
// cvModel使用找到的最佳模型(lrModel)，在测试文档上做预测
cvModel.transform(test.toDF)
    .select("id", "text", "probability", "prediction")
    .collect()
    .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double)
        =>println(s"($id, $text) --> prob=$prob, prediction=$prediction")
}
sc.stop()
```

11.18 小结

Spark MLlib 与 Spark ML 共同搭建了 Spark 目前的机器学习平台。Spark MLlib 作为机器学习的仓库，收纳了统计、分类、回归、决策树、过滤、聚类、特征提取与转型、数据挖掘等领域的大量算法实现。Spark MLlib 提供的 API 都容易使用，大大降低了用户使用这些复杂算法的门槛。Spark ML 作为高层次的 API，提供了机器学习的工作流处理能力，并提供交叉验证帮助开发者选择最佳的计算模型。

附录 A Appendix A

Utils

Utils 是 Spark 中最常用的工具类之一，如果不关心其实现也不会对阅读 Spark 源码有太大影响。下面将逐个介绍 Utils 提供的方法。

1. localHostName

功能描述：获取本地机器名。

```
def localHostName(): String = {
    customHostname.getOrElse(localIpAddressHostname)
}
```

2. getDefaultPropertiesFile

功能描述：获取默认的 Spark 属性文件。

```
def getDefaultPropertiesFile(env: Map[String, String] = sys.env): String = {
    env.get("SPARK_CONF_DIR")
        .orElse(env.get("SPARK_HOME").map { t => s"$t${File.separator}conf" })
        .map { t => new File(s"$t${File.separator}spark-defaults.conf")}
        .filter(_.isFile)
        .map(_.getAbsolutePath)
        .orNull
}
```

3. loadDefaultSparkProperties

功能描述：加载指定文件中的 Spark 属性，如果没有指定文件，则加载默认 Spark 属性文件的属性。

```
def loadDefaultSparkProperties(conf: SparkConf, filePath: String = null): String = {
    val path = Option(filePath).getOrElse(getDefaultPropertiesFile())
    Option(path).foreach { confFile =>
```

```
            getPropertiesFromFile(confFile).filter { case (k, v) =>
                k.startsWith("spark.")
            }.foreach { case (k, v) =>
              conf.setIfMissing(k, v)
              sys.props.getOrElseUpdate(k, v)
            }
        }
        path
    }
```

4. getCallSite

功能描述：获取当前 SparkContext 的当前调用栈，将栈里最靠近栈底的属于 Spark 或者 Scala 核心的类压入 callStack 的栈顶，并将此类的方法存入 lastSparkMethod；将栈里最靠近栈顶的用户类放入 callStack，将此类的行号存入 firstUserLine，类名存入 firstUserFile，最终返回的样例类 CallSite 存储了最短栈和长度默认为 20 的最长栈的样例类。在 JavaWordCount 例子中，获得的数据如下：

- 最短栈：JavaSparkContext at JavaWordCount.java:44；
- 最长栈：org.apache.spark.api.java.JavaSparkContext.<init>(JavaSparkContext.scala:61) org.apache.spark.examples.JavaWordCount.main(JavaWordCount.java:44)。

```
def getCallSite(skipClass: String => Boolean = coreExclusionFunction): CallSite = {
    val trace = Thread.currentThread.getStackTrace().filterNot { ste: StackTrace
        Element =>
      ste == null || ste.getMethodName == null || ste.getMethodName.contains("getStackTrace")
    }
    var lastSparkMethod = "<unknown>"
    var firstUserFile = "<unknown>"
    var firstUserLine = 0
    var insideSpark = true
    var callStack = new ArrayBuffer[String]() :+ "<unknown>"

    for (el <- trace) {
        if (insideSpark) {
            if (skipClass(el.getClassName)) {
                lastSparkMethod = if (el.getMethodName == "<init>") {
                    el.getClassName.substring(el.getClassName.lastIndexOf('.') + 1)
                } else {
                  el.getMethodName
                }
                callStack(0) = el.toString // Put last Spark method on top of the stack trace.
            } else {
                firstUserLine = el.getLineNumber
                firstUserFile = el.getFileName
                callStack += el.toString
                insideSpark = false
            }
        } else {
            callStack += el.toString
        }
```

```
    }
    val callStackDepth = System.getProperty("spark.callstack.depth", "20").toInt
    CallSite(
        shortForm = s"$lastSparkMethod at $firstUserFile:$firstUserLine",
        longForm = callStack.take(callStackDepth).mkString("\n"))
}
```

5. startServiceOnPort

功能描述：Scala 跟其他脚本语言一样，函数也可以传递，此方法正是通过回调 startService 这个函数来启动服务，并最终返回 startService 返回的 service 地址及端口。如果启动过程有异常，还会多次重试，直到达到 maxRetries 表示的最大次数。

```
def startServiceOnPort[T](
        startPort: Int,
        startService: Int => (T, Int),
        conf: SparkConf,
        serviceName: String = ""): (T, Int) = {
    require(startPort == 0 || (1024 <= startPort && startPort < 65536),
        "startPort should be between 1024 and 65535 (inclusive), or 0 for a random
            free port.")
    val serviceString = if (serviceName.isEmpty) "" else s" '$serviceName'"
    val maxRetries = portMaxRetries(conf)
    for (offset <- 0 to maxRetries) {
        val tryPort = if (startPort == 0) {
            startPort
        } else {
            ((startPort + offset - 1024) % (65536 - 1024)) + 1024
        }
        try {
            val (service, port) = startService(tryPort)
            logInfo(s"Successfully started service$serviceString on port $port.")
            return (service, port)
        } catch {
          case e: Exception if isBindCollision(e) =>
              if (offset >= maxRetries) {
                  val exceptionMessage =
                      s"${e.getMessage}: Service$serviceString failed after
                          $maxRetries retries!"
                  val exception = new BindException(exceptionMessage)
                  exception.setStackTrace(e.getStackTrace)
                  throw exception
              }
              logWarning(s"Service$serviceString could not bind on port $tryPort. " +
                  s"Attempting port ${tryPort + 1}.")
          }
    }
    throw new SparkException(s"Failed to start service$serviceString on port
        $startPort")
}
```

6. createDirectory

功能描述：用 spark+UUID 的方式创建临时文件目录，如果创建失败会多次重试，最多重试 10 次。

```
def createDirectory(root: String, namePrefix: String = "spark"): File = {
    var attempts = 0
    val maxAttempts = MAX_DIR_CREATION_ATTEMPTS
    var dir: File = null
    while (dir == null) {
        attempts += 1
        if (attempts > maxAttempts) {
            throw new IOException("Failed to create a temp directory (under " +
               root + ") after " +maxAttempts + " attempts!")
        }
        try {
            dir = new File(root, "spark-" + UUID.randomUUID.toString)
            if (dir.exists() || !dir.mkdirs()) {
                dir = null
            }
        } catch { case e: SecurityException => dir = null; }
    }

    dir
}
```

7. getOrCreateLocalRootDirs

功能描述：根据 spark.local.dir 的配置，作为本地文件的根目录，在创建一、二级目录之前要确保根目录是存在的。然后调用 createDirectory 创建一级目录。

```
private[spark] def getOrCreateLocalRootDirs(conf: SparkConf): Array[String] = {
    if (isRunningInYarnContainer(conf)) {
        getYarnLocalDirs(conf).split(",")
    } else {
      Option(conf.getenv("SPARK_LOCAL_DIRS"))
          .getOrElse(conf.get("spark.local.dir", System.getProperty("java.
              io.tmpdir")))
          .split(",")
          .flatMap { root =>
            try {
               val rootDir = new File(root)
               if (rootDir.exists || rootDir.mkdirs()) {
                   val dir = createDirectory(root)
                   chmod700(dir)
                   Some(dir.getAbsolutePath)
               } else {
                  logError(s"Failed to create dir in $root. Ignoring this directory.")
                  None
               }
            } catch {
              case e: IOException =>
              logError(s"Failed to create local root dir in $root. Ignoring this
                  directory.")
```

```
                None
            }
        }
        .toArray
    }
}
```

8. getLocalDir

功能描述：查询 Spark 本地文件的一级目录。

```
def getLocalDir(conf: SparkConf): String = {
    getOrCreateLocalRootDirs(conf)(0)
}
```

9. createTempDir

功能描述：在 Spark 一级目录下创建临时目录，并将目录注册到 shutdownDeletePaths：scala.collection.mutable.HashSet[String] 中。

```
def createTempDir(
        root: String = System.getProperty("java.io.tmpdir"),
        namePrefix: String = "spark"): File = {
    val dir = createDirectory(root, namePrefix)
    registerShutdownDeleteDir(dir)
    dir
}
```

10. registerShutdownDeleteDir

功能描述：将目录注册到 shutdownDeletePaths：scala.collection.mutable.HashSet[String] 中，以便在进程退出时删除。

```
def registerShutdownDeleteDir(file: File) {
    val absolutePath = file.getAbsolutePath()
    shutdownDeletePaths.synchronized {
        shutdownDeletePaths += absolutePath
    }
}
```

11. hasRootAsShutdownDeleteDir

功能描述：判断文件是否匹配关闭时要删除的文件及目录，shutdownDeletePaths：scala.collection.mutable.HashSet[String] 存储在进程关闭时要删除的文件及目录。

```
def hasRootAsShutdownDeleteDir(file: File): Boolean = {
    val absolutePath = file.getAbsolutePath()
    val retval = shutdownDeletePaths.synchronized {
        shutdownDeletePaths.exists { path =>
            !absolutePath.equals(path) && absolutePath.startsWith(path)
        }
    }
    if (retval) {
        logInfo("path = " + file + ", already present as root for deletion.")
    }
```

```
    retval
}
```

12. deleteRecursively

功能描述：用于删除文件或者删除目录及其子目录、子文件，并且从 shutdownDeletePaths：scala.collection.mutable.HashSet[String] 中移除此文件或目录。

```
def deleteRecursively(file: File) {
    if (file != null) {
        try {
            if (file.isDirectory && !isSymlink(file)) {
                var savedIOException: IOException = null
                for (child <- listFilesSafely(file)) {
                    try {
                        deleteRecursively(child)
                    } catch {
                      case ioe: IOException => savedIOException = ioe
                    }
                }
                if (savedIOException != null) {
                    throw savedIOException
                }
                shutdownDeletePaths.synchronized {
                    shutdownDeletePaths.remove(file.getAbsolutePath)
                }
            }
        } finally {
            if (!file.delete()) {
                if (file.exists()) {
                    throw new IOException("Failed to delete: " + file.getAbsolutePath)
                }
            }
        }
    }
}
```

13. getSparkClassLoader

功能描述：获取加载当前 class 的 ClassLoader。

```
def getSparkClassLoader = getClass.getClassLoader
```

14. getContextOrSparkClassLoader

功能描述：用于获取线程上下文的 ClassLoader，没有设置时获取加载 Spark 的 ClassLoader。

```
def getContextOrSparkClassLoader =
    Option(Thread.currentThread().getContextClassLoader).getOrElse(getSparkClassLoader)
```

15. newDaemonCachedThreadPool

功能描述：使用 Executors.newCachedThreadPool 创建的缓存线程池。

```
def newDaemonCachedThreadPool(prefix: String): ThreadPoolExecutor = {
    val threadFactory = namedThreadFactory(prefix)
```

```
    Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
}
```

16. doFetchFile

功能描述：使用 URLConnection 通过 HTTP、HTTPS、FIP 等协议下载文件。

```
private def doFetchFile(url: String, targetDir: File, filename: String, conf:
    SparkConf,securityMgr: SecurityManager, hadoopConf: Configuration) {
    val tempFile = File.createTempFile("fetchFileTemp", null, new File(targetDir.
       getAbsolutePath))
    val targetFile = new File(targetDir, filename)
    val uri = new URI(url)
    val fileOverwrite = conf.getBoolean("spark.files.overwrite", defaultValue =
       false)
    Option(uri.getScheme).getOrElse("file") match {
        case "http" | "https" | "ftp" =>
            logInfo("Fetching " + url + " to " + tempFile)
            var uc: URLConnection = null
            if (securityMgr.isAuthenticationEnabled()) {
                logDebug("fetchFile with security enabled")
                val newuri = constructURIForAuthentication(uri, securityMgr)
                uc = newuri.toURL().openConnection()
                uc.setAllowUserInteraction(false)
            } else {
                logDebug("fetchFile not using security")
                uc = new URL(url).openConnection()
            }
                val timeout = conf.getInt("spark.files.fetchTimeout", 60) * 1000
                uc.setConnectTimeout(timeout)
                uc.setReadTimeout(timeout)
                uc.connect()
                val in = uc.getInputStream()
                downloadFile(url, in, tempFile, targetFile, fileOverwrite)
            case "file" =>
                val sourceFile = if (uri.isAbsolute) new File(uri) else new
                   File(url)
                copyFile(url, sourceFile, targetFile, fileOverwrite)
            case _ =>
                val fs = getHadoopFileSystem(uri, hadoopConf)
                val in = fs.open(new Path(uri))
                downloadFile(url, in, tempFile, targetFile, fileOverwrite)
    }
}
```

17. fetchFile

功能描述：如果文件在本地有缓存，则从本地获取，否则通过 HTTP、HTTPS、FTP 等协议远程下载。最后对 .tar、.tar.gz 等格式的文件解压缩后，调用 shell 命令行的 chmod 命令给文件增加 a+x 的权限。

```
def fetchFile(
        url: String,
        targetDir: File,
```

```
        conf: SparkConf,
        securityMgr: SecurityManager,
        hadoopConf: Configuration,
        timestamp: Long,
        useCache: Boolean) {
    val fileName = url.split("/").last
    val targetFile = new File(targetDir, fileName)
    val fetchCacheEnabled = conf.getBoolean("spark.files.useFetchCache",
      defaultValue = true)
    if (useCache && fetchCacheEnabled) {
        val cachedFileName = s"${url.hashCode}${timestamp}_cache"
        val lockFileName = s"${url.hashCode}${timestamp}_lock"
        val localDir = new File(getLocalDir(conf))
        val lockFile = new File(localDir, lockFileName)
        val raf = new RandomAccessFile(lockFile, "rw")
        val lock = raf.getChannel().lock()
        val cachedFile = new File(localDir, cachedFileName)
        try {
            if (!cachedFile.exists()) {
                doFetchFile(url, localDir, cachedFileName, conf, securityMgr,
                  hadoopConf)
            }
        } finally {
            lock.release()
        }
        copyFile(
            url,
            cachedFile,
            targetFile,
            conf.getBoolean("spark.files.overwrite", false)
        )
    } else {
        doFetchFile(url, targetDir, fileName, conf, securityMgr, hadoopConf)
    }
    if (fileName.endsWith(".tar.gz") || fileName.endsWith(".tgz")) {
        logInfo("Untarring " + fileName)
        Utils.execute(Seq("tar", "-xzf", fileName), targetDir)
    } else if (fileName.endsWith(".tar")) {
        logInfo("Untarring " + fileName)
        Utils.execute(Seq("tar", "-xf", fileName), targetDir)
    }
    FileUtil.chmod(targetFile.getAbsolutePath, "a+x")
}
```

18. executeAndGetOutput

功能描述：执行一条 command 命令，并且获取它的输出。调用 stdoutThread 的 join 方法，让当前线程等待 stdoutThread 执行完成。

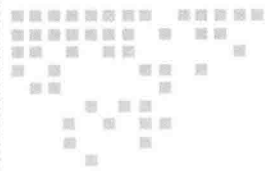
```
def executeAndGetOutput(
        command: Seq[String],
        workingDir: File = new File("."),
        extraEnvironment: Map[String, String] = Map.empty): String = {
```

```
    val builder = new ProcessBuilder(command: _*)
        .directory(workingDir)
    val environment = builder.environment()
    for ((key, value) <- extraEnvironment) {
        environment.put(key, value)
    }
    val process = builder.start()
    new Thread("read stderr for " + command(0)) {
        override def run() {
            for (line <- Source.fromInputStream(process.getErrorStream).getLines())
                {
                System.err.println(line)
            }
        }
    }.start()
    val output = new StringBuffer
    val stdoutThread = new Thread("read stdout for " + command(0)) {
        override def run() {
            for (line <- Source.fromInputStream(process.getInputStream).getLines()) {
                output.append(line)
            }
        }
    }
    stdoutThread.start()
    val exitCode = process.waitFor()
    stdoutThread.join()   // Wait for it to finish reading output
    if (exitCode != 0) {
        logError(s"Process $command exited with code $exitCode: $output")
        throw new SparkException(s"Process $command exited with code $exitCode")
    }
    output.toString
}
```

19. memoryStringToMb

功能描述：将内存大小字符串转换为以 MB 为单位的整型值。

```
def memoryStringToMb(str: String): Int = {
    val lower = str.toLowerCase
    if (lower.endsWith("k")) {
       (lower.substring(0, lower.length-1).toLong / 1024).toInt
    } else if (lower.endsWith("m")) {
      lower.substring(0, lower.length-1).toInt
    } else if (lower.endsWith("g")) {
      lower.substring(0, lower.length-1).toInt * 1024
    } else if (lower.endsWith("t")) {
      lower.substring(0, lower.length-1).toInt * 1024 * 1024
    } else {// no suffix, so it's just a number in bytes
      (lower.toLong / 1024 / 1024).toInt
    }
}
```



Appendix B 附录 B

Akka

1. Akka 简介

Akka 是一款提供了用于构建高并发的、分布式的、可伸缩的、基于 Java 虚拟机的消息驱动应用的工具集和运行时环境。从下面 Akka 官网提供的一段代码示例，可以看出 Akka 并发编程的简约。

```
case class Greeting(who: String)
class GreetingActor extends Actor with ActorLogging {
    def receive = {
        case Greeting(who) ⇒ log.info("Hello " + who)
    }
}
val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
greeter ! Greeting("Charlie Parker")
```

Akka 提供了分布式的框架，意味着用户不需要考虑如何实现分布式部署。Akka 官网提供了下面的示例演示如何获取远程 Actor 的引用。

```
// config on all machines
akka {
    actor {
        provider = akka.remote.RemoteActorRefProvider
        deployment {
            /greeter {
             remote = akka.tcp://MySystem@machine1:2552
        }
        }
    }
}
```

```
// ------------------------------
// define the greeting actor and the greeting message
case class Greeting(who: String) extends Serializable
class GreetingActor extends Actor with ActorLogging {
    def receive = {
        case Greeting(who) ⇒ log.info("Hello " + who)
}
}
// ------------------------------
// on machine 1: empty system, target for deployment from machine 2
val system = ActorSystem("MySystem")
// ------------------------------
// on machine 2: Remote Deployment - deploying on machine1
val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
// ------------------------------
// on machine 3: Remote Lookup (logical home of "greeter" is machine2, remote
    deployment is transparent)
val system = ActorSystem("MySystem")
val greeter = system.actorSelection("akka.tcp://MySystem@machine2:2552/user/greeter")
greeter ! Greeting("Sonny Rollins")
```

Actor 之间最终会构成一棵树，作为父亲的 Actor 应当对所有儿子的异常失败进行处理（监管）。Akka 给出了简单的示例，代码如下。

```
class Supervisor extends Actor {
    override val supervisorStrategy =
        OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
            case _: ArithmeticException      ⇒ Resume
            case _: NullPointerException     ⇒ Restart
            case _: Exception                ⇒ Escalate
        }
    val worker = context.actorOf(Props[Worker])
    def receive = {
        case n: Int => worker forward n
    }
}
```

Akka 的更多信息请访问官方网站：http://akka.io/。

2. AkkaUtils

AkkaUtils 是 Spark 对 Akka 相关 API 的又一层封装，这里对其常用的功能进行介绍。

（1）doCreateActorSystem

功能描述：创建 ActorSystem。

```
private def doCreateActorSystem(
        name: String,
        host: String,
        port: Int,
        conf: SparkConf,
        securityManager: SecurityManager): (ActorSystem, Int) = {
```

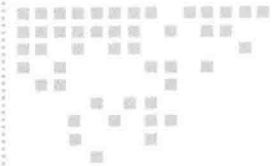
```
val akkaThreads   = conf.getInt("spark.akka.threads", 4)
val akkaBatchSize = conf.getInt("spark.akka.batchSize", 15)
val akkaTimeout = conf.getInt("spark.akka.timeout", 100)
val akkaFrameSize = maxFrameSizeBytes(conf)
val akkaLogLifecycleEvents = conf.getBoolean("spark.akka.logLifecycleEvents",
  false)
val lifecycleEvents = if (akkaLogLifecycleEvents) "on" else "off"
if (!akkaLogLifecycleEvents) {
    Option(Logger.getLogger("akka.remote.EndpointWriter")).map(l => l.setLevel
       (Level.FATAL))
}
val logAkkaConfig = if (conf.getBoolean("spark.akka.logAkkaConfig", false))
  "on" else "off"
val akkaHeartBeatPauses = conf.getInt("spark.akka.heartbeat.pauses", 6000)
val akkaFailureDetector =
    conf.getDouble("spark.akka.failure-detector.threshold", 300.0)
val akkaHeartBeatInterval = conf.getInt("spark.akka.heartbeat.interval", 1000)
val secretKey = securityManager.getSecretKey()
val isAuthOn = securityManager.isAuthenticationEnabled()
if (isAuthOn && secretKey == null) {
    throw new Exception("Secret key is null with authentication on")
}
val requireCookie = if (isAuthOn) "on" else "off"
val secureCookie = if (isAuthOn) secretKey else ""
logDebug("In createActorSystem, requireCookie is: " + requireCookie)
val akkaConf = ConfigFactory.parseMap(conf.getAkkaConf.toMap[String, String]).
  withFallback(
  ConfigFactory.parseString(
  s"""
  |akka.daemonic = on
  |akka.loggers = [""akka.event.slf4j.Slf4jLogger""]
  |akka.stdout-loglevel = "ERROR"
  |akka.jvm-exit-on-fatal-error = off
  |akka.remote.require-cookie = "$requireCookie"
  |akka.remote.secure-cookie = "$secureCookie"
  |akka.remote.transport-failure-detector.heartbeat-interval = $akkaHeartBeat-
     Interval s
  |akka.remote.transport-failure-detector.acceptable-heartbeat-pause =
     $akkaHeartBeatPauses s
  |akka.remote.transport-failure-detector.threshold = $akkaFailureDetector
  |akka.actor.provider = "akka.remote.RemoteActorRefProvider"
  |akka.remote.netty.tcp.transport-class = "akka.remote.transport.netty.
     NettyTransport"
  |akka.remote.netty.tcp.hostname = "$host"
  |akka.remote.netty.tcp.port = $port
  |akka.remote.netty.tcp.tcp-nodelay = on
  |akka.remote.netty.tcp.connection-timeout = $akkaTimeout s
  |akka.remote.netty.tcp.maximum-frame-size = ${akkaFrameSize}B
  |akka.remote.netty.tcp.execution-pool-size = $akkaThreads
  |akka.actor.default-dispatcher.throughput = $akkaBatchSize
  |akka.log-config-on-start = $logAkkaConfig
  |akka.remote.log-remote-lifecycle-events = $lifecycleEvents
  |akka.log-dead-letters = $lifecycleEvents
```

```
        |akka.log-dead-letters-during-shutdown = $lifecycleEvents
        """.stripMargin))
    val actorSystem = ActorSystem(name, akkaConf)
    val provider = actorSystem.asInstanceOf[ExtendedActorSystem].provider
    val boundPort = provider.getDefaultAddress.port.get
    (actorSystem, boundPort)
}
```

（2）makeDriverRef

功能描述：从远端 ActorSystem 中查找已经注册的某个 Actor。

```
def makeDriverRef(name: String, conf: SparkConf, actorSystem: ActorSystem):
   ActorRef = {
    val driverActorSystemName = SparkEnv.driverActorSystemName
    val driverHost: String = conf.get("spark.driver.host", "localhost")
    val driverPort: Int = conf.getInt("spark.driver.port", 7077)
    Utils.checkHost(driverHost, "Expected hostname")
    val url = s"akka.tcp://$driverActorSystemName@$driverHost:$driverPort/
       user/$name"
    val timeout = AkkaUtils.lookupTimeout(conf)
    logInfo(s"Connecting to $name: $url")
    Await.result(actorSystem.actorSelection(url).resolveOne(timeout), timeout)
}
```



Appendix C 附录 C

Jetty

1. Jetty 简介

Jetty 是一个开源的，以 Java 作为开发语言的 servlet 容器。它的 API 以一组 jar 包的形式发布。Jetty 容器可以实例化成一个对象，因而迅速为一些独立运行的 Java 应用提供网络和 web 服务。要为 Jetty 创建 servlet，就涉及 ServletContextHandler 的 API 使用。示例代码如下。

```
class HelloServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private String msg = "Hello World!";

    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        response.setContentType("text/html");
        response.setStatus(HttpServletResponse.SC_OK);
        response.getWriter().println("<h1>" + msg + "</h1>");
        response.getWriter().println("session=" + request.getSession(true).getId());
    }
}
public static void main(String[] args) throws Exception {
    Server server = new Server(8080);
    ServletContextHandler context = new ServletContextHandler();
    context.setContextPath("/");
    server.setHandler(context);
    // http://localhost:8080/hello
    context.addServlet(new ServletHolder(new HelloServlet()), "/hello");
    server.start();
    server.join();
}
```

如果想更深入了解 Jetty，请访问官网 http://www.eclipse.org/jetty/。

2. JettyUtils

JettyUtils是Spark对于Jetty相关API的又一层封装，这里对其中一些主要方法进行介绍。

（1）createServletHandler

功能描述：创建以给定路径为前缀的请求的响应处理。处理步骤如下：

1）调用createServlet，生成javax.servlet.http.HttpServlet的匿名内部类实例。此实例处理请求实际是使用servletParams的responder：Responder，此Responder类型发生隐式转换，会转换为用户传入的函数参数。

2）调用重载的createServletHandler方法，生成org.eclipse.jetty.servlet.ServletHolder，并最终生成ServletContextHandler。

createServletHandler的实现如下。

```
def createServletHandler[T <% AnyRef](
        path: String,
        servletParams: ServletParams[T],
        securityMgr: SecurityManager,
        basePath: String = ""): ServletContextHandler = {
    createServletHandler(path, createServlet(servletParams, securityMgr),
       basePath)
}
type Responder[T] = HttpServletRequest => T

class ServletParams[T <% AnyRef](val responder: Responder[T],
    val contentType: String,
    val extractFn: T => String = (in: Any) => in.toString) {}

    def createServlet[T <% AnyRef](
        servletParams: ServletParams[T],
        securityMgr: SecurityManager): HttpServlet = {
    new HttpServlet {
        override def doGet(request: HttpServletRequest, response: HttpServletResponse) {
            if (securityMgr.checkUIViewPermissions(request.getRemoteUser)) {
                response.setContentType("%s;charset=utf-8".format(servletParams.
                    contentType))
                response.setStatus(HttpServletResponse.SC_OK)
                val result = servletParams.responder(request)
                response.setHeader("Cache-Control", "no-cache, no-store, must-
                   revalidate")
                response.getWriter.println(servletParams.extractFn(result))
            } else {
                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED)
                response.setHeader("Cache-Control", "no-cache, no-store, must-
                   revalidate")
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
                   "User is not authorized to access this page.")
            }
        }
    }
```

```
}

def createServletHandler(
        path: String,
        servlet: HttpServlet,
        basePath: String): ServletContextHandler = {
    val prefixedPath = attachPrefix(basePath, path)
    val contextHandler = new ServletContextHandler
    val holder = new ServletHolder(servlet)
    contextHandler.setContextPath(prefixedPath)
    contextHandler.addServlet(holder, "/")
    contextHandler
}
```

（2）startJettyServer

功能描述：创建以给定路径为前缀的请求的响应处理。处理步骤如下：

1）将 SparkUI 中的全部 handler 加入 ContextHandlerCollection。

2）如果使用配置 spark.ui.filters 指定了 filter，则给所有 handler 增加 filter。

3）调用 Utils 的方法 startServiceOnPort，最终回调函数 connect。

startJettyServer 的实现如下。

```
def startJettyServer(
        hostName: String,
        port: Int,
        handlers: Seq[ServletContextHandler],
        conf: SparkConf,
        serverName: String = ""): ServerInfo = {
    val collection = new ContextHandlerCollection
    collection.setHandlers(handlers.toArray)
    addFilters(handlers, conf)
    def connect(currentPort: Int): (Server, Int) = {
        val server = new Server(new InetSocketAddress(hostName, currentPort))
        val pool = new QueuedThreadPool
        pool.setDaemon(true)
        server.setThreadPool(pool)
        server.setHandler(collection)
        try {
            server.start()
            (server, server.getConnectors.head.getLocalPort)
        } catch {
            case e: Exception =>
            server.stop()
            pool.stop()
            throw e
        }
    }
    val (server, boundPort) = Utils.startServiceOnPort[Server](port, connect,
       conf, serverName)
    ServerInfo(server, boundPort, collection)
}
```

附录 D Appendix D

Metrics

1. Metrics 简介

Metrics[⊖]是 codahale 提供的第三方测量仓库。Metrics 作为一款监控指标的测量类库，可以为第三方库提供辅助统计信息，还可以将测量数据发送给 Ganglia 和 Graphite，以提供图形化的监控。

Metrics 提供了 Gauge、Counter、Meter、Histogram、Timer 等测量工具类以及健康检查（health check）功能。

2. MetricRegistry

MetricRegistry 是 Metrics 提供的测量容器，这里先列出 MetricRegistry 的主要结构。

```
public class MetricRegistry implements MetricSet {
    private final ConcurrentMap<String, Metric> metrics;
    private final List<MetricRegistryListener> listeners;
}
```

从上面代码可以看出，MetricRegistry 中会缓存各种测量和监听器。下面对 MetricRegistry 中的一些方法进行介绍。

（1）notifyListenerOfAddedMetric

功能描述：当有新的 Metric 添加到 ConcurrentMap < String, Metric > metrics 时，调用此方法。根据 Metric 的子接口的不同，调用不同方法。例如，Gauge 则调用监听器的 onGaugeAdded；Counter 则调用监听器的 onCounterAdded；Histogram 则调用监听器的 onHistogramAdded。

⊖ Metrics 是测量框架的名称，Metric 是其中一个类。

```
private void notifyListenerOfAddedMetric(MetricRegistryListener listener, Metric
    metric, String name) {
        if (metric instanceof Gauge) {
            listener.onGaugeAdded(name, (Gauge<?>) metric);
        } else if (metric instanceof Counter) {
            listener.onCounterAdded(name, (Counter) metric);
        } else if (metric instanceof Histogram) {
            listener.onHistogramAdded(name, (Histogram) metric);
        } else if (metric instanceof Meter) {
            listener.onMeterAdded(name, (Meter) metric);
        } else if (metric instanceof Timer) {
            listener.onTimerAdded(name, (Timer) metric);
        } else {
            throw new IllegalArgumentException("Unknown metric type: " + metric.
                getClass());
        }
    }
```

（2）onMetricAdded

功能描述：当有新的 Metric 添加到 ConcurrentMap < String, Metric > metrics 时，调用此方法。遍历调用监听器缓存 List < MetricRegistryListener > listeners 中的所有监听器，调用 notifyListenerOfAddedMetric。

```
private void onMetricAdded(String name, Metric metric) {
    for (MetricRegistryListener listener : listeners) {
        notifyListenerOfAddedMetric(listener, metric, name);
    }
}
```

（3）register

功能描述：如果 metric 的类型是 Metric 并且 Metric 缓存 metrics：concurrent Map < String Metrics 中还没有此 Metric，则将它添加到 metrics；如果 Metric 的类型是 MetricSet，则 MetricSet 中包含的所有新的 Metric 添加到缓存 ConcurrentMap < String, Metric > metrics。以上添加过程都伴随 onMetricAdded 的调用。

```
public <T extends Metric> T register(String name, T metric) throws IllegalArgument-
   Exception {
        if (metric instanceof MetricSet) {
            registerAll(name, (MetricSet) metric);
        } else {
            final Metric existing = metrics.putIfAbsent(name, metric);
            if (existing == null) {
                onMetricAdded(name, metric);
            } else {
                throw new IllegalArgumentException("A metric named " + name + "
                   already exists");
            }
        }
        return metric;
    }
```

```
    private void registerAll(String prefix, MetricSet metrics) throws Illegal-
        ArgumentException {
          for (Map.Entry<String, Metric> entry : metrics.getMetrics().entrySet()) {
              if (entry.getValue() instanceof MetricSet) {
                  registerAll(name(prefix, entry.getKey()), (MetricSet) entry.
                    getValue());
              } else {
                  register(name(prefix, entry.getKey()), entry.getValue());
              }
          }
}
```

Appendix E 附录 E

Hadoop word count

这里主要演示 Hadoop1.0 版本中的 word count 例子，用于和 Spark 中的实现对比。

```
package org.apache.hadoop.examples;
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
public class WordCount {
    public static class Map extends MapReduceBase implements
            Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
```

```
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
public static class Reduce extends MapReduceBase implements
        Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
       OutputCollector<Text, IntWritable> output, Reporter reporter)
           throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}
}
```

Appendix F 附录 F

CommandUtils

CommandUtils 是 Spark 中最常用的工具类之一，所以有必要在这里作介绍。如果不太关心其实现，也不影响对 Spark 源码的阅读和原理的学习。我们要介绍的方法如下。

（1）buildProcessBuilder

功能描述：基于给定的参数创建 ProcessBuilder。

```
def buildProcessBuilder(
      command: Command,
      memory: Int,
      sparkHome: String,
      substituteArguments: String => String,
      classPaths: Seq[String] = Seq[String](),
      env: Map[String, String] = sys.env): ProcessBuilder = {
      val localCommand = buildLocalCommand(command, substituteArguments,
         classPaths, env)
    val commandSeq = buildCommandSeq(localCommand, memory, sparkHome)
    val builder = new ProcessBuilder(commandSeq: _*)
    val environment = builder.environment()
    for ((key, value) <- localCommand.environment) {
        environment.put(key, value)
    }
    builder
}
```

（2）buildLocalCommand

功能描述：通过复制 ApplicationDescription 中的类路径、包路径、环境变量、Java 选项参数等信息，在本地创建 Command。

```
private def buildLocalCommand(
      command: Command,
```

```
      substituteArguments: String => String,
      classPath: Seq[String] = Seq[String](),
      env: Map[String, String]): Command = {
    val libraryPathName = Utils.libraryPathEnvName
    val libraryPathEntries = command.libraryPathEntries
    val cmdLibraryPath = command.environment.get(libraryPathName)
    val newEnvironment = if (libraryPathEntries.nonEmpty && libraryPathName.
       nonEmpty) {
      val libraryPaths = libraryPathEntries ++ cmdLibraryPath ++ env.
         get(libraryPathName)
      command.environment + ((libraryPathName, libraryPaths.mkString(File.
          pathSeparator)))
    } else {
      command.environment
    }
    Command(
        command.mainClass,
        command.arguments.map(substituteArguments),
        newEnvironment,
        command.classPathEntries ++ classPath,
        Seq[String](), // library path already captured in environment variable
        command.javaOpts)
}
```

（3）buildCommandSeq

功能描述：用于构建命令行参数序列。

```
private def buildCommandSeq(command: Command, memory: Int, sparkHome: String):
   Seq[String] = {
    val runner = sys.env.get("JAVA_HOME").map(_ + "/bin/java").getOrElse("java")
    Seq(runner) ++ buildJavaOpts(command, memory, sparkHome) ++ Seq(command.
       mainClass) ++
       command.arguments
}
```

（4）buildJavaOpts

功能描述：依据指定的内存大小、SPARK_JAVA_OPTS，执行 /bin/compute-classpath 的 sh 或者 cmd 脚本计算当前环境的 classPath，依据 javaVersion 设置 -XX:MaxPermSize 等构造出 Java 选项。

```
private def buildJavaOpts(command: Command, memory: Int, sparkHome: String):
Seq[String] = {
    val memoryOpts = Seq(s"-Xms${memory}M", s"-Xmx${memory}M")
    // Exists for backwards compatibility with older Spark versions
    val workerLocalOpts = Option(getenv("SPARK_JAVA_OPTS")).map(Utils.
       splitCommandString)
      .getOrElse(Nil)
    if (workerLocalOpts.length > 0) {
      logWarning("SPARK_JAVA_OPTS was set on the worker. It is deprecated in Spark
          1.0.")
      logWarning("Set SPARK_LOCAL_DIRS for node-specific storage locations.")
    }
```

```
    // Figure out our classpath with the external compute-classpath script
    val ext = if (System.getProperty("os.name").startsWith("Windows")) ".cmd"
      else ".sh"
    val classPath = Utils.executeAndGetOutput(
        Seq(sparkHome + "/bin/compute-classpath" + ext),
        extraEnvironment = command.environment)
    val userClassPath = command.classPathEntries ++ Seq(classPath)
    val javaVersion = System.getProperty("java.version")
    val permGenOpt = if (!javaVersion.startsWith("1.8")) Some("-XX:MaxPermSize=128m")
      else None
    Seq("-cp", userClassPath.filterNot(_.isEmpty).mkString(File.pathSeparator)) ++
        permGenOpt ++ workerLocalOpts ++ command.javaOpts ++ memoryOpts
}
```

附录 G Appendix G

Netty

1. Netty 简介

Netty 是一个 NIO 客户端服务器框架，使得开发高性能、高可靠性的网络服务器和客户端程序变得快速且容易。它极大地简化和优化了网络编程，如 TCP 和 UDP 套接字服务器。

“快速和容易”并不意味着应用程序会有难维护和性能低的问题。Netty 是一个精心设计的框架，它从许多协议的实现中吸收了很多经验，比如 FTP、SMTP、HTTP 以及多种多样的二进制和基于文本的传统协议。因此 Netty 在不降低开发效率、性能、稳定性、灵活性的情况下，已经成功地找到了解决方法。

有关 Netty 的更多内容请访问 Netty 官网 http://netty.io/。

2. NettyUtils

NettyUtils 是 Spark 对于 Netty API 的又一层封装，这里对 NettyUtils 中的主要方法进行介绍。

（1）createThreadFactory

功能描述：创建线程工厂，此工厂生成的线程都使用给定的前缀名 threadPoolPrefix + "-" + 数字的格式命名。

```
public static ThreadFactory createThreadFactory(String threadPoolPrefix) {
    return new ThreadFactoryBuilder()
        .setDaemon(true)
        .setNameFormat(threadPoolPrefix + "-%d")
        .build();
}
```

（2）createEventLoop

功能描述：根据参数 IOMode，创建 Netty 的 EventLoopGroup。

```
public static EventLoopGroup createEventLoop(IOMode mode, int numThreads, String
    threadPrefix) {
    ThreadFactory threadFactory = createThreadFactory(threadPrefix);

    switch (mode) {
        case NIO:
            return new NioEventLoopGroup(numThreads, threadFactory);
        case EPOLL:
            return new EpollEventLoopGroup(numThreads, threadFactory);
        default:
            throw new IllegalArgumentException("Unknown io mode: " + mode);
    }
}
```

（3）getClientChannelClass

功能描述：根据参数 IOMode，返回正确的客户端 SocketChannel。

```
public static Class<? extends Channel> getClientChannelClass(IOMode mode) {
    switch (mode) {
        case NIO:
            return NioSocketChannel.class;
        case EPOLL:
            return EpollSocketChannel.class;
        default:
            throw new IllegalArgumentException("Unknown io mode: " + mode);
    }
}
```

（4）getServerChannelClass

功能描述：根据参数 IOMode，返回正确的服务端 SocketChannel。

```
public static Class<? extends ServerChannel> getServerChannelClass(IOMode mode) {
    switch (mode) {
        case NIO:
            return NioServerSocketChannel.class;
        case EPOLL:
            return EpollServerSocketChannel.class;
        default:
            throw new IllegalArgumentException("Unknown io mode: " + mode);
    }
}
```

（5）createFrameDecoder

功能描述：创建一个 LengthFieldBasedFrameDecoder。LengthFieldBasedFrameDecoder 的 5 个参数分别代表 frame 的最大长度、长度字段的偏移量、长度字段的字节数、需要排除的长度字段的字节数、长度字段的初始长度。所以创建的 LengthFieldBasedFrameDecoder 的前 8 字节代表 frame 的长度。LengthFieldBasedFrameDecoder 通常会被设置到 SocketChannel 的管道中，在所有 Decoder 之前使用。

```
public static ByteToMessageDecoder createFrameDecoder() {
    return new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 8, -8, 8);
}
```

（6）getRemoteAddress

功能描述：返回 Channel 的远端地址。

```
public static String getRemoteAddress(Channel channel) {
    if (channel != null && channel.remoteAddress() != null) {
        return channel.remoteAddress().toString();
    }
    return "<unknown remote>";
}
```

（7）createPooledByteBufAllocator

功能描述：创建一个汇集 ByteBuf 但对本地线程缓存禁用的分配器。为什么要对本地线程缓存禁用？因为 ByteBuf 都是由事件循环线程分配，所以线程本地缓存对于 TransportClient 是禁用的。但是 ByteBuf 的释放却是由 Executor 线程，而不是事件循环线程来完成。本地线程缓存经常会延迟 ByteBuf 的回收，导致巨大的内存消耗。

```
public static PooledByteBufAllocator createPooledByteBufAllocator(
        boolean allowDirectBufs,
        boolean allowCache,
        int numCores) {
    if (numCores == 0) {
        numCores = Runtime.getRuntime().availableProcessors();
    }
    return new PooledByteBufAllocator(
        allowDirectBufs && PlatformDependent.directBufferPreferred(),
        Math.min(getPrivateStaticField("DEFAULT_NUM_HEAP_ARENA"), numCores),
        Math.min(getPrivateStaticField("DEFAULT_NUM_DIRECT_ARENA"),
            allowDirectBufs ? numCores : 0),
        getPrivateStaticField("DEFAULT_PAGE_SIZE"),
        getPrivateStaticField("DEFAULT_MAX_ORDER"),
        allowCache ? getPrivateStaticField("DEFAULT_TINY_CACHE_SIZE") : 0,
        allowCache ? getPrivateStaticField("DEFAULT_SMALL_CACHE_SIZE") : 0,
        allowCache ? getPrivateStaticField("DEFAULT_NORMAL_CACHE_SIZE") : 0
    );
}
```

（8）getPrivateStaticField

功能描述：用于获得 Netty 的静态属性值。

```
private static int getPrivateStaticField(String name) {
    try {
        Field f = PooledByteBufAllocator.DEFAULT.getClass().getDeclaredField(name);
        f.setAccessible(true);
        return f.getInt(null);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
}
```

（9）askWithReply

功能描述：使用 ActorSystem 向 ActorRef 发送任何消息。发送每条消息的最大尝试次数

是 3 次，间隔为 3000 毫秒，请求超时时间是 30 秒。

```
def askWithReply[T](
        message: Any,
        actor: ActorRef,
        maxAttempts: Int,
        retryInterval: Int,
        timeout: FiniteDuration): T = {
    if (actor == null) {
        throw new SparkException("Error sending message as actor is null " +
            "[message = " + message + "]")
    }
    var attempts = 0
    var lastException: Exception = null
    while (attempts < maxAttempts) {
        attempts += 1
        try {
            val future = actor.ask(message)(timeout)
            val result = Await.result(future, timeout)
        if (result == null) {
            throw new SparkException("Actor returned null")
        }
        return result.asInstanceOf[T]
      } catch {
        case ie: InterruptedException => throw ie
        case e: Exception =>
            lastException = e
            logWarning("Error sending message in " + attempts + " attempts", e)
      }
      Thread.sleep(retryInterval)
    }

    throw new SparkException(
        "Error sending message [message = " + message + "]", lastException)
}
```

附录 H Appendix H

源码编译错误

笔者在编译过程中遇到很多问题，这里列出需要注意的 6 个主要问题。

（1）jar 包依赖错误

出现很多 maven 的 jar 包依赖错误，例如：

```
ArtifactTransferException: Failure to transfer asm:asm-commons:jar:3.1
from?http://mvnrepo.taobao.ali.com/mvn/repositorywas cached in the local
    repository, resolution will not be reattempted until the update interval of
    central has elapsed or updates are forced. Original error: Could not transfer
    artifact asm:asm-commons:jar:3.1 from/to snapshots (http://mvnrepo.taobao.
    ali.com/mvn/repository): The operation was cancelled.pom.xml/spark-hive-
    thriftserver_2.10line 20Maven Dependency Problem
```

类似这样的错误虽然多，但都很好解决，拿这个例子来说，只需要去本地 maven 仓库下的 asm\asm-commons 文件夹下，将 3.1 这个文件夹删除，然后在 Eclipse 里右击项目 spark-hive-thriftserver_2.10，然后选择 Maven 菜单下的 update project 更新 jar 包即可解决。

（2）单元测试错误

编译时报错：http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException:Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.10:test (default-test) on project hello: There are test failures.

如果单独进行 spark-streaming_2.10 的单元测试，全部是通过的，不得已只能在 spark-streaming_2.10 下的 pom 里找到如下这段配置，修改为 ture，当 Scala 单测失败时依然能编译通过，如图 H-1 所示。

（3）Scala 版本问题

出现 Scala 版本问题，错误信息如下：

```
<plugin>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest-maven-plugin</artifactId>
  <configuration>
      <testFailureIgnore>true</testFailureIgnore>
  </configuration>
</plugin>
```

图 H-1 spark-streaming_2.10 项目的 pom 修改

```
can't expand macros compiled by previous versions of
ScalaSparkSinkSuite.scala/
spark-streaming-flume-sink_2.10/src/test/scala/org/apache/spark/streaming/flume/
   sinkline 75Scala Problem
```

意思是 Scala 版本不对，解决方式：单击 Window 菜单，然后选择 Preferences 子菜单，在弹出的对话框中选择 Scala 下的 Installations，选择合适的 Scala 版本即可，如图 H-2 所示。

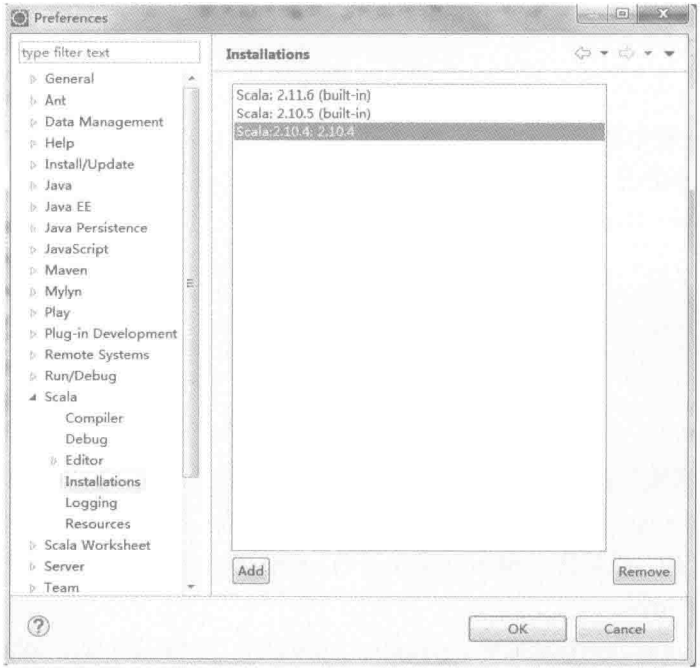


图 H-2 选择合适的 Scala 版本

（4）scalastyle-maven-plugin 修改

编译 spark-catalyst_2.10 时，出现错误：

```
http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException:Failed to
   execute goal
org.scalastyle:scalastyle-maven-plugin:0.4.0:check (default) on project。具体信
    息有Failed during scalastyle execution: Unable to find configuration file at
    location scalastyle-config.xml。
```

意思是找不到 scalastyle-config.xml 了，在 spark 源码的根目录下有 scalastyle-config.xml 文件，但是编译时的路径不对，在父项目 spark 下的 pom 文件中找到这段 scalastyle-maven-plugin 的配置，将其中的配置改为 scalastyle-config.xml 所在的路径，如图 H-3 所示。

```
<plugin>
  <groupId>org.scalastyle</groupId>
  <artifactId>scalastyle-maven-plugin</artifactId>
  <version>0.4.0</version>
  <configuration>
    <verbose>false</verbose>
    <failOnViolation>true</failOnViolation>
    <includeTestSourceDirectory>false</includeTestSourceDirectory>
    <failOnWarning>false</failOnWarning>
    <sourceDirectory>${basedir}/src/main/scala</sourceDirectory>
    <testSourceDirectory>${basedir}/src/test/scala</testSourceDirectory>
    <configLocation>D:\git\spark/scalastyle-config.xml</configLocation>
    <outputFile>D:\git\spark/scalastyle-output.xml</outputFile>
    <outputEncoding>UTF-8</outputEncoding>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

图 H-3　修改 scalastyle-config.xml 的路径

（5）jar 包盲眼问题

起名叫盲眼问题，是因为 jar 包就在那里，可是项目会报找不到这个 jar 包的错误，比如笔者本地的 tachyon 的 jar 包就找不到了，如图 H-4 所示。

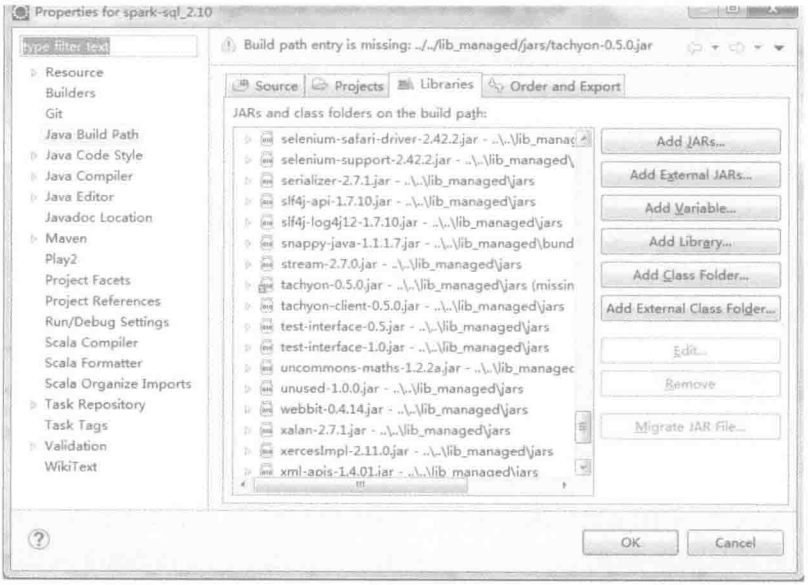


图 H-4　jar 包盲眼问题

解决起来很简单，只需要将此 jar 包依赖删除，重新添加一遍即可。

（6）项目间依赖盲眼

Spark 各个项目间会存在依赖关系，比如 spark-assembly_2.10 就依赖 spark-bagel_2.10、spark-core_2.10、spark-graphx_2.10、spark-mllib_2.10、spark-sql_2.10、spark-streaming_2.10 等项目。可是 spark-assembly_2.10 居然找不到这些依赖的项目，会出现图 H-5 中的错误。

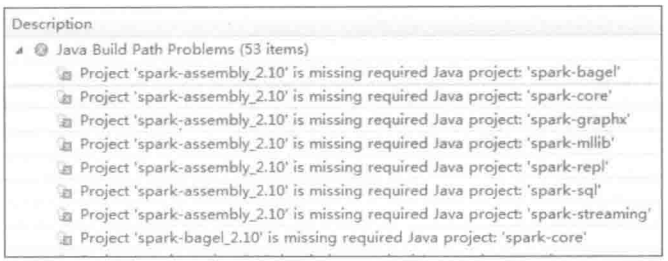


图 H-5 项目间依赖盲眼

打开 spark-assembly_2.10 的 build path 查看，如图 H-6 所示。

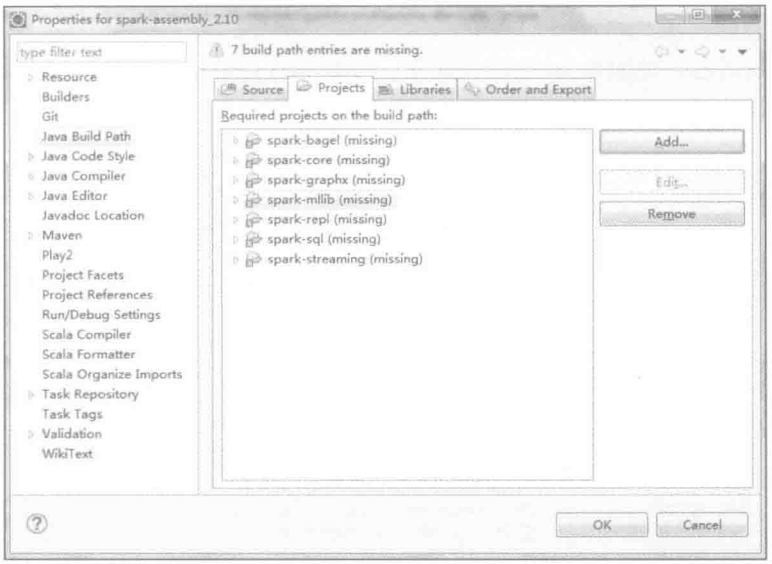


图 H-6 spark-assembly_2.10 依赖的项目

spark-assembly_2.10 没找到依赖的项目，可能是因为导入 Spark 的时候，项目名称改变的原因，不过解决起来还是很简单的，将这些项目依赖删除，重新添加进来即可，如图 H-7 所示。

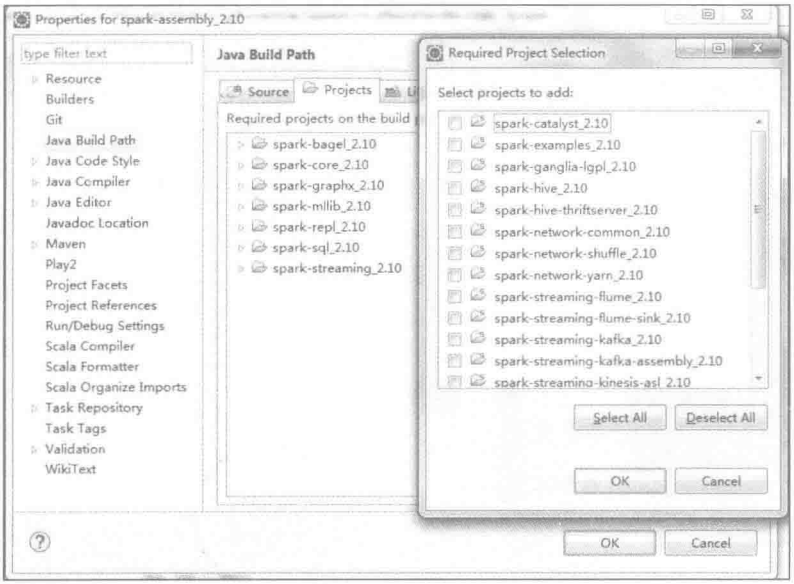
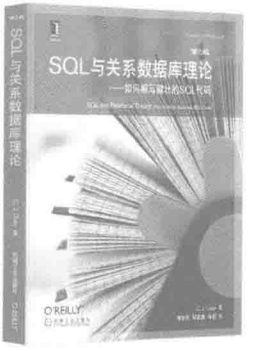
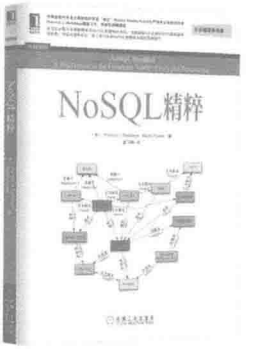


图 H-7　重新编辑 spark-assembly_2.10 依赖的项目

推荐阅读


SQL与关系数据库理论
O'REILLY


数据库
查询优化器
的艺术
原理解析与SQL性能优化


NoSQL精粹

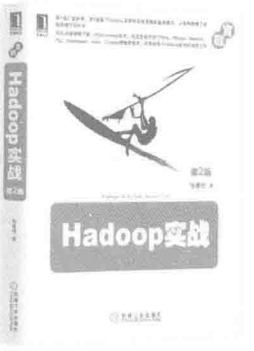

SAP HANA平台
软件开发实战


HBase企业应用
开发实战


Spark大数据处理
技术、应用与性能优化


DB2 SQL
Tuning Tips for z/OS Developers
DB2 SQL性能调优秘笈


深入理解大数据
大数据处理与编程实践
UNDERSTANDING BIG DATA
BIG DATA


Hadoop实战
Hadoop实战

伴随着互联网的不断演进，人类所面临的数据在体量，产生速度和多样性方面阶跃性发展，随之而来的是数据计算和处理方式的多样化，目前越来越多的数据处理链路是通过多种计算组合而成，例如批量与流式计算，图计算，交互式查询等。而以往几个独立计算系统“物理”拼装组合成的复杂系统在处理这些问题时，往往在成本和效率上产生瓶颈。Spark从迭代计算的数据复用切入，底层一个runtime来支撑多种计算模型，越来越受到业界的重视，社区发展非常迅速。而本书从源码分析角度深入剖析系统，希望读者不仅做到知其然，更要知其所以然，对Spark有更加深入的研究。本书作者在相关领域有多年丰富的实践和应用经验，相信通过研读本书必定可以给读者带来事半功倍的效果。

—— **强琦** 阿里云计算平台资深技术专家

这是一本不错的Spark的入门书籍，完全从工程师的视角出发，从安装到使用再到高阶应用。有些时候甚至有些啰嗦，但这不正是我们读者需要的么？作者用他专一的一线工程师视角与在阿里面临的场景结合，写作的落笔相当接地气。这是一本难得的工程师参考用书。

—— **张茂森** 阿里巴巴商家业务事业部资深数据挖掘专家

本书特色：

- 按照源码分析的习惯设计，条分缕析。
- 多图、多示例，帮读者快速在头脑中“建模”。
- 原理与实现剖析，帮助读者提升架构设计、程序设计等方面的能力。
- 尽可能保留较多的源码，方便离线和移动环境的阅读。

上架指导：计算机/数据挖掘
ISBN 978-7-111-52234-8
9 787111 522348 >
定价：99.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

[General Information]
书名=深入理解Spark 核心思想与源码分析
作者=耿嘉安著
页数=470
SS号=13952303
DX号=
出版日期=2016.01
出版社=北京机械工业出版社