本书全面、系统地介绍Spark源码
为读者提供了一系列分析源码的实用技巧
始终抓住资源分配、消息传递、容错处理等基本问题
一步步寻找答案，所有问题迎刃而解

Broadview
www.broadview.com.cn

Apache Spark 源码剖析

许鹏 | 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Apache Spark 源码剖析

许鹏 | 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书以Spark 1.02版本源码为切入点，着力于探寻Spark所要解决的主要问题及其解决办法，通过一系列精心设计的小实验来分析每一步背后的处理逻辑。

本书第3～5章详细介绍了Spark Core中作业的提交与执行，对容错处理也进行了详细分析，有助读者深刻把握Spark实现机理。第6～9章对Spark Lib库进行了初步的探索。在对源码有了一定的分析之后，读者可尽快掌握Spark技术。

本书对于Spark应用开发人员及Spark集群管理人员都有极好的学习价值；对于那些想从源码学习而又不知如何入手的读者，也不失为一种借鉴。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Apache Spark源码剖析 / 许鹏著. — 北京：电子工业出版社，2015.3
ISBN 978-7-121-25420-8
I. ① A…II. ① 许…III. ① 互联网络－网络服务器 ② 数据处理软件IV. ① TP368.5 ② TP274

中国版本图书馆CIP数据核字（2015）第010897号

策划编辑：付　睿
责任编辑：李云静
印　　刷：北京天来印务有限公司
装　　订：北京天来印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　印张：18.5　字数：432千字
版　　次：2015年3月第1版
印　　次：2015年3月第1次印刷
定　　价：68.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

笔者接触Spark时间不算很长，而本书之所以能够出版，凭借的是浓厚的兴趣和执着之心。

这一切还要从Storm说起。笔者一直在做互联网相关工作，但接触大数据的时间并不长，当时Hadoop和Storm等非常红火，引起了笔者的“窥视”之心。从2013年开始，笔者打算看看Hadoop的源码实现，观察其代码规模，发觉所花时间可能会很长。恰好其时Storm风头正劲，于是转向Storm源码，0.8版的Storm代码规模不过20 000行左右，感觉还是比较好入手的。

Storm源码分析期间，笔者还学习了Clojure、ZeroMQ、Thrift、ZooKeeper、LMAX Disruptor等新技术，对于实时流数据处理算是有了一个大概的了解。由于听说在实时流数据处理领域Spark技术也很强悍，而且在容错性方面具有天生的优势，更引发了笔者的兴趣，为了弄清楚究竟，于是开始了Spark的源码走读过程。

笔者是以读Spark论文开始的，说老实话觉得晦涩难懂，因为无法将其映射到内存使用、进程启动、线程运行、消息传递等基本问题上。或许换个方法会更好，故笔者选择直接从源码入手，如此一来事情反而变简单了。在源码分析的过程中，笔者始终抓住资源分配、消息传递、容错处理等基本问题设问，然后一步步努力寻找答案，所有的问题渐渐迎刃而解。

笔者关于源码分析有一个心得，就是要紧紧把握住计算的基本模型，然后结合新分析问题的业务领域，将业务上的新问题转换到计算处理的老套路上来，然后就可以以不变应万变，而不被一些新技术名词晃花了眼。这里所说的老套路是指从操作系统的角度来看，如果能事先深度了解操作系统，将对分析一些新应用程序大有裨益。

Spark源码采用Scala语言编写，那么阅读Spark源码之前，是否一定要先学Scala呢？笔者个人以为不必，只要你有一些Java或C++编程语言的基础，就可以开始看Spark源码，遇到不懂的地方再去学习，效率反而会大大提高，做到有的放矢。将学习中遇到的知识点，从函数式编程、

泛型编程、面向对象、并行编程等几个方面去整理归纳，这样能够快速将Scala语言的框架勾勒出来。

本书第1章和第2章简要介绍了大数据分析技术的产生背景和演进过程；第3～5章详细分析了Spark Core中的作业规划、提交及任务执行等内容，对于要深刻把握Spark实现机理的读者来说，这几章值得反复阅读；第6～9章就Spark提供的高级Lib库进行了简要的分析，分析的思路是解决的主要问题是什么、解决的方案是如何产生的，以及方案是如何通过代码来具体实现的。

在对源码有了一定的分析和掌握之后，再回过头来看一下Spark相关的论文，这时候对论文的理解可能会更顺畅。

Spark的整体框架非常庞大，涵盖的范围也很广，随着笔者在工作中使用得越来越具体，这样的感受也越来越深。另外，必须要说对于Spark来说，笔者所做的分析实在有限，个中错误在所难免，读者诸君还请多多谅解。

在本书成稿期间，电子工业出版社的付睿编辑和李云静编辑给出了极为详细的改进意见，在这里表示衷心的感谢。最后感谢家人的支持和鼓励，亲爱的老婆和懂事的儿子给了笔者坚持的理由和勇气。

许　鹏

2015年2月

目录

第一部分 Spark概述 1

第1章 初识Spark 3

1.1 大数据和Spark 3

1.1.1 大数据的由来 4

1.1.2 大数据的分析 4

1.1.3 Hadoop 5

1.1.4 Spark简介 6

1.2 与Spark的第一次亲密接触 7

1.2.1 环境准备 7

1.2.2 下载安装Spark 8

1.2.3 Spark下的WordCount 8

第二部分 Spark核心概念 13

第2章 Spark整体框架 15

2.1 编程模型 15

2.1.1 RDD 17

2.1.2 Operation 17

2.2 运行框架 18

2.2.1 作业提交 18

2.2.2 集群的节点构成 18

2.2.3 容错处理 19

2.2.4 为什么是Scala 19

2.3 源码阅读环境准备 19
2.3.1 源码下载及编译 19
2.3.2 源码目录结构 21
2.3.3 源码阅读工具 21
2.3.4 本章小结 22

第3章 SparkContext初始化 23

3.1 spark-shell 23
3.2 SparkContext的初始化综述 27
3.3 Spark Repl综述 30
3.3.1 Scala Repl执行过程 31
3.3.2 Spark Repl 32

第4章 Spark作业提交 33

4.1 作业提交 33
4.2 作业执行 38
4.2.1 依赖性分析及Stage划分 39
4.2.2 Actor Model和Akka 46
4.2.3 任务的创建和分发 47
4.2.4 任务执行 53
4.2.5 Checkpoint和Cache 62
4.2.6 WebUI和Metrics 62
4.3 存储机制 71
4.3.1 Shuffle结果的写入和读取 71
4.3.2 Memory Store 80
4.3.3 存储子模块启动过程分析 81
4.3.4 数据写入过程分析 82
4.3.5 数据读取过程分析 84
4.3.6 TachyonStore 88

第5章 部署方式分析 91

5.1 部署模型 91
5.2 单机模式 local 92
5.3 伪集群部署 local-cluster 93
5.4 原生集群Standalone Cluster 95
5.4.1 启动Master 96
5.4.2 启动Worker 97
5.4.3 运行spark-shell 102
5.4.4 容错性分析 106
5.5 Spark On YARN 112
5.5.1 YARN的编程模型 112
5.5.2 YARN中的作业提交 112
5.5.3 Spark On YARN实现详解 113
5.5.4 SparkPi on YARN 122

第三部分 Spark Lib 129

第6章 Spark Streaming 131

6.1 Spark Streaming整体架构 131
6.1.1 DStream 132
6.1.2 编程接口 133
6.1.3 Streaming WordCount 134
6.2 Spark Streaming执行过程 135
6.2.1 StreamingContext初始化过程 136
6.2.2 数据接收 141
6.2.3 数据处理 146
6.2.4 BlockRDD 155
6.3 窗口操作 158
6.4 容错性分析 159
6.5 Spark Streaming vs. Storm 165
6.5.1 Storm简介 165
6.5.2 Storm和Spark Streaming对比 168
6.6 应用举例 168
6.6.1 搭建Kafka Cluster 168
6.6.2 KafkaWordCount 169

第7章 SQL 173

7.1 SQL语句的通用执行过程分析 175
7.2 SQL On Spark的实现分析 178
7.2.1 SqlParser 178
7.2.2 Analyzer 184
7.2.3 Optimizer 191
7.2.4 SparkPlan 192
7.3 Parquet 文件和JSON数据集 196
7.4 Hive简介 197
7.4.1 Hive 架构 197
7.4.2 HiveQL On MapReduce执行过程分析 199
7.5 HiveQL On Spark详解 200
7.5.1 Hive On Spark环境搭建 206
7.5.2 编译支持Hadoop 2.x的Spark 211
7.5.3 运行Hive On Spark测试用例 213

第8章 GraphX 215

8.1 GraphX简介 215
8.1.1 主要特点 216
8.1.2 版本演化 216
8.1.3 应用场景 217

8.2 分布式图计算处理技术介绍 218
8.2.1 属性图 218
8.2.2 图数据的存储与分割 219
8.3 Pregel计算模型 220
8.3.1 BSP 220
8.3.2 像顶点一样思考 220
8.4 GraphX图计算框架实现分析 223
8.4.1 基本概念 223
8.4.2 图的加载与构建 226
8.4.3 图数据存储与分割 227
8.4.4 操作接口 228
8.4.5 Pregel在GraphX中的源码实现 230
8.5 PageRank 235
8.5.1 什么是PageRank 235
8.5.2 PageRank核心思想 235

第9章 MLLib 239

9.1 线性回归 239
9.1.1 数据和估计 240
9.1.2 线性回归参数求解方法 240
9.1.3 正则化 245
9.2 线性回归的代码实现 246
9.2.1 简单示例 246
9.2.2 入口函数 train 247
9.2.3 最优化算法 optimizer 249
9.2.4 权重更新 update 256
9.2.5 结果预测 predict 257
9.3 分类算法 257
9.3.1 逻辑回归 258
9.3.2 支持向量机 260
9.4 拟牛顿法 261
9.4.1 数学原理 261
9.4.2 代码实现 265
9.5 MLLib与其他应用模块间的整合 268

第四部分 附录 271

附录A Spark源码调试 273

附录B 源码阅读技巧 283

第一部分

Spark 概述

第 1 章 初识 Spark
1.1 大数据和 Spark
1.2 与 Spark 的第一次亲密接触

第1章

初识Spark

“物有本末，事有终始，知所先后，则近道矣。”

《大学》

1.1 大数据和Spark

大数据和大数据分析是目前IT领域最炙手可热的概念，不谈一下自己对大数据的看法都不好意思说自己是在IT圈混的。

那么大数据从何而来，大量的数据到底是如何产生的呢？这些问题鲜有人来回答。

同样对于大数据分析来说，也有不少问题：（1）对大数据要做哪些分析；（2）相对于小规模数据来说，分析中又有哪些不同；（3）这些不同会给理论研究和工程实施方面带来哪些难点。

谈到大数据，不由会想到Hadoop。现在Hadoop的流行已有相当一段时间了，大数据和Hadoop之间的关系如何，Hadoop解决了大数据中的哪些问题？

本书的主角是Spark，离开主角不提，却一开始聊什么大数据和Hadoop的关系，有没有搞错啊？

其实不然，要讲清楚Spark的由来，必然要涉及这几层关系：

（1） 大数据的由来。

（2） 大数据和Hadoop。

（3） Hadoop的不足及Spark的诞生。

任何工具的诞生都会涉及以下几个基本问题：（1） 现实问题的产生；（2） 理论模型的提出；（3） 工程实现。

1.1.1 大数据的由来

正所谓“物有本末，事有终始”，要想深入地搞清楚一件事情，理一理它的由来必定会有不少收获。

大数据是一个相对的概念，这个“大”是相对于“小”来说的，一般认为大数据具有如下3个特点。

- **Volume**：数据规模大。
- **Velocity**：处理速度快，时效性要求比较高。
- **Variety**：数据有丰富的多样性。

当然也有一种说法是4V，即加上一个Value。

以Google为例来说明大数据的产生。在互联网刚兴起的日子，尽管还是一种静态网页的展现方式，但对于普罗大众来说，却找到了一种很方便表达自己想法的途径。于是网页规模日趋扩大，内容也日渐丰富。

搜索引擎应时而起，面对这么多的网页、这么多的信息，如何快速找到自己感兴趣的内容呢？对于这个问题的解答，一种非常直观的思路就是先将网页抓取、存储起来，再按关键词进行查询。Yahoo和Google也是这么做的。

数以亿计的网页在抓取下来之后就变成了一个规模巨大的数据集。

网页抓取带来的第一个技术难题就是如何将抓取结果合理地存储起来。

1.1.2 大数据的分析

有数据就会有分析。

为了让数据产生更大的价值，需要对其进行相应的分析。这个时候就会遇到许多技术上的实际问题。

比如，数据规模大到一台机器无法处理时，如何在有限的时间内对整个数据集进行遍历及分析？

1.1.3 Hadoop

针对上述在数据存储和数据分析方面的技术挑战，Google抛出了自己的解决方案。即常说的三大论文：（1） MapReduce；（2） GFS；（3） BigTable。

这三大论文各自解决哪方面的问题呢？具体如下所示。

- **MapReduce**：计算框架。
- **GFS**：数据存储。
- **BigTable**：NoSQL始祖。

Hadoop是根据GFS和MapReduce两大论文所做的开源实现。讲到这里你大概明白了Hadoop所要解决的两大主要问题了：（1） 数据存储；（2） 分布式计算框架。

基于MapReduce这样一个计算框架对数据做哪些方面的分析，则是各显神通了，如Hive、Pig。这也是Hadoop生态圈异常庞杂的一个主因。

Hadoop生态圈非常庞大，看一看图1.1，会留下一个直观的印象。

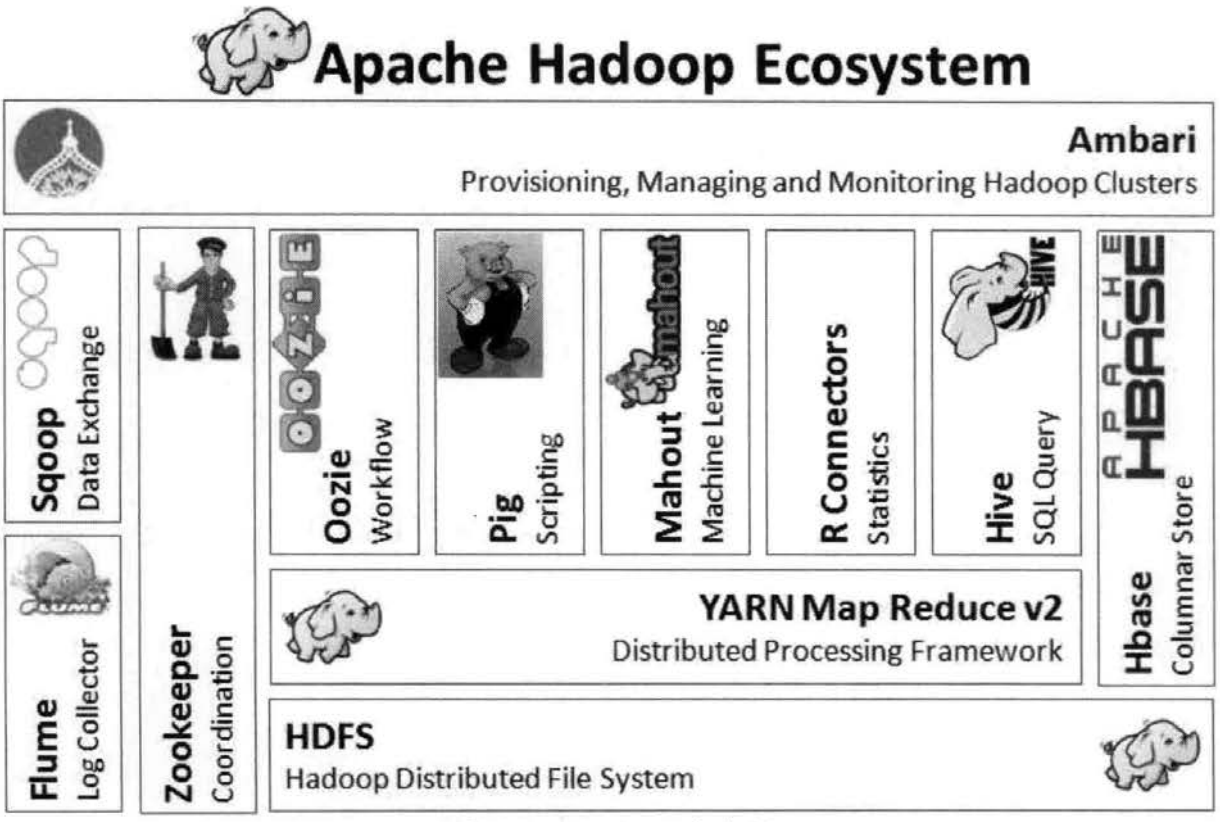


图 1.1 Hadoop生态圈

YARN是Hadoop2和Hadoop1的最大区别，将集群管理本身独立出来，而计算模型则更加专注于问题本身。

1.1.4　Spark简介

到了这里，终于可以讲一讲Spark了。这跟小说没什么分别，没有哪一部小说一开始就让主角上台表演的，精彩总是要慢慢地铺陈开来才行。

Spark由UC Berkeley的AMPLab出品，主要创作者是Matei Zaharia。目前Spark已经成为Apache的顶级项目，这也是称为Apache Spark的原因。

参照上面提到的Hadoop生态圈，问到的第一个问题就是Spark 处在哪一层，它要解决哪方面的问题呢？

好的，采用这样的思维方式，确定参考体系，然后找到问题的定位，将笛卡儿坐标在实际思维中加以运用。

在Hadoop的整个生态系统中，Spark和MapReduce在同一个层级，即主要解决分布式计算框架的问题。

与MapReduce提供的编程模型相比，Spark具有如下两个鲜明的特点：（1） 计算更为快速，速度可以提高10到100倍不等；（2） 计算过程中，如果某一节点出现问题，事件重演的代价远远小于MapReduce。

Spark是如何达到上述效果的，会在后续章节中一一详细讲解。

“天下武功，唯快不破”，算得快、算得准就是Spark的最大特色。

Spark用最精简的话来定义就是通用的分布式计算框架。如果非要加上一些定语的话，那就是适用于大数据的高可靠、高性能分布式并行计算框架。

从上面的简短定义可以看到Spark所要涉及的知识领域：

（1） 分布式并行处理，集群管理。
（2） 高可靠的两重含义，一是服务的有效性，二是计算结果的准确性。
（3） 高性能计算在可以接受的时间内完成。

从Spark的应用领域理论上来说，原先基于MapReduce来开发的分析工具都可以基于Spark来实现，通过这样一种迁移来达到更快的处理效果。

目前Spark已经对表 1.1中的应用开发提供了支持。

表 1.1　Spark支持的应用

Streaming	流数据的实时处理
SQL	类SQL的数据分析
GraphX	常用的图计算
MLLib	机器学习算法

Spark和Hadoop有如下几重关联：

（1） Spark和Hadoop中的MapReduce处在同一层面。

（2） Spark可以部署在YARN上。

（3） Spark原生支持对HDFS文件系统的访问。

至此，基本介绍清楚了Spark因何而生。以图 1.2来做个小小的总结。

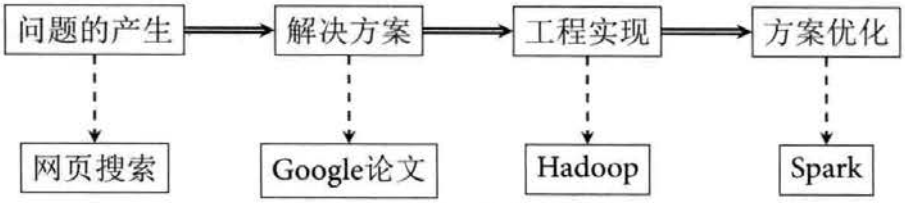


图 1.2 Spark的由来

1.2 与Spark的第一次亲密接触

简要介绍之后，来做一个简单实验，实际使用一下Spark，增加一点感性认识。

1.2.1 环境准备

在真正开始之前，我们还是先来提一提需要哪些准备工作。

- **机器配置**：Spark是非常“吃”内存的，即便是用于学习、不用于实际生产环境，建议内存也不少于4GB。
- **操作系统**：这个当然是Linux，在其所有的发行版中，建议使用Debian或Arch。
- **软件包**：JDK、Scala、Sbt、Maven。

本文后续的所有例子，都假设运行在Arch Linux之上。在Arch Linux，使用以下指令来安装上述软件包。

代码清单 1.1 安装Scala

```
pacman -S scala maven sbt
```

JDK的安装稍微复杂一些，在Linux环境下，由于License的问题，默认使用 OpenJDK。如果要使用Oracle提供的JDK的话，则在安装之外还需要做相应的配置。

安装及配置的具体步骤如下。

代码清单 1.2 安装JDK

```
yaourt -S jdk
```

上述指令将JDK安装在/opt/java目录。打开/etc/profile，修改PATH的值，同时添加JAVA_HOME变量。

代码清单 1.3　将Oracle JDK作为默认的JDK

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/bin:/opt/java/bin:"
export PATH
export JAVA_HOME="/opt/java"
```

1.2.2　下载安装Spark

从官方网站下载Spark发行版，本文以Spark 1.0为例。

下载pre-built版本，选择适用于Hadoop1或Hadoop2的版本。假设下载之后文件存储在$HOME/downloads目录，按以下指令进行解压。

代码清单 1.4　下载pre-built Spark

```
cd $HOME/downloads
tar zvxf spark-1.0.0-bin-hadoop2.tgz
export SPARK_HOME=$HOME/downloads/spark-1.0.0-bin-hadoop2
```

1.2.3　Spark下的WordCount

比如想统计一下某个文件中含有的单词数，WordCount就是一件经常发生的事情。只要文件不是太大，用简单的awk就可以搞定了。下面是用awk来进行单词个数统计的脚本。

代码清单 1.5　workdcount.awk

```
{
  for (i = 1; i<=NF; i++)
    freq[$i]++
}
END{
  for (word in freq)
  printf "%s\t%d\n",word,freq[word]
}
```

将上述脚本保存到文件wordcount.awk，使用该脚本很简单。

代码清单 1.6 使用wordcount.awk

```
gawk -f wordcount.awk filename
```

WordCount俨然已经成为分布式计算框架下的HelloWorld。那么用Spark来实现WordCount是一个什么样子呢？

步骤1：修改$SPARK_HOME/conf/spark-env.sh，在文件末尾添加如下内容。

代码清单 1.7 更改spark-env.sh

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

步骤2：启动spark-shell。

代码清单 1.8 运行spark-shell

```
$SPARK_HOME/bin/spark-shell
```

步骤3：依次执行以下语句。

代码清单 1.9 WordCount in spark-shell

```
val rawFile = sc.textFile("READEME.md")
val words = rawFile.flatMap(line=> line.split(" "))
val wordNumber = words.map(w => (w,1))
val wordCounts = wordNumber.reduceByKey(_ + _)
wordCounts.foreach(println)
```

上述代码的含义如下：

（1） 读取文件README.md。
（2） 以空格为拆分标志，将文件中的每一行分割为多个单词。
（3） 对每一个单词进行计数。
（4） 将单词进行分类合并，计算总的出现次数。
（5） 将所有单词出现的次数进行打印输出。

如果觉着上面每一步都要定义一个变量很烦的话，那么用下面这个样式，一句话就可以搞定了。

代码清单 1.10 WordCount的简明表述

```
sc.textFile("README")
```

```
  .flatMap(line=>line.split(" "))
  .map(w=>(w,1))
  .reduceByKey(_ + _)
  .foreach(println)
```

之所以能够这么方便，一个主要的原因就是由于Scala是一个支持FP的编程语言。

如果不使用reduceByKey这样一个简易的函数而要求出每个单词的个数，程序又该如何写呢?关键的地方是先使用groupByKey，然后再使用map来求和。

代码清单 1.11 不使用reduceByKey求WordCount

```
  sc.textFile("README.md")
  .flatMap(l=>l.split(" "))
  .map(w=>(w,1))
  .groupByKey()
  .map( (p: (String, Iterable[Int])) => (p._1,p._2.sum))
  .collect
```

利用spark-shell可以非常方便地进行交互式运行。下面介绍如何编写一个脱离 spark-shell而可以单独运行的Spark应用程序。

步骤1：编辑源文件。

代码清单 1.12 独立应用SimpleApp

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md"
     //将YOUR_SPARK_HOME替换成Spark安装的目录
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
```

```
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

步骤2：编辑simple.sbt。

代码清单 1.13 simple.sbt

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.10.4"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.0.2"
resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

相应的目录结构如下所示。

代码清单 1.14 SimpleApp的目录结构

```
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala
```

步骤3：编译打包。

代码清单 1.15 生成package

```
sbt package
```

步骤4：提交运行。

代码清单 1.16 运行SimpleApp

```
$SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/scala-2.10/simple-project_2.10-1.0.jar
```

使用ps来查看一下SimpleApp运行时对哪些Java库有依赖。

代码清单 1.17　ps -ef | grep -i simpleapp

```
/opt/java/bin/java -cp ::/root/downloads/spark/conf:/root/downloads/spark/
  assembly/target/scala-2.10/spark-assembly-1.1.0-SNAPSHOT-hadoop1.0.4.jar
-XX:MaxPermSize=128m -Djava.library.path= -Xms512m
-Xmx512m org.apache.spark.deploy.SparkSubmit
--class SimpleApp
--master local
target/scala-2.10/simple-project_2.10-1.0.jar
```

从检查的输出结果可以看出，SimpleApp运行时需要spark-assembly-1.1.0-SNAPSHOT-hadoop 1.0.4.jar 。

第二部分

Spark 核心概念

第 2 章 Spark 整体框架
- 2.1 编程模型
- 2.2 运行框架
- 2.3 源码阅读环境准备

第 3 章 SparkContext 初始化
- 3.1 spark-shell
- 3.2 SparkContext 的初始化综述
- 3.3 Spark Repl 综述

第 4 章 Spark 作业提交
- 4.1 作业提交
- 4.2 作业执行
- 4.3 存储机制

第 5 章 部署方式分析
- 5.1 部署模型
- 5.2 单机模式 local
- 5.3 伪集群部署 local-cluster
- 5.4 原生集群 Standalone Cluster
- 5.5 Spark On YARN

第2章

Spark整体框架

“君子务本，本立而道生。”

《论语》

本章从简单的WordCount例子出发，通过分析来推导出Spark的大致架构。

讲述的重点是Spark的框架为什么会是这样的，而不在于Spark框架已经就是这样的。

Spark作为一个非常优秀的分布式计算框架，面对分布式计算中的常见问题，提供的解决方案又有什么特点和优点？

2.1 编程模型

“读取一个文件，统计其中每个单词出现的次数”，这就是WordCount的需求描述。

当文件不是很大的时候，完全可以采用下述步骤来解决问题：

（1） 读取文件中的某一行。

（2） 将读入的内容根据指定的单词分割符分割成为多个单词。

（3） 对每一个单词分别计数，单词已经存在则加1，如果第一次出现则计为1。

（4） 重复上述步骤，直到文件遍历完毕。

如果文件内容很多，在一台机器上已经不能存储下该文件，则势必要将该文件采用HDFS的方式来存储。

在这种情况下，我们如果还是只用一台机器来进行单词统计，所花费的时间必定会很长，那有没有可能分布式并行处理，利用多台机器的优势，进而缩短整个处理的等待时长？

所以，现在的重点转移到研究WordCount问题有无并行化的可能上来？仔细观察会发现，整个单词统计的过程可以分为这几大步（见图2.1）：（1）读取文件；（2）单词分割；（3）单词计数；（4）单词归并。

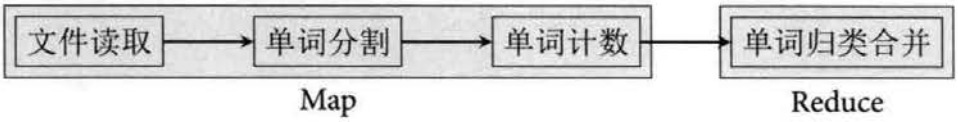


图 2.1　WordCount处理阶段划分

除了最后一步的归并操作，前面的三步是完全可以并行化处理的。经过进一步的抽象，将前面可以高度并行处理的操作归为Map阶段，而后面并行化程度不高的归为Reduce阶段。这就是MapReduce模型最简单的示例。

有了这个模型之后，再来看一看针对大文件的单词统计该如何实现。首先将一个文件分成多个部分，然后针对每个部分分别进行单词分割和单词计数。示意效果如图 2.2所示。

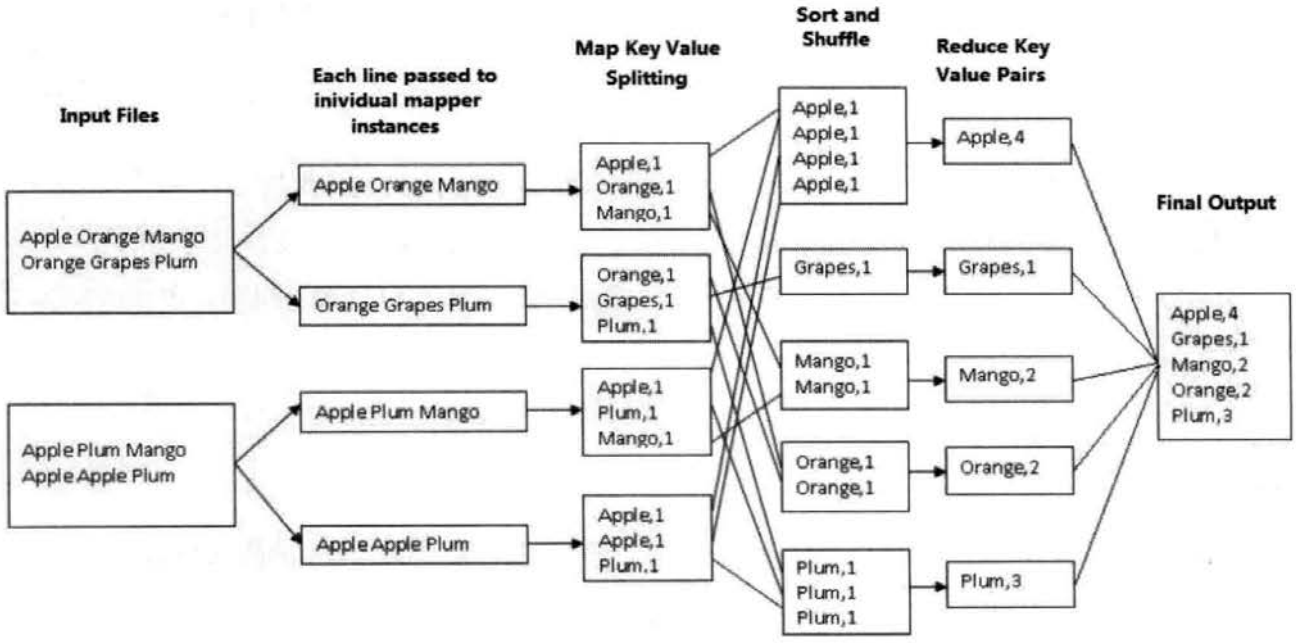


图 2.2　WordCount数据流转示意图

2.1.1 RDD

如果将前面提到的MapReduce模型换一种方式来展现的话，如图2.3所示，即对输入的数据经过一系列的操作后输出另一组数据。

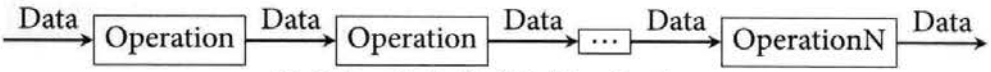


图 2.3 换个角度看MapReduce

可以看出上述的整个过程其实就是在不断重复数据输入和数据处理，针对某一具体的数据输入，有着特定的数据处理逻辑。

Spark中用RDD这样一个概念来包装数据输入和数据处理。有点类似于面向对象中类（Class）这样一个概念。

从字面上来看RDD（Resilient Distributed Dataset）是弹性分布式数据集的缩写，其主要特点如下所列：

- 数据全集被分割为多个正相交的数据子集，每个数据子集可以被派发到任一计算节点进行处理。
- 计算的中间结果会被保存。出于可靠性考虑，同一个计算结果会被保存于多个计算节点。
- 如果其中某一数据子集在处理中出现问题，针对该数据子集的处理会被重新调度进而重新处理。

2.1.2 Operation

Operation分为两种类型：一种是Transformation，另一种是Action。为什么要有这样的划分？这要从执行的角度来加以理解。

Transformation是领取任务的过程，而Action则是真正触发执行的过程。用个形象的比喻就是Transformation是一个做规划的过程，而Action则是将规划付诸实施的过程。

Action是触发将规划以任务（Job）的形式提交给计算引擎，由计算引擎将其转换为多个Task，然后分发到相应的计算节点，开始真正的逻辑处理的过程。

针对数据集，会有哪些Transformation，为什么Spark一开始就要支持这些 Transformation呢？大体的依据要落到关系代数上的图论和集合论中来了。针对数据集，从宏观的层面来说无外乎是交、并、差、子集。

有关Spark RDD所支持各种操作的具体用法可以参考http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html。

2.2 运行框架

2.2.1 作业提交

2.1节提到的RDD只是一个静态的模型，它所解决的只是一个规划问题。静态模型需要在真正运行起来之后才能将问题实实在在解决掉。

在RDD的Operation中有一类称为Action，Action区别于Transformation的显著特征就是会将针对原始输入的所有操作作为一个单一的作业提交到集群中真正执行。

Spark在接收到提交的作业后，会进行如下处理：

（1） RDD之间的依赖性分析。RDD之间的依赖形成一个有向无环图DAG，依赖关系的分析和判断由DAGScheduler负责。

（2） 根据DAG的分析结果将一个作业分成多个Stage。划分Stage的一个主要依据就是当前的计算因子输入是否是确定的，如果是则分在同一个Stage之中。

（3） DAGScheduler在确定完Stage之后，会向TaskScheduler提交任务集（Taskset），而TaskScheduler负责将这些任务一一分发到集群的计算节点（Executor）。

图 2.4是对刚才描述的一个形象化展示。

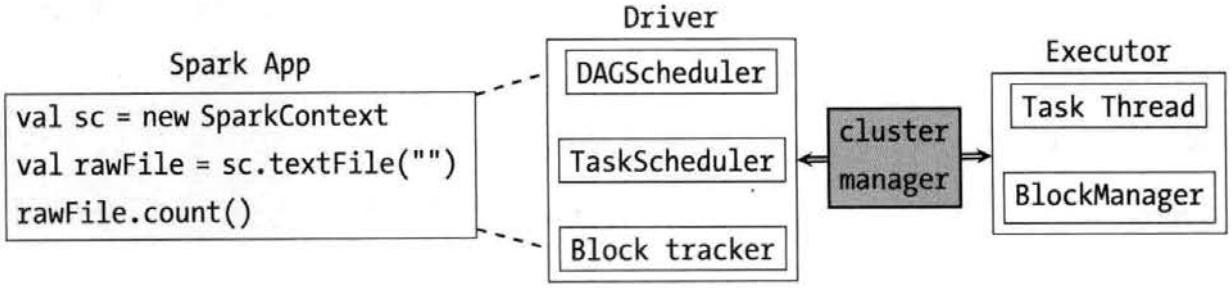


图 2.4 Spark Job提交到运行

2.2.2 集群的节点构成

前面提到Spark的作业会被分发到计算节点上进行真正的运算，那么Spark集群中有哪些类型的节点呢，它们之间的逻辑关系如何？本节将会对这些问题做出分析。

Spark集群由以下4个节点组成，分别是：（1）Driver，（2）Master，（3）Worker，（4）Executor。

它们之间的逻辑关系如图 2.5所示。

其中节点Driver有些特殊，严格来说它并不属于集群中单独的节点类型，它可以运行于集群内部，也可以独立运行于集群之外。

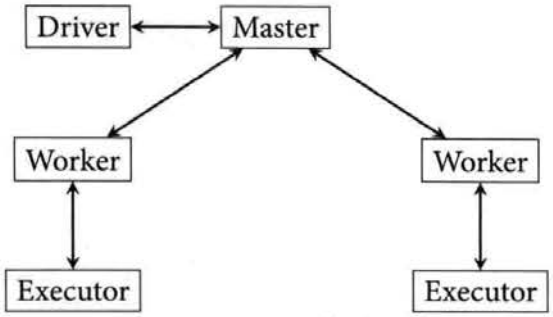


图 2.5 Spark集群组成

如果是跑在YARN Cluster中，则Driver也会运行于集群之内，Driver与Executor处在同一个层级。

2.2.3 容错处理

容错处理是分布式计算中最具挑战性的课题之一。其包含两大主要命题：（1） 确认任务失效，（2） 任务重演。

2.2.4 为什么是Scala

Spark为什么会选用Scala作为其编程语言，这是一个有意思的话题。

我们从编程模型和运行框架两个方面来看Spark这样一个系统会涉及哪方面的内容，然后再就可以入选的语言做一些特性上的比较，这样的分析过程对于我们了解选用Scala的原因将会大有裨益（见表2.1）。

表 2.1 编程语言特性比较

	序列化&反序列化	分布式框架	Hadoop	编码效率	FP	面向对象	泛型编程	IOC
C++	✗	✗	✗	✗	✓	✓	✓	✗
Erlang++	✗	✓	✗	✗	✓	✗	✗	✗
Java	✓	✗	✓	✗	✓	✓	✓	✓
Scala	✓	✓	✓	✓	✓	✓	✓	✓

2.3 源码阅读环境准备

2.3.1 源码下载及编译

有两种不同的途径下载源码。一种是去Spark官方站点spark.apache.org下载。另一种则是通过git来实现。下面着重介绍一下如何使用git来下载源码。

代码清单 2.1 获取Spark源码

```
git clone github.com/apache/spark
```

同步最新的改变，使用如下指令。

代码清单 2.2 同步最新代码

```
git pull origin master
```

有时候对源码做了一定修改，但又不想将其提交，在下一次同步的时候，采用如下指令执行。

代码清单 2.3 不保留本地修改，源码同步

```
git reset --hard
git pull origin master
```

编译很简单，首先确认Sbt已经安装。

代码清单 2.4 编译Spark

```
$SPARK_HOME/sbt/sbt assembly
```

Spark自带了一个编译脚本，名为make-distribution.sh。

代码清单 2.5 使用make—distribution.sh编译

```
export SCALA_HOME=/usr/share/scala
cd $SPARK_HOME
./make-distribution.sh
```

如果一切顺利，会在$SPARK_HOME/assembly/target/scala-2.10目录下生成目标文件，比如笔者机器上编译生成的文件如下所示。

代码清单 2.6 Spark编译结果

```
assembly/target/scala-2.10/spark-assembly-1.0.0-SNAPSHOT-hadoop1.0.4.jar
```

运行测试用例既然能够顺利地编译出jar文件，那么肯定也能改动两行代码来试试效果。如何知道自己的改动是否生效呢？运行测试用例是最好的办法。

假设已经修改了$SPARK_HOME/core下的某些源码，重新编译的话，使用如下指令。

代码清单 2.7 maven编译Spark

```
export SCALA_HOME=/usr/share/scala
mvn package -DskipTests
```

假设当前在$SPARK_HOME/core目录下，想要运行一下RandomSamplerSuite这个测试用例集合，使用以下指令即可。

代码清单 2.8 使用maven运行测试用例

```
export SPARK_LOCAL_IP=127.0.0.1
export SPARK_MASTER_IP=127.0.0.1
mvn -Dsuites=org.apache.spark.util.random.RandomSamplerSuite test
```

2.3.2 源码目录结构

表 2.2是Spark源码各目录的简要说明。

表 2.2 Spark源码目录分类

编译相关	Spark核心	Spark Lib	运行脚本和配置	虚拟化	示例	部署相关	python支持	repl	3pp
sbt assembly project	core	streaming sql graphx mllib	bin sbin conf	ec2 docker dev	examples data	yarn	python	repl	externals

2.3.3 源码阅读工具

以下几种都可以作为Spark的源码阅读工具，每种编辑器都各有所长：

- Intellij Idea
- Eclipse
- Emacs
- Sublime

推荐使用Intellij Idea作为Scala的开发工具，安装完Intellij Idea之后需要安装Scala插件。

2.3.4 本章小结

接下来的3幅图（见图2.6、图2.7和图2.8）反映了从静态到动态再到部署3个不同层面看Spark所要涉及的主要概念。

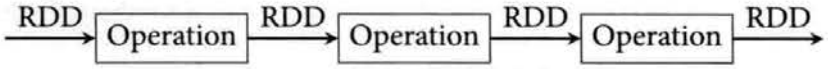


图 2.6 编程模型静态视图

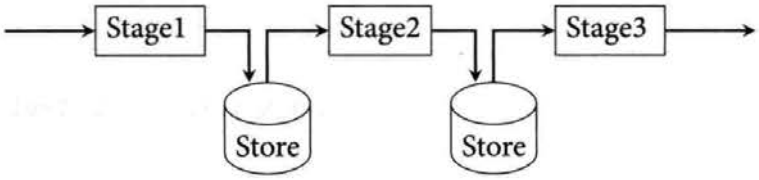


图 2.7 动态视图

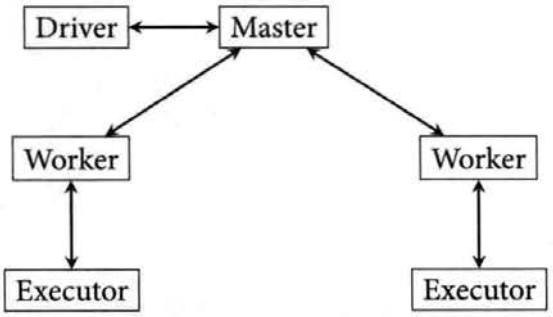


图 2.8 进程视图

第3章

SparkContext初始化

“合抱之木生于毫末；九层之台起于累土；千里之行始于足下。”

《道德经》

本章继续以WordCount为例来说明Spark Job的提交和运行，内容会涉及Spark Application初始化过程、DAG依赖性分析、任务的调度和派发、中间计算结果的存储和读取。

本章及第4章会对Spark Core中的函数调用和消息传递过程做深入而细致的分析，掌握好这两章将会为Spark应用开发打好坚实基础。

3.1 spark-shell

首次使用Spark一般都是从执行spark-shell开始的。当在键盘上敲入spark-shell并回车时，后面究竟发生了哪些事情呢？

代码清单 3.1 spark-shell脚本

```
export SPARK_SUBMIT_OPTS
$FWDIR/bin/spark-submit spark-shell "$@" --class org.apache.spark.repl.Main
```

可以看出spark-shell其实是对spark-submit的一层封装，但事情到这里还没有结束，毕竟还没有找到调用Java的地方，继续往下搜索看看spark-submit脚本的内容。

代码清单 3.2 spark-submit脚本

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

离目标越来越近了，spark-class中会调用到Java程序，与Java相关部分的代码摘录如下。

代码清单 3.3 spark-class脚本片段

```
# Find the java binary
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" $JAVA_OPTS "$@"
```

在某些情况下，可能需要对环境变量及JVM 启动参数做修改，还可能涉及ulimit中的某些配置项，那么可以将这些指令添加到spark-class中。

SparkSubmit当中定义了main函数，在它的处理中会将Spark Repl运行起来，Spark Repl能够接收用户的输入，通过编译与运行，返回结果给用户。这就是Spark具有交互处理能力的原因所在。

调用顺序如下所述：

（1） SparkSubmit

（2） repl.Main

（3） SparkILoop

用Java VisualVM做个小小的实验来验证上述分析。

第一步是在启动Spark Repl的时候打开JMX服务，为此需要修改spark-class文件，在JAVA_OPTS中添加与JMX相关的配置项，修改后的JAVA_OPTS内容如下所示。

代码清单 3.4 修改spark-class中的JAVA_OPTS

```
JAVA_OPTS="-XX:MaxPermSize=128m $OUR_JAVA_OPTS -Dcom.sun.management.jmxremote.port
    =8300 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.
    jmxremote.ssl=false -Djava.rmi.server.hostname=127.0.0.1"
```

第二步是运行Java VisualVM。

代码清单 3.5 启动jvisualvm

```
$JAVA_HOME/bin/jvisualvm
```

Java VisualVM运行后的效果如图 3.1所示。

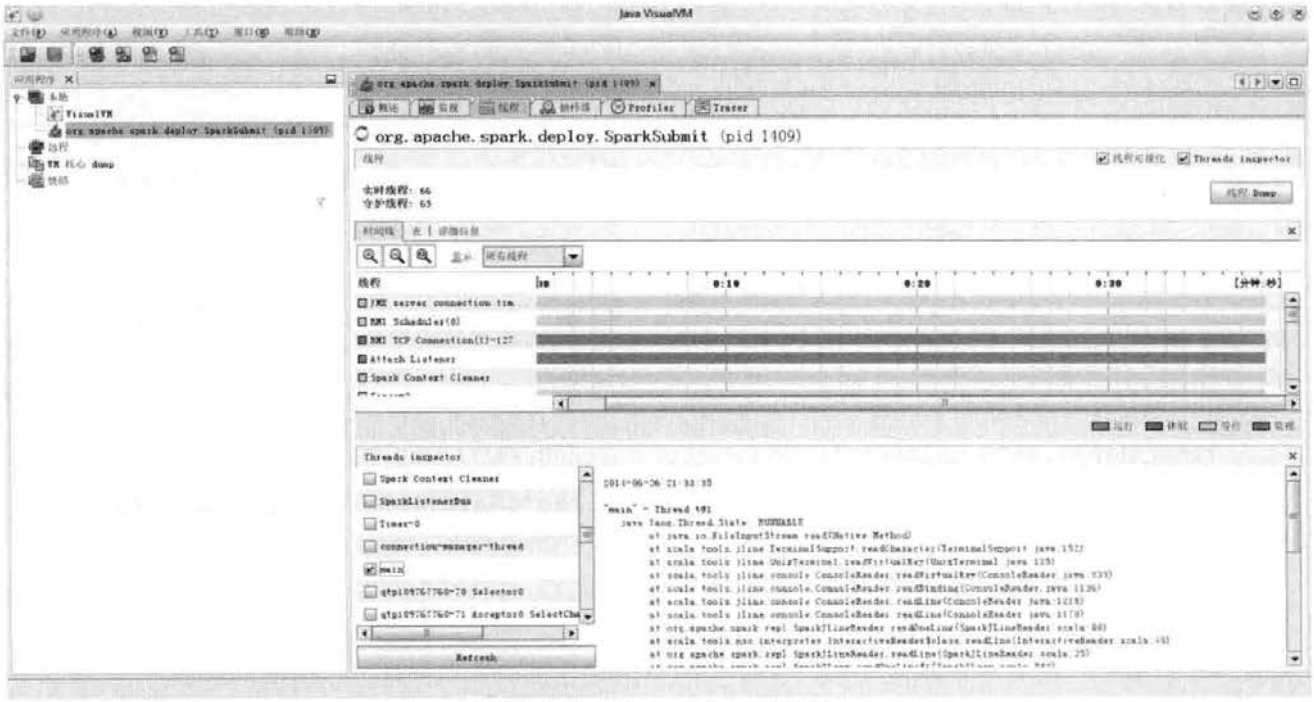


图 3.1 Java VisualVM

在右侧的Tab页中选取线程，在左下角是运行的各个线程，单击main函数，则会在右下侧显示线程的调用关系。

将thread dump的内容摘录如下。

代码清单 3.6 thread dump信息

```
"main" #1 prio=5 os_prio=0 tid=0x00007f91a4008800 nid=0x123b runnable [0x00007f91aaf68000]
java.lang.Thread.State: RUNNABLE
at org.apache.spark.repl.SparkILoop.loop(SparkILoop.scala:611)
```

```
at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply$mcZ$sp(SparkILoop.scala:936)
at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply(SparkILoop.scala:884)
at org.apache.spark.repl.SparkILoop$$anonfun$process$1.apply(SparkILoop.scala:884)
at scala.tools.nsc.util.ScalaClassLoader$.savingContextLoader(ScalaClassLoader.scala:135)
at org.apache.spark.repl.SparkILoop.process(SparkILoop.scala:884)
at org.apache.spark.repl.SparkILoop.process(SparkILoop.scala:982)
at org.apache.spark.repl.Main$.main(Main.scala:31)
at org.apache.spark.repl.Main.main(Main.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:483)
at org.apache.spark.deploy.SparkSubmit$.launch(SparkSubmit.scala:292)
at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:55)
at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```

上述thread dump中的信息非常清晰地给出了repl.main.Main是如何被调用到的，SparkILoop在初始化的过程中会创建SparkContext。

代码清单 3.7 initializeSpark

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

那么看一看createSparkContext函数的代码。

代码清单 3.8 createSparkContext

```
  def createSparkContext(): SparkContext = {
  val execUri = System.getenv("SPARK_EXECUTOR_URI")
  val jars = SparkILoop.getAddedJars
  val conf = 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)
    }
    if (System.getenv("SPARK_HOME") != null) {
      conf.setSparkHome(System.getenv("SPARK_HOME"))
    }
    sparkContext = new SparkContext(conf)
    logInfo("Created spark context..")
    sparkContext
  }
```

至此，从spark-shell到SparkContext被创建的路径全部打通，图3.2总结了调用路径。

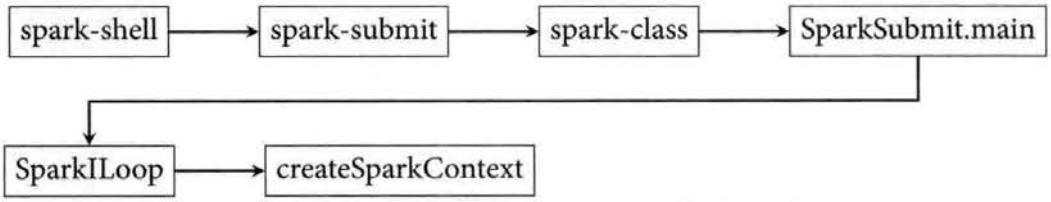


图 3.2 从spark-shell到SparkContext的函数调用路径

3.2 SparkContext的初始化综述

SparkContext是进行Spark应用开发的主要接口，是Spark上层应用与底层实现的中转站。

SparkContext在初始化过程中，主要涉及以下内容：

- SparkEnv
- DAGScheduler
- TaskScheduler
- SchedulerBackend
- WebUI

SparkContext的构造函数中最重要的入参是SparkConf。

步骤1：根据初始化入参生成SparkConf，再根据SparkConf来创建SparkEnv。SparkEnv中主要包含以下关键性组件：BlockManager，MapOutputTracker，ShuffleFetcher，ConnectionManager。

代码清单 3.9 生成SparkEnv

```
private[spark] val env = SparkEnv.create(
  conf,
  "",
  conf.get("spark.driver.host"),
  conf.get("spark.driver.port").toInt,
  isDriver = true,
  isLocal = isLocal)
SparkEnv.set(env)
```

SparkEnv的构造函数入参列表如下所示。

代码清单 3.10 SparkEnv

```
class SparkEnv (
    val executorId: String,
    val actorSystem: ActorSystem,
    val serializer: Serializer,
    val closureSerializer: Serializer,
    val cacheManager: CacheManager,
    val mapOutputTracker: MapOutputTracker,
    val shuffleManager: ShuffleManager,
    val broadcastManager: BroadcastManager,
    val blockManager: BlockManager,
    val connectionManager: ConnectionManager,
    val securityManager: SecurityManager,
    val httpFileServer: HttpFileServer,
    val sparkFilesDir: String,
    val metricsSystem: MetricsSystem,
    val conf: SparkConf) extends Logging
```

下面看一下这些入参的用途。

- **cacheManager**：用以存储中间计算结果。
- **mapOutputTracker**：用来缓存MapStatus信息，并提供从MapOutputMaster获取信息的功能。
- **shuffleManager**：路由维护表。
- **broadcastManager**：广播。
- **blockManager**：块管理。
- **connectionManager**：网络连接管理器。
- **securityManager**：安全管理。
- **httpFileServer**：文件存储服务器。
- **sparkFilesDir**：文件存储目录。
- **metricsSystem**：测量。
- **conf**：配置文件。

步骤2：创建TaskScheduler，根据Spark的运行模式来选择相应的SchedulerBackend，同时启动TaskScheduler，这一步至为关键。

代码清单 3.11 生成TaskScheduler

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

createTaskScheduler最为关键的一点就是根据MASTER环境变量来判断Spark当前的部署方式，进而生成相应的SchedulerBackend的不同子类。创建的SchedulerBackend放置到TaskScheduler中，在后续的Task分发过程中扮演重要作用。

createTaskScheduler函数的代码不具体列出，这里只列出其进行部署方式判断的正则表达式。

代码清单 3.12 createTaskScheduler

```
// Regular expression used for local[N] and local[*] master formats
val LOCAL_N_REGEX = """local\[([0-9\*]+)\]""".r
// Regular expression for local[N, maxRetries], used in tests with failing
// tasks
val LOCAL_N_FAILURES_REGEX = """local\[([0-9]+)\s*,\s*([0-9]+)\]""".r
// Regular expression for simulating a Spark cluster of [N, cores, memory]
// locally
val LOCAL_CLUSTER_REGEX = """local-cluster\[\s*([0-9]+)\s*,\s*([0-9]+)\s*,\s
*([0-9]+)\s*]""".r
// Regular expression for connecting to Spark deploy clusters
val SPARK_REGEX = """spark://(.*)""".r
```

```
    // Regular expression for connection to Mesos cluster by mesos:// or zk:// url
    val MESOS_REGEX = """(mesos|zk)://.*""".r
    // Regular expression for connection to Simr cluster
    val SIMR_REGEX = """simr://(.*)""".r
```

TaskScheduler.start的目的是启动相应的SchedulerBackend，并启动定时器进行检测。

代码清单 3.13 生成SchedulerBackend

```
override def start() {
    backend.start()
    if (!isLocal && conf.getBoolean("spark.speculation", false)) {
      logInfo("Starting speculative execution thread")
      import sc.env.actorSystem.dispatcher
      sc.env.actorSystem.scheduler.schedule(SPECULATION_INTERVAL milliseconds,
            SPECULATION_INTERVAL milliseconds) {
        checkSpeculatableTasks()
      }
    }
  }
```

步骤3：以上一步中创建的TaskScheduler实例为入参创建DAGScheduler并启动运行。

代码清单 3.14 生成DAGScheduler

```
  @volatile private[spark] var dagScheduler = new DAGScheduler(taskScheduler)
  dagScheduler.start()
```

步骤4：启动WebUI。

代码清单 3.15 WebUI

```
ui.start()
```

3.3 Spark Repl综述

Scala已经拥有Repl，为什么在Spark中还要自己另起炉灶，重写一套Repl，原因何在呢？

Scala原生的Repl，是使用Object来封装输入的代码，那这有什么不妥呢？是“序列化和反序列化”的问题啊。在反序列化的过程中，对象的构造函数会被再次调用，产生了副作用，这显

然不是我们所期望的。我们希望生成Class而不是Object。如果你不知道Object和Class的区别，没关系，看一下Scala的简明手册，马上就明白了。

3.3.1 Scala Repl执行过程

再啰唆一次，Scala是需要编译执行的，而Repl给我们的错觉是Scala是解释执行的。那我们在Repl中输入的语句是如何被真正执行的呢？

简要的步骤是这样的：

（1） 在Repl中输入的每一行语句，都会被封装为一个Object，这一工作主要由Interpreter完成。

（2） 对该Object进行编译。

（3） 由ClassLoader加载编译后的Java Bytecode。

（4） 执行引擎负责真正执行加载入内存的Bytecode。

做个小小的实验来验证所做的分析。

步骤1：启动Scala Repl。

代码清单 3.16 启动Scala Repl

```
scala -Dscala.repl.debug=true
```

步骤2：在Scala Repl中输入val c = 10。由于Debug模式已打开，因此可以看到生成的源码如下所示。

代码清单 3.17 Scala Repl的调试信息

```
object $read extends scala.AnyRef {
  def () = {
    super.;
    ()
  };
  object $iw extends scala.AnyRef {
    def () = {
      super.;
      ()
    };
    object $iw extends scala.AnyRef {
      def () = {
        super.;
```

```
        ()
      };
      val c = 10
    }
  }
}
```

3.3.2 Spark Repl

针对Spark Repl，做一个类似的实验。

步骤1：在启动spark-shell之前，先修改一下$SPARK_HOME/bin/spark-class，在JAVA_OPTS中加入以下内容。

代码清单 3.18 修改spark-class

```
-Dscala.repl.debug=true
```

步骤2：启动spark-shell，在Spark Repl中输入val b = 10，可以看到生成的代码是classbasedwrapper。由于生成的代码较长，故此这里不再列出。

Spark实现的自定义的Intepreter，最主要的是实现了SparkIMain。

第4章

Spark作业提交

"未知生，焉知死?"

《论语》

在第3章中分析了Application的启动过程。一个Application运行期间可以执行多个Spark作业，这些作业从提交到运行，分别需要经历哪些过程呢？本章将会为你娓娓道来。

4.1 作业提交

无论是聊Hadoop还是Spark，都习惯于用WordCount作为最开始的例子，WordCount 有点像学编程语言时的HelloWorld一样，是最基础的开始。

那么本章就以最简单的WordCount为例说明RDD从转换到作业提交的过程。

代码清单 4.1 WordCount

```
sc.textFile("README.md").flatMap(line=>line.split(" ")).map(word => (word, 1)).
   reduceByKey(_ + _)
```

上述一行简短的代码其实发生了很复杂的RDD转换。下面仔细解释每一步的转换过程和转换结果。

步骤1：`val rawFile = sc.textFile("README.md")`

textFile先是生成HadoopRDD，然后再通过map操作生成MappedRDD。如果在spark-shell中执行上述语句，得到的结果可以证明所做的分析。

代码清单 4.2 执行textFile及结果

```
scala> sc.textFile("README.md")
res0: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at :13
```

步骤2: `val splittedText = rawFile.flatMap(line => line.split(" "))`

flatMap将原来的MappedRDD转换为FlatMappedRDD。

代码清单 4.3 flatMap

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

步骤3：`val wordCount = splittedText.map(word => (word, 1))`

利用word生成相应的键值对，上一步的FlatMappedRDD被转换为MappedRDD。

步骤4：`val reduceJob = wordCount.reduceByKey(_ + _)`

步骤2和步骤3中使用到的Operation全部定义在RDD.scala中，而这里使用到的reduceByKey却在RDD.scala中见不到踪迹。reduceByKey的定义出现在源文件PairRDDFunctions.scala。

细心的你一定会问：reduceByKey不是MappedRDD的属性和方法啊，怎么能被MappedRDD调用呢？其实这背后发生了一个隐式的转换，该转换将MappedRDD转换为PairRDDFunctions。在PairRDDFunctions.scala中可以找到如下的定义。

代码清单 4.4 rddToPairRDDFunction

```
implicit def rddToPairRDDFunctions[K: ClassTag, V: ClassTag](rdd: RDD[(K, V)]) =
new PairRDDFunctions(rdd)
```

这种隐式的转换是Scala中一个重要的语法特征。

代码清单 4.5 reduceByKey

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)] = {
  reduceByKey(defaultPartitioner(self), func)
}
```

注意defaultPartitioner函数的调用，要获取当前RDD的Partition数目。在真正将Job提交执行之前，必须知道map中有多少Partition。

代码清单 4.6 defaultPartitioner

```
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)
  }
}
```

如果RDD中没有指定Partitioner，那么使用默认的HashPartitioner，Partitioner将在4.3.4节和4.3.5节中做更多的解释。

代码清单 4.7 HashPartitioner

```
class HashPartitioner(partitions: Int) extends Partitioner {
  def numPartitions = partitions

  def getPartition(key: Any): Int = key match {
    case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }

  override def equals(other: Any): Boolean = other match {
    case h: HashPartitioner =>
      h.numPartitions == numPartitions
    case _ =>
      false
  }

  override def hashCode: Int = numPartitions

}
```

步骤5: 触发执行reduceJob.foreach(println)。

foreach会调用sc.runJob，从而生成Job并提交到Spark集群中运行。有关作业执行的细节请看4.2节的详细分析。

先来看一下foreach的函数定义。

代码清单 4.8 foreach

```
def foreach(f: T => Unit) {
  sc.runJob(this, (iter: Iterator[T]) => iter.foreach(f))
}
```

在foreach函数中调用的runJob函数有多个变种，也就是实现了函数重载，这些重载的函数各自实现了哪些功能呢？

接下来就会解答这个问题。

步骤1：指定了Final RDD和作用于RDD上的Function。

代码清单 4.9 runJob（一）

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

步骤2：读取Final RDD的分区数，并指定是否允许本地执行。

代码清单 4.10 runJob（二）

```
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)
}
```

allowLocal表示在Standalone模式下，如果没有Shuffle过程，是否允许在Driver所在的JVM中执行Job。默认是不允许的，即不管是否需要Shuffle，都需要将任务中的Task分发到Cluster中执行。

步骤3：匿名函数转换。

代码清单 4.11 runJob（三）

```
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
}
```

步骤4：添加对Job计算结果的处理句柄。

代码清单 4.12 runJob（四）

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit) {
  if (dagScheduler == null) {
    throw new SparkException("SparkContext has been shutdown")
  }
  val callSite = getCallSite
  val cleanedFunc = clean(func)
  logInfo("Starting job: " + callSite.short)
  val start = System.nanoTime
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
    resultHandler, localProperties.get)
  logInfo("Job finished: " + callSite.short + ", took " + (System.nanoTime -
  start) / 1e9 + " s")
  rdd.doCheckpoint()

}
```

注意在此处调用clean(func)。

代码清单 4.13 clean

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

ClosureCleaner的主要功能

当Scala在创建一个闭包时，需要先判定哪些变量会被闭包所使用并将这些需要使用的变量存储在闭包之内。这一特性使得闭包可以在创建闭包的作用范围之外也能得以正确执行。

但如SI-1419中所反映的那样，Scala有时会捕捉太多不必要的外部变量。在大多数情况下，这样子操作不会有什么副作用，只是这些多余的变量没有被使用罢了。但对于Spark来说，由于这些闭包可能会在其他的机器上执行，故此，多余的外部变量一方面浪费了网络带宽，另一方面可能就是由于外部变量并不支持序列化操作进而导致整个闭包的序列化操作出错。

为了解决这个潜在的问题，Spark专门写了ClosureCleaner来移除那些不必要的外部变量。经过清理的闭包函数能够得以正常地序列化，并可以在任意的机器上执行。

理解了ClosureCleaner存在的原因，也就会明白为什么在写Spark Application的时候，经常会遇到的“Task Not Serializable”是在什么地方报错的了。产生无法序列化的原因就是在RDD的操作中引用了无法序列化的变量。

4.2 作业执行

作业提交执行的完整流程如图 4.1所示。

在任务提交过程中主要涉及Driver和Executor两个节点。

Driver侧在任务提交过程中最主要解决如下几个问题：

（1） RDD依赖性分析，以生成DAG。

（2） 根据RDD DAG将Job分割为多个Stage。

（3） Stage一经确认，即生成相应的Task，将生成的Task分发到Executor执行。

Executor节点在接收到执行任务的指令后，启动新的线程，运行接收到的任务，并将任务的处理结果返回。

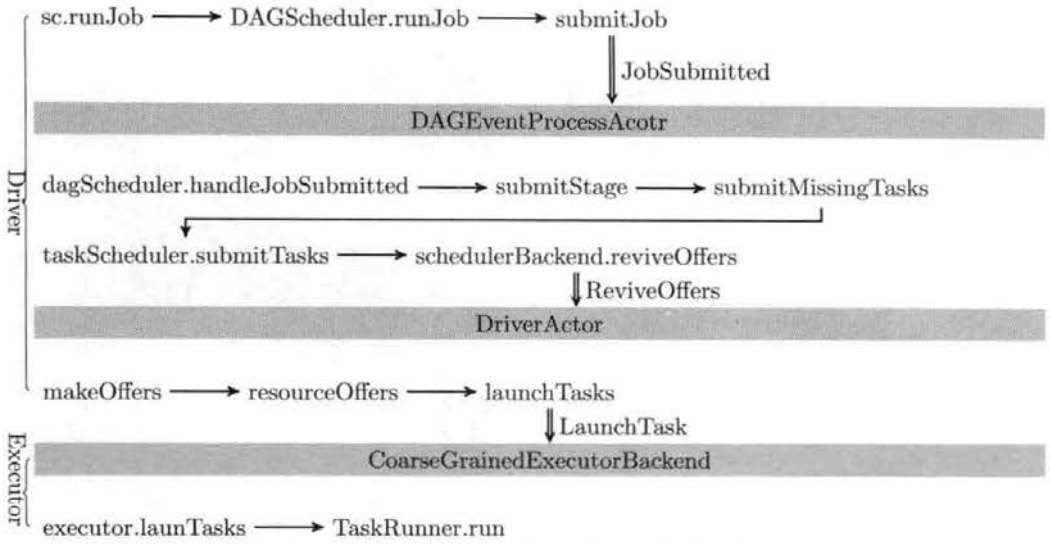


图 4.1 作业提交执行期的函数调用

4.2.1 依赖性分析及Stage划分

Spark中将RDD之间的依赖分为窄依赖和宽依赖。

窄依赖是指父RDD的所有输出都会被指定的子RDD消费，也就是输出路径是固定的。宽依赖是指父RDD的输出会由不同的子RDD消费，即输出路径不固定。

图 4.2和图 4.3是两者的形象化表示。

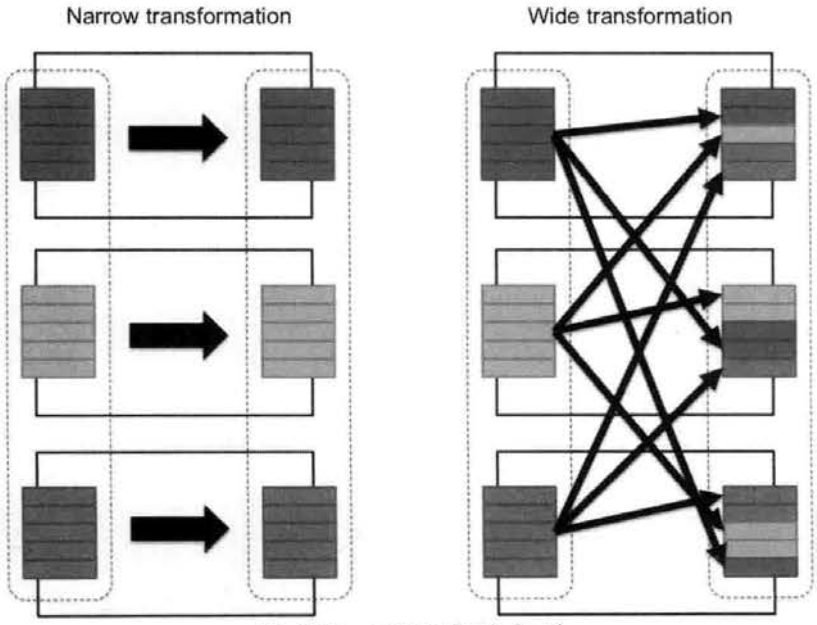


图 4.2 RDD依赖类型

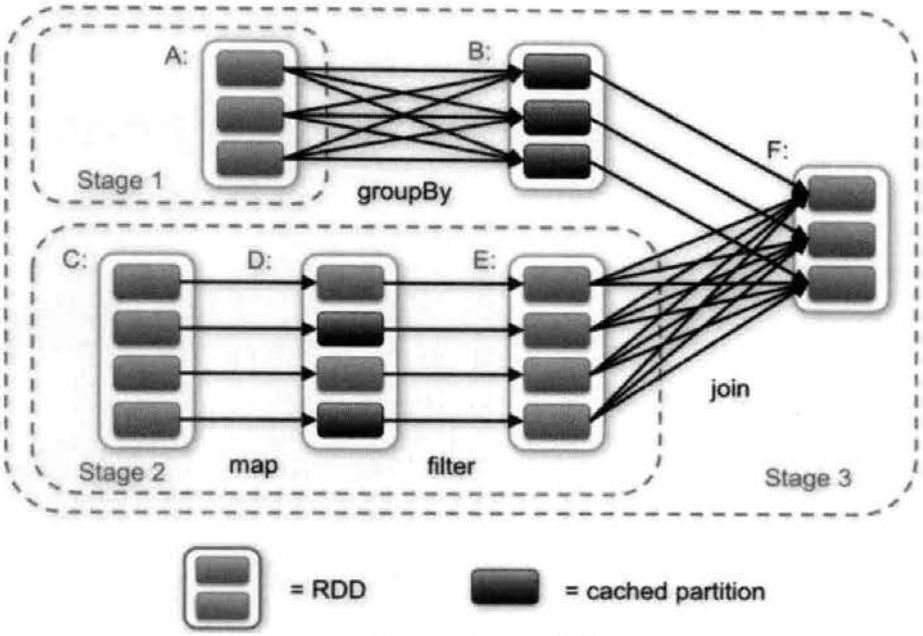


图 4.3　RDD依赖

下列转换（Transformation）导致窄依赖：

- map
- flatMap
- filter
- sample

导致宽依赖的转换（Transformation）如下：

- sortByKey
- reduceByKey
- groupByKey
- cogroupByKey
- join
- cartensian

调度器（Scheduler）会计算RDD之间的依赖关系，将拥有持续窄依赖的RDD归并到同一个Stage中，而宽依赖则作为划分不同Stage的判断标准。

函数handleJobSubmitted和submitStage主要负责依赖性分析，对其处理逻辑做进一步的分析。

handleJobSubmitted最主要的工作是生成finalStage，并根据finalStage来产生ActiveJob。

代码清单 4.14　handleJobSubmitted

```
private[scheduler] def handleJobSubmitted(jobId: Int,
```

```
    finalRDD: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    allowLocal: Boolean,
    callSite: CallSite,
    listener: JobListener,
    properties: Properties = null)
{
  var finalStage: Stage = null
  try {
    // New stage creation may throw an exception if, for example, jobs are run
    // on a HadoopRDD whose underlying HDFS files have been deleted.
    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()
    logInfo("Got job %s (%s) with %d output partitions (allowLocal=%s)".format(
      job.jobId, callSite.short, partitions.length, allowLocal))
    logInfo("Final stage: " + finalStage + "(" + finalStage.name + ")")
    logInfo("Parents of final stage: " + finalStage.parents)
    logInfo("Missing parents: " + getMissingParentStages(finalStage))
    if (allowLocal && finalStage.parents.size == 0 && partitions.length == 1) {
      // Compute very short actions like first() or take() with no parent stages
      // locally.
      listenerBus.post(SparkListenerJobStart(job.jobId, Array[Int](), properties
  ))
      runLocally(job)
```

```
    } else {
      jobIdToActiveJob(jobId) = job
      activeJobs += job
      resultStageToJob(finalStage) = job
      listenerBus.post(SparkListenerJobStart(job.jobId, jobIdToStageIds(jobId).
  toArray,
        properties))
      submitStage(finalStage)
    }
  }
  submitWaitingStages()
}
```

newStage用于创建一个新的Stage。

代码清单 4.15 newStage

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

Stage的初始化参数：在创建一个Stage之前，我们必须知道该Stage需要从多少个Partition读入数据，这个数值直接影响要创建多少个Task。

代码清单 4.16 Stage

```
private[spark] class Stage(
    val id: Int, //Stage的序号，数值越大，
                 //越优先执行，如3,2,1,0
    val rdd: RDD[_], //归属于本Stage的最后一个rdd
    val numTasks: Int, //创建的Task数目，
                       //等于父rdd的输出Partition数目
    val shuffleDep: Option[ShuffleDependency[_, _, _]],
                       //是否存在ShuffleDependency
    val parents: List[Stage], //父Stage列表
    val jobId: Int, //作业Id
    val callSite: CallSite)
```

也就是说在创建Stage的时候，其实已经清楚该Stage需要从多少不同的Partition读入数据，并写出到多少个不同的Partition中，即输入和输出的个数均已明确。

ActiveJob的初始化参数如下。

代码清单 4.17 ActiveJob

```
  private[spark] class ActiveJob(
    val jobId: Int, //每个作业都分配一个唯一的Id
    val finalStage: Stage, //最终的Stage
    val func: (TaskContext, Iterator[_]) => _,
    //作用于最后一个Stage上的函数
    val partitions: Array[Int], //分区列表，
                                //注意这里表示需要从多少个分区读入数据并进行处理
    val callSite: CallSite,
    val listener: JobListener,
    val properties: Properties) {
  val numPartitions = partitions.length
  val finished = Array.fill[Boolean](numPartitions)(false)
  var numFinished = 0
}
```

submitStage处理流程如下所述：

- 所依赖的Stage是否都已经完成，如果没有则先执行所依赖的Stage。
- 如果所有的依赖已经完成，则提交自身所处的Stage。

代码清单 4.18 submitStage

```
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)
        runningStages += stage
      } else {
        for (parent <- missing) {
          submitStage(parent)
        }
        waitingStages += stage
      }
    }
  } else {
    abortStage(stage, "No active job for stage " + stage.id)
  }
}
```

getMissingParentStages通过图的遍历，来找出所依赖的所有父Stage。

代码清单 4.19 getMissingParentStages

```
private def getMissingParentStages(stage: Stage): List[Stage] = {
  val missing = new HashSet[Stage]
  val visited = new HashSet[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[_] =>
              visit(narrowDep.rdd)
          }
        }
      }
    }
    visit(stage.rdd)
    missing.toList
  }
```

Stage的划分是如何确定的呢？其判断的重要依据就是是否存在ShuffleDependency，如果有则创建一个新的Stage。那么又是如何知道是否存在ShuffleDependency的呢？这取决于RDD的转换本身了。以下RDD会返回ShuffleDependency：

- ShuffledRDD
- CoGroupedRDD
- SubtractedRDD

假设今后需要新建一种RDD，就需要明确其Dependency类型，具体就是重载getDependencies函数，如ShuffledRDD中的实现。

代码清单 4.20 getDependencies

```
override def getDependencies: Seq[Dependency[_]] = {
  List(new ShuffleDependency(prev, part, serializer, keyOrdering, aggregator,
  mapSideCombine))
}
```

Stage划分完毕就已经明确了如下内容：

（1） 产生的Stage需要从多少个Partition中读取数据。

（2） 产生的Stage会生成多少Partition。

（3） 产生的Stage是否属于ShuffleMap类型。

确认Partition以决定需要产生多少不同的Task，ShuffleMap类型判断来决定生成的Task类型。在Spark中共分为两种Task，分别是ShuffleMapTask和ResultTask。

4.2.2 Actor Model和Akka

在作业提交及执行期间，Spark集群之间存在大量的消息交互，要想理解这些消息是如何传递的，就有必要简单了解一下Actor Model及Akka的实现。

Actor Model最适合用于解决并发编程问题。每一个Actor都是独立的实体，它们之间没有任何继承关系，所有的交互通过消息传递来完成。

每一个Actor的行为规范起来只有3种：（1） 消息接收，（2） 消息处理，（3） 消息发送。

图 4.4是Actor Model的抽象示意。

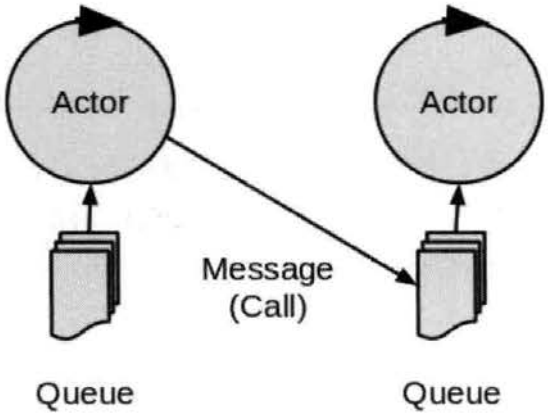


图 4.4 Actor Model

为什么不用共享内存的方式来进行消息交互呢？采用内存共享技术最主要的是会导致并发问题，为了解决状态不一致又要引入锁，由于锁的引入又会导致死锁，同时性能会下降。所以，Erlang语言之父Joe Armstrong不无激进地说“共享内存是程序开发中的万恶之源”。

通过最精简的HelloWorld来看看Scala中的消息发送及消息接收用法。

代码清单 4.21 HelloWorld in Akka

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
```

```
class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  // default Actor constructor
  val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

上述程序中的要点分列如下：

- 创建Actor。
- 消息发送使用!
- Actor中必须实现receive函数来处理接收到的消息。

Actor Model使用最为成功的语言当属Erlang无疑，Akka在很大程度上借鉴了Erlang的设计思想。有关Akka和Erlang的简要对比总结如下：

- 针对虚拟机内部的消息转发，Erlang采用copy-on-send策略，Akka采用共享内存。
- Erlang是针对每个Erlang进程的内存回收，而Akka采用的是JVM GC针对整个JVM进程。
- OTP为Erlang上的应用开发提供了极大便利，而Akka可以让应用开发非常方便地使用已有的Java资源。
- Erlang中的进程调度策略是固定不可设置的，而Akka提供了大量的配置选项。
- Erlang拥有一个杀手级应用热部署（hot code swapping）或者说不宕机进行代码升级，Akka则在这方面落后许多。

4.2.3 任务的创建和分发

Spark将由Executor执行的Task分为ShuffleMapTask和ResultTask两种，可以简单地将其对应于Hadoop中的Map和Reduce。

submitMissingTasks负责创建新的Task。

每个Stage生成Task的时候根据Stage中的isShuffleMap标记确定Task的类型，如果标记为真，则创建ShuffleMapTask；否则创建ResultTask。

属于同一个Stage的Task是可以并发执行的，那么决定同一个Stage要生成多少个Task又是由哪些因素决定的呢？从源码中可以看出Partitions决定了每一个Stage中生成的Task个数，Partitions是事先可以计算出来的。Partition在生成Job的时候就已经明确，如果有疑问，可以返回4.1节复习一下。

需要特别指出的是Task的个数不等于真正并发执行的个数，比如总共生成了8个Task，但只有2个Core，那么需要分成4个批次，每次并发执行两个Task。

代码清单 4.22 submitMissingTask

```
if (stage.isShuffleMap) {
  for (p <- 0 until stage.numPartitions if stage.outputLocs(p) == Nil) {
    val locs = getPreferredLocs(stage.rdd, p)
    //stage.id Stage的序号
    //stage.rdd 属于该Stage的最后一个rdd，即finalRDD
    //stage.shuffleDep  ShuffleDependency
    //p  Partition索引，表示从哪个Partition读取数据
    //locs 最适合的执行位置
    tasks += new ShuffleMapTask(stage.id, stage.rdd, stage.shuffleDep.get, p,
  locs)
  }
} else {
  // This is a final stage; figure out its job's missing partitions
  val job = resultStageToJob(stage)
  for (id <- 0 until job.numPartitions if !job.finished(id)) {
    val partition = job.partitions(id)
    val locs = getPreferredLocs(stage.rdd, partition)
    //stage.id  Stage的序号
    //stage.rdd 属于该Stage的finalRDD
    //job.func  计算函数
    //partition 读入的数据分区索引，表示mapId
    //locs
    //id        输出的分区索引，表示reduceId
    tasks += new ResultTask(stage.id, stage.rdd, job.func, partition, locs, id)
    //可以看出Final Stage一定生成ResultTask，同时finalStage
```

```
    //的输出分区和输入分区数目相等
  }
}
```

从上述一段代码的分析中除了知道根据哪些因素来创建ShuffleMapTask及 ResultTask之外，还需要把握好每种类型的Stage的输入分区和输出分区数目是如何确定的。

一旦任务类型及任务个数确定之后，剩下的工作就是将这些任务派发到各个Executor，由Executor启动相应的线程来执行。这也是从计划到真正执行的过渡阶段。

TaskschedulerImpl发送ReviveOffers消息给DriverActor，DriverActor在收到ReviveOffers消息后，调用makeOffers处理函数。

代码清单 4.23 makeOffers

```
def makeOffers() {
  launchTasks(scheduler.resourceOffers(
  executorHost.toArray.map {case (id, host) => new WorkerOffer(id, host,
  freeCores(id))}))
}
```

makeOffers的处理逻辑如下所述：

（1） 找到空闲的Executor，分发的策略是随机分发，即尽可能将任务平摊到各个Executor。

（2） 如果有空闲的Executor，就将任务列表中的部分任务利用launchTasks发送给指定的Executor。

任务分发策略是随机分发的，即将任务随机发送到各个Executor中。资源分配的工作由resourceOffers函数处理。

代码清单 4.24 resourceOffers

```
def resourceOffers(offers: Seq[WorkerOffer]): Seq[Seq[TaskDescription]] =
    synchronized {
    SparkEnv.set(sc.env)

    // Mark each slave as alive and remember its hostname
    for (o <- offers) {
      executorIdToHost(o.executorId) = o.host
      if (!executorsByHost.contains(o.host)) {
        executorsByHost(o.host) = new HashSet[String]()
        executorAdded(o.executorId, o.host)
```

```
    }
  }

  // Randomly shuffle offers to avoid always placing tasks on the same set of
  // workers.
  val shuffledOffers = Random.shuffle(offers)
  // Build a list of tasks to assign to each worker.
  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))
  }

  // Take each TaskSet in our scheduling order, and then offer it each
  // node in increasing order of locality levels so that it gets a chance
  // to launch local tasks on all of them.
  var launchedTask = false
  for (taskSet <- sortedTaskSets; maxLocality <- TaskLocality.values) {
    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
            activeExecutorIds += 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
  }
```

SchedulerBackend负责将新创建的Task分发给Executor，从launchTasks的代码中可以看出在发送LaunchTasks指令之前需要将TaskDescription序列化。

先来看一下TaskDescription的定义。

代码清单 4.25 TaskDescription

```
private[spark] class TaskDescription(
    val taskId: Long,
    val executorId: String,
    val name: String,
    val index: Int,    // Index within this task's TaskSet
    _serializedTask: ByteBuffer)
  extends Serializable {

  // Because ByteBuffers are not serializable, wrap the task in a
  // SerializableBuffer
  private val buffer = new SerializableBuffer(_serializedTask)

  def serializedTask: ByteBuffer = buffer.value

  override def toString: String = "TaskDescription(TID=%d, index=%d)".format(
    taskId, index)
}
```

代码清单 4.26 launchTasks

```
def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
    val ser = SparkEnv.get.closureSerializer.newInstance()
    //序列化TaskDescription
    val serializedTask = ser.serialize(task)
    if (serializedTask.limit >= akkaFrameSize - 1024) {
      val taskSetId = scheduler.taskIdToTaskSetId(task.taskId)
      scheduler.activeTaskSets.get(taskSetId).foreach { taskSet =>
        try {
          var msg = "Serialized task %s:%d was %d bytes which " +
          "exceeds spark.akka.frameSize (%d bytes). " +
          "Consider using broadcast variables for large values."
          msg = msg.format(task.taskId, task.index, serializedTask.limit,
akkaFrameSize)
          taskSet.abort(msg)
        } catch {
          case e: Exception => logError("Exception in error callback", e)
        }
      }
    }
    else {
      freeCores(task.executorId) -= scheduler.CPUS_PER_TASK
      executorActor(task.executorId) ! LaunchTask(new SerializableBuffer(
serializedTask))
    }}}
```

在执行Task的时候所要依赖的文件和Jar包是在哪里指定的呢？回过头看一下TaskSetManager.resourceOffer中最为关键的一句就知道答案了。

代码清单 4.27 serializedTask

```
val serializedTask = Task.serializeWithDependencies(
task, sched.sc.addedFiles, sched.sc.addedJars, ser)
```

4.2.4 任务执行

LaunchTask消息被Executor接收，Executor会使用launchTask对该消息进行处理。

这里需要注意的是如果Executor没有注册到Driver，即便接收到LaunchTask指令，也不会做任何处理。

代码清单 4.28 CoarseGrainedSchedulerBackend.launchTasks

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

在TaskRunner的反序列化过程。

代码清单 4.29 TaskRunner.run

```
    SparkEnv.set(env)
    Accumulators.clear()
    val (taskFiles, taskJars, taskBytes) = Task.deserializeWithDependencies(
    serializedTask)
    updateDependencies(taskFiles, taskJars)
    task = ser.deserialize[Task[Any]](taskBytes,
                                Thread.currentThread.getContextClassLoader)
```

解决依赖性问题：updateDependencies。

代码清单 4.30 updateDependecies

```
private def updateDependencies(newFiles: HashMap[String, Long], newJars: HashMap[
    String, Long]) {
    synchronized {
      // Fetch missing dependencies
      for ((name, timestamp) <- newFiles if currentFiles.getOrElse(name, -1L) <
    timestamp) {
        logInfo("Fetching " + name + " with timestamp " + timestamp)
```

```
        Utils.fetchFile(name, new File(SparkFiles.getRootDirectory), conf, env.
    securityManager)
        currentFiles(name) = timestamp
      }
      for ((name, timestamp) <- newJars if currentJars.getOrElse(name, -1L) <
    timestamp) {
        logInfo("Fetching " + name + " with timestamp " + timestamp)
        Utils.fetchFile(name, new File(SparkFiles.getRootDirectory), conf, env.
    securityManager)
        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)
        }
      }
    }
  }
```

Utils.fetchFile从HttpFileServer上获取所依赖的文件，依赖文件上传到HttpFileServer是发生在Submit的时候。支持的文件存储方式如下：

- HttpFileServer
- HDFS
- 本地文件

Utils.fetchFile在开始获取文件之前，首先要调用createTempDir来创建存储文件的临时目录。目录名通过java.io.tmpdir指定，默认是在/tmp目录下。

代码清单 4.31 createTempDir

```
def createTempDir(root: String = System.getProperty("java.io.tmpdir")): File = {
    var attempts = 0
    val maxAttempts = 10
    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: IOException => ; }
  }

  registerShutdownDeleteDir(dir)

  // Add a shutdown hook to delete the temp dir when the JVM exits
  Runtime.getRuntime.addShutdownHook(new Thread("delete Spark temp dir " + dir)
  {
    override def run() {
      // Attempt to delete if some patch which is parent of this is not already
      // registered.
      if (! hasRootAsShutdownDeleteDir(dir)) Utils.deleteRecursively(dir)
    }
  })
  dir
}
```

Shuffle Task

TaskRunner会启动一个新的线程，这没有问题。问题是如何在run中去调用用户自己定义的处理函数呢？也就是说作用于RDD上的Operation是如何真正起作用的呢？

```
TaskRunner.run
    └ Task.run
        └ Task.runTask
            └ RDD.iterator
                └ RDD.computeOrReadCheckpoint
                    └ RDD.compute
```

下面来看一看ShuffleMapTask中的runTask函数的实现，注意要返回MapStatus。

代码清单 4.32　ShuffleMapTask中的runTask函数的实现

```
override def runTask(context: TaskContext): MapStatus = {
  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
  )
    for (elem <- rdd.iterator(split, context)) {
      writer.write(elem.asInstanceOf[Product2[Any, Any]])
    }
    writer.stop(success = true).get
  } catch {
    case e: Exception =>
      if (writer != null) {
        writer.stop(success = false)
      }
      throw e
  } finally {
    context.executeOnCompleteCallbacks()
  }
}
```

Iterator很重要，看一看每个RDD中Iterator的定义。

代码清单 4.33　RDD.iterator

```
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)
    }
  }
```

或许当看到RDD.compute函数定义时，还是觉着f没有被调用。下面以MappedRDD的compute定义为例。

代码清单 4.34 MappedRDD.compute

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

注意，这里最容易产生错觉的地方就是map函数，这里的map不是RDD中的map，而是Scala中定义的Iterator的成员函数map。

函数调用的堆栈输出如下。

代码清单 4.35 调用堆栈输出

```
org.apache.spark.rdd.HadoopRDD.compute(HadoopRDD.scala:149)
org.apache.spark.rdd.HadoopRDD.compute(HadoopRDD.scala:64)
org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:241)
org.apache.spark.rdd.RDD.iterator(RDD.scala:232)
org.apache.spark.rdd.MappedRDD.compute(MappedRDD.scala:31)
org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:241)
org.apache.spark.rdd.RDD.iterator(RDD.scala:232)
org.apache.spark.rdd.FlatMappedRDD.compute(FlatMappedRDD.scala:33)
org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:241)
org.apache.spark.rdd.RDD.iterator(RDD.scala:232)
org.apache.spark.rdd.MappedRDD.compute(MappedRDD.scala:31)
org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:241)
org.apache.spark.rdd.RDD.iterator(RDD.scala:232)
org.apache.spark.rdd.MapPartitionsRDD.compute(MapPartitionsRDD.scala:34)
org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:241)
org.apache.spark.rdd.RDD.iterator(RDD.scala:232)
```

```
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:161)
org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:102)
org.apache.spark.scheduler.Task.run(Task.scala:53)
org.apache.spark.executor.Executor$TaskRunner$$anonfun$run$1.apply$mcV$sp(
    Executor.scala:211)
```

Reduce Task

compute的计算过程对于ShuffleMapTask比较复杂，绕的圈圈比较多；而对于ResultTask则就直接许多。

与ShuffleMapTask不同，ResultTask的runTask没有明确返回值，在后续的handleTaskCompletion函数中可以进一步发现这样处理的原因。

代码清单 4.36 runTask

```
override def runTask(context: TaskContext): U = {
    metrics = Some(context.taskMetrics)
    try {
      func(context, rdd.iterator(split, context))
    } finally {
      context.executeOnCompleteCallbacks()
    }
  }
```

结果返回

Task在执行时，会有大量的数据交互，这些数据可以分成3种不同类型：

- 状态相关，如StatusUpdate。
- 中间结果。
- 计量相关的数据 Metrics Data。

图 4.5展示了哪些节点会与Task进行这些数据交互。

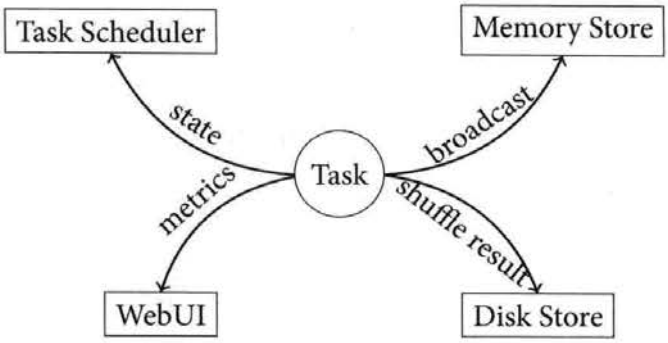


图 4.5 Task生成的数据分类

Task执行完毕，通过statusUpdate通知ExecuteBackend，结果保存在DirectTaskResult。

代码清单 4.37 Task Result

```
val directResult = new DirectTaskResult(valueBytes, accumUpdates,
task.metrics.getOrElse(null))
val serializedDirectResult = ser.serialize(directResult)
logInfo("Serialized size of result for " + taskId + " is " +
serializedDirectResult.limit)
val serializedResult = {
  if (serializedDirectResult.limit >= akkaFrameSize - 1024) {
    logInfo("Storing result for " + taskId + " in local BlockManager")
    val blockId = TaskResultBlockId(taskId)
    env.blockManager.putBytes(
    blockId, serializedDirectResult, StorageLevel.MEMORY_AND_DISK_SER)
    ser.serialize(new IndirectTaskResult[Any](blockId))
  } else {
    logInfo("Sending result for " + taskId + " directly to driver")
    serializedDirectResult
  }
}
execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
```

ShuffleMapTask和ResultTask返回的结果中有哪些不同？

ShuffleMapTask需要返回MapStatus，而ResultTask只需要告知是否已经成功完成执行。

SchedulerBack接收到Executor发送过来的StatusUpdate。

代码清单 4.38 Receive StatusUpdate

```
case StatusUpdate(executorId, taskId, state, data) =>
scheduler.statusUpdate(taskId, state, data.value)
if (TaskState.isFinished(state)) {
  if (executorActor.contains(executorId)) {
    freeCores(executorId) += scheduler.CPUS_PER_TASK
    makeOffers(executorId)
  } else {
    // Ignoring the update since we don't know about the executor.
    val msg = "Ignored task status update (%d state %s) from unknown executor %s
   with ID %s"
    logWarning(msg.format(taskId, state, sender, executorId))
  }
}
```

SchedulerBackend接收到StatusUpdate之后做如下判断：如果任务已经成功处理，则将其从监视的列表中删除。如果整个作业中的所有任务都已经完成，则将占用的资源释放。

TaskSchedulerImpl将当前顺利完成的任务放入完成队列，同时取出下一个等待运行的Task。

DAGScheduler中的handleTaskCompletion，针对ResultTask和ShuffleMapTask区别对待结果。

如果ReduceTask执行成功，DAGScheduler会发出TaskSucceed来通知对整个作业执行情况感兴趣的监听者，如JobWaiter。

JobWaiter中的taskSucceeded函数会根据当前已经完成的任务数之和是否等于事先提交的任务总数来判定整个作业执行结束与否。

如果判定结果是任务顺利完成，则会置_jobFinished标记为真，同时通知相应的等待线程状态发生变化。

代码清单 4.39 JobWaiter.taskSucceeded

```
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()
    }
  }
```

taskSucceeded发出的通知会被awaitResult函数接收到。如果整个作业完成，则awaitResult结束等待。

代码清单 4.40 awaitResult

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

讲述到这里的时候，就重新回到runJob的处理流程。至此讲述完了作业的创建与执行。

代码清单 4.41 DAGScheduler.runJob

```
  def runJob[T, U: ClassTag](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      callSite: String,
      allowLocal: Boolean,
      resultHandler: (Int, U) => Unit,
      properties: Properties = null)
  {
    val waiter = submitJob(rdd, func, partitions, callSite, allowLocal,
    resultHandler, properties)
    waiter.awaitResult() match {
      case JobSucceeded => {}
      case JobFailed(exception: Exception) =>
        logInfo("Failed to run " + callSite)
```

```
            throw exception
        }
    }
```

一个Spark Application可以由多个作业（Job）组成，如果Application退出，情况会怎样呢？这些会放到第5章中应用结束时的清理流程来做详细描述。

4.2.5 Checkpoint和Cache

出于容错及效率方面的考虑，有时候需要将计算的中间结果进行持久化保存，原因是为了后面再次利用到该RDD的时候不需要重新计算。

中间结果的存储有两种方式：一种是Checkpoint，另一种是Cache。

Checkpoint将计算结果写入HDFS文件系统中，但不会保存RDD Linerage。Cache则不同，Cache会将数据缓存到内存，如果内存不足则会写入磁盘；同时会将RDD Lineage也保存下来。

Checkpoing分成两种不同类型：

（1） Data Checkpoint

（2） Metadata Checkpoint

有关Metadata Checkpoint会在6.1节中做详细介绍。

4.2.6 WebUI和Metrics

当用户在使用Spark时，无论是对于Spark Cluster的运行情况还是Spark Application运行时的一些细节，我们都希望能够有一个直观的手段可以观察到。

这一部分工作在Spark中由WebUI及Metrics来完成，它们各自的目标不同，需要分开阐述。

WebUI

先看一幅图感受一下Spark WebUI。假设当前已经在本机运行Standalone Cluster模式，输入http://127.0.0.1:8080，将会看到如图 4.6所示的页面。

Driver Application默认会打开4040端口进行Http监听，可以看到Application相关的详细信息。

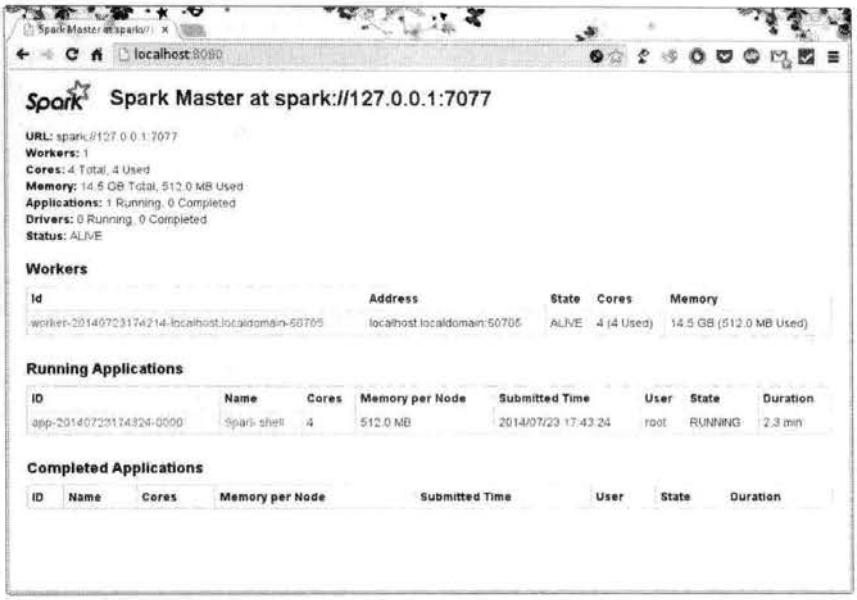


图 4.6 Spark Master页面

图4.7显示了每个Stage的详细信息，如果想在一个Driver Application中创建多个SparkContext，就必须为每个SparkContext指定不同的WebUI Port；否则会报地址已经被占用的错误。WebUI使用的端口通过配置项spark.ui.port来指定。

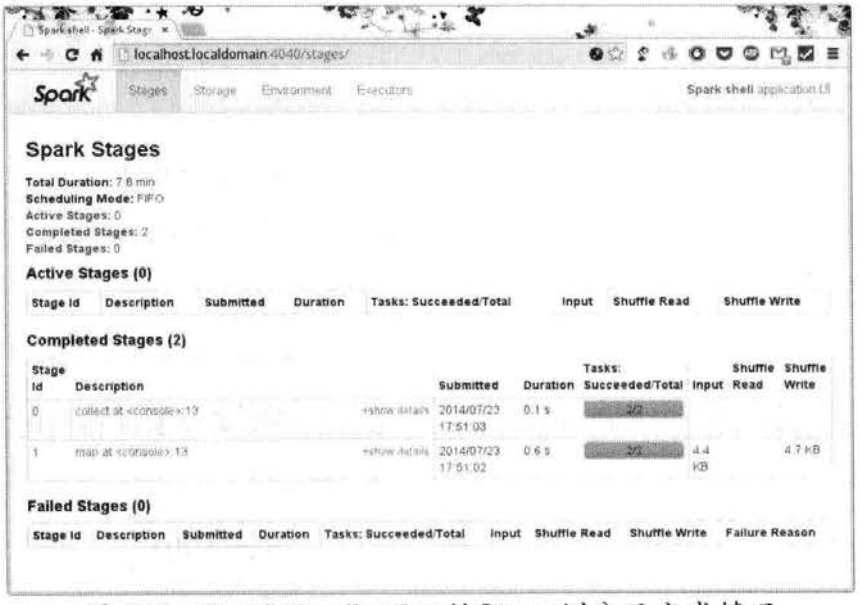


图 4.7 Spark Application的Stage划分及完成情况

单击某一个Stage，会进入显示Stage中各个Task执行详细信息的页面，如图 4.8所示。

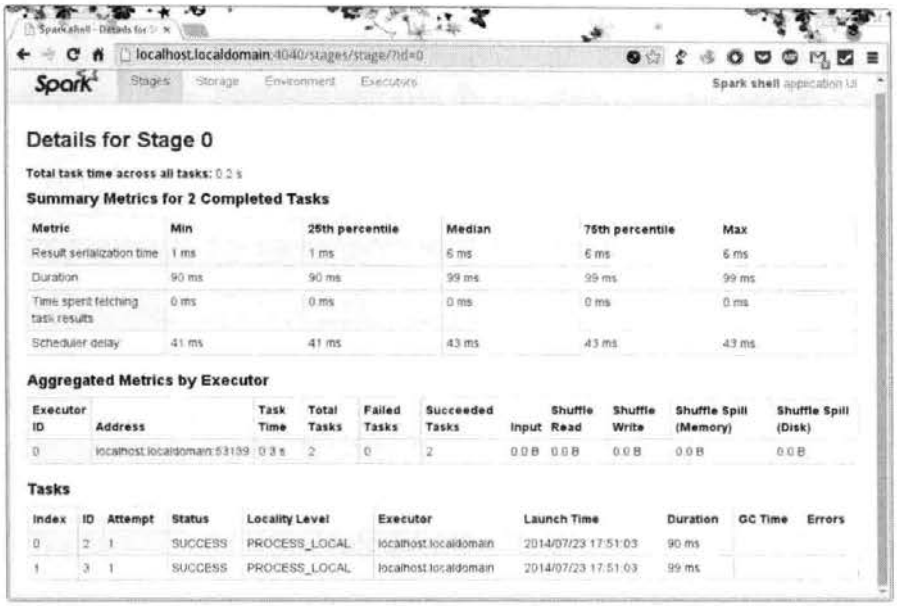


图 4.8 Stage执行情况汇总

本节要讨论的重点是Http Server是如何启动的，页面中的数据是从哪里获取到的？

Spark中用到的Http Server是Jetty，Jetty采用Java编写，是非常轻巧的Servlet Engine和Http Server。能够嵌入到用户程序中执行，不用像Tomcat或JBoss那样需要自己独立的JVM进程。

SparkUI在SparkContext初始化的时候创建。

代码清单 4.42 在SparkContext创建SparkUI

```
// Initialize the Spark UI, registering all associated listeners
private[spark] val ui = new SparkUI(this)
ui.bind()
```

initialize的主要工作是注册页面处理句柄，WebUI的子类需要实现自己的initialize函数（见图4.9）。

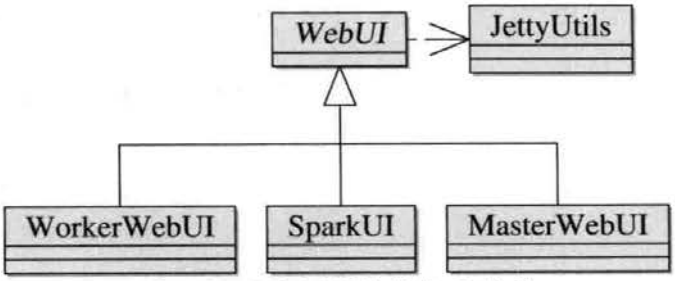


图 4.9 WebUI的类体系结构

bind将真正启动JettyServer。

代码清单 4.43 WebUI.bind

```
def bind() {
    assert(!serverInfo.isDefined, "Attempted to bind %s more than once!".format(
    className))
    try {
      //启动JettyServer
      serverInfo = Some(startJettyServer("0.0.0.0", port, handlers, conf))
      logInfo("Started %s at http://%s:%d".format(className, publicHostName,
    boundPort))
    } catch {
      case e: Exception =>
        logError("Failed to bind %s".format(className), e)
        System.exit(1)
    }
  }
```

在startJettyServer函数中将JettyServer运行起来的关键处理函数是connect。

代码清单 4.44 connect

```
    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()
      } match {
        case s: Success[_] =>
          (server, server.getConnectors.head.getLocalPort)
        case f: Failure[_] =>
          val nextPort = (currentPort + 1) % 65536
          server.stop()
          pool.stop()
```

```
        val msg = s"Failed to create UI on port $currentPort. Trying again on
    port $nextPort."
        if (f.toString.contains("Address already in use")) {
          logWarning(s"$msg - $f")
        } else {
          logError(msg, f.exception)
        }
        connect(nextPort)
      }
    }
```

页面中的数据是如何获取的呢？这就要归功于SparkListener了，典型的观察者设计模式。当有与Stage及Task相关的事件发生时，这些Listener都将收到通知，并进行数据更新。

需要指出的是，数据尽管得以自动更新，但页面并没有更新，仍需要手工刷新才能得到最新的数据。

图 4.10显示的是SparkUI中注册了哪些SparkListener子类。

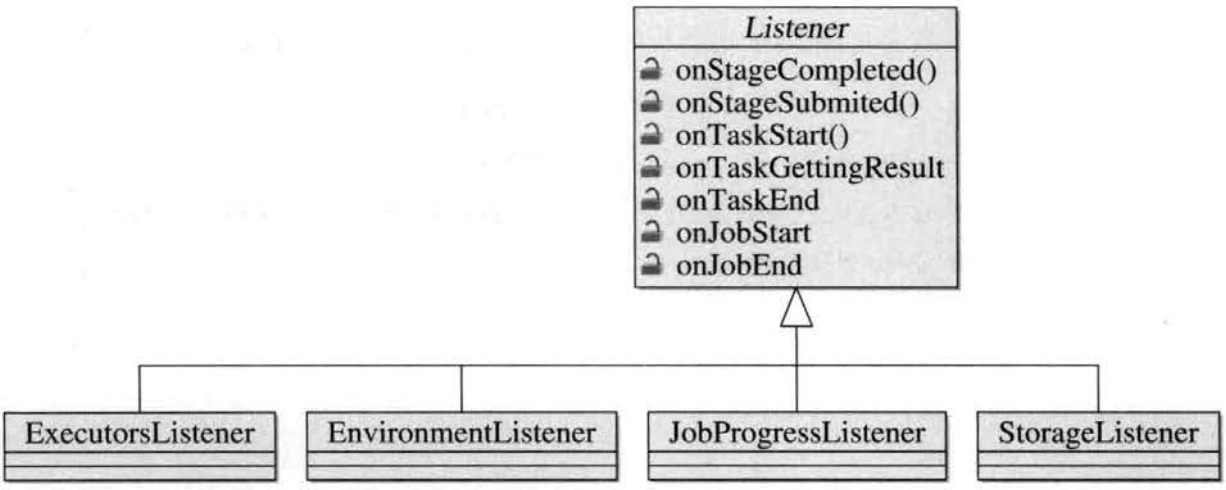


图 4.10 SparkListener的类继承体系

下面来看一看这些子类是在什么时候注册进去的。注意研究一下SparkUI.initialize函数就会发现，listener是在此时被注入的。

代码清单 4.45 SparkUI.initialize

```
def initialize() {
  listenerBus.addListener(storageStatusListener)
  val jobProgressTab = new JobProgressTab(this)
```

```
    attachTab(jobProgressTab)
    attachTab(new StorageTab(this))
    attachTab(new EnvironmentTab(this))
    attachTab(new ExecutorsTab(this))
    attachHandler(createStaticHandler(SparkUI.STATIC_RESOURCE_DIR, "/static"))
    attachHandler(createRedirectHandler("/", "/stages", basePath = basePath))
    attachHandler(
      createRedirectHandler("/stages/stage/kill", "/stages", jobProgressTab.
    handleKillRequest))
    if (live) {
      sc.env.metricsSystem.getServletHandlers.foreach(attachHandler)
    }
  }
```

下面举一个实际例子来看看Notifier发送Event的时刻，比如有任务提交的时候，resourceOffer → taskStarted → handleBeginEvent。

代码清单 4.46 handleBeginEvent

```
  private[scheduler] def handleBeginEvent(task: Task[_], taskInfo: TaskInfo) {
    listenerBus.post(SparkListenerTaskStart(task.stageId, taskInfo))
    submitWaitingStages()
  }
```

post其实是向listenerBus的消息队列中添加一个消息，真正将消息发送出去的是另一个处理线程listenerThread。

代码清单 4.47 listenerThread.run

```
  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)
      }
    }
  }
```

Option(event).foreach(postToAll)负责将事件通知给各个Observer。

postToAll的函数实现如下。

代码清单 4.48 postToAll

```
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 SparkListenerShutdown =>
    }
  }
```

Metrics

在系统设计中，测量模块是不可或缺的组成部分。通过这些测量数据来感知系统的运行情况。

在Spark中，测量模块由MetricsSystem来担任。MetricsSystem中有3个重要的概念，分述如下。

- **Instance**：表示谁在使用MetricsSystem。目前已知的有Master、Worker、Executor和Client Driver会创建MetricsSystem用以测量。
- **Source**：表示数据源，从哪里获取数据。
- **Sinks**：数据目的地，将从Source获取的数据发送到哪里。Spark目前支持将测量数据保存或发送到如下目的地。
  - **ConsoleSink**——输出到Console。
  - **CSVSink**——定期保存为CSV文件。
  - **JmxSink**——注册到JMX，以通过JMXConsole来查看。
  - **MetricsServlet**——在SparkUI中添加MetricsServlet，用以查看Task运行时的测量数据。
  - **GraphiteSink**——发送给Graphite以对整个系统（不仅仅包括Spark）进行监控。

下面从MetricsSystem的创建、数据源的添加、数据更新与发送几个方面来跟踪一下源码。

MetricsSystem依赖于由codahale提供的第三方库Metrics，可以在metrics.codahale.com找到更为详细的介绍。

以Driver Application为例，Driver Application首先会初始化SparkContext，在SparkContext的初始化过程中就会创建MetricsSystem。具体调用关系如下：SparkContext.init→ SparkEnv.init→ MetricsSystem.createMetricsSystem。

注册数据源，继续以SparkContext为例。

代码清单 4.49 add source

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

private def initDriverMetrics() {
```

```
    SparkEnv.get.metricsSystem.registerSource(dagSchedulerSource)
    SparkEnv.get.metricsSystem.registerSource(blockManagerSource)
  }

  initDriverMetrics()
```

数据读取由Sink来完成，在Spark中创建的Sink子类如图 4.11所示。

读取最新的数据，以CsvSink为例，最主要的就是创建CsvReporter，启动之后会定期更新最近的数据到Console。不同类型的Sink所使用的Reporter是不一样的。

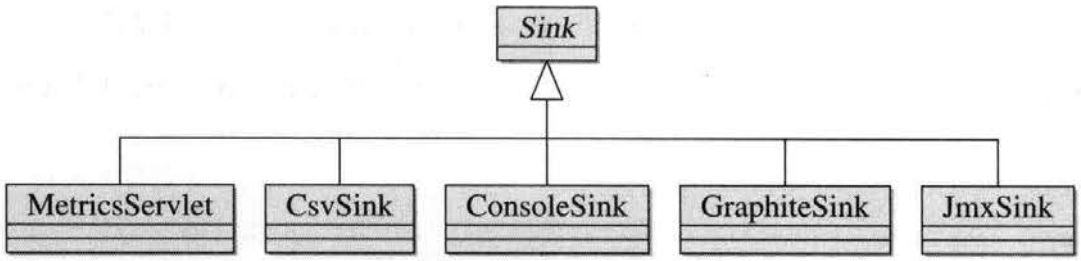


图 4.11 Sink类体系结构

代码清单 4.50 start

```
val reporter: CsvReporter = CsvReporter.forRegistry(registry)
    .formatFor(Locale.US)
    .convertDurationsTo(TimeUnit.MILLISECONDS)
    .convertRatesTo(TimeUnit.SECONDS)
    .build(new File(pollDir))

override def start() {
  reporter.start(pollPeriod, pollUnit)
}
```

Spark 中关于 Metrics 子系统的配置文件详见 conf/metrics.properties。默认的 Sink 是 MetricsServlet，在任务提交执行之后，输入http://127.0.0.1:4040/metrics/json会得到以JSON格式保存的Metrics信息。

4.3 存储机制

4.3.1 Shuffle结果的写入和读取

通过上面的分析知道，WordCount这个Job在最终提交之后，被DAGScheduler分为两个Stage：第一个Stage是ShuffleMaptask，第二个Stage是ResultTask。

ShuffleMapTask会产生临时计算结果，这些数据会被ResulTask作为输入而读取。图 4.12是该过程的形象化展示。

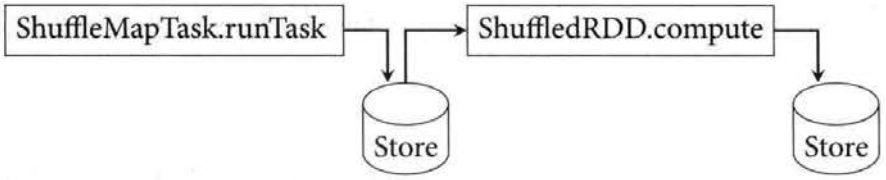


图 4.12 中间结果的保存和读取

那么ShuffleMapTask的计算结果是如何被ResultTask取得的呢？这个过程简述如下。

（1） ShffuleMapTask将计算的状态（注意不是具体的计算数值）包装为MapStatus返回给DAGScheduler。

（2） DAGScheduler将MapStatus保存到MapOutputTrackerMaster中。

（3） ResultTask在调用到ShuffleRDD时会利用BlockStoreShuffleFetcher的fetch方法去获取数据。

（a） 第一件事就是咨询MapOutputTrackerMaster所要取的数据的location。

（b） 根据返回的结果调用BlockManager.getMultiple获取真正的数据。

MapStatus的数据结构如图4.13所示。理解好这幅图能对Shuffle的数据写入和读取有深刻掌握。

- 每一个ShuffleMapTask都会用一个MapStatus来保存计算结果。
- MapStatus由blockmanagerid和byteSize构成，blockmanagerid表示这些计算的中间结果实际数据在哪个BlockManager，bytesize表示不同reduceid所要读取的数据的大小。

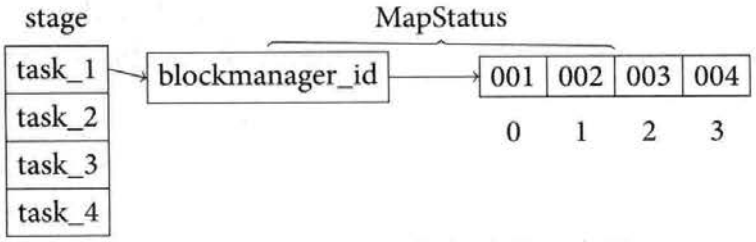


图 4.13 MapStatus数据结构示意图

Shuffle结果写入

下面先来看数据写入过程：

```
ShuffleMapTask.runTask
  HashShuffleWriter.write
    BlockObjectWriter.write
```

HashShuffleWriter.write中主要处理两件事：

（1） 判断是否需要进行聚合，比如<hello,1>和<hello,1>都要写入的话，那么先生成<hello,2>，然后再进行后续的写入工作。

（2） 利用Partitioner函数来决定<k,val>写入哪一个文件中。

结果写入时的一个主要问题就是：已经知道shuffle_id、map_id和要写入的elem，如何找到对应的写入文件。

每一个临时文件由三元组(shuffle_id,map_id,reduce_id)来决定。当前已经知道了两个，还剩下一个reduce_id待确定。

reduce_id是使用Partitioner计算出来的结果，输入的是elem的键值。

代码清单 4.51 HashShuffleWriter.write

```
override def write(records: Iterator[_ <: Product2[K, V]]): Unit = {
    val iter = if (dep.aggregator.isDefined) {
      if (dep.mapSideCombine) {
        //汇聚工作，reduceByKey是一分为二的，一部分在
        //shufflemaptask中进行聚合，
        //另一部分在resulttask中聚合
        dep.aggregator.get.combineValuesByKey(records, context)
      } else {
        records
      }
    } else if (dep.aggregator.isEmpty && dep.mapSideCombine) {
      throw new IllegalStateException("Aggregator is empty for map-side combine")
    } else {
      records
    }

    for (elem <- iter) {
```

```
      //elem是类似于<k,val>的键值对，以k为参数用Partitioner计算其对应的hash值，
      //从而选取相应的writer来写入整个elem。
      //那么这里的Partitioner是什么呢？默认是HashPartitioner
      val bucketId = dep.partitioner.getPartition(elem._1)
      shuffle.writers(bucketId).write(elem)
    }
  }
```

在HashShuffleWriter.write中使用到的Shuffle由ShuffleBlockManager中的forMapTask函数生成，注意forMapTask产生Writer的代码逻辑。

每个Writer分配一下文件，文件名由三元组(shuffle_id,map_id,reduce_id)组成。如果知道了这个三元组，就可以找到对应的文件。

如果Consolidation开关没有打开，那么在一个Task中，有多少个输出的Partition就会有多少个中间文件。

代码清单 4.52 forMapTask

```
val writers: Array[BlockObjectWriter] = if (consolidateShuffleFiles) {
        fileGroup = getUnusedFileGroup()
        Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
          val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
          blockManager.getDiskWriter(blockId, fileGroup(bucketId), serializer,
    bufferSize)
        }
      } else {
        Array.tabulate[BlockObjectWriter](numBuckets) { bucketId =>
          val blockId = ShuffleBlockId(shuffleId, mapId, bucketId)
          val blockFile = blockManager.diskBlockManager.getFile(blockId)
          // Because of previous failures, the shuffle file may already exist on
          // this machine.
          // If so, remove it.
          if (blockFile.exists) {
            if (blockFile.delete()) {
              logInfo(s"Removed existing shuffle file $blockFile")
            } else {
              logWarning(s"Failed to remove existing shuffle file $blockFile")
            }
```

```
        }
        blockManager.getDiskWriter(blockId, blockFile, serializer, bufferSize)
      }
    }
```

getFile负责将三元组(shuffle_id,map_id,reduce_id)映射到文件名。

代码清单 4.53 getFile

```
def getFile(filename: String): File = {
    // Figure out which local directory it hashes to, and which subdirectory
    // in that
    val hash = Utils.nonNegativeHash(filename)
    val dirId = hash % localDirs.length
    val subDirId = (hash / localDirs.length) % subDirsPerLocalDir

    // Create the subdirectory if it doesn't already exist
    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)
  }

  def getFile(blockId: BlockId): File = getFile(blockId.name)
```

目录结构如下所述。

代码清单 4.54 shuffle文件的目录结构

```
/tmp/spark-local-20140723092540-7f24
/tmp/spark-local-20140723092540-7f24/0d
/tmp/spark-local-20140723092540-7f24/0d/shuffle_0_0_1
/tmp/spark-local-20140723092540-7f24/0d/shuffle_0_1_0
/tmp/spark-local-20140723092540-7f24/0c
/tmp/spark-local-20140723092540-7f24/0c/shuffle_0_0_0
/tmp/spark-local-20140723092540-7f24/0e
/tmp/spark-local-20140723092540-7f24/0e/shuffle_0_1_1
```

DiskBlockObjectWriter负责将数据真正写入磁盘。

代码清单 4.55 DiskBlockObjectWriter.write

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

Shuffle结果读取

ShuffledRDD的compute函数是读取ShuffleMapTask计算结果的触发点。

代码清单 4.56 ShuffledRDD.compute

```
override def compute(split: Partition, context: TaskContext): Iterator[P] = {
  val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]
  SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.
  index + 1, context)
    .read()
    .asInstanceOf[Iterator[P]]
}
```

shuffleManager.getReader返回的是HashShuffleReader，所以看一看HashShuffleReader中的read函数的具体实现。

代码清单 4.57 HashShuffleReader.read

```
override def read(): Iterator[Product2[K, C]] = {
  val iter = BlockStoreShuffleFetcher.fetch(handle.shuffleId, startPartition,
  context,
    Serializer.getSerializer(dep.serializer))

  if (dep.aggregator.isDefined) {
    if (dep.mapSideCombine) {
      new InterruptibleIterator(context, dep.aggregator.get.
  combineCombinersByKey(iter, context))
    } else {
      new InterruptibleIterator(context, dep.aggregator.get.combineValuesByKey(
  iter, context))
    }
  } else if (dep.aggregator.isEmpty && dep.mapSideCombine) {
    throw new IllegalStateException("Aggregator is empty for map-side combine")
  } else {
    iter
  }
}
```

一路辗转，终于来到了读取过程中非常关键的所在——BlockStoreShuffleFetcher。

BlockStoreShuffleFetcher需要回答如下问题：

- 所要获取的mapid的MapStatus的内容是什么。
- 如何根据获得的MapStatus去相应的BlockManager获取具体的数据。

BlockStoreShuffleFetcher的fetch函数伪码如下。

代码清单 4.58 fetch函数片段

```
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)))
}
val blockFetcherItr = blockManager.getMultiple(blocksByAddress, serializer)
val itr = blockFetcherItr.flatMap(unpackBlock)
```

一个ShuffleMapTask会生成一个MapStatus，MapStatus中含有当前 ShuffleMapTask产生的数据落到各个Partition中的大小。如果大小为0，则表示该分区没有数据产生。

索引即reduceId。如果array(0) == 0，就表示上一个ShuffleMapTask中生成的数据中没有任意的内容可以作为reduceId为0的ResultTask的输入。如果不能理解，可以返回仔细看一下MapStatus的结构图。

每一个分区中的数据大小是用一个byte来表示的。这里就有问题了：一个byte最多只能表示255，如何表示更大的size呢？这里使用到了巧妙的转换，使用1.1作为对数底，可以将2^8，转换为1.1^{256}。

compressSize和decompressSize的作用，将数据的大小用另一种进制来表示，这样可以让表达的空间从0至255转换到0至35 903 328 256。单个存储的大小可以高达近35GB。

代码清单 4.59 compressSize

```
def compressSize(size: Long): Byte = {
    if (size == 0) {
      0
    } else if (size <= 1L) {
      1
    } else {
      math.min(255, math.ceil(math.log(size) / math.log(LOG_BASE)).toInt).toByte
    }
```

```
  }

  /**
   * Decompress an 8-bit encoded block size, using the reverse operation of
    compressSize.
   */
  def decompressSize(compressedSize: Byte): Long = {
    if (compressedSize == 0) {
      0
    } else {
      math.pow(LOG_BASE, compressedSize & 0xFF).toLong
    }
  }
```

Shuffle_id唯一标识了一个Job中的Stage，这一Stage是作为ReduceTask所在Stage的直接上游。需要遍历该Stage中每一个Task产生的mapStatus来获知是否有当前ResultTask需要读取的数据。

去除掉size为0的，就能够完整拼凑出三元组(shuffle_id,map_id,reduce_id)，同时知道了该文件所属的blockmanagerid。数据获取的过程就很简单了。

BlockManager.getMultiple 用于读取 BlockManager 中的数据，根据配置确定生成 tNettyBlockFetcherIterator 还是 BasicBlockFetcherIterator。

如果所要获取的文件落在本地，则调用getLocal读取；否则发送请求到远端BlockManager。看一下BlockFetcherIterator的initialize函数。

代码清单 4.60 BlockFetcherIterator.initialize

```
  override def initialize() {
    // Split local and remote blocks.
    val remoteRequests = splitLocalRemoteBlocks()
    // Add the remote requests into our queue in a random order
    fetchRequests ++= Utils.randomize(remoteRequests)

    // Send out initial requests for blocks, up to our maxBytesInFlight
    while (!fetchRequests.isEmpty &&
      (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <=
  maxBytesInFlight)) {
      sendRequest(fetchRequests.dequeue())
```

```
    }

    val numFetches = remoteRequests.size - fetchRequests.size
    logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs
  (startTime))

    // Get Local Blocks
    startTime = System.currentTimeMillis
    getLocalBlocks()
    logDebug("Got local blocks in " + Utils.getUsedTimeMs(startTime) + " ms")
}
```

Spark对内存的要求较高，分析到这里，我们基本上可能已经明了内存都用到了哪里。

在ShuffleMaptask和ResultTask中，由于需要先将计算结果保存在内存，然后再写入到磁盘，由此，如果每一个数据分区的数据很大，则会消耗大量的内存。具体体现在以下几个地方：

- 每个Writer开启100KB的缓存。
- Records会占用大量内存。
- 在ResultTask的combine阶段，利用Hashmap来缓存数据。如果读取的数据量很大，或者分区很多，都有可能导致内存不足错误的发生。

采用伪分布模式，做两个小实验来看看生成的Shuffle文件。

代码清单 4.61 以local-cluster方式运行spark-shell

```
MASTER=local-cluster[2,2,512] bin/spark-shell
```

进入spark-shell之后，输入以下内容。

代码清单 4.62 WordCount

```
val rawFile = sc.textFile("README.md")
val splitLine = rawFile.faltMap(l => l.split(" "))
val words = splitLine.map(w => (w,1))
val wc = words.reduceByKey(_ + _)
wc.collect
```

生成的文件在/tmp目录之下，文件名如下所示。

```
drwxr-xr-x  4 root root     80 7月  16 14:52 spark-local-20140716144947-a511
drwxr-xr-x 12 root root    240 7月  16 14:52 spark-local-20140716144951-5cda
```

```
drwxr-xr-x 14 root root    280 7月  16 14:52 spark-local-20140716144951-c6f6
```

4.3.2 Memory Store

在4.2节中讲到可以将计算结果显式地缓存起来，优先放到内存。那么如果想取得缓存里的数据，具体的过程又是啥样子？

同刚才所述的Shuffle结果的存储与获取相比，要简单得多。先来看一看如图4.14所示存储子系统的整体框架。

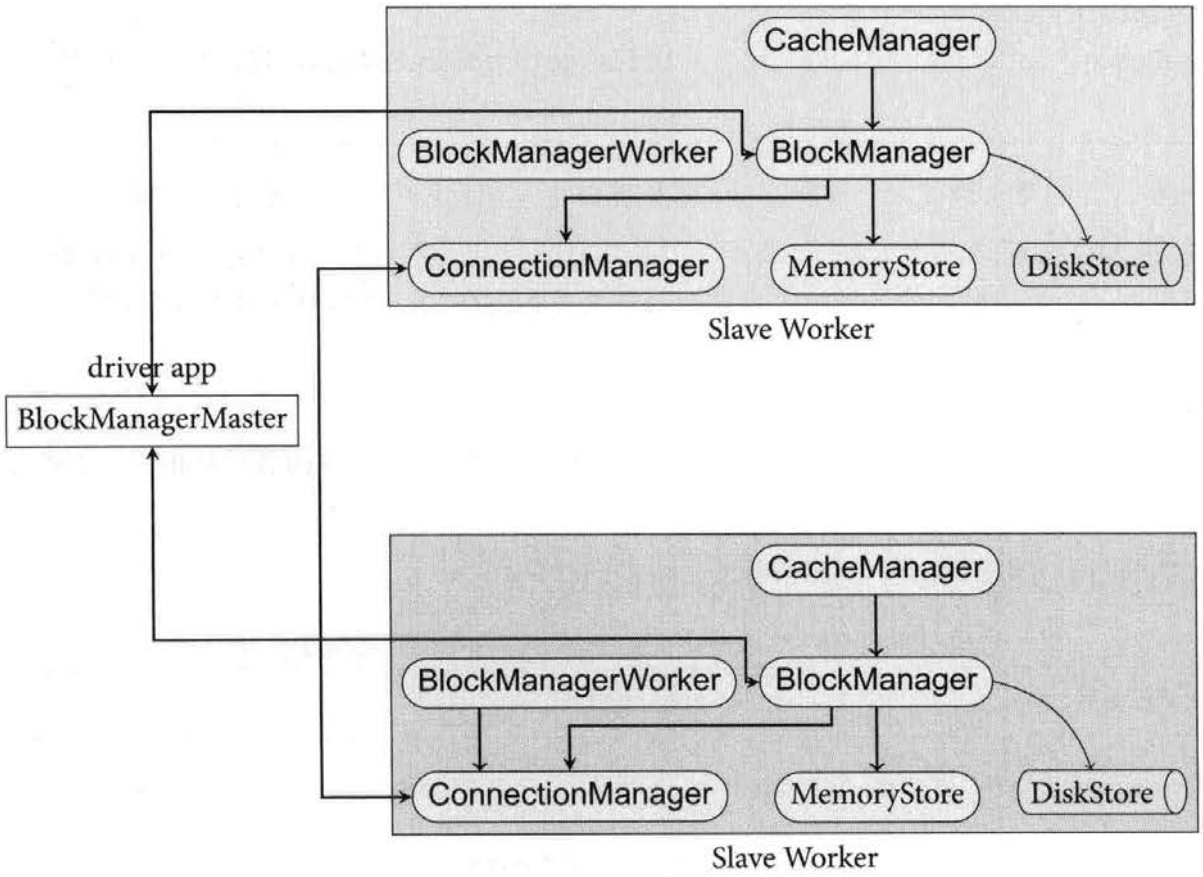


图 4.14 存储子系统

图 4.14是Spark存储子系统中几个主要模块的关系示意图，现简要说明如下。

- CacheManager：RDD 在进行计算的时候，通过 CacheManager 来获取数据，并通过 CacheManager 来存储计算结果。
- BlockManager：CacheManager在进行数据读取和存取的时候主要依赖BlockManager接口来操作，BlockManager决定数据是从内存（MemoryStore）还是从磁盘（DiskStore）中获取。
- MemoryStore：负责将数据保存在内存或从内存读取。

- DiskStore：负责将数据写入磁盘或从磁盘读入。
- BlockManagerWorker：数据写入本地的MemoryStore或DiskStore是一个同步操作。为了容错可能还需要将数据复制到别的计算节点，以便数据丢失的时候还能够恢复。数据复制的操作是异步完成，由BlockManagerWorker来处理这一部分事情。
- ConnectionManager：负责与其他计算节点建立连接，并负责数据的发送和接收。
- BlockManagerMaster：注意该模块只运行在Driver Application所在的Executor，功能是负责记录下所有BlockId存储在哪个SlaveWorker上。比如RDD Task运行在机器A，所需要的 BlockId 为 3，但在机器 A 上没有 BlockId 为 3 的数据，这个时候 Slave Worker 需要向 BlockManagerMaster 询问数据存储的位置，然后再通过ConnectionManager去获取。

4.3.3 存储子模块启动过程分析

上述各个存储子模块由SparkEnv来创建，创建过程在SparkEnv.create中完成。

代码清单 4.63 BlockManagerMaster的创建过程

```
val blockManagerMaster = new BlockManagerMaster(registerOrLookup(
"BlockManagerMaster",
new BlockManagerMasterActor(isLocal, conf)), conf)
val blockManager = new BlockManager(executorId, actorSystem, blockManagerMaster,
   serializer, conf)

val connectionManager = blockManager.connectionManager
val broadcastManager = new BroadcastManager(isDriver, conf)
val cacheManager = new CacheManager(blockManager)
```

这段代码容易让人疑惑，看起来像是在所有的 Cluster Node 上都创建了 BlockManagerMasterActor。其实不然，仔细看registerOrLookup函数的实现。如果当前节点是Driver则创建这个Actor，否则建立到Driver的连接，取得BlockManagerMaster的Actor。

代码清单 4.64 registerOrLookup

```
def registerOrLookup(name: String, newActor: => Actor): ActorRef = {
  if (isDriver) {
    logInfo("Registering " + name)
    actorSystem.actorOf(Props(newActor), name = name)
  } else {
    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://spark@$driverHost:$driverPort/user/$name"
      val timeout = AkkaUtils.lookupTimeout(conf)
      logInfo(s"Connecting to $name: $url")
      Await.result(actorSystem.actorSelection(url).resolveOne(timeout), timeout)
    }
  }
```

BlockManager作为客户端，其初始化过程中一个主要的动作就是向BlockManagerMaster发起注册，同时启动定时器，以便发送心跳消息。

代码清单 4.65 BlockManagerMaster.initialize

```
  private def initialize(): Unit = {
    master.registerBlockManager(blockManagerId, maxMemory, slaveActor)
    BlockManagerWorker.startBlockManagerWorker(this)
    if (!BlockManager.getDisableHeartBeatsForTesting(conf)) {
      heartBeatTask = actorSystem.scheduler.schedule(0.seconds, heartBeatFrequency
    .milliseconds) {
        Utils.tryOrExit { heartBeat() }
      }
    }
  }
```

4.3.4 数据写入过程分析

数据写入的简要流程如图4.15所示，简述如下：

（1） RDD.iterator是与Storage子系统交互的入口。

（2） CacheManager.getOrCompute调用BlockManager的put接口来写入数据。

（3） 数据优先写入MemoryStore，即内存。如果MemoryStore中的数据已满，则将最近使用次数不频繁的数据写入磁盘。

（4） 通知BlockManagerMaster有新的数据写入，在BlockManagerMaster中保存元数据。

（5） 如果数据备份数目大于1，则将写入的数据与其他Slave Worker进行同步。

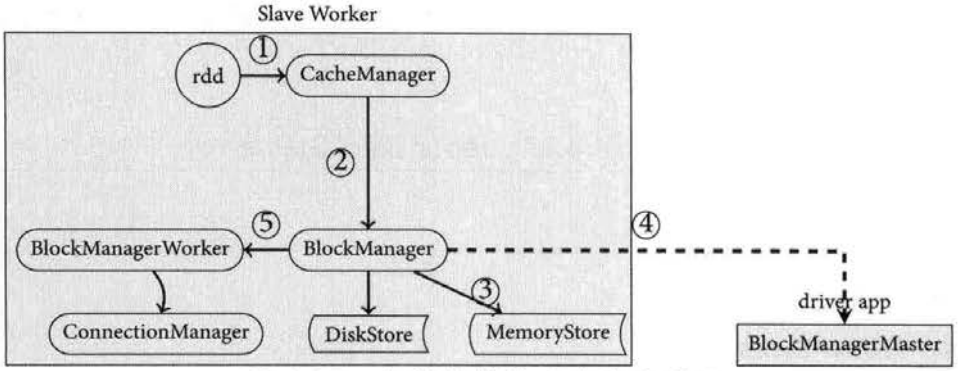


图 4.15 中间计算结果存储子系统

数据写入过程中最为重要的一个函数就是doPut。在默认情况下，Spark只会将RDD的计算结果保存到内存中，这个时候是不会产生备份的。如果该RDD的计算结果丢失，则重新计算。

如果要将RDD的计算结果存储到多个节点，那么在调用persist函数时，将参数指定为MEMORY_ONLY_2或是MEMORY_AND_DISK_2。

当指定的备份数目大于1时，doPut函数会调用replicate将数据备份到其他节点。

代码清单 4.66 replicate

```
private def replicate(blockId: BlockId, data: ByteBuffer, level: StorageLevel) {
    val tLevel = StorageLevel(
      level.useDisk, level.useMemory, level.useOffHeap, level.deserialized, 1)
    if (cachedPeers == null) {
      cachedPeers = master.getPeers(blockManagerId, level.replication - 1)
    }
    for (peer: BlockManagerId <- cachedPeers) {
      val start = System.nanoTime
      data.rewind()
      logDebug("Try to replicate BlockId " + blockId + " once; The size of the
    data is " + data.limit() + " Bytes. To node: " + peer)
      if (!BlockManagerWorker.syncPutBlock(PutBlock(blockId, data, tLevel),
        new ConnectionManagerId(peer.host, peer.port))) {
        logError("Failed to call syncPutBlock to " + peer)
      }
      logDebug("Replicated BlockId " + blockId + " once used " +
        (System.nanoTime - start) / 1e6 + " s; The size of the data is " +
        data.limit() + " bytes.")
    }
  }
```

doPut在完成数据的存储之后，会利用reportBlockStatus告知BlockManagerMaster数据写入的详细信息。

代码清单 4.67 reportBlockStatus

```
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")
      // Re-registering will report our new block for free.
      asyncReregister()
    }
    logDebug(s"Told master about block $blockId")
  }
```

4.3.5 数据读取过程分析

数据读取的入口函数是BlockManager.get函数，其主要处理逻辑是先尝试从本地获取，如果所要获取的内容不在本地，则发起远程获取。

代码清单 4.68 get

```
def get(blockId: BlockId): Option[Iterator[Any]] = {
    val local = getLocal(blockId)
    if (local.isDefined) {
      logInfo("Found block %s locally".format(blockId))
      return local
    }
    val remote = getRemote(blockId)
    if (remote.isDefined) {
      logInfo("Found block %s remotely".format(blockId))
      return remote
    }
```

```
    None
  }
```

如何知道本地有哪些内容在呢？可以通过简单的实验来验证。中间的临时文件在/tmp/目录下，使用lsof指令可以看到进程打开了哪些文件。

代码清单 4.69 列出进程打开的文件

```
lsof -p pid
```

仔细看一下远程获取的处理步骤。远程获取的代码调用路径为getRemote→ doGetRemote。在doGetRemote中最主要的就是调用BlockManagerWorker.syncGetBlock来从远程获得数据。

代码清单 4.70 syncGetBlock

```
def syncGetBlock(msg: GetBlock, toConnManagerId: ConnectionManagerId): ByteBuffer
    = {
    val blockManager = blockManagerWorker.blockManager
    val connectionManager = blockManager.connectionManager
    val blockMessage = BlockMessage.fromGetBlock(msg)
    val blockMessageArray = new BlockMessageArray(blockMessage)
    val responseMessage = connectionManager.sendMessageReliablySync(
        toConnManagerId, blockMessageArray.toBufferMessage)
    responseMessage match {
      case Some(message) => {
        val bufferMessage = message.asInstanceOf[BufferMessage]
        logDebug("Response message received " + bufferMessage)
        BlockMessageArray.fromBufferMessage(bufferMessage).foreach(blockMessage =>
     {
            logDebug("Found " + blockMessage)
            return blockMessage.getData
          })
      }
      case None => logDebug("No response message received")
    }
    null
  }
```

上述这段代码中最有意思的莫过于sendMessageReliablySync，远程数据读取毫无疑问是一个异步I/O操作，这里的代码怎么写起来就像是在进行同步的操作一样呢？也就是说如何知道对方发送回来响应的呢？

别急，继续去看看sendMessageReliably的定义。

代码清单 4.71 sendMessageReliably

```
def sendMessageReliably(connectionManagerId: ConnectionManagerId, message: Message
    )
      : Future[Option[Message]] = {
    val promise = Promise[Option[Message]]
    val status = new MessageStatus(
      message, connectionManagerId, s => promise.success(s.ackMessage))
    messageStatuses.synchronized {
      messageStatuses += ((message.id, status))
    }
    sendMessage(connectionManagerId, message)
    promise.future
  }
```

如果这个Future执行完毕，则返回s.ackMessage。再看看这个ackMessage是在什么地方被写入的。看一看ConnectionManager.handleMessage中的代码片段。

代码清单 4.72 ackMessage写入过程

```
case bufferMessage: BufferMessage => {
  if (authEnabled) {
    val res = handleAuthentication(connection, bufferMessage)
    if (res == true) {
      // message was security negotiation so skip the rest
      logDebug("After handleAuth result was true, returning")
      return
    }
  }
  if (bufferMessage.hasAckId) {
    val sentMessageStatus = messageStatuses.synchronized {
      messageStatuses.get(bufferMessage.ackId) match {
```

```
        case Some(status) => {
          messageStatuses -= bufferMessage.ackId
          status
        }
        case None => {
          throw new Exception("Could not find reference for received ack message
    " + message.id)
          null
        }
      }
    }
    sentMessageStatus.synchronized {
      sentMessageStatus.ackMessage = Some(message)
      sentMessageStatus.attempted = true
      sentMessageStatus.acked = true
      sentMessageStaus.markDone()
    }
```

注意，此处所调用的sentMessageStatus.markDone就会调用在sendMessageReliablySync中定义的promise.Success。不妨看看MessageStatus的定义。

代码清单 4.73 MessageStatus

```
class MessageStatus(
      val message: Message,
      val connectionManagerId: ConnectionManagerId,
      completionHandler: MessageStatus => Unit) {

    var ackMessage: Option[Message] = None
    var attempted = false
    var acked = false

    def markDone() { completionHandler(this) }
  }
```

4.3.6 TachyonStore

在4.3节中讲到Cache时，计算结果会存储到内存当中，当内存不足的时候，写到外部磁盘。

一切看起来没有什么问题，真的是这样吗？

其实将中间结果放到当前JVM的内存中，就是让执行计算的JVM充当了两个角色，既是计算引擎，又是存储引擎，带来的问题如下：

- 计算引擎中的错误如果导致JVM进程退出，则存储的所有内容全部丢失。
- 大量的Cache使得JVM发生GC的概率大大增强，严重影响计算性能。

有没有一种方案将存储引擎和计算引擎剥离开，同时使得缓存数据的存储和读取又很快的方案呢？

答案是肯定的，这就是Spark社区的新星产品Tachyon。

Tachyon是一个高容错的分布式文件系统，允许文件以内存的速度在集群框架中进行可靠的共享，就像Spark和MapReduce那样。通过利用信息继承、内存侵入，Tachyon获得了高性能。Tachyon工作集文件缓存在内存中，并可以让不同的 Jobs/Queries及框架都能以内存的速度来访问缓存文件。因此，Tachyon可以减少那些需要经常使用的数据集通过访问磁盘来获得的次数。

Tachyon的整体架构如图 4.16所示。

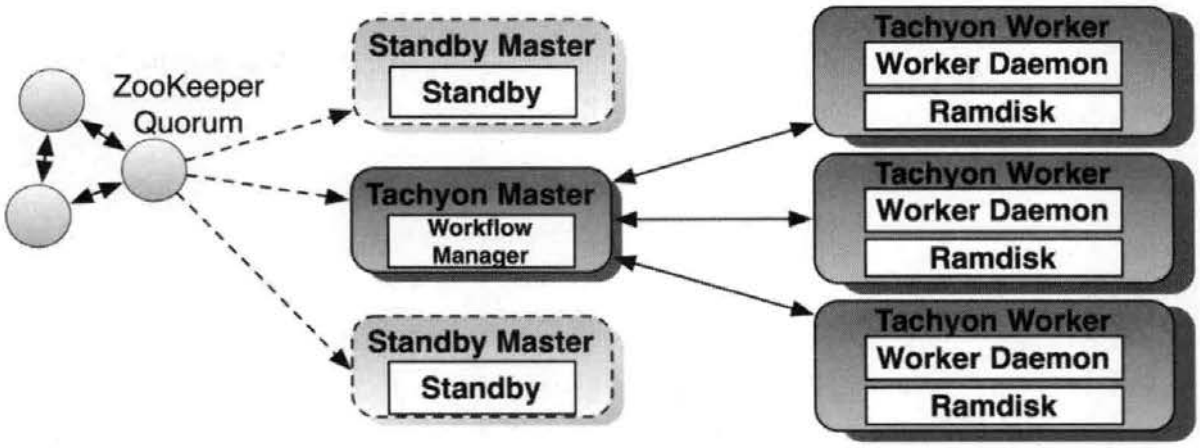


图 4.16 Tachyon架构

Tachyon以常见的Master/Worker的方式组织集群，由Master节点负责管理、维护文件系统Metadata，文件数据维护在Worker节点的内存中。

在容错性方面，主要的技术要点如下：

- 底层支持Plugable的文件系统，如HDFS用于用户指定文件的持久化。
- 使用Journal机制持久化文件系统的Metadata。
- 利用ZooKeeper构建Master的HA。

- 采用和Spark RDD类似的Lineage的思想用于灾难恢复。

对于Tachyon本身的架构不做过多的分析，这里主要关注Spark如何与Tachyon结合。

在Spark的最新源码中，Storage子系统引入了TachyonStore。TachyonStore是在内存中实现了HDFS文件系统的接口，其主要目的就是尽可能地利用内存来作为数据持久层，避免过多的磁盘读写操作。

读取存储在Tachyon中的文件。

代码清单 4.74 使用Tachyon来读取文件

```
val file = sc.textFile("tachyon://ip:port/path")
```

将中间结果保存到Tachyon中。

代码清单 4.75 保存数据到Tachyon

```
val file = sc.textFile("tachyon://ip:port/path")
file.persist(OFF_HEAP)
```

putInfoTachyonStore负责将数据真正存储到Tachyon中。

代码清单 4.76 putIntoTachyonStore

```
private def putIntoTachyonStore(
      blockId: BlockId,
      bytes: ByteBuffer,
      returnValues: Boolean): PutResult = {
    // So that we do not modify the input offsets !
    // duplicate does not copy buffer, so inexpensive
    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)
    }
  }
```

第5章

部署方式分析

“乃至童子戏，聚沙为佛塔。如是诸人等，皆已成佛道。”

《妙法莲花经》

Spark部署主要涉及的是计算资源的管理。不同的部署方式，资源管理的参与者和方法均有很大的差异。本章将对常用的部署方式做详尽的分析。

5.1 部署模型

Spark支持以下几种部署模式：

（1） 单机模式。
（2） 伪集群模式。
（3） 独立集群，笔者喜欢称之为原生集群模式。
（4） YARN集群。
（5） Mesos。

Spark在运行态中，主要由Driver、Master、Worker、Executor这4种不同类型的组件构成。

在不同的部署模式下，它们有的是运行于一个独立的JVM进程，有的是运行于不同的进程。下面几节内容将具体描述它们在特定部署模式下的运行情况。

不同的部署模式，差异主要体现在运行资源的管理和分配以及容错处理上。

5.2　单机模式 local

单机模式下，Driver、Master、Worker和Executor都运行在同一个JVM进程之中。

简单实验一下，以单机模式运行Spark。

代码清单 5.1　单机模式运行spark-shell

```
MASTER=local bin/spark-shell
```

利用jps检查运行着的Java进程，会发现只有一个与Spark相关的JVM进程。

代码清单 5.2　jps -ml

```
9162 org.apache.spark.deploy.SparkSubmit spark-shell --class org.apache.spark.
  repl.Main
```

在SparkContext的createTaskScheduler函数中，如果判定Spark是以local模式运行的，则所创建的ExecutorBackend是LocalBackend。

代码清单 5.3　createTaskScheduler中创建单机模式的backend

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

在local模式下，如果没有显式指定spark.default.parallelism，那么任务的并发度为1，即每次只有一个Task运行。

在local模式下，容错性是最差的，不管什么原因都很容易导致JVM进程退出，这时所有任务将无法恢复，运行结果将丢失。所以，local模式主要的功能体现在调试方便上。

有一点需要指出的是，如果使用bin/run-example来运行Spark例子时，不显式地指定MASTER=local，这个时候MASTER的值为local[*]。

local是单个线程来执行任务，而local[*]会创建多个线程来执行任务，具体线程个数与运行机器的Core数目相关，可以参考spark.default.parallelism参数的具体解释。

5.3 伪集群部署 local-cluster

为了尽可能接近于集群部署，Spark支持一种伪集群方式，Master、Worker、Executor都运行在本机，较之Standalone集群方式，其有两个显著特点：

- Master和Worker运行于同一个JVM中。
- Master、Worker、Executor都运行于同一台机器，无法跨机器运行。

下面看一下SparkContext中的createTaskScheduler函数针对local-cluster这种部署方式的逻辑处理。

代码清单 5.4 创建支持local-cluster模式的backend

```
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()
}
scheduler
```

在local-cluster模式下会创建LocalSparkCluster，进而生成Master和Worker的实例，让两者运行于同一JVM中。看一下LocalSparkCluster中start函数的实现即可证明这一点。

代码清单 5.5　LocalSparkCluster.start函数

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

    /* Start the Master */
    val conf = new SparkConf(false)
    val (masterSystem, masterPort, _) = Master.startSystemAndActor(localHostname,
    0, 0, conf)
    masterActorSystems += masterSystem
    val masterUrl = "spark://" + localHostname + ":" + masterPort
    val masters = Array(masterUrl)

    /* Start the Workers */
    for (workerNum <- 1 to numWorkers) {
      val (workerSystem, _) = Worker.startSystemAndActor(localHostname, 0, 0,
    coresPerWorker,
        memoryPerWorker, masters, null, Some(workerNum))
      workerActorSystems += workerSystem
    }

    masters
  }
```

做个简单的实验，将Spark运行于local-cluster模式。

代码清单 5.6　Spark运行于local-cluster

```
MASTER=local-cluster[2,2,512] bin/spark-shell
```

配置运行的时候需要注意为每个 Executor 指定的内存数量必须与 spark-defaults.conf 中指定的数值一致。如例子中指定的是 512MB，那么如果配置了 spark-defaults.conf，其中的 spark.executor.memory 的值也必须是 512MB，否则运行会报错。

检查有哪些JVM进程被创建。

代码清单 5.7 ps -ef|grep -i java

```
4822 org.apache.spark.executor.CoarseGrainedExecutorBackend
4813 org.apache.spark.executor.CoarseGrainedExecutorBackend
4477 org.apache.spark.deploy.SparkSubmit
```

容错性如何呢？如果Executor异常退出，Worker会将其重新拉起；而如果Worker或Master出错导致JVM退出，则整个Spark Cluster彻底失效。

5.4 原生集群Standalone Cluster

不管local模式还是local-cluster模式，其主要目的均是为了开发调试。鉴于其容错性能很差，故此绝对不可用于实际生产环境。

Spark提供的原生集群（Standlaone Cluster）模式在集群规模不是非常大的情况下，可以用于实际生产。

如图 5.1所示，组成Cluster的三大主要节点，即Master、Worker和Executor各自运行于独立的JVM进程。

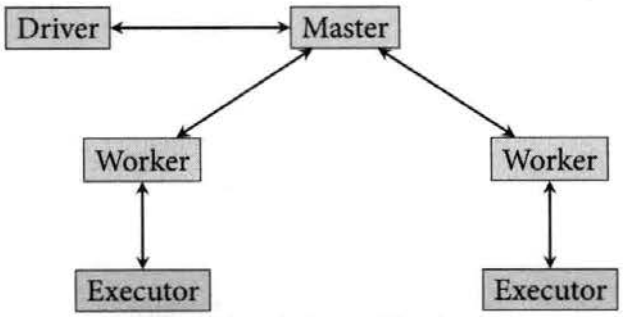


图 5.1 Standalone Cluster组成

在 Standalone Cluster 模式下，Driver 不能运行于 Cluster 内部，只能独立运行，所以在看 Master.scala 中接收到 RegisterApplication 消息的处理逻辑时要加以注意。

Standalone集群由3个不同种类的节点组成，分别如下所示。

- Master：主控节点，可以类比为董事长或总舵主。在整个集群之中，最多只有一个Master处在Active状态。
- Worker：工作节点，这个是Manager，是分舵主。在整个集群中，可以有多个Worker。如果Worker为零，则什么事也做不了。
- Executor：干苦力活的，直接受Worker掌控。一个Worker可以启动多个Executor。

这3种不同类型的节点各自运行于独立的JVM进程之中。

提交到Standalone集群的应用程序被称为Spark Client，Driver Application运行于Spark Client之中。

图 5.2总结了正常情况下Standalone集群的启动以及应用提交时各节点之间有哪些消息交互。

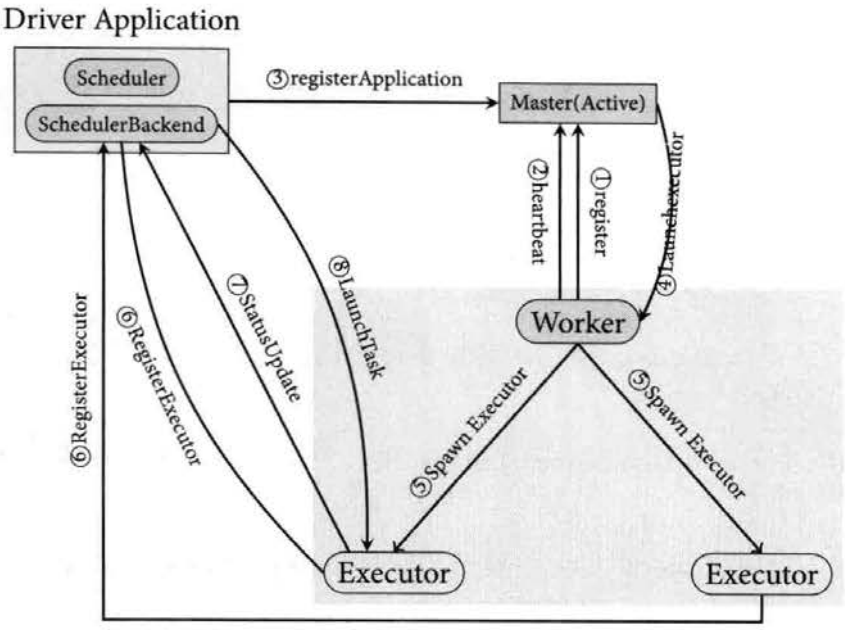


图 5.2　Standalone Cluster启动过程

5.4.1　启动Master

先做个实验，看看如何启动Standalone Cluster模式。

第一步启动Master。

代码清单 5.8　运行Master

```
$SPARK_HOME/sbin/start_master.sh
```

在start_master.sh中最关键的一句如下。

代码清单 5.9　start_master.sh

```
"$sbin"/spark-daemon.sh start org.apache.spark.deploy.master.Master 1 --ip
  $SPARK_MASTER_IP --port $SPARK_MASTER_PORT --webui-port
  $SPARK_MASTER_WEBUI_PORT
```

检测Master的JVM进程。

代码清单 5.10 检测Master进程

```
19787 org.apache.spark.deploy.master.Master
```

Master的运行日志在$SPARK_HOME/logs目录下，在实际测试中如果想查看有哪些Worker已经连接上来，可以通过分析该文件得知。

Master在启动函数分析：（1）配置信息读取，功能在MasterArguments；（2）创建Actor。

MasterArguments要读取的环境变量如表 5.1所示。

表 5.1 Master读取的环境变量

spark_master_host	监听地址
spark_master_port	监听端口
spark_master_webui_port	webui 监听的端口

5.4.2 启动Worker

运行Worker。

代码清单 5.11 运行Worker

```
./bin/spark-class org.apache.spark.deploy.worker.Worker spark://localhost:7077
```

Worker运行时，需要注册到指定的Master URL，Master URL需要和spark-env.sh中指定的spark_master_ip一致，上述例子中的URL就是spark://localhost:7077。

在 Worker 启动过程中，自然少不了读取配置相关的信息，这一部分的代码详见 WorkerArguments。

Worker需要向Master汇报自己所在机器的Core数目和内存容量，这些信息是如何获取的呢？Worker首先读取系统信息，即实际的CPU Core数目和物理内存，使用的函数分别为inferDefaultCores和inferDefaultMemory。

代码清单 5.12 获取机器CPU个数和内存大小

```
def inferDefaultCores(): Int = {
  Runtime.getRuntime.availableProcessors()
}

def inferDefaultMemory(): Int = {
  val ibmVendor = System.getProperty("java.vendor").contains("IBM")
  var totalMb = 0
```

```
    try {
      val bean = ManagementFactory.getOperatingSystemMXBean()
      if (ibmVendor) {
        val beanClass = Class.forName("com.ibm.lang.management.
    OperatingSystemMXBean")
        val method = beanClass.getDeclaredMethod("getTotalPhysicalMemory")
        totalMb = (method.invoke(bean).asInstanceOf[Long] / 1024 / 1024).toInt
      } else {
        val beanClass = Class.forName("com.sun.management.OperatingSystemMXBean")
        val method = beanClass.getDeclaredMethod("getTotalPhysicalMemorySize")
        totalMb = (method.invoke(bean).asInstanceOf[Long] / 1024 / 1024).toInt
      }
    } catch {
      case e: Exception => {
        totalMb = 2*1024
        System.out.println("Failed to get total physical memory. Using " + totalMb
     + " MB")
      }
    }
    // Leave out 1 GB for the operating system, but don't return a negative
    // memory size
    math.max(totalMb - 1024, 512)
  }
```

如果与内存和CPU Core相关的环境变量存在，则使用环境变量中指定的值。表5.2显示了与Worker相关的环境变量。

表 5.2　Worker读取的环境变量

SPARK_WORKER_PORT	监听地址
SPARK_WORKER_CORES	CPU Core数目
SPARK_WORKER_MEMORY	Worker可使用内存
SPARK_WORKER_WEBUI_PORT	WEBUI 监听的端口
SPARK_WORKER_DIR	Worker目录

Worker启动后会向Master发起注册，在注册消息中说明本Worker Node含有的Core数目和可用内存。具体逻辑见下面的tryRegisterAllMasters。

调用顺序是preStart→registerWithMaster→tryRegisterAllMasters。

代码清单 5.13 tryRegisterAllMasters

```
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)
  }
}
```

Master侧收到RegisterWorker通知，做如下处理。

（1） 如果收到消息的Master处于Standby状态，则不做任何响应。

（2） 判断是否存在重复的Worker ID，如果是则拒绝注册。

（3） 如果当前Master是Active的Master，同时不存在重复的Worker ID，则：

（a） 抽取注册上来的Worker的消息并加以保存，为今后Master在异常退出后的恢复做准备。

（b） 发送响应给Worker，确认注册成功。

（c） 调用Schedule函数，将已经提交但没有分配资源的Application分发到新加入的Worker Node。

代码清单 5.14 Master.scala对RegisterWorker消息的处理

```
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)
      }
    }
  }
```

Worker在收到Master确认注册成功的消息RegisteredWorker之后，会注册定时处理函数，以定期向Master发送心跳消息SendHeartbeat。

代码清单 5.15　processing for RegisteredWorker

```
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) {
        context.system.scheduler.schedule(CLEANUP_INTERVAL_MILLIS millis,
          CLEANUP_INTERVAL_MILLIS millis, self, WorkDirCleanup)
      }

case SendHeartbeat =>
      masterLock.synchronized {
        if (connected) { master ! Heartbeat(workerId) }
      }
```

那么Master是如何确认Worker一直是活着还是挂掉的呢？这个处理逻辑比较有意思，Master也会启动一个定时器，在定时处理函数中对原先处于Alive状态的Worker进行状态判定。

代码清单 5.16 Master设置检测Worker的定时器

```
context.system.scheduler.schedule(0 millis, WORKER_TIMEOUT millis, self,
    CheckForWorkerTimeOut)
```

如果当前时间减去Worker最近一次的状态更新时间小于定时器时长的话，就认为该Worker还处于Alive状态；否则认为该Worker因为没有发送心跳消息而“挂”了。

代码清单 5.17 定时处理函数

```
case CheckForWorkerTimeOut => {
  timeOutDeadWorkers()
}
def timeOutDeadWorkers() {
  // Copy the workers into an array so we don't modify the hashset while
  // iterating through it
  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
        // we've seen this DEAD worker in the UI, etc. for long enough; cull it
      }
    }
  }
```

如果判定结果认为相应的Worker已经不再存活，那么利用removeWorker函数通知Driver Application。

如果管理一个有多个Worker参与的Standalone Cluster，手工去一一启动Worker是非常烦琐的，Spark为此提供了相应的脚本。

SPARK_HOME/sbin/start-slaves.sh负责启动各个Slaves上的Worker进程。此处需要注意的是，运行Master和Worker的用户组和用户名需要一致；否则会在Worker创建Executor时由于连接无法建立而出错。

5.4.3　运行spark-shell

Master和Worker启动完毕之后，Executor是什么时候被启动的呢？

Executor不是显式地被拉起来的，而是在Application注册到Master时被带起的。

spark-shell属于Application，启动spark-shell的指令如下。

代码清单 5.18　运行spark-shell

```
MASTER=spark://localhost:7077 $SPARK_HOME/bin/spark-shell
```

Standalone模式下每个Application的运行日志存储在$SPARK_HOME/works目录中。

在 Standalone Cluster 部署模式下，Master 和 Worker 都运行在独立的 JVM 进程，Application Driver 也运行于独立的 JVM 进程。Application Driver 和 SparkSubmit 运行于同一个进程，参见 SparkSubmit 中的 main 函数。

具体调用过程是这样的：spark-shell→spark-submit→spark-class。

下面看一下spark-submit脚本中的最后一行，可以发现其最终使用的是SparkSubmit。

代码清单 5.19　spark-submit

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

SparkSubmit根据输入参数会创建SparkDeploySchedulerBackend（见图5.3）。

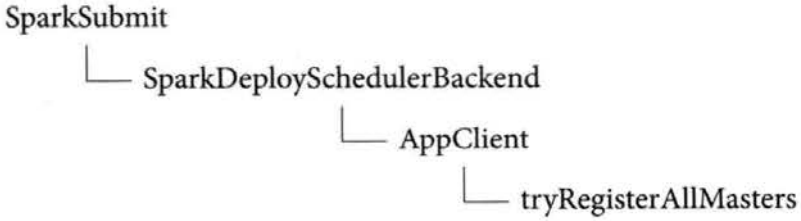


图 5.3　SparkSubmit引发的函数调用关系图

AppClient专门用来和Master进行消息交互。调用关系如下：registerWithMaster→tryRegisterAllMasters。

代码清单 5.20 tryRegisterAllMasters

```
def tryRegisterAllMasters() {
  for (masterUrl <- masterUrls) {
    logInfo("Connecting to master " + masterUrl + "...")
    val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
    actor ! RegisterApplication(appDescription)
  }
}
```

spark-shell作为Application，在Master侧处理的分支是RegisterApplication，具体处理代码如下。

代码清单 5.21 Master中的RegisterApplication消息处理分支

```
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()
  }
}
```

每当有新的Application注册到Master时，Master都要调用Schedule函数将Application发送到相应的Worker，在对应的Worker启动相应的ExecutorBackend。

有两种分发Application的原则：一是尽可能将任务分发到各个Worker Node，二是将任务分发到尽可能少的Worker Node。那么一个Application可以占用集群中多少个Core呢？如果不显式指定的话，可以占用Integer.MAX_VALUE。很恐怖的一个数字。如果不想让其他提交的Application一直处于饥饿状态，那么最好为每个Application指定占用的最大CPU Core数目。

设置每个Application所占用最大Core数目的配置项是spark.cores.max。

将任务分发到尽可能多的Worker Node的处理逻辑如下，每次在各个Worker Node上分配一个空闲的Core。

代码清单 5.22　schedule（一）

```
if (spreadOutApps) {
  // Try to spread out each app among all the nodes, until it has all its cores
  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
    }
    // Now that we've decided how many cores to give on each node,
    // let's actually give them
    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
      }
    }
  }
}
```

将任务分配到尽可能少的Worker Node处理逻辑，就是一次在某一个节点上占用尽可能多的空闲Core，代码如下。

代码清单 5.23　schedule（二）

```
// Pack each app into as few nodes as possible until we've assigned
// all its cores
```

```
for (worker <- workers if worker.coresFree > 0 && worker.state == WorkerState.
  ALIVE) {
  for (app <- waitingApps if app.coresLeft > 0) {
    if (canUse(app, worker)) {
      val coresToUse = math.min(worker.coresFree, app.coresLeft)
      if (coresToUse > 0) {
        val exec = app.addExecutor(worker, coresToUse)
        launchExecutor(worker, exec)
        app.state = ApplicationState.RUNNING
      }
    }
  }
}
```

如果Application不是一个常驻应用，如流数据处理，那么过一段时间之后，会正常运行结束。

那么Master和Executor是如何感知到Application已经退出了呢？

先从Executor说起。当Application Driver退出的时候，CoarseGrainedExecutorBackend会收到系统通知DisassociatedEvent，这个消息是由Akka定义规范的。一旦收到该消息，Executor认为Application Driver 已经退出，自己没有服务的对象了，就会主动退出。

代码清单 5.24 Executor主动退出

```
case x: DisassociatedEvent =>
  logError(s"Driver $x disassociated! Shutting down.")
  System.exit(1)
```

Master也会收到DisassociatedEvent，处理逻辑较Executor复杂一些，将注册上来的Application删除。

代码清单 5.25 Master监听到Application退出

```
  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() }
  }
```

当Spark运行于Standalone Cluster模式下，Executor的启动参数受控于Driver机器上的SPARK_HOME/conf/spark-defaults.conf文件。

比如为每个Executor指定的运行时的最大内存为4GB，那么只需要修改运行Driver Application所在机器上的spark-defaults.conf中的配置项即可。

顺带多补充一句：如果在spark.executor.memory中指定的内存是5GB，而加入到Cluster中的Worker说它只有3GB的内存可用，那么将无法为提交的Application创建任务Executor，因为内存需求无法得到满足。

5.4.4　容错性分析

在Standalone模式下，Cluster的启动及任务提交执行流程简述如下：

（1）启动Master，Master启动完毕等待新的Application提交上来。

（2）启动Worker，Worker在启动后会向Master发起注册，注册成功之后，会定期发送心跳消息给Master。

（3）如果有新的Application提交到Master，Master会根据资源使用情况要求Worker启动相应的Executor。

（4）新启动的Executor注册到Application中的Driver，并定期发送心跳消息。

（5）Application中的SchedulerBackend将作业中的Task分配到各个注册上来的Executor执行。

下面就Standalone Cluster运行模式下，各节点出现异常情况时对运行作业会带来哪些影响进行详细的分析。

异常场景1：Worker异常退出

如图 5.4所示，在Spark运行过程中，可能会碰到Worker异常退出的情况。当Worker退出时，整个集群会有哪些“故事”发生呢？

- Worker异常退出。在做实验的时候，可以通过kill指令故意将Worker杀死。
- Worker在退出之前，会将自己所管控的所有“小弟”Executor干掉。
- Worker需要定期向Master发送心跳消息。现在Worker进程都已经“玩完”了，哪有心跳消息，所以Master会在超时处理中意识到有一个“分舵”离开了。
- Master非常伤心，伤心的Master将情况汇报给了相应的Driver。

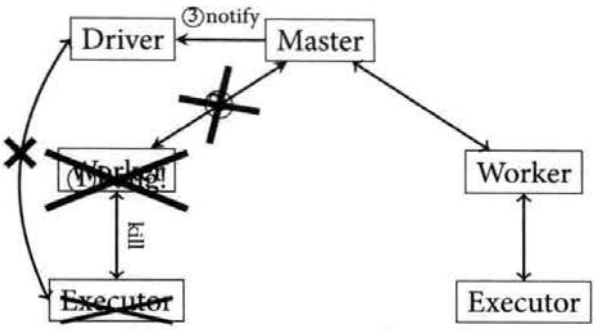


图 5.4 Worker异常退出

Worker是如何保证在退出前将所有归自己管控的Executor全部kill的呢？Java中有一个退出处理句柄，通过添加相应的回调函数来实现进程退出前的清理工作。

Worker进程中的ExecutorRunner具体负责Executor的启动和停止，每一个Executor进程对应于一个ExecutorRunner，ExecutorRunner担任监工角色。

下面具体来看ExecutorRunner中的start函数实现。

代码清单 5.26 start in ExecutorRunner

```
def start() {
  workerThread = new Thread("ExecutorRunner for " + fullId) {
    override def run() { fetchAndRunExecutor() }
  }
  workerThread.start()
  // Shutdown hook that kills actors on shutdown.
  shutdownHook = new Thread() {
    override def run() {
      killProcess(Some("Worker shutting down"))
    }
  }
  Runtime.getRuntime.addShutdownHook(shutdownHook)
}
```

killProcess一方面停止Executor进程，一方面将停止的结果反馈给Worker本身。

代码清单 5.27 killProcess

```
private def killProcess(message: Option[String]) {
  var exitCode: Option[Int] = None
  if (process != null) {
```

```
      logInfo("Killing process!")
      process.destroy()
      process.waitFor()
      if (stdoutAppender != null) {
        stdoutAppender.stop()
      }
      if (stderrAppender != null) {
        stderrAppender.stop()
      }
      exitCode = Some(process.waitFor())
    }
    worker ! ExecutorStateChanged(appId, execId, state, message, exitCode)
  }
```

在自己挂掉之前，一定要将“小弟”先干掉，干这种勾当最著名的莫过于宋江了。

Application Driver对于Worker的状态不感兴趣，只想知道Executor当前的情况是什么。

Application Driver 又是如何意识到Executor已经失去联系了呢？这要归功于Master。Master收到Worker上报来的消息，会将失联的Executor通知给Application Driver。具体代码如下。

代码清单 5.28　Master对Executor状态变化的处理逻辑

```
  case ExecutorStateChanged(appId, execId, state, message, exitStatus) => {
    val execOption = idToApp.get(appId).flatMap(app => app.executors.get(execId))
    execOption match {
      case Some(exec) => {
        exec.state = state
        exec.application.driver ! ExecutorUpdated(execId, state, message,
    exitStatus)
        if (ExecutorState.isFinished(state)) {
          val appInfo = idToApp(appId)
          // Remove this executor from the worker and app
          logInfo("Removing executor " + exec.fullId + " because it is " + state)
          appInfo.removeExecutor(exec)
          exec.worker.removeExecutor(exec)

          val normalExit = exitStatus.exists(_ == 0)
          // Only retry certain number of times so we don't go into
```

```
        // an infinite loop.
        if (!normalExit && appInfo.incrementRetryCount < ApplicationState.
    MAX_NUM_RETRY) {
          schedule()
        } else if (!normalExit) {
          logError("Application %s with ID %s failed %d times, removing it".
    format(
          appInfo.desc.name, appInfo.id, appInfo.retryCount))
          removeApplication(appInfo, ApplicationState.FAILED)
        }
      }
    }
    case None =>
    logWarning("Got status update for unknown executor " + appId + "/" + execId)
  }
}
```

即便 Master 没有将状态及时反馈给 Application Driver，Application Driver 也会由于收到 DisassociatedEvent 而意识到 Executor 已经退出。

代码清单 5.29 DisassociatedEvent

```
case DisassociatedEvent(_, address, _) =>
  addressToExecutorId.get(address).foreach(removeExecutor(_,
   "remote Akka client disassociated"))
```

异常场景2：Executor异常退出

Executor作为Standalone集群部署方式下的底层员工，一旦异常退出，其后果会是什么呢（见图5.5）？

（1） Executor异常退出，ExecutorRunner注意到异常，将情况通过ExecutorStateChanged汇报给Master。

（2） Master收到通知之后，非常不高兴，竟然有“小弟”要跑路，那还了得，要求Executor所属的Worker再次启动。

（3） Worker收到LaunchExecutor指令，再次启动Executor。

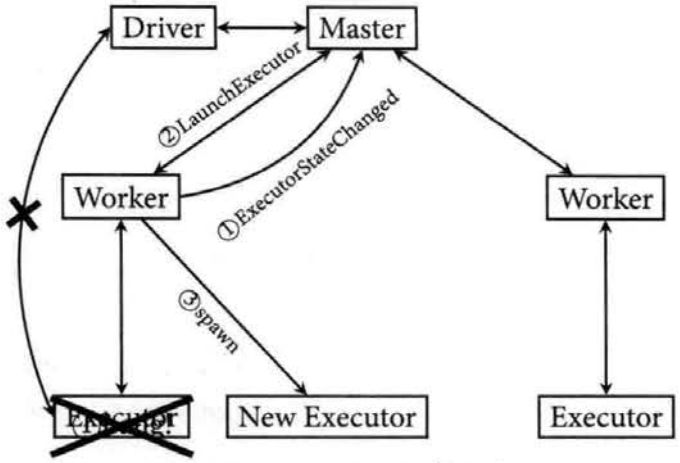


图 5.5　Executor异常退出

异常场景3：Master 异常退出

Worker和Executor异常退出的场景都讲到了，现在只剩下最后一种情况了：Master挂掉了怎么办（见图5.6）？

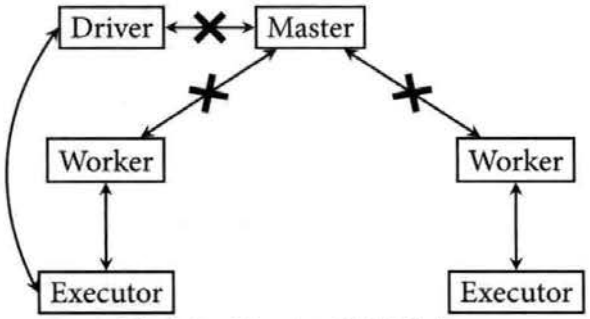


图 5.6　Master异常退出

“带头大哥”如果不在了，会是什么后果呢？

（1） Worker没有汇报的对象了，也就是如果Executor再次跑飞，Worker是不会将Executor启动起来的，“大哥”没给指令无法向集群提交新的任务。

（2） 老的任务即便结束了，占用的资源也无法清除，因为资源清除的指令是Master发出的。

怎么样，知道后果很严重了吧？别看“老大”平时不干活，要真的不在，仅凭“小弟们”是不行的。

那么怎么解决Master单点失效的问题呢？

你可能会说再加一个Master就是了，两个老大。两个老大如果同时具有指挥权，结果也将是灾难性的，“一副扑克牌只能有一个大王”。

可以设立一个副职人员，当目前的正职挂掉之后，副职接管。也就是同一时刻，有且只有一个Active Master。

主意不错，如何实现呢？使用ZooKeeper的ElectLeader功能，效果如图 5.7所示。

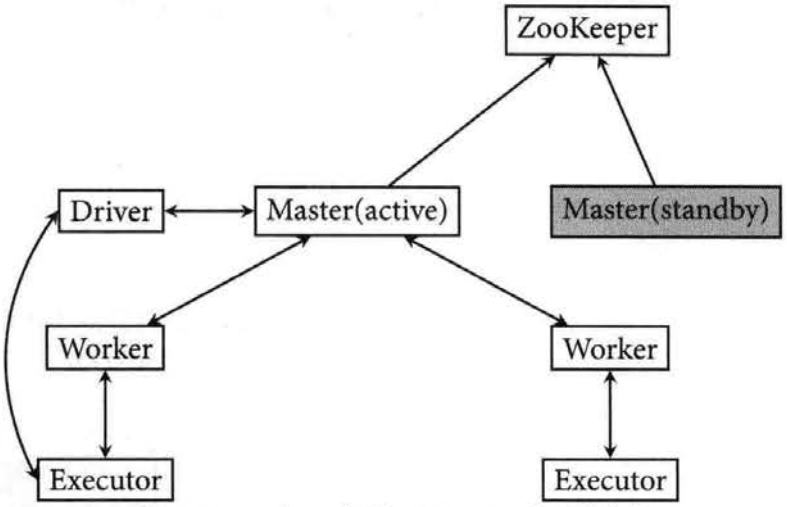


图 5.7 一主一备的双Master部署模式

假设ZooKeeper集群已经安装设置成功，那么如何启动Standalone集群中的节点呢？有哪些地方需要做特殊配置？

修改 conf/spark-env.sh。在 conf/spark-env.sh 中，为 SPARK_DAEMON_JAVA_OPTS 添加如表 5.3所列的选项。

表 5.3 ZooKeeper集群运行时的修改参数列表

配置项	说明
spark.deploy.recoveryMode	值设置为ZooKeeper表示支持备机方案，默认值是NONE
spark.deploy.zookeeper.url	ZooKeeper 集群的 URL 地址（如 192.168.1.100:2181，192.168.1.101:2181）
spark.deploy.zookeeper.dir	ZooKeeper中存储recovery state的目录

设置SPARK_DAEMON_JAVA_OPTS的实际例子如下。

代码清单 5.30 spark.deploy.recoveryMode

```
SPARK_DAEMON_JAVA_OPTS="$SPARK_DAEMON_JAVA_OPTS -Dspark.deploy.recoveryMode=
    ZOOKEEPER"
```

应用程序运行的时候，指定多个Master地址，用逗号分开，如下所示。

代码清单 5.31　设定Master

```
MASTER=spark://192.168.100.101:7077,spark://192.168.100.102:7077 bin/spark-shell
```

5.5　Spark On YARN

5.5.1　YARN的编程模型

YARN的基本架构如图 5.8所示，由三大功能模块组成，分别是：RM （ResourceManager）、NM （NodeManager）、AM（ApplicationMaster）。

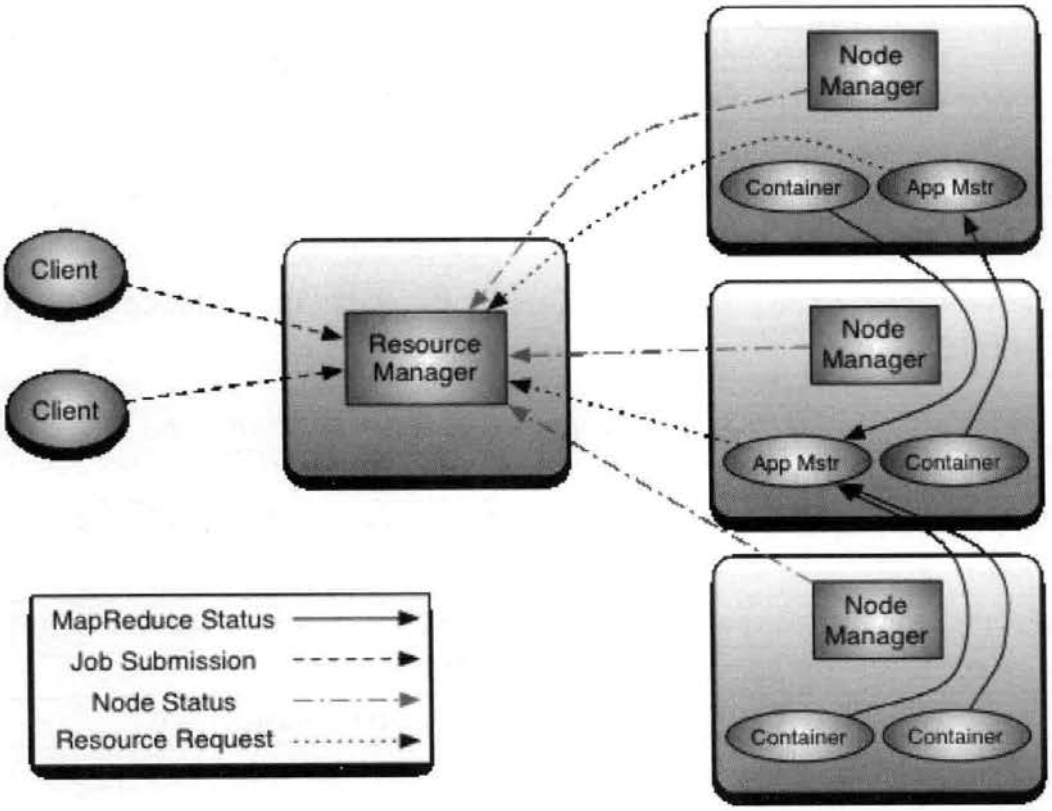


图 5.8　YARN架构

5.5.2　YARN中的作业提交

具体步骤如下（见图5.9）：

（1）用户通过 Client 向 ResourceManager 提交 Application，ResourceManager 根据用户请求分配合适的 Container，然后在指定的 NodeManager 上运行 Container 以启动 ApplicationMaster。

（2） ApplicationMaster启动完成后，向ResourceManager注册自己。

（3） 对于用户的Task，ApplicationMaster需要首先跟ResourceManager进行协商以获取运行用户Task所需要的Container，在获取成功后，ApplicationMaster将任务发送给指定的NodeManager。

（4） NodeManager启动相应的Container，并运行用户Task。

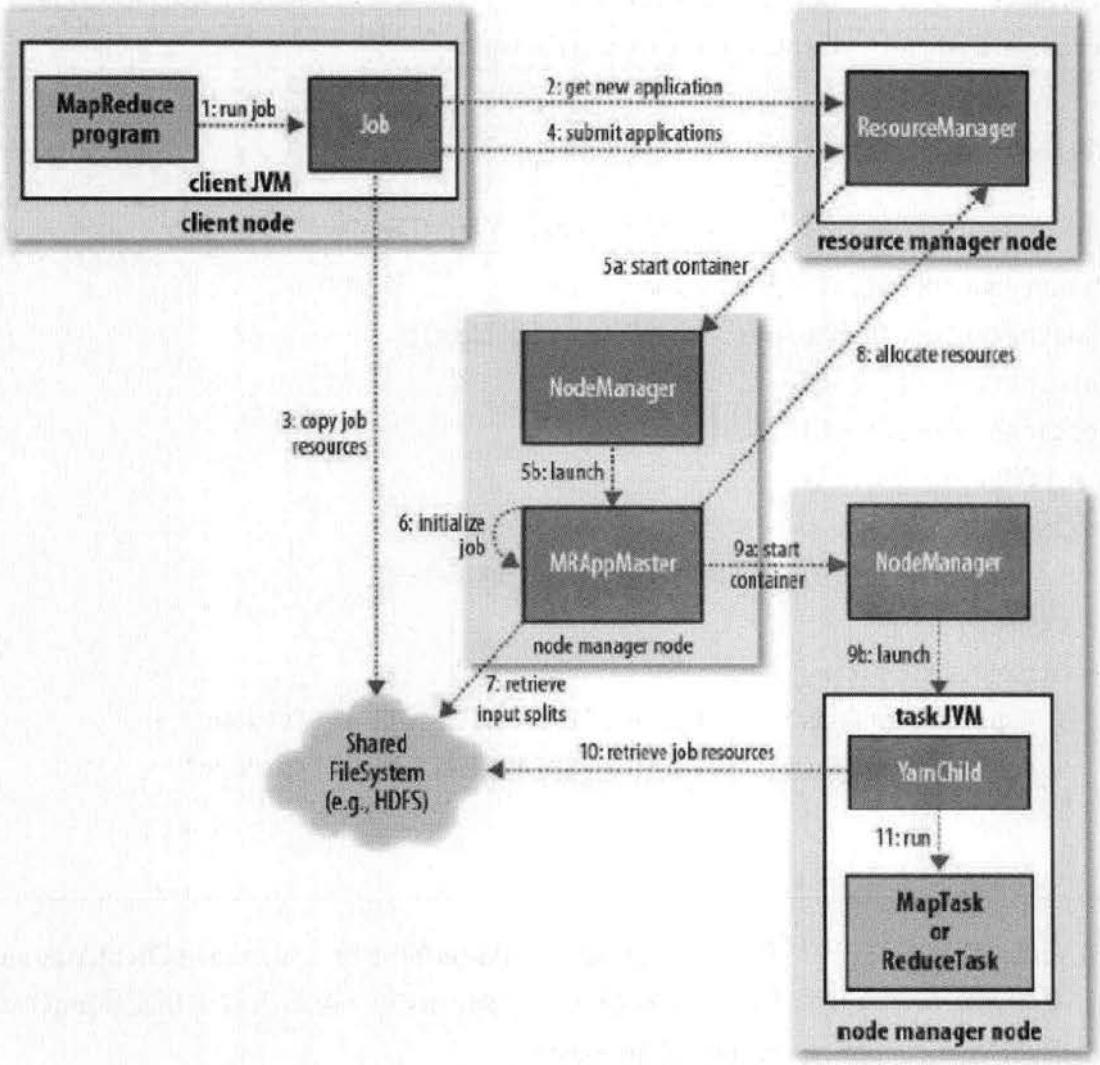


图 5.9 部署YARN应用

上面说了一大堆，说白了就是在编写 YARN Application 时，主要是实现 Client 和 ApplicationMaster。实例请参考 github 上的 simple-yarn-app。

5.5.3 Spark On YARN实现详解

表 5.4列出了为了适配YARN，需要做哪些更改，与之前Standalone模式间的对应关系是什么。代码走读时，分析的重点是ApplicationMaster、YarnClusterSchedulerBackend和YarnClusterScheduler。

表 5.4　Standalone vs. YARN

Standalone	YARN	Notes
Client	Client	standalone请参考spark.deploy目录
Master	ApplicationMaster	
Worker	ExecutorRunnable	
Scheduler	YarnClusterScheduler	
SchedulerBackend	YarnClusterSchedulerBackend	

一般来说，在Client中会显式地指定启动ApplicationMaster的类名，如下面的代码所示。

代码清单 5.32　Yarn Client

```
ContainerLaunchContext amContainer =
Records.newRecord(ContainerLaunchContext.class);
amContainer.setCommands(
Collections.singletonList(
"$JAVA_HOME/bin/java" +
" -Xmx256M" +
" com.hortonworks.simpleyarnapp.ApplicationMaster" +
" " + command +
" " + String.valueOf(n) +
" 1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout" +
" 2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr"
)
);
```

但在yarn.Client中并没有直接指定ApplicationMaster的类名，而是通过ClientArguments进行了封装，真正指定启动类的名称的地方在ClientArguments中。构造函数中指定了amClass的默认值是org.apache.spark.deploy.yarn.ApplicationMaster。

将SparkPi部署到YARN上，具体指令如下。

代码清单 5.33　run SparkPi on YARN

```
$ SPARK_JAR=./assembly/target/scala-2.10/spark-assembly-0.9.1-hadoop2.0.5-alpha.
  jar \
./bin/spark-class org.apache.spark.deploy.yarn.Client \
--jar examples/target/scala-2.10/spark-examples-assembly-0.9.1.jar \
--class org.apache.spark.examples.SparkPi \
```

```
--args yarn-standalone \
--num-workers 3 \
--master-memory 4g \
--worker-memory 2g \
--worker-cores 1
```

从输出的日志可以看出，AM指定的是org.apache.spark.deploy.yarn.ApplicationMaster。

代码清单 5.34 提交SparkPi运行时产生的日志

```
13/12/29 23:33:25 INFO Client: Command for starting the Spark ApplicationMaster:
    $JAVA_HOME/bin/java -server -Xmx4096m -Djava.io.tmpdir=$PWD/tmp org.apache.
    spark.deploy.yarn.ApplicationMaster --class org.apache.spark.examples.SparkPi
    --jar examples/target/scala-2.9.3/spark-examples-assembly-0.8.1-incubating.jar
     --args  'yarn-standalone'  --worker-memory 2048 --worker-cores 1 --num-
    workers 3 1> /stdout 2> /stderr
```

目录结构：忽略掉alpha目录，alpha表示Hadoop的版本为0.23和2.0.x。将重点放在stable版本上，下述代码全部指的是stable版本。

代码清单 5.35 YARN目录结构

```
stable common pom.xml README.md
```

注意stable和common目录，启动顺序如下所述：

（1） ApplicationMaster作为YARN应用中的ApplicationMaster最先启动。

（2） 向RM申请Container。

（3） 申请成功后，向指定的NM发送指令以启动Container。

（4） 在ApplicationMaster中启动监听线程，以监控运行着的ExecutorContainer。

图 5.10是将Spark应用部署在YARN上的示意图。

下面就源码部分进行走读。

步骤1：注册ApplicationMaster。

代码清单 5.36 register in ApplicationMaster.scala

```
def register(master: ApplicationMaster) {
  applicationMasters.add(master)
}
```

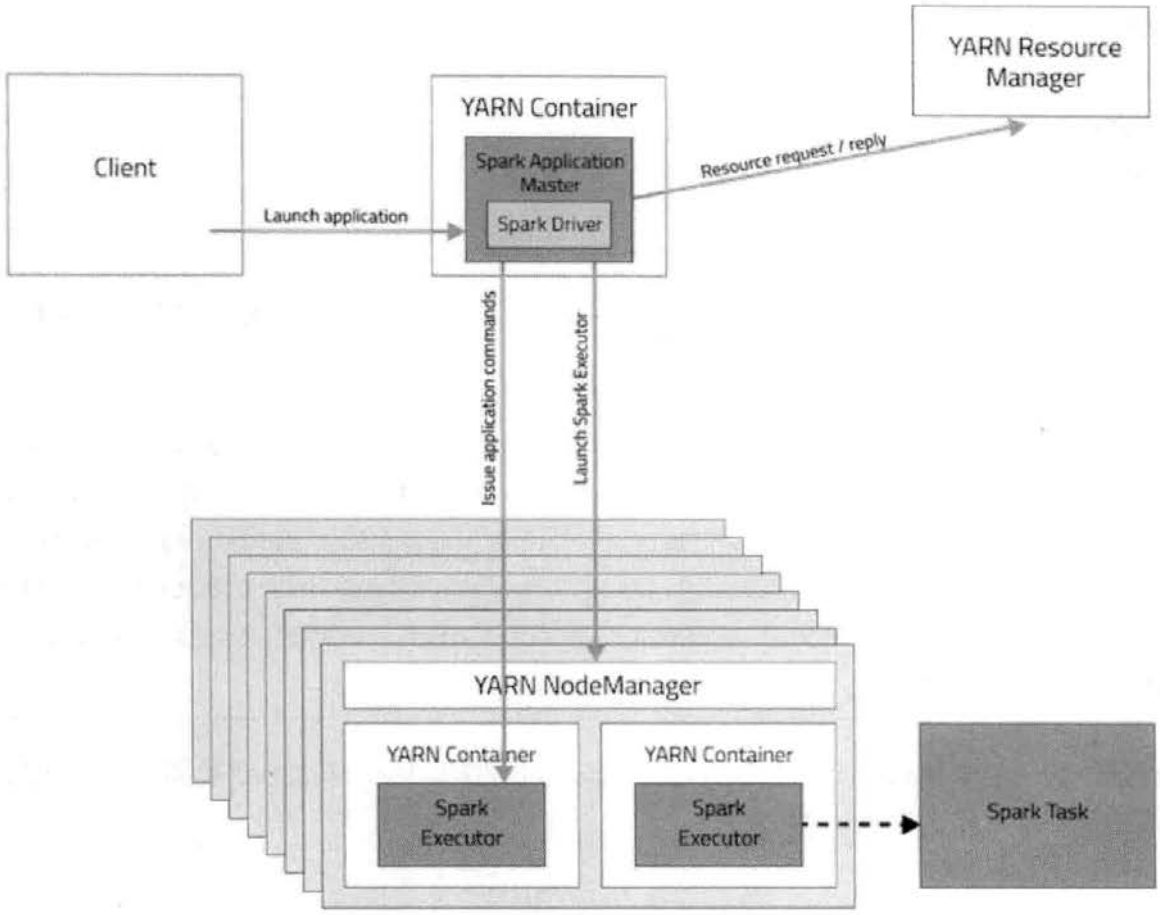


图 5.10　Spark On YARN

步骤2：启动用户应用Application Driver。注意ApplicationMaster和Application Driver运行于同一个JVM进程。

代码清单 5.37　startUserClass

```
private def startUserClass(): Thread = {
  logInfo("Starting the user JAR in a separate Thread")
  System.setProperty("spark.executor.instances", args.numExecutors.toString)
  val mainMethod = Class.forName(
    args.userClass,
    false,
    Thread.currentThread.getContextClassLoader).getMethod("main", classOf[Array[
  String]])
  val t = new Thread {
    override def run() {
      var succeeded = false
```

```
        try {
          // Copy
          val mainArgs = new Array[String](args.userArgs.size)
          args.userArgs.copyToArray(mainArgs, 0, args.userArgs.size)
          mainMethod.invoke(null, mainArgs)
          // Some apps have "System.exit(0)" at the end.  The user thread will
          // stop here unless it has an uncaught exception thrown out.
          // It needs a shutdown hook to set SUCCEEDED.
          succeeded = true
        } finally {
          logDebug("Finishing main")
          isLastAMRetry = true
          if (succeeded) {
            ApplicationMaster.this.finishApplicationMaster(FinalApplicationStatus.
    SUCCEEDED)
          } else {
            ApplicationMaster.this.finishApplicationMaster(FinalApplicationStatus.
    FAILED)
          }
        }
      }
    }
    t.setName("Driver")
    t.start()
    t
  }
```

如果 Driver Application 启动成功，会向 ApplicationMaster 注册已经完成初始化工作的 SparkContext。

代码清单 5.38 postStartHook

```
override def postStartHook() {
    val sparkContextInitialized = ApplicationMaster.sparkContextInitialized(sc)
    super.postStartHook()
    if (sparkContextInitialized){
      ApplicationMaster.waitForInitialAllocations()
```

```
      // Wait for a few seconds for the slaves to bootstrap and register with
      // master - best case attempt
      // TODO It needn't after waitBackendReady
      Thread.sleep(3000L)
    }
    logInfo("YarnClusterScheduler.postStartHook done")
  }
```

SparkContext初始化完成之后，调用registerApplicationMaster。

代码清单 5.39　registerApplicationMaster

```
private def registerApplicationMaster(): RegisterApplicationMasterResponse = {
    logInfo("Registering the ApplicationMaster")
    amClient.registerApplicationMaster(Utils.localHostName(), 0, uiAddress)
  }
```

步骤3：启动Executor，这里要注意概念的转换。在Standalone Cluster模式中，有Worker这一角色，在YARN中这一角色被NodeManager所替换。由NodeManager来负责Executor的启动与停止。

调用顺序为allocateExecutors→yarnAllocator.allocateResources。

代码清单 5.40　allocateMissingExecutor

```
private def allocateExecutors() {
    try {
      logInfo("Requesting" + args.numExecutors + " executors.")
      // Wait until all containers have launched
      yarnAllocator.addResourceRequests(args.numExecutors)
      yarnAllocator.allocateResources()
      // Exits the loop if the user thread exits.

      var iters = 0
      while (yarnAllocator.getNumExecutorsRunning < args.numExecutors &&
    userThread.isAlive) {
        checkNumExecutorsFailed()
        allocateMissingExecutor()
        yarnAllocator.allocateResources()
```

```
        if (iters == ApplicationMaster.ALLOCATOR_LOOP_WAIT_COUNT) {
          ApplicationMaster.doneWithInitialAllocations()
        }
        Thread.sleep(ApplicationMaster.ALLOCATE_HEARTBEAT_INTERVAL)
        iters += 1
      }
    } finally {
      // In case of exceptions, etc - ensure that the loop in
      // ApplicationMaster#sparkContextInitialized() breaks.
      ApplicationMaster.doneWithInitialAllocations()
    }
    logInfo("All executors have launched.")
  }
```

这里稍微注释一下：在YARN中，ApplicationMaster通过发送AllocateRequest来向RM请求资源，RM在AllocateResponse中将结果返回给ApplicationMaster。

具体到 Spark 中，这一处理逻辑集中于 YarnAllocationHandler 的 alloationResources 函数中，ApplicationMaster接收到AllocationResponse之后，创建ExecutorRunnable，进而调用NodeManagerClient 中的 startContainer 来启动相应的 Container。

代码清单 5.41 创建ExecutorRunnable

```
val executorRunnable = new ExecutorRunnable(
container,
conf,
sparkConf,
driverUrl,
executorId,
executorHostname,
executorMemory,
executorCores)
new Thread(executorRunnable).start()
```

在ExecutorRunnable中使用startContainer来启动Container。

代码清单 5.42 startContainer in ExecutorRunnable

```
def startContainer = {
```

```
    logInfo("Setting up ContainerLaunchContext")

    val ctx = Records.newRecord(classOf[ContainerLaunchContext])
      .asInstanceOf[ContainerLaunchContext]

    ctx.setContainerId(container.getId())
    ctx.setResource(container.getResource())
    val localResources = prepareLocalResources
    ctx.setLocalResources(localResources)

    val env = prepareEnvironment
    ctx.setEnvironment(env)

    ctx.setUser(UserGroupInformation.getCurrentUser().getShortUserName())

    val credentials = UserGroupInformation.getCurrentUser().getCredentials()
    val dob = new DataOutputBuffer()
    credentials.writeTokenStorageToStream(dob)
    ctx.setContainerTokens(ByteBuffer.wrap(dob.getData()))

    val commands = prepareCommand(masterAddress, slaveId, hostname,
     executorMemory,executorCores, localResources)
    logInfo("Setting up executor with commands: " + commands)
    ctx.setCommands(commands)

    // Send the start request to the ContainerManager
    val startReq = Records.newRecord(classOf[StartContainerRequest])
    .asInstanceOf[StartContainerRequest]
    startReq.setContainerLaunchContext(ctx)
    cm.startContainer(startReq)
  }
```

步骤4: 启动监控线程至这里为止，Executor启动没有问题。剩下的疑问是如果ExecutorContainer退出了，ApplicationMaster是如何感知到的。

ApplicationMaster 会通过 launchReporterThread 来监控 Container 的运行情况。如果失效的 Container 数目没超过最大容忍阈值，则重启失效的 Container；否则整个应用退出。

代码清单 5.43 launchReporterThread

```
private def launchReporterThread(): Thread = {
  // Ensure that progress is sent before YarnConfiguration.
  // RM_AM_EXPIRY_INTERVAL_MS elapses.
  val expiryInterval = yarnConf.getInt(YarnConfiguration.
  RM_AM_EXPIRY_INTERVAL_MS, 120000)

  // we want to be reasonably responsive without causing too many
  // requests to RM.
  val schedulerInterval =
    sparkConf.getLong("spark.yarn.scheduler.heartbeat.interval-ms", 5000)

  // must be <= timeoutInterval / 2.
  val interval = math.max(0, math.min(expiryInterval / 2, schedulerInterval))

  val t = new Thread {
    override def run() {
      while (userThread.isAlive) {
        checkNumExecutorsFailed()
        allocateMissingExecutor()
        logDebug("Sending progress")
        yarnAllocator.allocateResources()
        Thread.sleep(interval)
      }
    }
  }
  // Setting to daemon status, though this is usually not a good idea.
  t.setDaemon(true)
  t.start()
  logInfo("Started progress reporter thread - heartbeat interval : " + interval)
  t
}
```

如果失效的Container数目过大，整个应用则失败退出。

代码清单 5.44　checkNumExecutorsFailed

```
private def checkNumExecutorsFailed() {
   if (yarnAllocator.getNumExecutorsFailed >= maxNumExecutorFailures) {
     finishApplicationMaster(FinalApplicationStatus.FAILED,
       "max number of executor failures reached")
   }
 }
```

5.5.4　SparkPi on YARN

安装Hadoop

步骤1：创建用户组及用户。

代码清单 5.45　添加用户

```
groupadd hadoop
useradd -b /home -m -g hadoop hduser
```

步骤2: 下载Hadoop运行版。假设当前是以root用户登录的，现在要切换成用户hduser。

代码清单 5.46　切换用户为hduser

```
id ##检验一下切换是否成功，如果一切ok，将显示下列内容
uid=1000(hduser) gid=1000(hadoop) groups=1000(hadoop)
```

下载Hadoop 2.4并解压。

代码清单 5.47　下载Hadoop

```
cd /home/hduser
wget http://mirror.esocc.com/apache/hadoop/common/hadoop-2.4.0/hadoop-2.4.0.tar.gz
tar zvxf hadoop-2.4.0.tar.gz
```

步骤3: 设置环境变量。

代码清单 5.48　设置hadoop相关的环境变量

```
export HADOOP_HOME=$HOME/hadoop-2.4.0
export HADOOP_MAPRED_HOME=$HOME/hadoop-2.4.0
```

```
export HADOOP_COMMON_HOME=$HOME/hadoop-2.4.0
export HADOOP_HDFS_HOME=$HOME/hadoop-2.4.0
export HADOOP_YARN_HOME=$HOME/hadoop-2.4.0
export HADOOP_CONF_DIR=$HOME/hadoop-2.4.0/etc/hadoop
```

为了避免每次都要重复设置这些变量，可以将上述语句加入.bashrc文件。

步骤4: 创建目录。

接下来创建的目录是为Hadoop中HDFS相关的namenode，即datanode使用。

代码清单 5.49 创建namenode和datanode

```
mkdir -p $HOME/yarn_data/hdfs/namenode
mkdir -p $HOME/yarn_data/hdfs/datanode
```

步骤5: 修改Hadoop配置文件，下列文件需要相应的配置。

（1） yarn-site.xml

（2） core-site.xml

（3） hdfs-site.xml

（4） mapred-site.xml

切换到Hadoop安装目录。

代码清单 5.50 切换到Hadoop根目录

```
cd $HADOOP_HOME
```

修改etc/hadoop/yarn-site.xml，在<configuration>和</configuration>之间添加如下内容，其他文件的添加位置与此一致。

代码清单 5.51 yarn-site.xml

```
<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>mapreduce_shuffle</value>
</property>
<property>
   <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
   <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

修改etc/hadoop/core-site.xml。

代码清单 5.52　core-site.xml

```
<property>
   <name>fs.default.name</name>
   <value>hdfs://localhost:9000</value>
   <!--YarnClient会用到该配置项-->
</property>
```

修改etc/hadoop/hdfs-site.xml。

代码清单 5.53　hdfs-site.xml

```
   <property>
   <name>dfs.replication</name>
   <value>1</value>
 </property>
 <property>
   <name>dfs.namenode.name.dir</name>
   <value>file:/home/hduser/yarn_data/hdfs/namenode</value>
   <!--节点格式化中被用到-->
 </property>
 <property>
   <name>dfs.datanode.data.dir</name>
   <value>file:/home/hduser/yarn_data/hdfs/datanode</value>
 </property>
```

修改etc/hadoop/mapred-site.xml。

代码清单 5.54　mapred-site.xml

```
<property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
</property>
```

步骤6: 格式化namenode。

代码清单 5.55 格式化namenode

```
bin/hadoop namenode -format
```

步骤7: 启动namenode。

代码清单 5.56 启动namenode

```
sbin/hadoop-daemon.sh start namenode
```

步骤8: 启动datanode。

代码清单 5.57 启动datanode

```
sbin/hadoop-daemon.sh start datanode
```

步骤9: 启动ResourceManager（RM）。

代码清单 5.58 启动RM

```
sbin/yarn-daemon.sh start resourcemanager
```

步骤10: 启动NodeManager（NM）。

代码清单 5.59 启动NM

```
sbin/yarn-daemon.sh start nodemanager
```

步骤11: 启动Job History Server。

代码清单 5.60 启动historyserver

```
sbin/mr-jobhistory-daemon.sh start historyserver
```

验证一下Hadoop搭建成功与否的最好办法就是在上面跑个WordCount试试。

代码清单 5.61 创建file

```
$mkdir in
$cat > in/file
This is one line
This is another line
```

将文件复制到HDFS中。

代码清单 5.62　复制文件到HDFS

```
$bin/hdfs dfs -copyFromLocal in /in
```

运行WordCount。

代码清单 5.63　运行WordCount

```
bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.0.jar
    wordcount /in /out
```

查看运行结果。

代码清单 5.64　检查执行结果

```
bin/hdfs dfs -cat /out/*
```

运行SparkPi

代码清单 5.65　运行SparkPi

```
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
SPARK_JAR=./assembly/target/scala-2.10/spark-assembly_2.10-0.9.1-hadoop2.2.0.jar \
./bin/spark-class org.apache.spark.deploy.yarn.Client \
--jar ./examples/target/scala-2.10/spark-examples_2.10-assembly-0.9.1.jar \
--class org.apache.spark.examples.JavaSparkPi \
--args yarn-standalone \
--num-workers 1 \
--master-memory 512m \
--worker-memory 512m \
--worker-cores 1
```

运行结果保存在相关Application的stdout目录，使用以下指令可以找到。

代码清单 5.66　检查运行结果

```
  cd $HADOOP_HOME
find . -name "*stdout"
```

假设找到的文件为./logs/userlogs/application_1400479924971_0002/container_1400479924971_0002_01_000001/stdout，使用cat命令可以看到结果。

代码清单 5.67 stdout

```
cat ./logs/userlogs/application_1400479924971_0002/
    container_1400479924971_0002_01_000001/stdout
Pi is roughly 3.14028
```

第三部分

Spark Lib

第 6 章 Spark Streaming
- 6.1 Spark Streaming 整体架构
- 6.2 Spark Streaming 执行过程
- 6.3 窗口操作
- 6.4 容错性分析
- 6.5 Spark Streaming vs. Storm
- 6.6 应用举例

第 7 章 SQL
- 7.1 SQL 语句的通用执行过程分析
- 7.2 SQL On Spark 的实现分析
- 7.3 Parquet 文件和 JSON 数据集
- 7.4 Hive 简介
- 7.5 HiveQL On Spark 详解

第 8 章 GraphX
- 8.1 GraphX 简介
- 8.2 分布式图计算处理技术介绍
- 8.3 Pregel 计算模型
- 8.4 GraphX 图计算框架实现分析
- 8.5 PageRank

第 9 章 MLLib
- 9.1 线性回归
- 9.2 线性回归的代码实现
- 9.3 分类算法
- 9.4 拟牛顿法
- 9.5 MLLib 与其他应用模块间的整合

第6章

Spark Streaming

“欲常常而见之，故源源而来。”

《孟子·万章上》

从本章开始讲述Spark的实际应用。Spark Streaming能够对流数据以近乎实时的速度处理。采用了不同于一般的流式数据处理模型，该模型使得Spark Streaming有非常高的处理速度，与Storm相比拥有更高的吞吐能力。

6.1 Spark Streaming整体架构

随着社交网络的兴起，实时流数据的处理变得越来越迫切，比如在使用微博的时候，想知道当前最热门的话题有哪些，想知道当前最新的粉丝数等，这些都会牵扯到实时流数据的处理。

与一般的文件型（即内容已经固定）数据源相比，所谓的流数据拥有如下的特点：

- 数据一直处在变化中。
- 数据无法回退。
- 数据一直源源不断地涌进。

6.1.1 DStream

针对实时流数据的特点，Spark有何应对之道呢？

用一句话来概括Spark Streaming的处理思路，那就是“将连续的数据持久化、离散化，然后进行批量处理”（见图6.1）。

下面来看一看这么做的原因。

- **数据持久化**：将网络上接收到的数据先暂时存储下来，为后续的处理出错时事件重演提供可能。
- **离散化**：数据源源不断地涌进，永远没有一个尽头，就像周星驰的喜剧中所说，崇拜之情如黄河之水绵绵不绝，一发而不可收。既然不能穷尽，那么就将其按时间分片。比如采用1分钟为时间间隔，那么在连续的1分钟内收集到的数据作为一个处理单元。
- **分片处理**：将持久化下来的数据分批进行处理，处理机制套用之前的RDD模式。

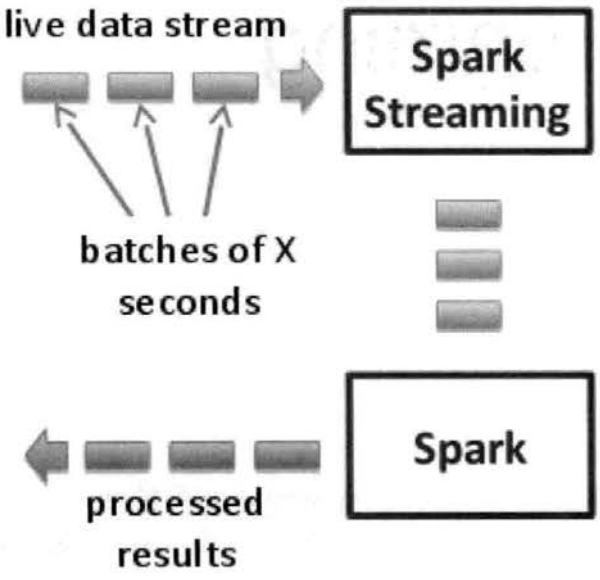


图 6.1 Spark Streaming处理示意图

如果说RDD是Spark Core中的一等公民，那么在Spark Streaming中，一等公民非DStream（Discretized Stream）莫属。

DStream表示从数据源获取持续性的数据流以及经过转换后的数据流。DStream由持续的RDD序列组成。DStream与RDD的关系如图 6.2所示。

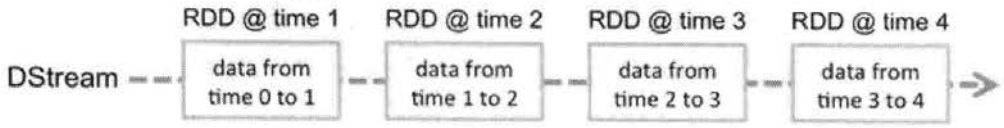


图 6.2 DStream与RDD的对应关系

从实际应用开发的角度来看，DStream可以说是对RDD的又一层封装。如果打开DStream.scala和RDD.Scala进行对比，发现几乎RDD上的所有Operation在DStream中都有相应的定义。

作用于DStream上的Operation分成两类：

- Transformation 转换操作。
- Output 表示输出结果，目前支持的有print、saveAsObjectFiles、saveAsTextFiles、saveAsHadoop-Files。

图 6.3是Spark Streaming的主要架构，从图中可以看出其由3个主要模块构成。

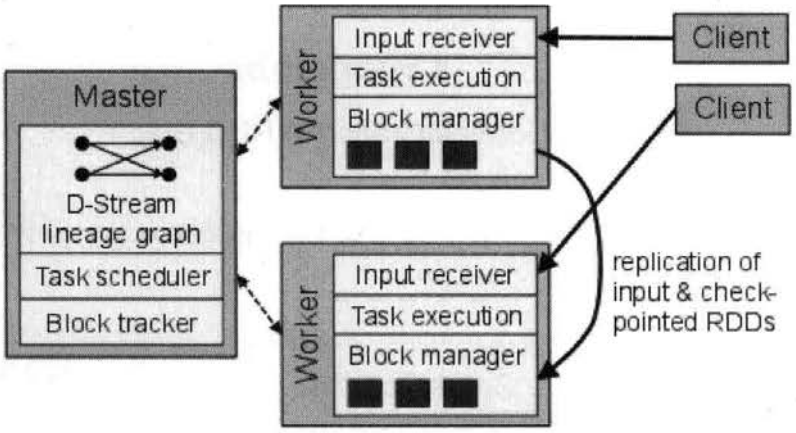


图 6.3 Spark流处理架构

- **Master**：Master主要记录DStream之间的依赖关系或者说是血缘关系，并负责任务调度以生成新的RDD。
- **Worker节点**：从网络接收数据，存储并执行RDD的计算。
- **Client**：Client负责向Spark Streaming中灌入数据。

6.1.2 编程接口

在实际应用中，用户对DStream的操作是通过StreamingContext来完成的（见图6.4），这点犹如Spark Core中SparkContext与RDD的关系。

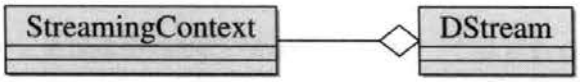


图 6.4 StreamingContext和DStream的引用关系

DStream之间的转换所形成的依赖关系全部保存于DStreamGraph当中，DStreamGraph对于后面定期生成RDD Graph至关重要。图 6.5反映了两者之间的转换。

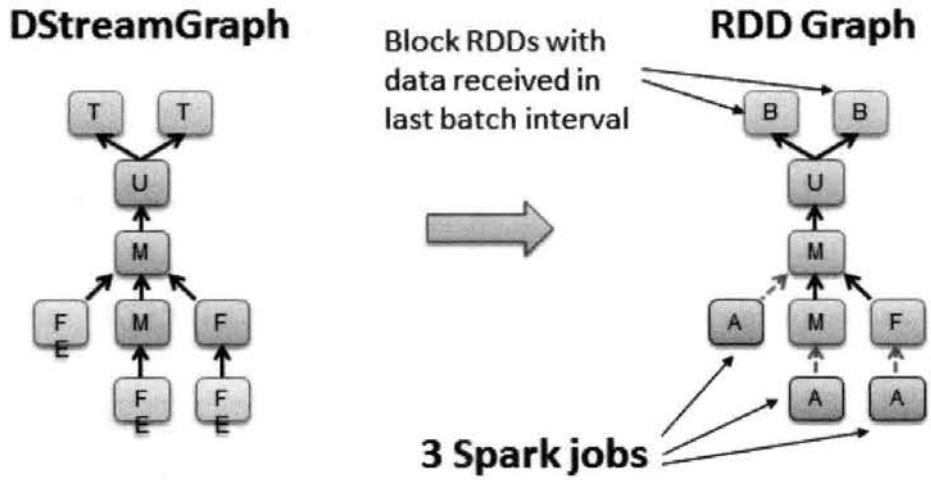


图 6.5 从DStreamGraph生成RDD Graph

指定了DStream之间的转换关系之后，利用StreamingContext.start函数来真正启动运行。图 6.6反映了从提交到运行的过程。

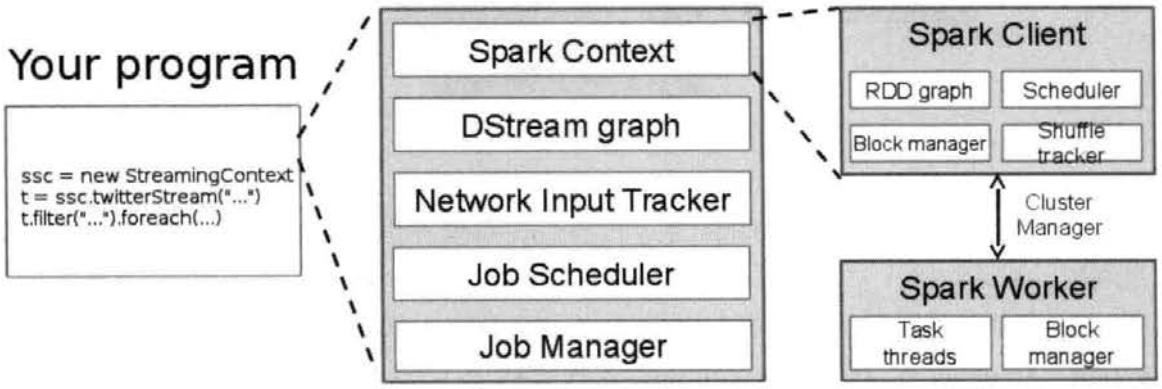


图 6.6 Spark Streaming App

6.1.3 Streaming WordCount

前面简要介绍了Spark Streaming的架构及主要组成构件之后，下面运行一个实例来增强一下大家的感性认识。

步骤1：使用netcat模拟服务器侧。netcat是Linux网络工具中的“瑞士军刀”，有了它，省去了好多事情，不用自己去写个Server或Client的模拟器了。

代码清单 6.1 使用netcat模拟服务器端

```
nc -lk 9999
```

步骤2：运行spark-shell。

代码清单 6.2 以伪集群方式运行spark-shell

```
SPARK_JAVA_OPTS=-Dspark.cleaner.ttl=10000 MASTER=local-cluster[2,2,1024] bin/spark
    -shell
```

步骤3: 在spark-shell中输入如下内容。

代码清单 6.3 streaming wordcount

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._
val ssc = new StreamingContext(sc, Seconds(3))
//连接到netcat
val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap( _.split(" "))
val pairs = words.map(word => (word,1))
val wordCount = pairs.reduceByKey(_ + _)
wordCount.print()
ssc.start()
ssc.awaitTermination()
```

当ssc.start()执行之后，在netcat一侧尝试输入一些内容并回车，spark-shell上就会显示出最新的统计结果。

6.2 Spark Streaming执行过程

光有感性上的认识还是远远不够的，还要对其内部实现机理和运行过程有个详细分析才好。

内部实现的分析沿循传统套路：

- 主要部件的初始化过程。
- 网络侧接收到的数据如何存储到内存。

- 如何根据存储下来的数据生成相应的Spark Job。

Streaming作业的提交和执行过程如图 6.7所示。

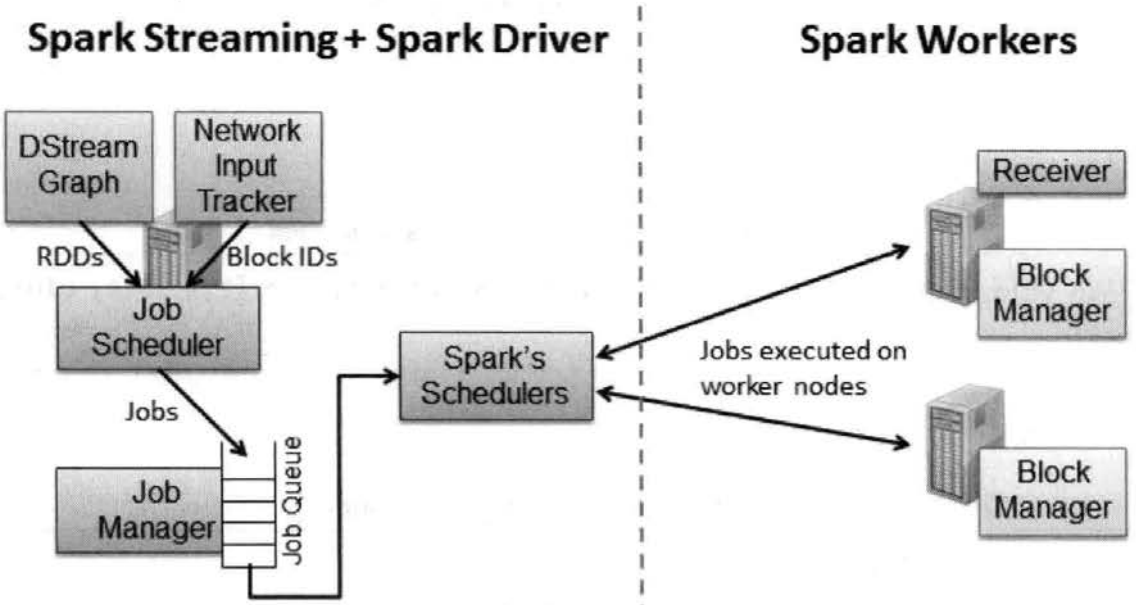


图 6.7 Streaming作业的提交和执行过程

如何能达到分析的目的呢？下面就利用Streaming WordCount这个例子作为入口，详细研究其每一阶段代码是如何实现并运行的。

6.2.1 StreamingContext初始化过程

代码清单 6.4 创建StreamingContext

```
val ssc = new StreamingContext("local[2]", "NetworkWordCount", Seconds(1))
```

Streaming Application中最开始的地方就是创建StreamingContext。对于StreamingContext的构造函数来说，最主要的入参如下。

- SparkContext：任务最终通过SparkContext接口提交到Spark Cluster运行。
- Checkpoint：检查点。
- Duration：根据多少时长创建一个batch。

图 6.8显示了StreamingContext最主要的几个成员变量，各个成员的主要功能说明见表 6.1。

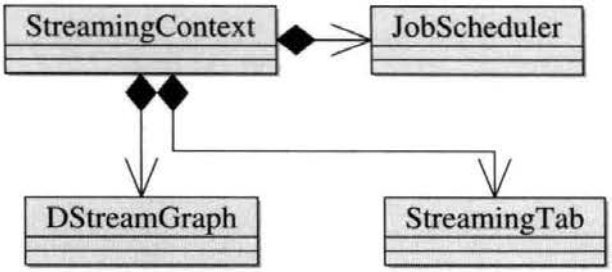


图 6.8 StreamingContext

表 6.1 StreamingContext的主要成员变量

JobScheduler	用于定期生成Spark Job
DStreamGraph	包含DStream之间依赖关系的容器
StreamingTab	用以Spark Streaming运行作业的监控

代码清单 6.5 StreamingContext.init

```
private[streaming] val conf = sc.conf
private[streaming] val env = SparkEnv.get
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 nextReceiverInputStreamd = new AtomicInteger(0)

private[streaming] var checkpointDir: String = {
  if (isCheckpointPresent) {
    sc.setCheckpointDir(cp_.checkpointDir)
```

```
      cp_.checkpointDir
    } else {
      null
    }
  }

  private[streaming] val checkpointDuration: Duration = {
    if (isCheckpointPresent) cp_.checkpointDuration else graph.batchDuration
  }

  private[streaming] val scheduler = new JobScheduler(this)
  private[streaming] val waiter = new ContextWaiter
  private[streaming] val uiTab = new StreamingTab(this)

  /** Register streaming source to metrics system */
  private val streamingSource = new StreamingSource(this)
  SparkEnv.get.metricsSystem.registerSource(streamingSource)

  /** Enumeration to identify current state of the StreamingContext */
  private[streaming] object StreamingContextState extends Enumeration {
    type CheckpointState = Value
    val Initialized, Started, Stopped = Value
  }

  import StreamingContextState._
  private[streaming] var state = Initialized
```

利用刚刚初始化后的ssc来生成DStream，socketTextStream的返回值是DStream。

代码清单 6.6 连接Server

```
val lines = ssc.socketTextStream("localhost", 9999)
```

socketTextStream会调用socketStream来创建SocketInputDstream。

代码清单 6.7 socketTextStream

```
def socketTextStream(
    hostname: String,
```

```
    port: Int,
    storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ): ReceiverInputDStream[String] = {
  socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel
  )
}
```

SocketInputDStream的类继承体系，如图6.9所示。

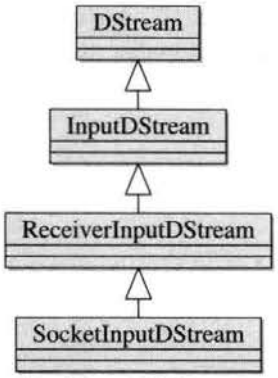


图 6.9 SocketInputDStream类继承体系

在InputStream的init函数中可以看到InputStream需要将自己添加到DStreamGraph中。

代码清单 6.8 InputStream.init

```
ssc.graph.addInputStream(this)
```

SocketInputDStream类实现中，最主要就是定义getReceiver函数，在getReceiver函数中只做一件事情，即产生一个新的SocketReceiver。

代码清单 6.9 SocketInputDStream.getReceiver

```
def getReceiver(): Receiver[T] = {
  new SocketReceiver(host, port, bytesToObjects, storageLevel)
}
```

创建完 SocketReceiver 之后，接下来的工作就是对 DStream 进行一系列的操作转换，对 Streaming 的实际应用开发也就集中在这样一个阶段。

代码清单 6.10 DStream转换过程

```
// Split each line into words
val words = lines.flatMap(_.split(" "))
```

```
import org.apache.spark.streaming.StreamingContext._
// Count each word in each batch
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)

// Print a few of the counts to the console
wordCounts.print()
```

在上述转换过程中，最为独特的一句就是print方法的调用。print属于输出操作，在针对DStream的所有转换操作中，如下操作归类为输出操作：

- print
- foreachRDD
- saveAsObjectFiles
- saveAsTextFiles
- saveAsHadoopFiles

上面涉及输出操作其实最终都会调用到 ForEachDStream，ForEachDStream 迥异于其他 DStream 子类的唯一地方就是重载了 generateJob。ForEachDStream类的实现如下所述。

代码清单 6.11 ForEachDStream

```
  private[streaming]
class ForEachDStream[T: ClassTag] (
    parent: DStream[T],
    foreachFunc: (RDD[T], Time) => Unit
  ) extends DStream[Unit](parent.ssc) {

  override def dependencies = List(parent)

  override def slideDuration: Duration = parent.slideDuration

  override def compute(validTime: Time): Option[RDD[Unit]] = None

  override def generateJob(time: Time): Option[Job] = {
    parent.getOrCompute(time) match {
      case Some(rdd) =>
        val jobFunc = () => {
```

```
          foreachFunc(rdd, time)
        }
        Some(new Job(time, jobFunc))
      case None => None
    }
  }
}
```

明确了输出操作之后，规划工作完成，提交运行，见证奇迹的时刻到了。

代码清单 6.12 执行StreamingSparkContext

```
ssc.start()             // Start the computation
ssc.awaitTermination()  // Wait for the computation to terminate
```

6.2.2 数据接收

图 6.10反映了 ssc.start 触发的运行逻辑，简述如下：调用 JobScheduler.start 函数，由 JobScheduler 依次启动以下三大功能模块。

- 监控模块。
- 数据接收模块。
- 启动定期生成Spark Job的JobGenerator。

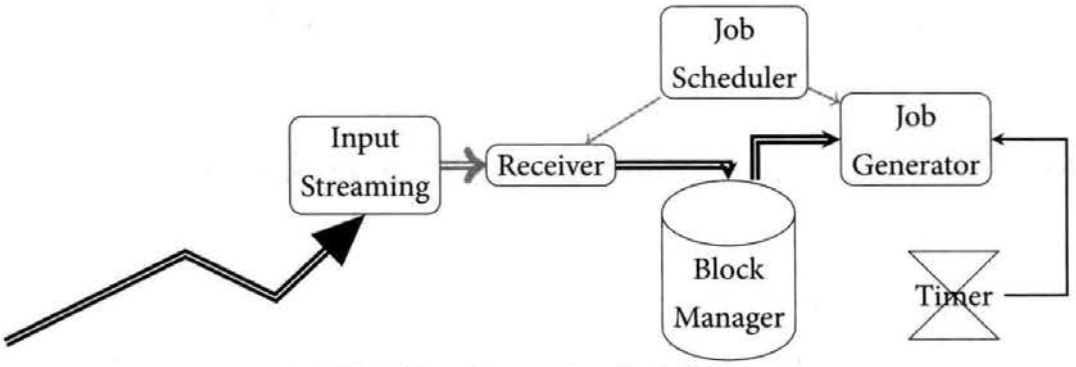


图 6.10 Streaming数据接收

Receiver运行于Worker启动的Executor中，而非Drirver Application。在下述的分析中会有对此结论的详细解释。

以SocketReceiver为例，看一看程序是如何一步步打开网络端口进行数据获取的。

代码清单 6.13 JobScheduler.start

```
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")
}
```

JobScheduler.start → ReceiverTracker.start。在 ReceiverTracker.start 函数中根据是否有 InputDStream 存在，进而打开相应的 Receiver 线程。

代码清单 6.14 ReceiverTracker.start

```
  def start() = synchronized {
  if (actor != null) {
    throw new SparkException("ReceiverTracker already started")
  }

  if (!receiverInputStreams.isEmpty) {
    actor = ssc.env.actorSystem.actorOf(Props(new ReceiverTrackerActor),
      "ReceiverTracker")
    receiverExecutor.start()
    logInfo("ReceiverTracker started")
  }
}
```

receiverTracker 是类 ReceiverLauncher 的实例，在 receiverLauncher 中开启的新线程会调用 startReceivers。

代码清单 6.15 ReceiverLauncher.start

```
@transient val thread  = new Thread() {
  override def run() {
    try {
      SparkEnv.set(env)
      startReceivers()
    } catch {
      case ie: InterruptedException => logInfo("ReceiverLauncher interrupted")
    }
  }
}

def start() {
  thread.start()
}
```

ReceiverLauncher中的startReceiver函数处理逻辑：

（1） 定义RDD。

（2） 定义作用于RDD上的函数startReceiver。

Receiver真正的执行是在Worker启动的Executor JVM之内。

代码清单 6.16 startReceivers

```
private def startReceivers() {
  val receivers = receiverInputStreams.map(nis => {
    val rcvr = nis.getReceiver()
    rcvr.setReceiverId(nis.id)
    rcvr
  })

  // Right now, we only honor preferences if all receivers have them
  val hasLocationPreferences = receivers.map(_.preferredLocation.isDefined).
   reduce(_ && _)

  // Create the parallel collection of receivers to distributed them on
```

```
  // the worker nodes
  val tempRDD =
    if (hasLocationPreferences) {
      val receiversWithPreferences = receivers.map(r => (r, Seq(r.
   preferredLocation.get)))
      ssc.sc.makeRDD[Receiver[_]](receiversWithPreferences)
    } else {
      ssc.sc.makeRDD(receivers, receivers.size)
    }

  // Function to start the receiver on the worker node
  val startReceiver = (iterator: Iterator[Receiver[_]]) => {
    if (!iterator.hasNext) {
      throw new SparkException(
        "Could not start receiver as object not found.")
    }
    val receiver = iterator.next()
    val executor = new ReceiverSupervisorImpl(receiver, SparkEnv.get)
    executor.start()
    executor.awaitTermination()
  }
  // Run the dummy Spark job to ensure that all slaves have registered.
  // This avoids all the receivers to be scheduled on the same node.
  if (!ssc.sparkContext.isLocal) {
    ssc.sparkContext.makeRDD(1 to 50, 50).map(x => (x, 1)).reduceByKey(_ + _, 20)
   .collect()
  }

  // Distribute the receivers and start them
  logInfo("Starting " + receivers.length + " receivers")
  ssc.sparkContext.runJob(tempRDD, startReceiver)
  logInfo("All of the receivers have been terminated")
}
```

运行于 Executor 之上的 startReceiver 函数，会首先启动 ReceiverSupervisor，然后由 ReceiverSupervisor 来触发 Receiver 执行。

代码清单 6.17 ReceiverSupervisor.startReceiver

```
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))
  }
}
```

Receiver的类继承体系如图 6.11所示。继续以SocketReceiver为例，再继续往前跟踪一步，就能看到Socket被真正地创建。

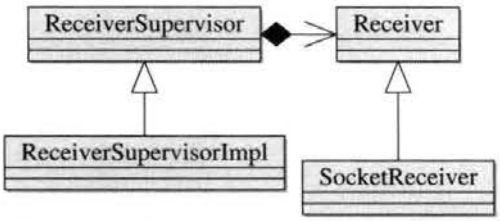


图 6.11 SocketReceiver继承体系

代码清单 6.18 SocketInputDStream.receive

```
def onStart() {
    // Start the thread that receives data over a connection
    new Thread("Socket Receiver") {
      setDaemon(true)
      override def run() { receive() }
    }.start()
  }

  /** Create a socket connection and receive data until receiver is stopped */
  def receive() {
    var socket: Socket = null
```

```
    try {
      logInfo("Connecting to " + host + ":" + port)
      socket = new Socket(host, port)
      logInfo("Connected to " + host + ":" + port)
      val iterator = bytesToObjects(socket.getInputStream())
      while(!isStopped && iterator.hasNext) {
        store(iterator.next)
      }
      logInfo("Stopped receiving")
      restart("Retrying connecting to " + host + ":" + port)
    } catch {
      case e: java.net.ConnectException =>
        restart("Error connecting to " + host + ":" + port, e)
      case t: Throwable =>
        restart("Error receiving data", t)
    } finally {
      if (socket != null) {
        socket.close()
        logInfo("Closed socket to " + host + ":" + port)
      }
    }
  }
```

再次强调，数据接收需要注意的一个主要问题是Receiver运行在Worker机器上的Executor JVM进程之中，而非Driver Application的JVM之内。

6.2.3 数据处理

有输入就要有输出，如果没有输出，则前面所做的所有动作全部没有意义，那么如何将这些输入和输出绑定起来呢？这个问题的解决就依赖于DStreamGraph，DStreamGraph记录输入的Stream和输出的Stream。

```
  private val inputStreams = new ArrayBuffer[InputDStream[_]]()
  private val outputStreams = new ArrayBuffer[DStream[_]]()

  var rememberDuration: Duration = null
  var checkpointInProgress = false
```

如果说InputDStream是用来解决启动Socket进行数据接收的问题，那么 OutputDStream存在的一个重要原因就是利用其来生成Spark Job。

Outputstreams 中的元素是在有 Output 类型的 Operation 作用于 DStream 上时自动添加到 DStreamGraph 中的。

Outputstream区别于Inputstream的一个重要地方就是会重载generateJob。

代码实现层面从两个角度来说，一个是控制层面（Control Panel），另一个是数据层面（Data Panel）。

Spark Streaming的数据接收过程的控制层面大致如图 6.12所示。

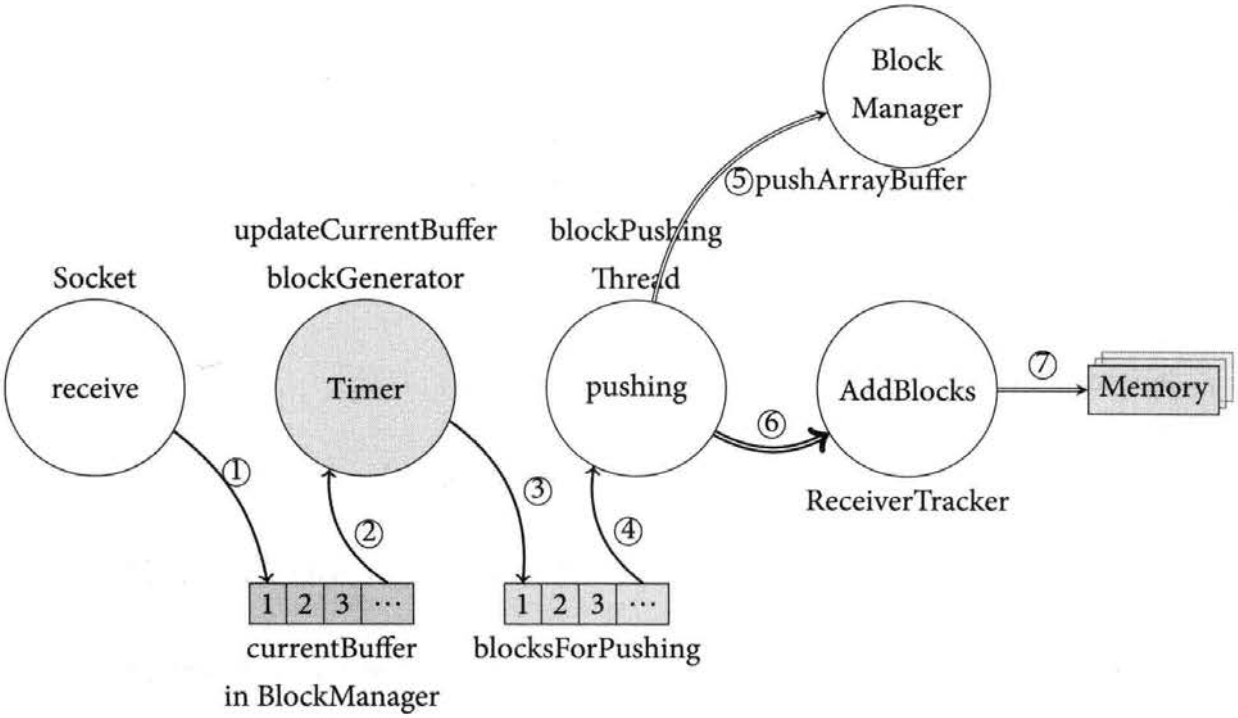


图 6.12 Streaming中数据的接收和存储

下面简要讲解一下图 6.12的意思：

- 数据真正接收到发生在SocketReceiver.receive函数中，将接收到的数据放入BlockGenerator.currentBuffer。
- 在BlockGenerator中有一个重复定时器，处理函数为updateCurrentBuffer，updateCurrentBuffer将当前buffer中的数据封装为一个新的Block，放入blocksForPush队列。
- 同样是在BlockGenerator中有一个BlockPushingThread，其职责就是不停地将 blocksForPush 队列中的成员通过pushArrayBuffer函数传递给BlockManager，让BlockManager将数据存储到MemoryStore中。
- pushArrayBuffer 还会将已经由 BlockManager 存储的 Block 的 id 号传递给 ReceiverTracker，

ReceiverTracker 会将存储的 blockId 放到对应 StreamId 的队列中。

图6.13展示了Socket接收时触发的函数调用路径。

```
socket.receive
    └ receiver.store
        └ pushSingle
            └ blockgenerator.updateCurrentBuffer
                └ blockgenerator.keepPushBlocks
                    └ pushArrayBuffer
                        └ ReceiverTracker.addBlocks
```

图 6.13 Socket接收时触发的函数调用路径

pushArrayBuffer函数的定义如下。

代码清单 6.19 pushArrayBuffer

```
def pushArrayBuffer(
    arrayBuffer: ArrayBuffer[_],
    optionalMetadata: Option[Any],
    optionalBlockId: Option[StreamBlockId]
  ) {
  val blockId = optionalBlockId.getOrElse(nextBlockId)
  val time = System.currentTimeMillis
  blockManager.put(blockId, arrayBuffer.asInstanceOf[ArrayBuffer[Any]],
    storageLevel, tellMaster = true)
  logDebug("Pushed block " + blockId + " in " + (System.currentTimeMillis -
   time)  + " ms")
  reportPushedBlock(blockId, arrayBuffer.size, optionalMetadata)
}
```

Spark Streaming数据处理高效的原因之一就是批量地进行数据分析，那么这些批量的数据是如何聚集起来的呢？

换种方式来表述这个问题，在某一时刻，接收到的数据是单一的，也就是我们最多只能组成<t,data>这种数据元组。而在runJob的时候是批量地提取和分析数据的。这个批量数据的组成是在什么时候完成的呢？

图 6.14大致勾勒出一条新的message被socketreceiver接收之后，是如何通过一系列的处理而放入到BlockManager中，并同时由ReceiverTracker记录下相应的元数据的。具体步骤如下。

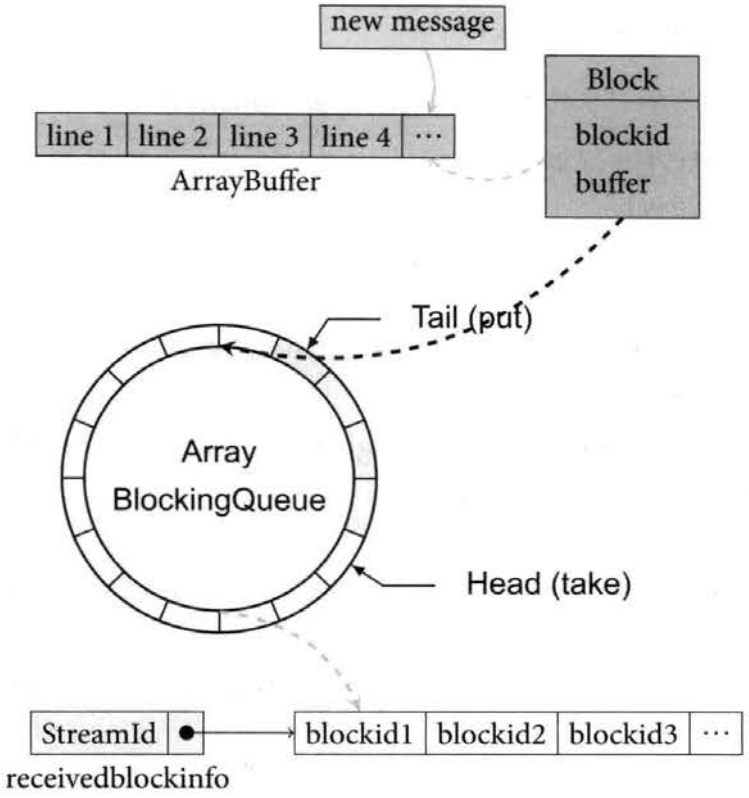


图 6.14 Block的生成和存放

图 6.14的理解如下：

（1） 首先new message被放入blockManager.currentBuffer。

（2） 定时器超时处理过程。将整个 currentBuffer 中的数据打包成一条 Block，放入 ArrayBlockingQueue，该数据结构支持 FIFO。

（3） keepPushingBlocks 将每一条 Block（Block 中包含时间戳，接收到的原始数据）让 BlockManager 进行保存，同时通知 ReceiverTracker 已经将哪些 Block 存储到了 BlockManager 中。

（4） ReceiverTracker将每一个Stream接收到但还没有进行处理的Block放入receiverBlockInfo，使用的数据结构是Hashmap。在后面的generateJobs中会从receiverBlockInfo提取数据以生成相应的RDD。

数据处理中最重要的函数就是generateJobs。generateJobs会引发下述的函数调用过程，具体的代码就不一一罗列了。

- jobgenerator.generateJobs → dstreamgraph.generateJobs → dstream.generateJob → getOrCompute → compute 生成RDD。
- Job调用job.func。

JobGenerator.generateJobs函数定义如下。

代码清单 6.20 generateJobs

```
private def generateJobs(time: Time) {
  SparkEnv.set(ssc.env)
  Try(graph.generateJobs(time)) match {
    case Success(jobs) =>
      val receivedBlockInfo = graph.getReceiverInputStreams.map { stream =>
        val streamId = stream.id
        val receivedBlockInfo = stream.getReceivedBlockInfo(time)
        (streamId, receivedBlockInfo)
      }.toMap
      jobScheduler.submitJobSet(JobSet(time, jobs, receivedBlockInfo))
    case Failure(e) =>
      jobScheduler.reportError("Error generating jobs for time " + time, e)
  }
  eventActor ! DoCheckpoint(time)
}
```

我们先来谈一谈数据处理阶段是如何与上述接收阶段中存储下来的数据挂上钩的。

假设上一次进行RDD处理发生在时间点t1，现在是时间点t2，那么在<t2,t1>之间有哪些Block没有被处理呢？

想必你已经知道答案了，没有被处理的Block全部保存在ReceiverTracker的receiverBlockInfo之中。

在 generateJob 时，每一个 DStream 都会调用 getReceivedBlockInfo。读者可能会说没有跟 ReceiverTracker 中的 receivedBlockInfo 连起来啊！别急！且看数据输入的源头 ReceiverInputDStream 中的 getReceivedBlockInfo 是如何定义的。代码列举如下。

代码清单 6.21 getReceivedBlockInfo

```
private[streaming] def getReceivedBlockInfo(time: Time) = {
  receivedBlockInfo(time)
}
```

那么，此处的receivedBlockInfo(time)是从何而来的呢？这个要看ReceivedInputDStream中的compute函数实现。

代码清单 6.22 ReceivedInputDStream.compute

```
override def compute(validTime: Time): Option[RDD[T]] = {
    // If this is called for any time before the start time of the context,
    // then this returns an empty RDD. This may happen when recovering from a
    // master failure
    if (validTime >= graph.startTime) {
      val blockInfo = ssc.scheduler.receiverTracker.getReceivedBlockInfo(id)
      receivedBlockInfo(validTime) = blockInfo
      val blockIds = blockInfo.map(_.blockId.asInstanceOf[BlockId])
      Some(new BlockRDD[T](ssc.sc, blockIds))
    } else {
      Some(new BlockRDD[T](ssc.sc, Array[BlockId]()))
    }
  }
```

至此终于看到了receiverTracker中的getReceivedBlockInfo被调用，也就是说将接收阶段的数据和目前处理阶段的输入通道打通了。

函数调用路径：从 generateJobs 到 sparkcontext.submitJobs。这个时候要注意注册为 DStream-Graph 中的 OutputStream 上的操作会引发 SparkContext.runJobs 被调用。我们以 print 函数为例看一下调用过程。

代码清单 6.23 print

```
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()
  }
```

注意rdd.take，其会引发runJob调用。不信的话，可以看一看take定义中调用runJob的片段。

代码清单 6.24 rdd.take

```
  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
}
```

传递给ForEachDStream构造函数的参数，一定会显式地调用 SparkContext.runJob。可以继续以saveAsHadoopFile为例来验证一下推断的准确与否。

代码清单 6.25 PairDStreamFunctions.saveAsHadoopFiles

```
  def saveAsHadoopFiles(
      prefix: String,
      suffix: String,
      keyClass: Class[_],
      valueClass: Class[_],
      outputFormatClass: Class[_ <: OutputFormat[_, _]],
      conf: JobConf = new JobConf
    ) {
    val saveFunc = (rdd: RDD[(K, V)], time: Time) => {
      val file = rddToFileName(prefix, suffix, time)
      rdd.saveAsHadoopFile(file, keyClass, valueClass, outputFormatClass, conf)
    }
    self.foreachRDD(saveFunc)
  }
```

定义在 PairDStreamFunctions 中的 saveAsHadoopFiles 传递给 foreachRDD 的参数中会调用到定义于 PairRDDFunctions 中的 saveAsHadoopFile，被 saveAsHadoopFile 调用到的 saveAsHadoopDataset 会显式调用 SparkContext.runJob。

也就是说其调用路径如图 6.15所示。

在print函数中另一个需要注意的地方就是rdd.take函数的算法逻辑。rdd.take 有可能会多次触发sc.runJob函数调用。

下面举个小的例子来说明。

PairDStreamFunctions.saveAsHadoopFiles
└ PairRDDFunctions.saveAsHadoopFile
└ saveAsHadoopDataSet
└ SparkContext.runJob

图 6.15 saveAsHadoopFiles到runJob的函数调用路径

代码清单 6.26 触发一次runJob的take

```
sc.parallelize(Seq(1,2,3,4,5,6,7,8,9),3).take(1)
```

从3个分区中取出一个元素，由于在第一个分区就能满足需求，所以只执行一次 runJob。

下面再来看另一个例子。

代码清单 6.27 触发多次runJob的take

```
sc.parallelize(Seq(1,2,3,4,5,6,7,8,9),3).take(1)
```

从3个分区中取出5个元素，由于一个分区最多只有3个元素，所以需要多次扫描。故runJob被触发了两次。

而print中是take(11)，如果在分片时间内只输入了少量内容，则无法达到11，故 runJob会被执行多次。

最后附上rdd.take的函数定义。

代码清单 6.28 rdd.take

```
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) {
      // The number of partitions to try in this iteration.
      // It is ok for this number to be
      // greater than totalParts because we actually cap it
      // at totalParts in runJob.
      var numPartsToTry = 1
```

```
    if (partsScanned > 0) {
      // If we didn't find any rows after the first iteration, just try
      // all partitions next.
      // Otherwise, interpolate the number of partitions we need to try,
      // but overestimate it by 50%.
      if (buf.size == 0) {
        numPartsToTry = totalParts - 1
      } else {
        numPartsToTry = (1.5 * num * partsScanned / buf.size).toInt
      }
    }
    numPartsToTry = math.max(0, numPartsToTry)
    // guard against negative num of partitions

    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
}
```

小结一下数据处理过程：

（1） 用time为关键字去取出在此时间之前加入的所有blockIds。

（2） 真正提交运行的时候，RDD中的blockfetcher以blockId为关键字去BlockManagerMaster获取真正的数据，即从Socket上接收到的原始数据。

处理结束时的清除工作如下。

代码清单 6.29 clearMetadata

```
private def clearMetadata(time: Time) {
```

```
    ssc.graph.clearMetadata(time)

    // If checkpointing is enabled, then checkpoint,
    // else mark batch to be fully processed
    if (shouldCheckpoint) {
      eventActor ! DoCheckpoint(time)
    } else {
      markBatchFullyProcessed(time)
    }
  }
```

6.2.4 BlockRDD

经过前面的分析知道，在定义的Batch Duration内容接收到的内容，会被创建为BlockRDD，然后针对该BlockRDD进行各种Operation。

换句话来说，BlockRDD是Spark Streaming产生的Job里头最源头的RDD。这时候有一个问题很容易被想到，那就是在一个分片时间（Batch Duration）的时间窗口内接收到的数据在处理的时候，是作为几个数据分区来进行处理的呢？

继续以WordCount为例，在netcat侧输入的每一行会被作为一个Block存入 BlockManager，假设分片时间（Batch Duration）定的是60 秒，在这期间输入了3行内容，那么创建3个Block，也就是3个Partition。

生成Block的时间间隔由参数spark.streaming.blockInterval决定，其默认值为200毫秒。

分别从源码和运行日志来验证这个结论。

步骤1: 在ReceivedInputDStream的compute函数中会创建BlockRDD。

代码清单 6.30 ReceivedInputDStream.compute

```
override def compute(validTime: Time): Option[RDD[T]] = {
    // If this is called for any time before the start time of the context,
    // then this returns an empty RDD. This may happen when recovering from a
    // master failure
    if (validTime >= graph.startTime) {
      val blockInfo = ssc.scheduler.receiverTracker.getReceivedBlockInfo(id)
      receivedBlockInfo(validTime) = blockInfo
      val blockIds = blockInfo.map(_.blockId.asInstanceOf[BlockId])
      Some(new BlockRDD[T](ssc.sc, blockIds))
```

```
    } else {
      Some(new BlockRDD[T](ssc.sc, Array[BlockId]()))
    }
  }
```

注意：入参blockIds反映的就是在分片时间（Batch Duration）内生成的Block数目。

步骤2: 根据blockIds来生成BlockRDD的Partitions。

代码清单 6.31 BlockRDD.getPartitions

```
class BlockRDD[T: ClassTag](@transient sc: SparkContext, @transient val blockIds:
    Array[BlockId])
  extends RDD[T](sc, Nil) {

  @transient lazy val locations_ = BlockManager.blockIdsToHosts(blockIds, SparkEnv
    .get)
  @volatile private var _isValid = true

  override def getPartitions: Array[Partition] = {
    assertValid()
    (0 until blockIds.size).map(i => {
      new BlockRDDPartition(blockIds(i), i).asInstanceOf[Partition]
    }).toArray
  }
}
```

日志分析

在netcat侧输入的每一行内容在存入BlockManager时，都会有如下的日志输出。

代码清单 6.32 存储Block的日志

```
INFO MemoryStore: ensureFreeSpace(18) called with curMem=0, maxMem=278019440
INFO MemoryStore: Block input-0-1408761997600 stored as bytes in memory (
    estimated size 18.0 B, free 265.1 MB)
INFO BlockManagerInfo: Added input-0-1408761997600 in memory on localhost.
    localdomain:33184 (size: 18.0 B, free: 265.1 MB)
INFO BlockManagerMaster: Updated info of block input-0-1408761997600
```

```
INFO SendingConnection: Initiating connection to [localhost.localdomain
   /127.0.0.1:33184]
INFO SendingConnection: Connected to [localhost.localdomain/127.0.0.1:33184], 1
   messages pending
INFO ConnectionManager: Accepted connection from [localhost.localdomain
   /127.0.0.1]
WARN BlockManager: Block input-0-1408761997600 already exists on this machine;
   not re-adding it
INFO BlockGenerator: Pushed block input-0-1408761997600
```

当Batch Duration时间窗口过期后，会生成相应的Spark Job，相应的日志中可以发现生成的Task数目与Block数目是一致的。

下述的日志内容是假设在Batch Duration中只输入了3行信息，所以只会有3个 Task生成。

代码清单 6.33 JobGenerator生成的Job运行时的日志

```
INFO DAGScheduler: Got job 1 (take at DStream.scala:608) with 1 output partitions
    (allowLocal=true)
INFO DAGScheduler: Final stage: Stage 1(take at DStream.scala:608)
INFO DAGScheduler: Parents of final stage: List(Stage 2)
INFO DAGScheduler: Missing parents: List(Stage 2)
INFO DAGScheduler: Submitting Stage 2 (MappedRDD[3] at map at MappedDStream.scala
   :35), which has no missing parents
INFO DAGScheduler: Submitting 3 missing tasks from Stage 2 (MappedRDD[3] at map
   at MappedDStream.scala:35)
INFO TaskSchedulerImpl: Adding task set 2.0 with 3 tasks
INFO TaskSetManager: Starting task 0.0 in stage 2.0 (TID 1, localhost,
   PROCESS_LOCAL, 1686 bytes)
INFO TaskSetManager: Starting task 1.0 in stage 2.0 (TID 2, localhost,
   PROCESS_LOCAL, 1686 bytes)
INFO TaskSetManager: Starting task 2.0 in stage 2.0 (TID 3, localhost,
   PROCESS_LOCAL, 1686 bytes)
INFO DAGScheduler: Got job 1 (take at DStream.scala:608) with 1 output partitions
    (allowLocal=true)
INFO DAGScheduler: Final stage: Stage 1(take at DStream.scala:608)
INFO DAGScheduler: Parents of final stage: List(Stage 2)
INFO DAGScheduler: Missing parents: List(Stage 2)
```

```
INFO DAGScheduler: Submitting Stage 2 (MappedRDD[3] at map at MappedDStream.scala
   :35), which has no missing parents
INFO DAGScheduler: Submitting 3 missing tasks from Stage 2 (MappedRDD[3] at map
   at MappedDStream.scala:35)
INFO TaskSchedulerImpl: Adding task set 2.0 with 3 tasks
INFO TaskSetManager: Starting task 0.0 in stage 2.0 (TID 1, localhost,
   PROCESS_LOCAL, 1686 bytes)
INFO TaskSetManager: Starting task 1.0 in stage 2.0 (TID 2, localhost,
   PROCESS_LOCAL, 1686 bytes)
INFO TaskSetManager: Starting task 2.0 in stage 2.0 (TID 3, localhost,
   PROCESS_LOCAL, 1686 bytes)
```

6.3 窗口操作

Spark Streaming中提供了一组窗口操作，通过使用滑动窗口技术来对大规模数据的增量更新进行统计分析。

举一个实际的例子，比如想知道新浪微博中最近5分钟的热门话题是哪些，并且要能够反映出热门话题变化的一个趋势，那么利用窗口操作会是一个非常好的解决办法。

滑动窗口原理如图 6.16所示。

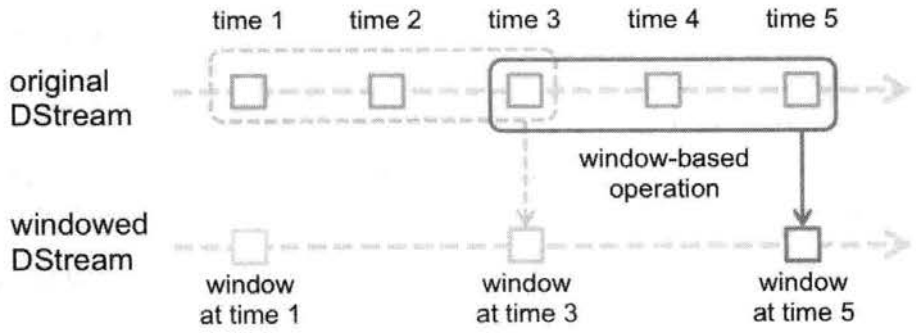


图 6.16 滑动窗口示意图

任何基于窗口的操作都需要指定两个参数：一个是窗口总的长度，另一个是滑动窗口的间隔。需要注意的是这两个参数的值必须是批量处理时间间隔的倍数。比如目前批量处理的时间间隔是2秒，那么窗口长度和滑动窗口间隔只能是2、4、6、8等。

比如想知道过去30秒内某个单词出现的次数，每10秒更新一次结果，那么可以使用如下的代码。

代码清单 6.34 实时更新单词出现次数

```
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),
    Seconds(30), Seconds(10))
```

6.4 容错性分析

JobGenerator.generateJobs 函数的最后会发出 DoCheckpoint 通知，该通知会让相应的 Actor 将 DStreamCheckpointData 写入 HDFS 文件。我们来看一看为什么需要写入 CheckpointData 及哪些东西是包含在 CheckpointData 中的。

在6.2.3节，我们已经分析到在 generateJobs 的时候会生成多个 Job，它们会通过 SparkContext.runJob 接口而发送到 Cluster 中被真正执行。这里有两个问题（见图6.17）。

- **问题1**：假设在t2，Worker挂掉了，挂掉的Worker直到t3才完全恢复。由于挂掉的原因，上一次generateJobs生成的Job不一定被完全处理了（也许有些已经处理了，有些还没有处理），所以需要重新再提交一次。这里有一个问题，那就是可能导致针对同一批数据有重复处理的情况发生，从而无法达到exactly-once的语义效果。
- **问题2**：在<t2,t3>这一段挂掉的时间之内，没有新的数据被接收，所以 Spark Streaming 的SocketReceiver 适合用来充当 Client 侧而不是 Server 侧。SocketReceiver 读取到的数据应该存在一个具有冗余备份机制的内存数据库或缓存队列里，如 Kafaka。对问题2，Spark Streaming本身是解决不了的。当然这里再往下细究的话，会牵出负载均衡的问题。

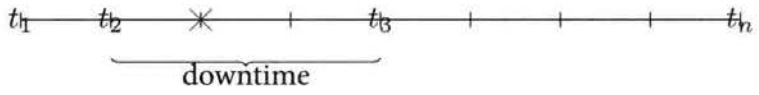


图 6.17 流处理故障发生时点示意图

代码清单 6.35 JobGenerator.doCheckpoint

```
private def doCheckpoint(time: Time) {
  if (shouldCheckpoint && (time - graph.zeroTime).isMultipleOf(ssc.
  checkpointDuration)) {
    logInfo("Checkpointing graph for time " + time)
    ssc.graph.updateCheckpointData(time)
```

```
      checkpointWriter.write(new Checkpoint(ssc, time))
    }
  }
```

Receiver需不需要区分自己到底是第一次启动还是第二次启动呢？

Checkpoint的成员变量有哪些呢？我们看一看其结构定义就清楚了。

代码清单 6.36 Checkpoint

```
private[streaming]
class Checkpoint(@transient ssc: StreamingContext, val checkpointTime: Time)
  extends Logging with Serializable {
  val master = ssc.sc.master
  val framework = ssc.sc.appName
  val sparkHome = ssc.sc.getSparkHome.getOrElse(null)
  val jars = ssc.sc.jars
  val graph = ssc.graph
  val checkpointDir = ssc.checkpointDir
  val checkpointDuration = ssc.checkpointDuration
  val pendingTimes = ssc.scheduler.getPendingTimes().toArray
  val delaySeconds = MetadataCleaner.getDelaySeconds(ssc.conf)
  val sparkConfPairs = ssc.conf.getAll

  def sparkConf = {
    new SparkConf(false).setAll(sparkConfPairs)
      .remove("spark.driver.host")
      .remove("spark.driver.port")
  }

  def validate() {
    assert(master != null, "Checkpoint.master is null")
    assert(framework != null, "Checkpoint.framework is null")
    assert(graph != null, "Checkpoint.graph is null")
    assert(checkpointTime != null, "Checkpoint.checkpointTime is null")
    logInfo("Checkpoint for time " + checkpointTime + " validated")
  }
}
```

代码清单 6.37 DStreamCheckpointData

```
class DStreamCheckpointData[T: ClassTag] (dstream: DStream[T])
  extends Serializable with Logging {
  protected val data = new HashMap[Time, AnyRef]()

  // Mapping of the batch time to the checkpointed RDD file of that time
  @transient private var timeToCheckpointFile = new HashMap[Time, String]
  // Mapping of the batch time to the time of the oldest checkpointed RDD
  // in that batch's checkpoint data
  @transient private var timeToOldestCheckpointFileTime =
          new HashMap[Time, Time]
  @transient private var fileSystem : FileSystem = null
  protected[streaming] def currentCheckpointFiles = data.asInstanceOf[HashMap
  [Time, String]]
```

generatedRDDs被包含在Graph里面。所以不要突然之间惊惶失措，发觉没有将generatedRDDs保存起来。

Checkpoint的数据通过CheckpointwriteHandler真正地写入HDFS，通过CheckpiontReader而读入。CheckpointReader在重启的时候会被使用到，判断是第一次干净地启动还是因错误而重启，判断的依据全部在cp这个变量。

为了达到重启之后而自动地检查并载入相应的Checkpoint数据，在创建StreamingContext的时候就不能简单地通过调用new StreamingContext来完成，而是利用getOrCreate函数，代码示例如下。

代码清单 6.38 getOrCreate

```
def getOrCreate(
      checkpointPath: String,
      creatingFunc: () => StreamingContext,
      hadoopConf: Configuration = new Configuration(),
      createOnError: Boolean = false
    ): StreamingContext = {
    val checkpointOption = try {
      CheckpointReader.read(checkpointPath,  new SparkConf(), hadoopConf)
    } catch {
      case e: Exception =>
```

```
        if (createOnError) {
          None
        } else {
          throw e
        }
    }
    //注意_的理解，表示将checkpointOption中
    //的每个成员作为StreamingContext构造函数
    //的第二个入参
    checkpointOption.map(new StreamingContext(null, _, null)).getOrElse(
    creatingFunc())
  }
```

上述代码的map函数中对StreamingContext的构造表明，入参sc_和batchDur_均为空，而cp_是从文件读入的。仔细看看StreamingContext的默认构造函数。

代码清单 6.39 StreamingContext

```
class StreamingContext private[streaming] (
  sc_ : SparkContext,
  cp_ : Checkpoint,
  batchDur_ : Duration
) extends Logging
```

在StreamingContext的init函数中非常主要的就是创建DStreamGraph实例。如果入参cp_非空，则表示DStreamGraph也需要从CheckpointFile恢复。

代码清单 6.40 DStreamGraph

```
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
    }
  }
```

一旦StreamingContext从CheckpointFile中读取恢复后，接下来的工作自然是重新启动Receiver来接收数据。此种情况下的Receiver启动工作与初次启动 Receiver并无任何差异。

这里需要注意的是当Driver退出的时候，原先在某一个Worker节点上的Executor 进程内运行着的Receiver也会由于Executor的退出而停止工作。也就是说 Receiver不会在Driver退出期间继续接收数据。这是完全遵循Spark集群资源管理原则的。如有问题，可以返回第 5章查看相应的内容。

在介绍完getOrCreate函数调用后引发的逻辑处理步骤之后，以RecoverableNetworkWordCount为例看一下getOrCreate函数的具体使用。

代码清单 6.41 RecoverableNetworkWordCount

```
object RecoverableNetworkWordCount {
  def createContext(ip: String, port: Int, outputPath: String) = {
    // If you do not see this printed, that means the StreamingContext has been
    // loaded from the new checkpoint
    println("Creating new context")
    val outputFile = new File(outputPath)
    if (outputFile.exists()) outputFile.delete()
    val sparkConf = new SparkConf().setAppName("RecoverableNetworkWordCount")
    // Create the context with a 1 second batch size
    val ssc = new StreamingContext(sparkConf, Seconds(1))

    // Create a socket stream on target ip:port and count the
    // words in input stream of \n delimited text (eg. generated by 'nc')
    val lines = ssc.socketTextStream(ip, port)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.foreachRDD((rdd: RDD[(String, Int)], time: Time) => {
      val counts = "Counts at time " + time + " " + rdd.collect().mkString("[", ",
     ", "]")
      println(counts)
      println("Appending to " + outputFile.getAbsolutePath)
      Files.append(counts + "\n", outputFile, Charset.defaultCharset())
```

```
    })
    ssc
  }

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.exit(1)
    }
    val Array(ip, IntParam(port), checkpointDirectory, outputPath) = args
    val ssc = StreamingContext.getOrCreate(checkpointDirectory,
      () => {
        createContext(ip, port, outputPath)
      })
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Master重启之后，会再次主动创建Receiver，将Receiver派发到某一特定的 Executor之上。

如果Worker挂掉怎么办？区分两种不同的情况：一个是普通Worker，上面没有运行Receiver；另一个是运行了Receiver。如果异常退出的Worker上没有运行Receiver，Worker被重启后对于整个计算结果没有影响，也不会有数据丢失。

对于运行了Receiver的Worker节点来说，如果其异常退出，则存在数据丢失的可能。考虑这种情况：数据已经从Kafka队列获取，但没有在Spark Cluster中进行复制，这样Worker重启后，原来的数据丢失。

如果Driver挂掉，怎么办？Driver如果异常退出，再次重启会通过Checkpoint Data来恢复退出前的处理场景。也就是，Master异常退出对于最终计算结果无影响。

能否达到exactly-once的处理效果？如果最终计算结果保存到HDFS，则能达到 exactly-once的处理效果，因为即便负责最终输出的Worker由于异常退出而重启进而重新恢复退出前的数据处理，数据被再次写入HDFS，由于HDFS中对同一位置的写入策略是覆盖性写入，所以最终结果一致。

但如果是写入数据库，则计算结果有被重复写入的可能，即存在重复计数的错误。不能达到exactly-once的效果。

6.5 Spark Streaming vs. Storm

6.5.1 Storm简介

Storm架构如图 6.18所示。

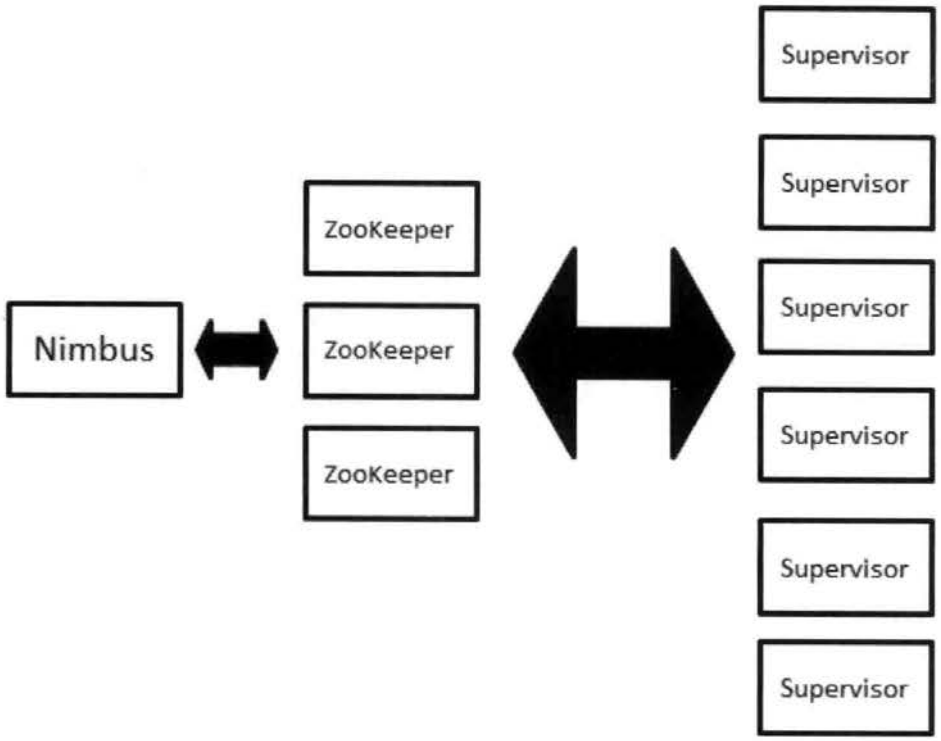


图 6.18 Storm架构图

构成Storm集群的主要节点如下。

- **Client**：提交应用，有点类似于Spark中的SparkSubmit。
- **Nimbus**：Master节点，兼具Spark中的Driver和Master的功能。
- **Supervisor**：负责监控Worker进程，类似于Spark中的Worker。
- **Worker**：Worker进程，类似于Spark中的Executor。
- **Executor**：Executor是具体运行各个任务的线程，类似于Spark中的TaskRunner。

与Spark采用Akka作为集群间通信框架不同，Storm主要依赖于ZooKeeper来维护整个集群，集群之间的消息通信采用ZeroMQ作为消息发送的组件。自Storm 0.90 版之后，已经可以使用Netty作为进程间通信的组件。

在JVM进程之中各线程之间的消息传递则采用了Disruptor Pattern，这是非常高效的线程间

消息发送机制。

Storm 中使用 Topology 作为应用开发的高级抽象。一个 Topology 由 Spout 和 Bolt 组成，在 Spout 和 Bolt 之间流动的是各种 Tuple。

Topology的整体示意如图 6.19所示。

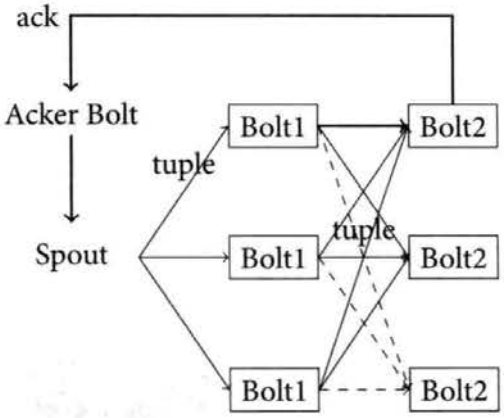


图 6.19 Storm Topology

图 6.20是Storm中Tuple被Worker接收及由Executor处理之后消息发送的示意图。

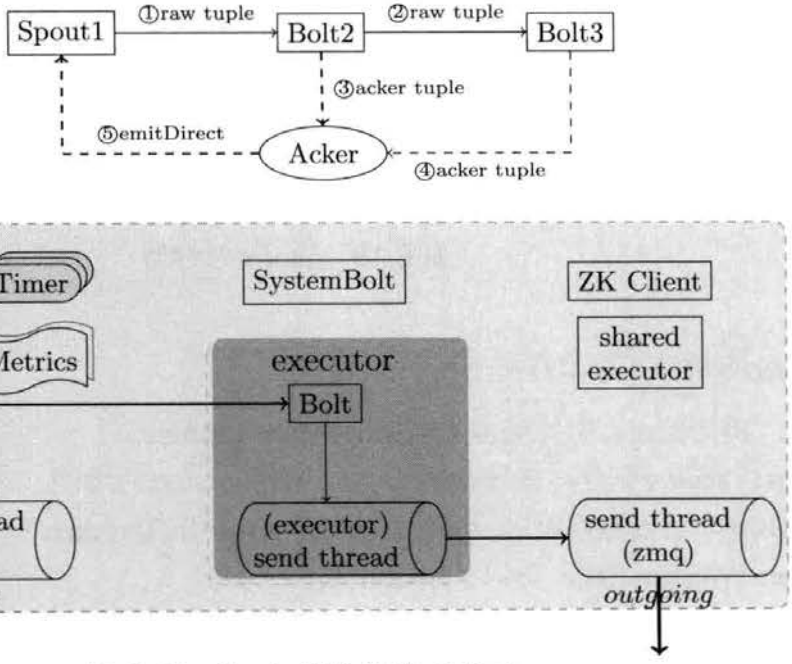


图 6.20 Tuple消息接收及处理

Storm最原始的Topology只能保证每个Tuple至少被处理一次，那么是如何知道一个Tuple已经被处理了呢？

原理其实很简单，也就是每个数字与自己相异或的结果为零，$A \oplus A = 0$。

每当源头的Spout向外发送一个Tuple的时候，Ack Bolt中都会记录下该新发送的Tuple Id，当下游的Bolt处理过完该Tuple并向下一跳的Bolt发送新产生的Tuple时，会将新Tuple和刚处理的Tuple两者Id异或的结果发送给Ack Bolt。Ack Bolt将收到的值与原先持有的值继续实行异或操作，当最原始的Tuple派发出来的Tuple都已经被处理时，Ack Bolt针对该Tuple异或出来的结果必然为零。

如果超时后，计算结果为非零，则需要重新发送原始的Tuple。

没有处理和多次处理都会导致计算结果不准确。为了达到**只处理一次**的效果，Storm又提供了TridentTopology来保证exactly-once。

在编程接口上，从某种程度上来说Storm的TridentTopology非常类似于Spark的 DStream。下列代码片段显示了如何使用TridentTopology来写WordCount。

代码清单 6.42 WordCount in TridentTopology

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout1", spout).parallelismHint
  (16).each(new Fields("sentence"),
new Split(), new Fields("word")).groupBy(new Fields("word")).persistentAggregate
  (new MemoryMapState.Factory(),
new Count(), new Fields("count")).parallelismHint(16);
```

在TridentTopology中有相应的TridentSpout，它又是如何知道消息已经被下游处理了呢？

（1） 在每跳中认为自己处理完毕的时候，它都会告诉下一跳，即下游，我给你发送了多少Tuple。如果下游将上游发送过来的确认消息与自身确实已经处理的消息比对一致的话，则认为处理都完成，于是发送ack。

（2） 问题的关键变成每一个Bolt是如何判断自己已经处理完毕的呢？请看下一步。

（3） 总有一个Bolt是没有上游的，即TridentSpoutExecutor，它只会收到启动指令，但不接收真正的业务数据，于是它会告诉下一跳：我发了多少Tuple给你。

在集群资源管理方面，Storm的容错性也是很不错的。就不同节点失效后给整个集群带来的影响如下所述。

- **Worker失效**：Worker节点一直处于Supervisor的监视下，如果Worker失效，就会被Supervisor重新拉起。
- **Supervisor**：如果Supervisor异常退出，一般会由监控它们的进程如runit或其他类似的工具

来将其重新拉起。对于Worker进程来说，Supervisor的重启过程对其无任何影响。

- **Nimbus**：像Supervisor一样，Nimbus也会被runit重新带起。

Nimbus是整个Storm集群中唯一存在单点失效可能的节点。如果没有Nimbus，则无法提交新任务到Storm集群，同时如果有Worker异常退出，则失效的工作也不会被重新分配到其他可工作的机器。

6.5.2 Storm和Spark Streaming对比

表 6.2总结了Storm和Spark Streaming两者之间的区别。

表 6.2 Storm与Spark Streaming比较

	Storm	Spark Streaming
处理模型	record-at-a-time	批量处理
消息获取模式	push	pull
延迟	Sub-second	few second
吞吐量	低	高
exactly-once	通过Trident Topology支持	支持
Cluster Rebalance	支持	不支持
集群管理	ZooKeeper	Akka
实现语言	Clojure	Scala
API支持的语言	Clojure，Java	Scala，Java
并发数动态可调	支持	不支持
社区活跃程度	活跃	活跃
大公司支持	Hortonwork	Cloudera

在实时性上Storm更优一些，在吞吐量上Spark略胜一筹。

6.6 应用举例

Kafka结合Spark的运行示例如下（见图6.21）。

6.6.1 搭建Kafka Cluster

介绍Kafka的特性和实验的例子如下。

步骤1：启动Kafka Server。

代码清单 6.43 启动Kafka集群

```
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

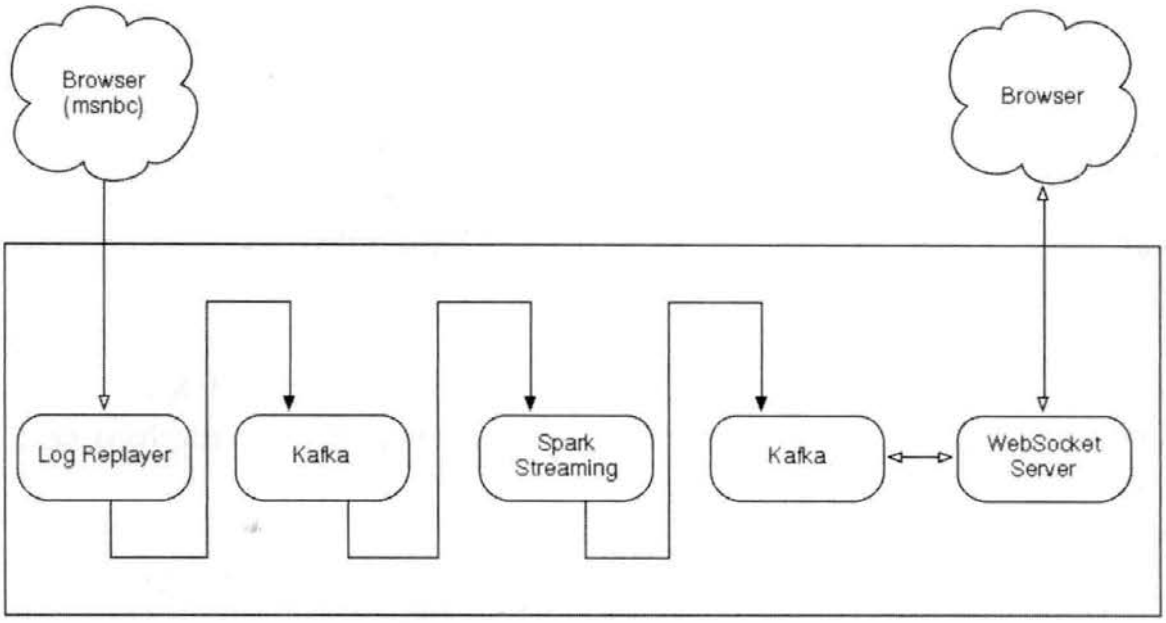


图 6.21 Spark Streaming应用

步骤2：创建Topic。

代码清单 6.44 创建Topic

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --
    partitions 1 --topic test
```

步骤3：作为Producer发送消息。

代码清单 6.45 消息发送

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

6.6.2 KafkaWordCount

代码清单 6.46 KafkaWordCount

```
object KafkaWordCount {
  def main(args: Array[String]) {
    if (args.length < 4) {
```

```
      System.err.println("Usage: KafkaWordCount <zkQuorum> <group> <topics> <
    numThreads>")
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()

    val Array(zkQuorum, group, topics, numThreads) = args
    val sparkConf = new SparkConf().setAppName("KafkaWordCount")
    val ssc =  new StreamingContext(sparkConf, Seconds(2))
    ssc.checkpoint("checkpoint")

    val topicpMap = topics.split(",").map((_,numThreads.toInt)).toMap
    val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicpMap).map(_._2)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1L))
      .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(2), 2)
    wordCounts.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```

在运行KafkaWordCount之前，先将ConsoleProducer和ConsoleConsumer停止运行。

步骤1：运行KafkaWordCountProducer。

代码清单 6.47 运行KafkaWordCountProducer

```
bin/run-example org.apache.spark.examples.streaming.KafkaWordCountProducer
    localhost:9092 test 3 5
```

解释一下参数的意思，localhost:9092表示Producer的地址和端口，test表示Topic，3表示每秒发多少条消息，5表示每条消息中有几个单词。

步骤2：运行KafkaWordCount。

代码清单 6.48 运行KafkaWordCount

```
bin/run-example org.apache.spark.examples.streaming.KafkaWordCount localhost:2181
    test-consumer-group test 1
```

在此解释一下参数的意思：

- localhost:2181表示ZooKeeper的监听地址。
- test-consumer-group 表示 consumer-group 的名称，必须和 $KAFKA_HOME/config/consumer.properties 中 group.id 的配置内容一致。
- test表示topic，1表示线程数。

第7章

SQL

“不特推陈出新，饶有别致。”

《秋灯丛话》

在Spark 1.0中有一个新增的功能，即对SQL的支持，也就是说可以用SQL来对数据进行查询。这对于DBA来说无疑是一大福音，因为以前的知识继续生效，而无须去学什么Scala或其他Script。

一般来说任意一个SQL子系统都需要有parser、optimizer、execution三大功能模块，在Spark中这些又都是如何实现的呢？这些实现又有哪些亮点和问题？带着这些疑问，本章做一些比较深入的分析。

SQL模块分析有几大难点，分别是：

（1） SQL分析和执行的通用过程，这个与是否用Spark无关，应该是非常基本的问题。

（2） Spark SQL中具体实现时的整体架构。

（3） 源码阅读时碰到的Scala特殊语法，也就是常说的语法糖问题。

SQL是一种标准，一种用来进行数据分析的标准，已经存在多年。

在大数据的背景下，随着数据规模的日渐增大，原有的分析技巧是否就过时了呢？答案显然是否定的。原来的分析技巧在既有的分析维度上依然保持有效，当然对于新的数据我们想挖掘出更多有意思、有价值的内容，这个目标可以交给数据挖掘或者机器学习去完成。

那么原有的数据分析人员如何快速地转换到Big Data的平台上来呢？去重新学一种脚本吗？直接用Scala或Python去编写RDD？显然这样的代价太高，学习成本大。数据分析人员希望底层存储机制和分析引擎的变换不要对上层分析的应用有直接的影响，需求用一句话来表达就是，“直接使用SQL语句来对数据进行分析”。

这也是为什么Hive兴起的原因了，Hive的流行直接证明这种设计贴近市场需求。但由于Hive是采用了Hadoop的MapReduce作为分析执行引擎，其处理速度不尽如人意。Spark以快著称，很快有高人基于Hive写出了Shark。据称Shark的运行速度比Hive快100多倍，Shark取得了非常不俗的成绩，赢得了极好的口碑。

然而Shark毕竟是游离于Spark之外的一个项目，不受Spark节制，而Spark开发团队的目标是将对SQL的支持放到Spark的核心功能中。以上分析就是Spark中SQL功能的由来。

在互联网企业和有大数据处理需求的传统企业中，基于Hadoop构建的数据仓库的数据来源主要有以下几个，详见表7.1。

表 7.1 数据仓库中的数据来源汇总

Flume、Sribe、Chukwa	日志收集和分析系统把来自Apache/Nginx的日志收集到HDFS上，然后通过Hive查询
Sqoop	把用户和业务维度数据（一般存储在Oracle/MySQL中）定期导入Hive，那么OLTP数据就有了一个用于OLAP的副本了
ETL	通过ETL工具从其他外部DW数据源里导入的数据

目前所有的SQL On Hadoop产品其实都是在某个或者某些特定领域内适合的，没有“银弹”（Silver Bullet）。像当年Oracle/Teradata这样的满足几乎所有企业级应用的产品在大数据时代是不现实的。所以，每一种SQL On Hadoop产品都在尽量满足某一类应用的特征。典型需求如下：

- Interactive Query （ms～3min）。
- Data Analyst，reporting query （3min～20min）。
- Data Mining，modeling and large ETL （20 min ～ hr ～ day）。
- 机器学习需求（通过MapReduce/MPI/Spark等计算模型来满足）。

像其他章一样，在继续深入探究Spark SQL的具体实现之前，先看一个简单的使用实例。

代码清单 7.1 spark sql example

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc);
import sqlContext._
case class Person(name: String, age: Int)
val person = sc.textFile("examples/src/main/resources/people.txt").map(_.split("
```

```
    )).map(p => Person(p(0), p(1).trim.toInt))
  person.registerAsTable("person")
  val teenagers = sql("SELECT name, age FROM person WHERE age >= 13 and age <= 19"
    )
  teenagers.map(t => "name:" + t(0)).collect().foreach(println)
```

上述代码的逻辑非常清晰，就是将存在于person.txt中年龄介于13至19岁的年轻人名字打印出来。

7.1 SQL语句的通用执行过程分析

SQL语句大家都很熟悉，那么你有没有仔细想过其由几大部分组成呢？可能你会说："这还用问，不就是'select * from tablex where f1=?'？有什么好想的。"

还是先来看看再说吧，说不定有些新的思维在里面呢。

图 7.1是对最简单的SQL语句的重新标注。SELECT表示是一种具体的操作，即查询数据；"f1,f2,f3"表示返回的结果；tableX是数据源；condition部分是查询条件。

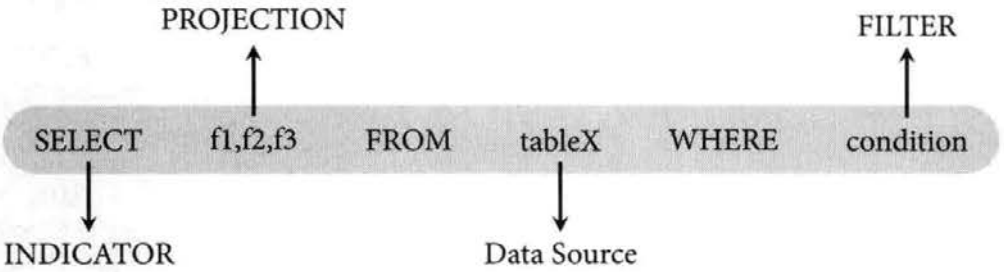


图 7.1 SQL语句组成

你有没有发觉SQL表达式中的顺序与常见的RDD处理逻辑在表达顺序上有差异？两者之间在顺序上的差异如图 7.2所示。

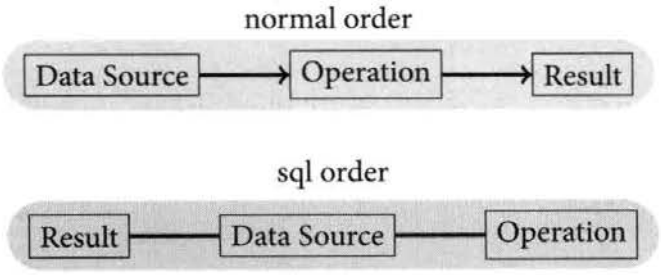


图 7.2 RDD和SQL表达顺序比较

SQL语句在分析执行过程中会经历如图7.3所示的几个步骤：

（1） 语法解析。
（2） 操作绑定。
（3） 优化执行策略。
（4） 交付执行。

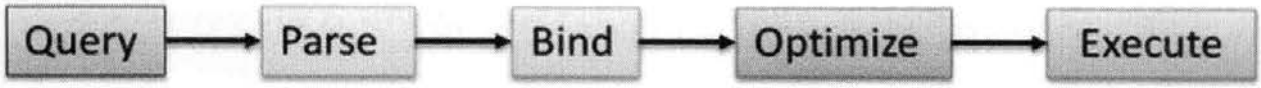


图 7.3 SQL执行过程

语法解析之后，会形成一棵语法树，树中的每个节点是执行的规则（Rule），整棵树被称为执行策略。图7.4是解析之后形成的语法树。

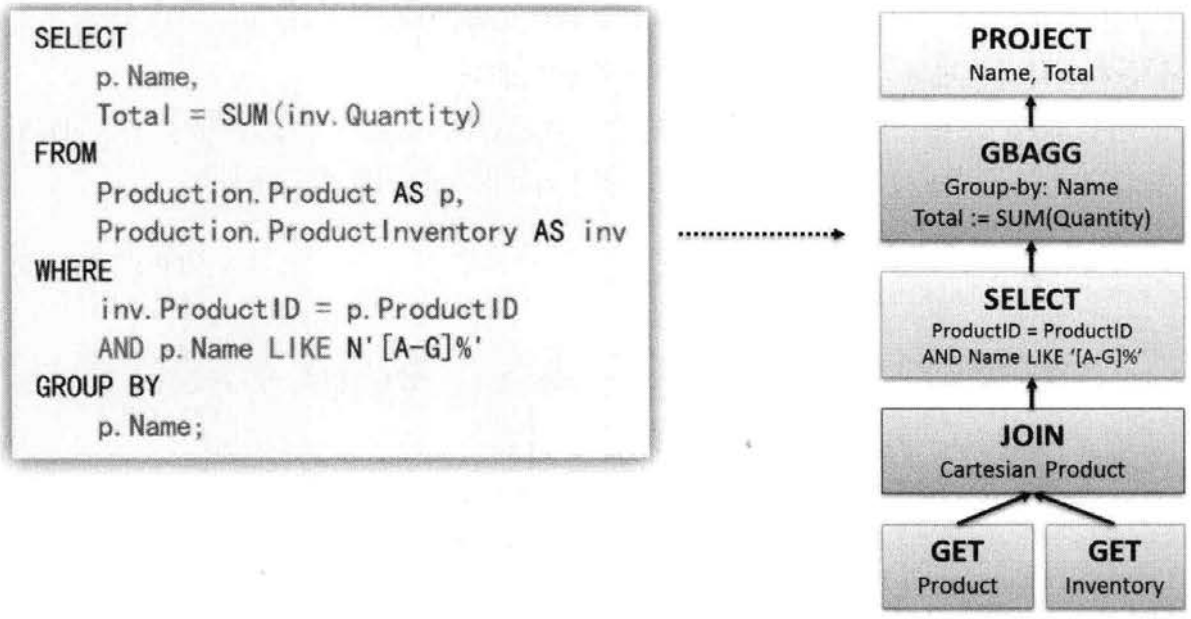


图 7.4 SQL语法树

形成上述执行策略树还只是第一步，因为这个执行策略可以进行优化。所谓的优化就是对树中的节点进行合并或是进行顺序上的调整。

以大家熟悉的Join操作为例，图7.5给出一个Join优化的示例。A Join B等同于B Join A，但是顺序的调整可能给执行的性能带来极大的影响。

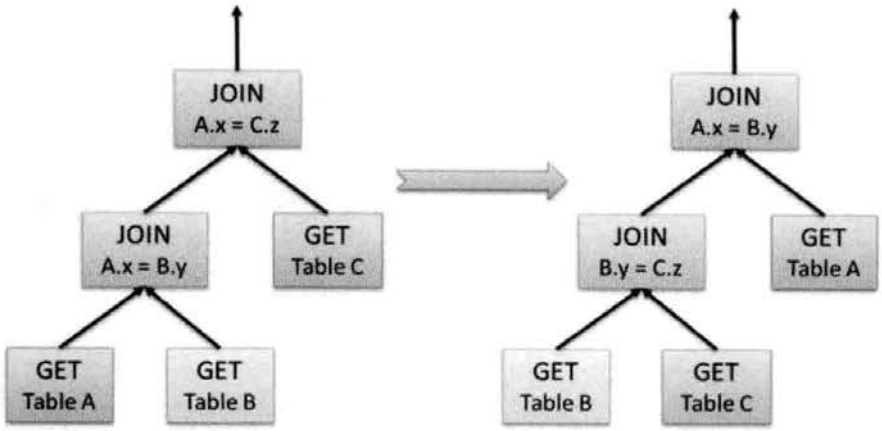


图 7.5 SQL策略优化

再举一例，一般来说尽可能地先实施聚合操作（Aggregate），然后再合并（Join），如图7.6所示。

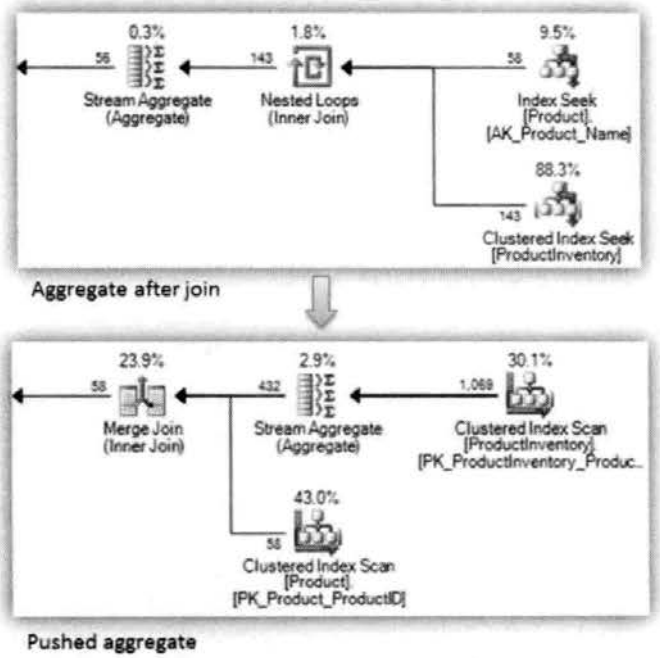


图 7.6 先聚合后合并

7.2 SQL On Spark的实现分析

7.2.1 SqlParser

有了上述内容的铺垫，想必你已经意识到Spark如果要很好地支持SQL，势必也要完成解析、优化、执行的三大过程。

整个SQL部分的代码大致分类如图 7.7所示。

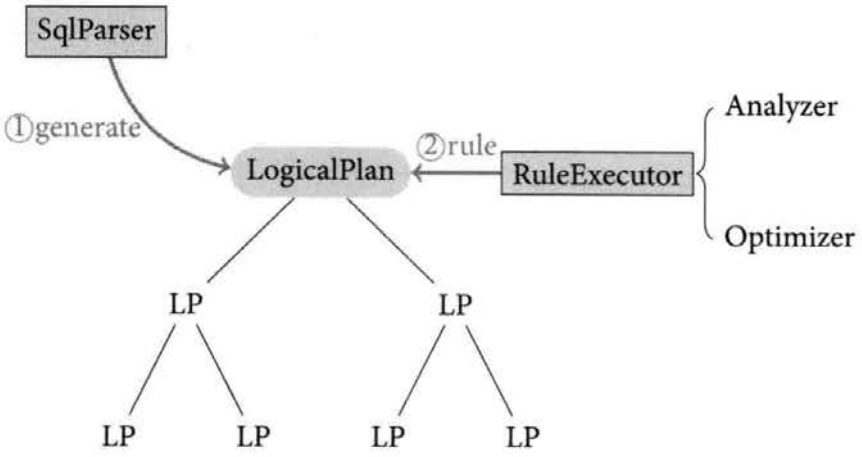


图 7.7 SQL On Spark

处理步骤顺序如下所述：

（1） SqlParser生成LogicPlan Tree。

（2） Analyzer和Optimizer将各种Rule作用于LogicalPlan Tree。

（3） 最终优化生成的LogicalPlan生成Spark RDD。

（4） 最后将生成的RDD交由Spark执行。

图 7.8是Spark官方给出的解析执行过程的整体框架图。

一般来说Spark每支持一种新的应用开发，都会引入一个新的Context及相应的RDD，对于SQL这一特性来说，引入的就是SQLContext和SchemaRDD。

在SQLContext中定义的sql函数用以解析生成SchemaRDD，且看看sql函数的具体实现。

代码清单 7.2 sql

```
def sql(sqlText: String): SchemaRDD = new SchemaRDD(this, parseSql(sqlText))
```

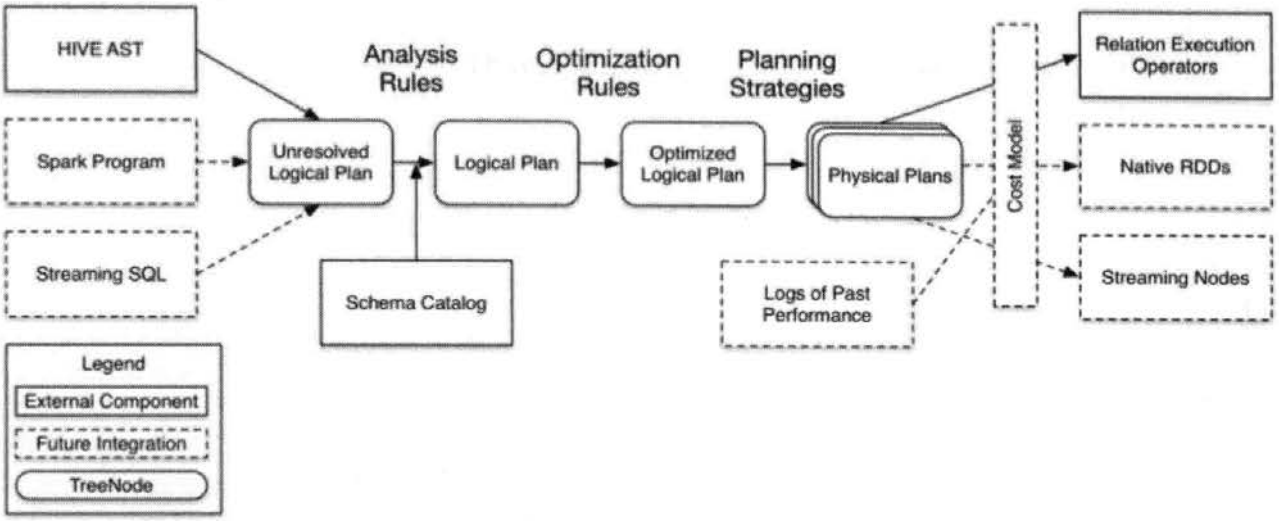


图 7.8 Spark SQL处理的整体流程

parseSql(sqlText)负责生成LogicalPlan，parseSql是SqlParser的一个实例，代码如下。

代码清单 7.3 parseSql

```
protected[sql] val parser = new catalyst.SqlParser

protected[sql] def parseSql(sql: String): LogicalPlan = parser(sql)
```

由于apply函数可以不被显式调用，所以parseSql(sqlText)一句其实会隐式地调用SqlParser中的apply函数。

代码清单 7.4 apply

```
def apply(input: String): LogicalPlan = {
    if (input.trim.toLowerCase.startsWith("set")) {
      input.trim.drop(3).split("=", 2).map(_.trim) match {
        case Array("") => // "set"
          SetCommand(None, None)
        case Array(key) => // "set key"
          SetCommand(Some(key), None)
        case Array(key, value) => // "set key=value"
          SetCommand(Some(key), Some(value))
      }
    } else {
      //这是最让人头痛和摸不着头脑的一句代码，
```

```
    //下文会有详细的分析
    phrase(query)(new lexical.Scanner(input)) match {
      case Success(r, x) => r
      case x => sys.error(x.toString)
    }
  }
}
```

SqlParser类的继承体系如图 7.9所示。

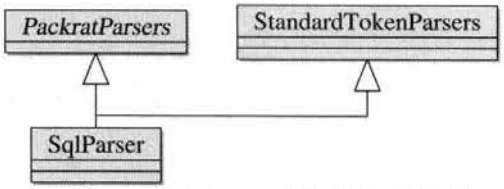


图 7.9 SqlParser的类体系结构

最让人头疼的一行代码就是phrase(query)(new lexical.Scanner(input))了，翻译过来就是如果输入的input字符串符合Lexical中定义的规则，则继续使用query处理。

看一看phrase的具体定义，再来解释()()的意思是什么。

代码清单 7.5 phrase

```
// phrase(query)(new lexical.Scanner(input))
// query是phrase中的参数p，而new lexical.Scanner(input)
//则是apply中的参数in Scala中的currying function
//特性不是那么容易好理解的
  override def phrase[T](p: Parser[T]) = {
  val q = super.phrase(p)
  new PackratParser[T] {
    def apply(in: Input) = in match {
      case in: PackratReader[_] => q(in)
      case in => q(new PackratReader(in))
    }
  }
}
```

解析器是编译原理中的主要话题之一，解析器先要进行词法分析再进行语法分析，词法分析的主要目的就是将输入的语句解析为一个个Token。

而语法分析的目的是在词法分析的基础上将单词序列组合成各类语法短语。对于SqlParser来说，就是组合成各个LogicalPlan。

看一下query的定义是什么。

代码清单 7.6 query

```
protected lazy val query: 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 ~ opt(DISTINCT) ^^^ { (q1: LogicalPlan, q2: LogicalPlan) => Distinct
  (Union(q1, q2)) }
    )
  | insert | cache
)
```

其中 ^^^是 Parser 中的函数，其详细解释见表7.2中的说明，有关源码详见 scala.util.parsing.combinator.Parsers$Parser。

表 7.2 Parser中的特殊符号含义

<~	只保留左侧内容 A<~B，只停留A
~>	只保留右侧内容 A~>，只保留B
^^	根据匹配结果生成语法短语
^^^	将语法短语转换为另外的值，注意与^^的区别
~	连接符 A~B 表示模式匹配是B紧跟于A之后
|	或者 A|B 表示模式要么由A组成，要么由B组成

有关Parser中这些特殊符号的具体含义，请参考scala.util.parsing.combinator.Parsers$Parser中的API。

有了表 7.2中的解释之后，再以select为例说明解析过程、词法分析和语法分析。

代码清单 7.7 select

```
protected lazy val select: Parser[LogicalPlan] =
    SELECT ~> opt(DISTINCT) ~ projections ~
    opt(from) ~ opt(filter) ~
```

```
    opt(grouping) ~
    opt(having) ~
    opt(orderBy) ~
    opt(limit) <~ opt(";") ^^ {
      case d ~ p ~ r ~ f ~ g ~ h ~ o ~ l  =>
        val base = r.getOrElse(NoRelation)
        val withFilter = f.map(f => Filter(f, base)).getOrElse(base)
        val withProjection =
          g.map {g =>
            Aggregate(assignAliases(g), assignAliases(p), withFilter)
          }.getOrElse(Project(assignAliases(p), withFilter))
        val withDistinct = d.map(_ => Distinct(withProjection)).getOrElse(
    withProjection)
        val withHaving = h.map(h => Filter(h, withDistinct)).getOrElse(
    withDistinct)
        val withOrder = o.map(o => Sort(o, withHaving)).getOrElse(withHaving)
        val withLimit = l.map { l => Limit(l, withOrder) }.getOrElse(withOrder)
        withLimit
  }
```

以relations为例看看如何根据匹配到的字符串生成LogicalPlan。

代码清单 7.8 relations

```
  protected lazy val relations: Parser[LogicalPlan] =
  %%下述语句表示将以匹配到的r1,r2为入参生成Join
  relation ~ "," ~ relation ^^ { case r1 ~ _ ~ r2 => Join(r1, r2, Inner, None) } |
  relation
```

根据Join类的继承关系（见图 7.10）可以看出，Join继承自LogicalPlan。

Join的定义如代码清单 7.9所示。

代码清单 7.9 Join

```
case class Join(
  left: LogicalPlan,
  right: LogicalPlan,
  joinType: JoinType,
```

```
  condition: Option[Expression]) extends BinaryNode {

  override def references = condition.map(_.references).getOrElse(Set.empty)

  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
    }
  }
}
```

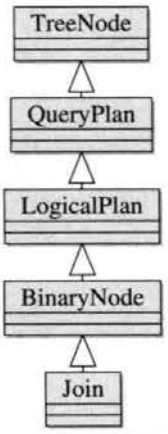


图 7.10 Join的类继承体系

到了这里终于看到有LogicalPlan了，也就是说将普通的String转换成LogicalPlan在这里发生了。

代码清单 7.10 select

```
val teenagers = sql("SELECT name, age FROM person WHERE age >= 13 and age <= 19")
```

在spark-shell中运行本章开头的例子时，将调试级别设置为Debug，可以看到经过SqlParser的分析之后得到的结果如下。

代码清单 7.11 SqlParser分析结果

```
Project ['name]
 Filter (('age >= 13) && ('age <= 19))
  UnresolvedRelation None, people, None
```

query这段代码同时说明，在目前的Spark SQL中仅支持Select、Insert和Cache这3种操作，至于Delete、Update暂不支持。

7.2.2 Analyzer

第一阶段，将String转换为Unresolved Logicalplan，第二阶段是将UnResolved的对象变成Resolved。

由于Scala支持惰性执行的特性，第二阶段Analyzer的调用关系就不是很明显了。所以第一步就是找到Analyzer被执行的触发点。在Spark Core部分的源码分析中得知，在Job执行的时候，需要通过getDependencies来解决RDD的依赖关系。

SQL中就利用了这一执行路径来让真正需要执行Job的时候，才对Unresolved LogicalPlan进行分析及优化。

代码清单 7.12 getDependencies

```
override protected def getDependencies: Seq[Dependency[_]] =
   List(new OneToOneDependency(queryExecution.toRdd))
```

queryExecution定义于SchemaRDDLike中。

代码清单 7.13 queryExecution

```
  lazy val queryExecution = sqlContext.executePlan(baseLogicalPlan)
```

executePlan定义在源文件SQLContext.scala中。

代码清单 7.14 executePlan

```
  protected[sql] def executePlan(plan: LogicalPlan): this.QueryExecution =
  new this.QueryExecution { val logical = plan }
```

怎么可以创建抽象类的实例？我的世界坍塌了，呵呵。不要紧张，这在Scala的世界是允许的，只不过Scala是隐含地创建了一个QueryExecution的子类并初始化而已，Java里的原则还是对的，人家背后有“猫腻”。

Ok，轮到第二阶段中最重要的角色QueryExecution闪亮登场了。

代码清单 7.15 QueryExeution

```
protected abstract class QueryExecution {
  def logical: LogicalPlan

  lazy val analyzed = analyzer(logical)
  lazy val optimizedPlan = optimizer(analyzed)
  lazy val sparkPlan = planner(optimizedPlan).next()
  lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)

  /** Internal version of the RDD. Avoids copies and has no schema */
  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 = stringOrError(executedPlan)

  override def toString: String =
  s"""== Logical Plan ==
  |${stringOrError(analyzed)}
  |== Optimized Logical Plan ==
  |${stringOrError(optimizedPlan)}
  |== Physical Plan ==
  |${stringOrError(executedPlan)}
  """.stripMargin.trim

  def debugExec() = DebugQuery(executedPlan).execute().collect()
}
```

QueryExecution在处理过程中的三大步，这三大步分成两个主要阶段：一个是LogicalPlan的处理，另一个是PhysicalPlan的处理。

- lazy val analyzed = analyzer(logical)
- lazy val optimizedPlan = optimizer(analyzed)
- lazy val sparkPlan = planner(optimizedPlan).next()

无论Analyzer还是Optimizer，它们都是RuleExecutor的子类，类的继承关系如图7.11所示。

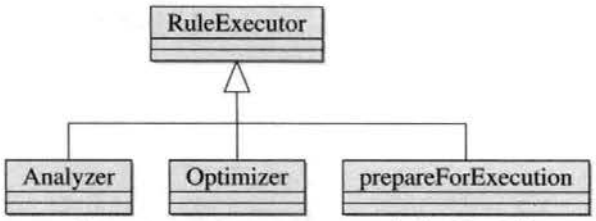


图 7.11 RuleExecutor类体系结构

RuleExecutor的默认处理函数是apply，对所有的子类都是一样的。RuleExecutor的apply函数定义如下。

代码清单 7.16 RuleExecutor.apply

```
def apply(plan: TreeType): TreeType = {
    var curPlan = plan

    batches.foreach { batch =>
      val batchStartPlan = curPlan
      var iteration = 1
      var lastPlan = curPlan
      var continue = true

      // Run until fix point (or the max number of iterations as specified
      // in the strategy.
      while (continue) {
        curPlan = batch.rules.foldLeft(curPlan) {
          case (plan, rule) =>
            val result = rule(plan)
            if (!result.fastEquals(plan)) {
              logger.trace(
                s"""
                  |=== Applying Rule ${rule.ruleName} ===
                  |${sideBySide(plan.treeString, result.treeString).mkString("\n")
    }
```

```
            """.stripMargin)
        }

        result
      }
      iteration += 1
      if (iteration > batch.strategy.maxIterations) {
        logger.info(s"Max iterations ($iteration) reached for batch ${batch.name
}")
        continue = false
      }

      if (curPlan.fastEquals(lastPlan)) {
        logger.trace(s"Fixed point reached for batch ${batch.name} after
$iteration iterations.")
        continue = false
      }
      lastPlan = curPlan
    }

    if (!batchStartPlan.fastEquals(curPlan)) {
      logger.debug(
        s"""
        |=== Result of Batch ${batch.name} ===
        |${sideBySide(plan.treeString, curPlan.treeString).mkString("\n")}
      """.stripMargin)
    } else {
      logger.trace(s"Batch ${batch.name} has no effect.")
    }
  }

  curPlan
}
```

对于RuleExecutor的子类来说，最主要的是定义自己的batches。下面看analyzer中的batches是如何定义的。

代码清单 7.17 定义于Analyzer中的规则

```
val batches: Seq[Batch] = Seq(
    Batch("MultiInstanceRelations", Once,
      NewRelationInstances),
    Batch("CaseInsensitiveAttributeReferences", Once,
      (if (caseSensitive) Nil else LowercaseAttributeReferences :: Nil) : _*),
    Batch("Resolution", fixedPoint,
      ResolveReferences ::
      ResolveRelations ::
      NewRelationInstances ::
      ImplicitGenerate ::
      StarExpansion ::
      ResolveFunctions ::
      GlobalAggregates ::
      typeCoercionRules :_*),
    Batch("AnalysisOperators", fixedPoint,
      EliminateAnalysisOperators)
  )
```

Batch中定义了一系列的规则，这里再次出现语法糖问题。::表示cons的意思，即连接生成一个list。

Batch构造函数中需要指定一系列的规则（Rule），这些规则的目的见表 7.3。

表 7.3 定义于Analyzer中各规则的用途

MultiInstanceRelations	如果一个实例在Logical Plan里出现了多次，则会应用NewRelationInstances
LowercaseAttributeReferences	将匹配到的属性统一转换为小写
ResolveReferences	将SQL Parser解析出来的UnresolvedAttribute全部都转为对应的实际的catalyst.expressions.AttributeReference AttributeReferences
ResolveRelations	将Unresolved Relation转换为Resovled Relation
ImplicitGenerate	如果在select中只有一个表达式，而且这个表达式是一个generator，那么适用这条规则
StarExpansion	在Project操作符里，如果是*符号，即select * 语句，可以将所有的references都展开，即将select * 中的*展开成实际的字段
ResolveFunctions	它和ResolveReferences差不多，这里主要是对udf进行 resolve

续表

GlobalAggregate	全局的聚合，如果遇到了Project就返回一个Aggregate
typeCoercionRules	Hive里的兼容SQL语法
EliminateAnalysisOperators	将分析的操作符移除，这里仅支持两种：一种是Subquery；另一种是 LowerCaseSchema。这些节点都会从Logical Plan里移除

有了这么多的 Analyze Rule 之后，来看一看在 SqlParser 结束之后，有一个 Unresolved LogicalPlan 是如何在这一阶段被处理的。

所谓的Unresolved LogicalPlan，用一个直白的语句来表示就是要找到数据源是什么。在SQL Parser阶段，尽管知道People是一个表名，但并不知道这个表中的真正数据存储于何处。而ResolveRelations就是要解决这一问题的。

代码清单 7.18 ResolveRelations

```
object ResolveRelations extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case UnresolvedRelation(databaseName, name, alias) =>
    //去catalog中查找tablename对应到的LogicalPlan
      catalog.lookupRelation(databaseName, name, alias)
  }
}
```

lookupRelation定义于SimpleCatalog类中。

代码清单 7.19 lookupRelation

```
    override def lookupRelation(
      databaseName: Option[String],
      tableName: String,
      alias: Option[String] = None): LogicalPlan = {
    val (dbName, tblName) = processDatabaseAndTableName(databaseName, tableName)
    val table = tables.get(tblName).getOrElse(sys.error(s"Table Not Found:
    $tableName"))
    val tableWithQualifiers = Subquery(tblName, table)

    // If an alias was specified by the lookup, wrap the plan in a subquery so
    // that attributes are properly qualified with this alias.
```

```
    alias.map(a => Subquery(a, tableWithQualifiers)).getOrElse(tableWithQualifiers
    )
  }
```

查找这一端的线索已经明了，现在问题转换成这个Tables中的内容是什么时候被注入的。

回过头再看一下Demo中的源码，再次审视这两行代码。

代码清单 7.20 registerAsTable

```
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(",
    ")).map(p => Person(p(0), p(1).trim.toInt))
people.registerAsTable("people")
```

people.registerAsTable这句代码的执行中隐含了createSchemaRDD的调用，createSchemaRDD中一个重要的内容就是生成相应的LogicalPlan，将数据源注册和LogicalPlan关联起来。

代码清单 7.21 createSchemaRDD

```
implicit def createSchemaRDD[A <: Product: TypeTag](rdd: RDD[A]) =
  new SchemaRDD(this, SparkLogicalPlan(ExistingRdd.fromProductRdd(rdd)))
```

registerAsTable引发的调用路径如下：SchemaRdd.registerAsTable→SqlContext.registerRDDAsTable→Catalog.registerTable。

registerTable将tableName插入到tables中。

代码清单 7.22 Catalog.registerAsTable

```
override def registerTable(
    databaseName: Option[String],
    tableName: String,
    plan: LogicalPlan): Unit = {
  val (dbName, tblName) = processDatabaseAndTableName(databaseName, tableName)
  tables += ((tblName, plan))
}
```

到此可以将Demo中的Select语句理解清楚了。registerAsTable将数据源与 logicalPlan结合，而在ResolveRelations中将SQL语句中没有绑定到特定操作的 people与LogicalPlan关联起来，这样一来打通了数据获取的通路，回答了下面几个问题：

- 数据从哪里来——registerAsTable。

- 经过哪些操作——过滤条件 Filter。
- 抽取哪些有用信息——Project。

7.2.3 Optimizer

Optimizer部分所做的操作是对LogicalPlan进行优化，使用到的规则详见表 7.4，在代码的理解方面与Analyzer并无二致。

表 7.4 Optimizer规则

CombineLimits	如果出现了两个Limit，则将两个Limit合并为一个。要求一个Limit是另一个Limit的grandChild
NullPropagation	将Expression Expressions替换为等价的Literal值的优化，并且能够避免NULL值在SQL语法树的传播
ConstantFolding	常量合并属于Expression优化的一种。对于可以直接计算的常量，不用放到物理执行里去生成对象来计算了，可以直接在计划里就计算出来
BooleanSimplification	这个是对布尔表达式的优化
CombineFilters	合并两个相邻的Filter，它和上述CombineLimits差不多
Filter Pushdown	过滤器下推，原理就是更早地过滤掉不需要的元素来减少开销
ColumnPruning	列裁剪用，常用于聚合操作、Join操作、合并相邻Project 的列

代码清单 7.23 Optimizer

```
object Optimizer extends RuleExecutor[LogicalPlan] {
  val batches =
    Batch("Combine Limits", FixedPoint(100),
      CombineLimits) ::
    Batch("ConstantFolding", FixedPoint(100),
      NullPropagation,
      ConstantFolding,
      LikeSimplification,
      BooleanSimplification,
      SimplifyFilters,
      SimplifyCasts,
      SimplifyCaseConversionExpressions) ::
    Batch("Filter Pushdown", FixedPoint(100),
      CombineFilters,
```

```
      PushPredicateThroughProject,
      PushPredicateThroughJoin,
      ColumnPruning) :: Nil
}
```

7.2.4 SparkPlan

经过SqlParser、Analyzer、Optimizer，现在需要将LogicalPlan转换为RDD才可以在Spark Cluster上真正地进行数据分析。

LogicalPlan到RDD的转换过程中引入了SparkPlan，也就是图 7.8中的Physical Plans的部分。

SparkPlan的主要任务就是生成RDD，在此之前是LogicalPlan到SparkPlan的转换过程。

代码清单 7.24 sparkPlan

```
lazy val sparkPlan = planner(optimizedPlan).next()
```

代码清单 7.24中的代码将LogicalPlan转换为SparkPlan。

在阶段3最主要的代码就两行。

```
  lazy val executePlan: SparkPlan = prepareForExecution(sparkPlan)
  lazy val toRdd: RDD[Row] = executedPlan.execute()
```

SparkPlanner利用SparkStrategies将LogicalPlan转换为SparkPlan。图7.12反映了SparkPlanner的类继承关系。图7.13反映了QueryPlan类体系结构。

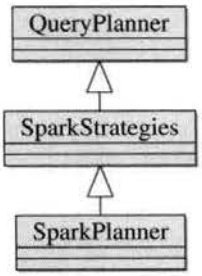


图 7.12 SparkPlanner

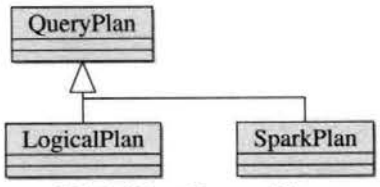


图 7.13 QueryPlan

Planner(optimizedPlan)会调用定义于QueryPlanner中的apply函数，函数实现见代码清单7.25。

代码清单7.25 apply

```
def apply(plan: LogicalPlan): Iterator[PhysicalPlan] = {
  // Obviously a lot to do here still...
  val iter = strategies.view.flatMap(_(plan)).toIterator
  assert(iter.hasNext, s"No plan for $plan")
  iter
}
```

strategies具体的定义见于SparkPlanner。

代码清单7.26 定义于SparkPlanner中的转换策略

```
val strategies: Seq[Strategy] =
  CommandStrategy(self) ::
  TakeOrdered ::
  PartialAggregation ::
  LeftSemiJoin ::
  HashJoin ::
  InMemoryScans ::
  ParquetOperations ::
  BasicOperators ::
  CartesianProduct ::
  BroadcastNestedLoopJoin :: Nil
```

表7.5中有各种类型的Join，现简述一下其区别。

- Join：是最简单的关联操作，两边关联只取交集。
- Outer Join：分为Left Outer Join、Right Outer Join和Full Outer Join。
  - Left Outer Join——是以左表驱动的，右表不存在的key均赋值为null。
  - Right Outer Join——是以右表驱动的，左表不存在的key均赋值为null。
  - Full Outer Join——全表关联，将两表完整地进行笛卡儿积操作，左右表均可赋值为null。
- Semi Join：最主要的使用场景就是解决exist in这样的需求。
- 交叉连接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉连接也称作笛卡儿积。

表 7.5 SparkPlan的转换策略

CommandStrategy	专门针对Command类型的Logical Plan
TakeOrdered	用于Limit操作。如果有Limit和Sort操作，采用该规则
PartialAggregation	部分聚合的策略，即有些聚合操作可以在Local里面完成的，就在Local里完成，而不必去shuffle所有的字段
LeftSemiJoin	最主要的使用场景就是解决exist in
HashJoin	HashJoin在并行和扩展方面优于其他连接，在执行时由构建和裁剪两步组成
InMemoryScans	对InMemoryRelation使用这个Logical Plan规则
ParquetOperations	支持ParquetOperations的读写、插入Table等
BasicOperators	有Project、Filter、Sample、Union、Limit、TakeOrdered、Sort、ExistingRdd
CartesianProduct	笛卡儿积的Join。有待过滤条件的Join，主要是利用RDD的cartesian实现的
BroadcastNestedLoopJoin	用于Left Outer Join、Right Outer Join、Full Outer Join这3种类型的Join

LeftSemiJoin类似于SQL Server中的Exists/In查询子句，在SQL Server中使用 Exist/In关键字来实现，如下所示。

代码清单 7.27 SQL Server中的LeftSemiJoin

```
SELECT a.key, a.value FROM a WHERE a.key IN (SELECT b.key FROM b);
```

使用Hive格式来表达，如下所示。

代码清单 7.28 LeftSemiJoin的Hive表示

```
SELECT a.key, a.val FROM a LEFT SEMI JOIN b ON (a.key = b.key)
```

与LogicalPlan不同，SparkPlan最重要的区别就是有execute函数。

以Sample为例，看出其实现的区别。继承自LogicalPlan的Sample，定义如代码清单 7.29所示。

代码清单 7.29 Sample（一）

```
case class Sample(fraction: Double, withReplacement: Boolean, seed: Long, child:
  LogicalPlan)
  extends UnaryNode {

override def output = child.output
```

```
  override def references = Set.empty
}
```

而继承自SparkPlan的Sample，其定义如下。

代码清单 7.30 Sample（二）

```
  case class Sample(fraction: Double, withReplacement: Boolean, seed: Long, child:
    SparkPlan)
  extends UnaryNode
{
  override def output = child.output

  // TODO: How to pick seed?
  override def execute() = child.execute().sample(withReplacement, fraction, seed)
}
```

针对SparkPlan的具体实现，又要分成UnaryNode、LeafNode和BinaryNode，简要来说即单目运算符操作、叶子节点、双目运算符操作。每个子类的具体实现可以自行参考源码。

execute返回RDD，看一看toRDD的内容是什么。executedPlan其实是Final Plan，由最后一个Plan向前推，直到所有的Plan都转换为RDD，这有点类似于Job提交中的Final Stage。

代码清单 7.31 toRdd

```
lazy val toRdd: RDD[Row] = executedPlan.execute()
```

代码清单 7.32 日志中输出的toRdd依赖关系

```
MapPartitionsRDD[9] at mapPartitions at basicOperators.scala:40
  MapPartitionsRDD[8] at mapPartitions at basicOperators.scala:53
  MapPartitionsRDD[4] at mapPartitions at basicOperators.scala:219
  MappedRDD[3] at map at LocalSqlQuery.scala:23
  MappedRDD[2] at map at LocalSqlQuery.scala:23
  examples/src/main/resources/people.txt MappedRDD[1] at textFile at LocalSqlQuery
    .scala:23
  examples/src/main/resources/people.txt HadoopRDD[0] at textFile at LocalSqlQuery
    .scala:23
```

从toRdd的dependency也可以看出，该Spark Job只会有一个Stage，因为没有牵涉到任何的Shuffle过程。

如何得到上述的打印信息？可以改写一下SchemaRDD的getDependencies，将原有内容改成如下所述。RDD中有一个很有用的函数toDebugString。

代码清单 7.33 SchemaRDD.getDependencies

```
override protected def getDependencies: Seq[Dependency[_]] = {
  println(queryExecution.toRdd.toDebugString)
  List(new OneToOneDependency(queryExecution.toRdd))
}
```

这一点也可以从执行日志上得到进一步的证明。

代码清单 7.34 执行日志

```
INFO scheduler.DAGScheduler: Final stage: Stage 0(collect at LocalSqlQuery.scala
    :31)
INFO scheduler.DAGScheduler: Parents of final stage: List()
INFO scheduler.DAGScheduler: Missing parents: List()
```

7.3 Parquet 文件和JSON数据集

Spark SQL目前支持将Parquet文件和JSON文件作为数据源，同时支持将SchemaRDD中的内容保存为Parquet文件。

代码清单 7.35显示了从Parquet中读取数据并存储的过程。

代码清单 7.35 将SchemaRDD保存为Parquet文件

```
import sqlContext.createSchemaRDD

case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/people.txt").map(_.split(","
    )).map(p => Person(p(0), p(1).trim.toInt))
people.registerAsTable("people")

//saveAsParquetFile会在当前目录创建新的目录people.parquet
//people.parquet目录列表
// _metadata
```

```
// part-r-1.parquet
// part-r-2.parquet
// _SUCCESS
people.saveAsParquetFile("people.parquet")

val parquetFile = sqlContext.parquetFile("people.parquet")

//Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerAsTable("parquetFile")
val teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND
    age <= 19")
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

7.4 Hive简介

Spark 1.0中添加的SQL模块，对Hive中的HiveQL也提供了良好的支持。本节将详细分析Spark是如何完成对HQL的支持的。

以下部分摘自*Hadoop Definite Guide*中的“Hive”一章：

*Hive*由*Facebook*出品，其设计之初目的是让精通*SQL*技能的分析师能够对*Facebook*存放在*HDFS*上的大规模数据集进行分析和查询。

*Hive*大大简化了对大规模数据集的分析门槛（不再要求分析人员具有很强的编程能力），因而迅速流行起来，成为*Hadoop*生成圈上的杀手级应用（*Killer Application*）。目前已经有很多组织把*Hive*作为一个通用的、可伸缩数据处理平台。

7.4.1 Hive 架构

Hive所有的数据都存在HDFS中，在Hive中有以下几种数据模型。

- Table（表）：Table和关系型数据库中的表是相对应的，每个表都有一个对应的HDFS目录。表中的数据经序列化后存储在该目录，Hive同时支持表中的数据存储在其他类型的文件系统中，如NFS或本地文件系统。
- 分区（Partition）：Hive中的分区起到的作用有点类似于RDBMS中的索引功能，每个Partition都有一个对应的目录，这样在查询的时候，可以减少数据规模。
- 桶（Bucket）：即使将数据分区之后，每个分区的规模也有可能仍很大。这时，按照关键字的Hash结果将数据分成多个Bucket，每个Bucket对应于一个文件。

HiveQL是Hive支持的类似于SQL的查询语言。HiveQL大体可以分成下面几种类型。

- DDL（data definition language）：比如创建数据库（create database），创建表（create table），数据库和表的删除。
- DML（data manipulation language）：数据的添加、查询。
- UDF（user defined function）：Hive还支持用户自定义查询函数。

由图 7.14可以看出，Hive的整体架构可以分成以下几大部分：

- 用户接口支持CLI、JDBC和WebUI。
- Driver Driver负责将用户指令翻译转换为相应的MapReduce Job。
- MetaStore 元数据存储仓库。像数据库和表的定义这些内容就属于元数据这个范畴，默认使用的是Derby存储引擎。

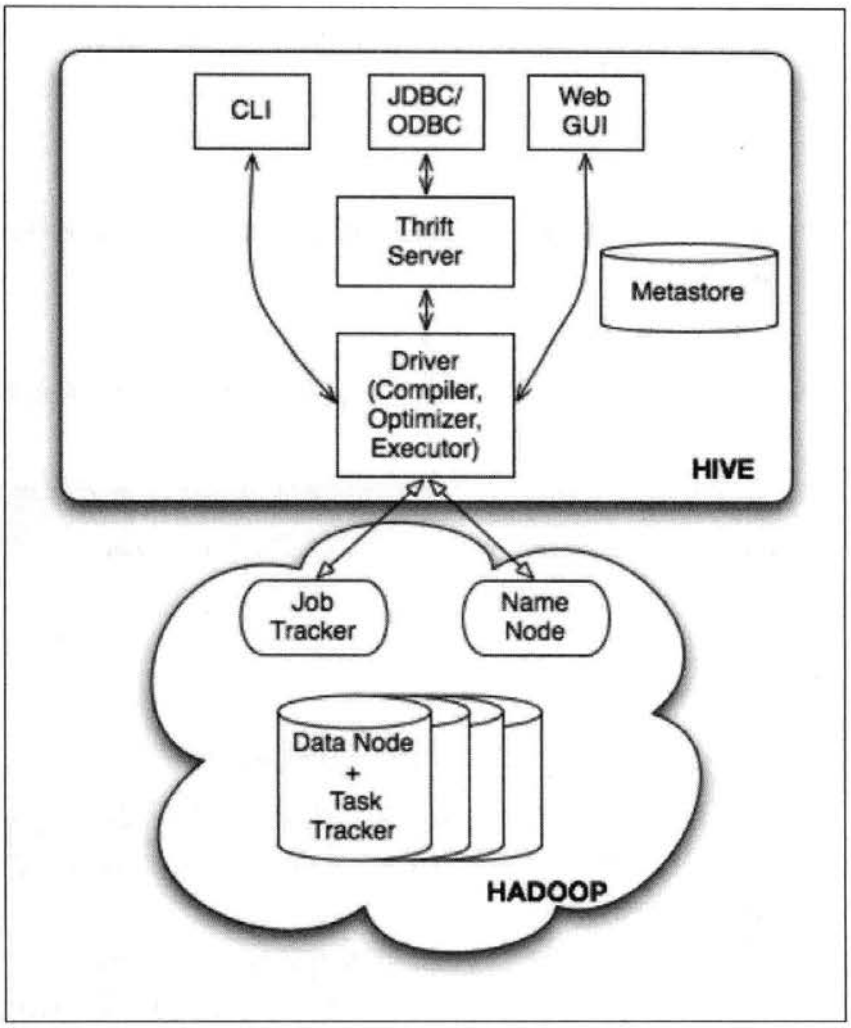


图 7.14 Hive架构

7.4.2 HiveQL On MapReduce执行过程分析

HiveQL的执行过程如下所述。

（1） Parser：将HiveQL解析为相应的语法树。
（2） Semantic Analyser：语义分析。
（3） LogicalPlan Generating：生成相应的LogicalPlan。
（4） QueryPlan Generating：生成QueryPlan。
（5） Optimizer：查询优化器。

最终生成MapReduce的Job，交付给Hadoop的MapReduce计算框架具体运行。

最好的学习就是实战，本节还是以一个具体的例子来结束吧。

前提条件是已经安装好Hadoop，具体安装可以参考源码走读之10：http://www.cnblogs.com/hseagle/category/569175.html。

warehouse目录用来存储Raw Data。

代码清单 7.36 创建warehouse

```
$HADOOP_HOME/bin/hadoop fs -mkdir       /tmp
$HADOOP_HOME/bin/hadoop fs -mkdir       /user/hive/warehouse
$HADOOP_HOME/bin/hadoop fs -chmod g+w   /tmp
$HADOOP_HOME/bin/hadoop fs -chmod g+w   /user/hive/warehouse
```

结合源码，我们再对一个简单的例子做如下说明。

可能你会想，既然Spark也支持HQL，那么我原先用Hive CLI创建的数据库和表用Spark能不能访问到呢？答案或许会让你很纳闷："在默认的配置下是不行的。"为什么？

Hive中的Meta Data采用的存储引擎是Derby，该存储引擎只能有一个访问用户。同一时刻只能有一个人访问，即便以同一用户登录访问也不行。针对这个局限，解决方法就是将MetaStore存储在MySQL或者其他可以多用户访问的数据库中。

具体实例：

- 创建表。
- 导入数据。
- 查询。
- 删除表。

在启动spark-shell之前，需要先设置环境变量HIVE_HOME和HADOOP_HOME。

启动spark-shell之后，执行如下代码。

代码清单 7.37 HiveQL On Spark

```
  val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

// Importing the SQL context gives access to all the public SQL functions
// and implicit conversions.
import hiveContext._

hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
hql("FROM src SELECT key, value").collect().foreach(println)
hql("drop table src")
```

create操作会在/user/hive/warehouse/目录下创建src目录，可以用以下指令来验证：

```
$HADOOP_HOME/bin/hdfs dfs -ls /user/hive/warehouse/
```

drop表时，不仅MetaStore中相应的记录被删除，而且原始数据Raw File本身也会被删除，即在warehouse目录下对应某个表的目录会被整体删除掉。

上述create、load及query操作对MetaStore和Raw Data的影响可以用图 7.15表示。

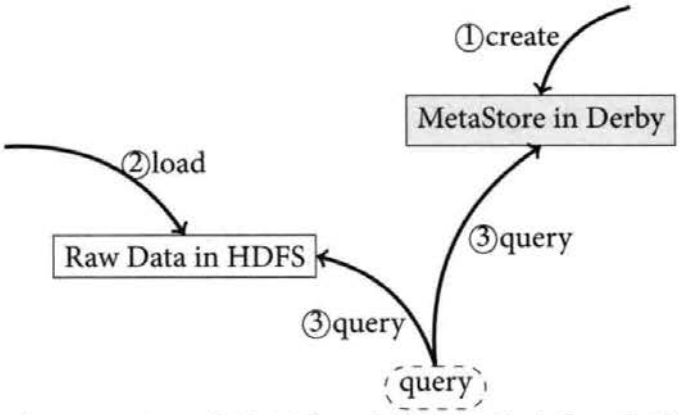


图 7.15 Hive中表创建及查询之间的关系示意图

7.5 HiveQL On Spark详解

前面花了大量篇幅介绍了Hive由来、框架及HiveQL执行过程。那么这些东西跟我们标题中所称的Hive On Spark有什么关系呢？

答案是这样的。Hive的整体解决方案很不错，但有一些地方还值得改进，其中之一就是："从查询提交到结果返回需要相当长的时间，查询耗时太长。"之所以查询时间很长，一个主要的原因就是由于Hive原生是基于MapReduce的。有没有办法改善呢？你一定想到了："不是生成MapReduce Job，而是生成Spark Job"，充分利用Spark的快速执行能力来缩短HiveQL的响应时间。

表 7.6是Spark 1.0中所支持的lib库。sql是其唯一新添加的lib库，可见sql在Spark 1.0中的地位之重要。

表 7.6 Spark支持的lib库

streaming	graphx	sql	mllib
Spark			

HiveContext是Spark提供的用户接口，HiveContext继承自SQLContext。

让我们回顾一下，SqlContext中牵涉到的类及其间关系如图 7.7所示。

既然是继承自SQLContext，那么我们将普通SQL与HiveQL分析执行步骤做一个对比，可以得到图 7.16。

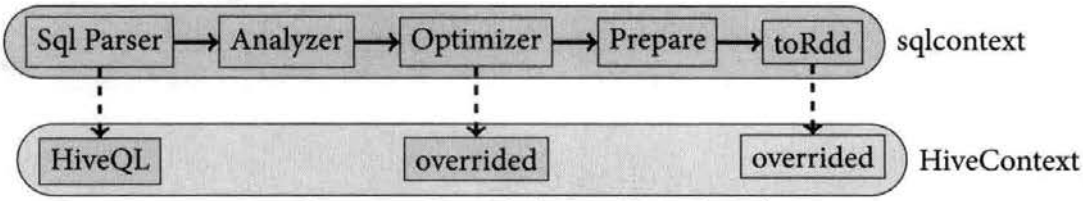


图 7.16 HiveQL与SQL执行过程对比

有了上述比较，就能抓住源码分析时需要把握的几个关键点：

- Entrypoint HiveContext.scala
- QueryExecution HiveContext.scala
  - parser HiveQL.scala
  - optimizer

使用到的数据有两种：

- Schema Data 像数据库的定义和表的结构，这些都存储在MetaStore中。
- Raw Data 即要分析的文件本身。

HiveQL是整个程序分析的入口点，而HQL是HiveQL的缩写形式。

代码清单 7.38 hiveql

```
def hiveql(hqlQuery: String): SchemaRDD = {
```

```
    val result = new SchemaRDD(this, HiveQl.parseSql(hqlQuery))
    // We force query optimization to happen right away instead of letting
    // it happen lazily like when using the query DSL. This is so
    // DDL commands behave as expected.  This is only generates the RDD
    // lineage for DML queries, but does not perform any execution.
    result.queryExecution.toRdd
    result
  }
```

上述HiveQL的定义与SQL的定义几乎一模一样，唯一的不同是SQL中使用parseSql的结果作为SchemaRDD的入参，而HiveQL中使用HiveQL.parseSql作为SchemaRDD的入参。

parseSql的函数定义如代码所示，解析过程中将指令分成两大类：

- nativecommand 非select语句，这类语句的特点是执行时间不会因为条件的不同而有很大的差异，基本上都能在较短的时间内完成。
- 非nativecommand 主要是select语句。

代码清单 7.39 parseSql

```
def parseSql(sql: String): LogicalPlan = {
    try {
      if (sql.toLowerCase.startsWith("set")) {
        NativeCommand(sql)
      } else if (sql.toLowerCase.startsWith("add jar")) {
        AddJar(sql.drop(8))
      } else if (sql.toLowerCase.startsWith("add file")) {
        AddFile(sql.drop(9))
      } else if (sql.startsWith("dfs")) {
        DfsCommand(sql)
      } else if (sql.startsWith("source")) {
        SourceCommand(sql.split(" ").toSeq match { case Seq("source", filePath) =>
     filePath })
      } else if (sql.startsWith("!")) {
        ShellCommand(sql.drop(1))
      } else {
        val tree = getAst(sql)
```

```
      if (nativeCommands contains tree.getText) {
        NativeCommand(sql)
      } else {
        nodeToPlan(tree) match {
          case NativePlaceholder => NativeCommand(sql)
          case other => other
        }
      }
    }
  } catch {
    case e: Exception => throw new ParseException(sql, e)
    case e: NotImplementedError => sys.error(
      s"""
        |Unsupported language features in query: $sql
        |${dumpTree(getAst(sql))}
      """.stripMargin)
  }
}
```

哪些指令是nativeCommand呢？答案在HiveQL.scala中的nativeCommands变量，列表很长，代码就不一一列出了。

对于非nativeCommand，最重要的解析函数就是nodeToPlan。

Spark对HiveQL所做的优化主要体现在Query相关的操作，其他的依然使用Hive的原生执行引擎。

在logicalPlan到physicalPlan的转换过程中，toRdd最关键的元素。

代码清单 7.40 toRdd

```
override lazy val toRdd: RDD[Row] =
analyzed match {
  case NativeCommand(cmd) =>
  val output = runSqlHive(cmd)

  if (output.size == 0) {
    emptyResult
  } else {
```

```
    val asRows = output.map(r => new GenericRow(r.split("\t").asInstanceOf[Array
  [Any]]))
    sparkContext.parallelize(asRows, 1)
  }
  case _ =>
  executedPlan.execute().map(_.copy())
}
```

由于nativeCommand是一些不怎么耗时的操作，因此直接使用Hive中原有的exeucte engine来执行即可。这些Command的执行示意如图7.17所示。

图 7.17 native command执行过程

在Hive解析过程中添加了两个规则（Rule），分别是HiveTypeCoercion和PreInsertionCasts。

代码清单 7.41 typeCorecionRules

```
val typeCoercionRules =
List(PropagateTypes, ConvertNaNs, WidenTypes, PromoteStrings,
BooleanComparisons, BooleanCasts, StringToIntegralCasts,
FunctionArgumentConversion)
```

PreInsertionCasts存在的目的就是确保在数据插入执行之前，相应的表已经存在。

代码清单 7.42 optimizedPlan

```
override lazy val optimizedPlan =
optimizer(catalog.PreInsertionCasts(catalog.CreateTables(analyzed)))
```

此处要注意的是Catalog的用途，Catalog是HiveMetastoreCatalog的实例。

HiveMetastoreCatalog是Spark中对Hive Metastore访问的wrapper。HiveMetastoreCatalog通过调用相应的Hive API可以获得数据库中的表及表的分区，也可以创建新的表和分区。

HiveMetastoreCatalog中会通过Hive Client来访问MetaStore中的元数据，使用了大量的Hive API，如图7.18所示。其中包括了广为人知的deSer library。

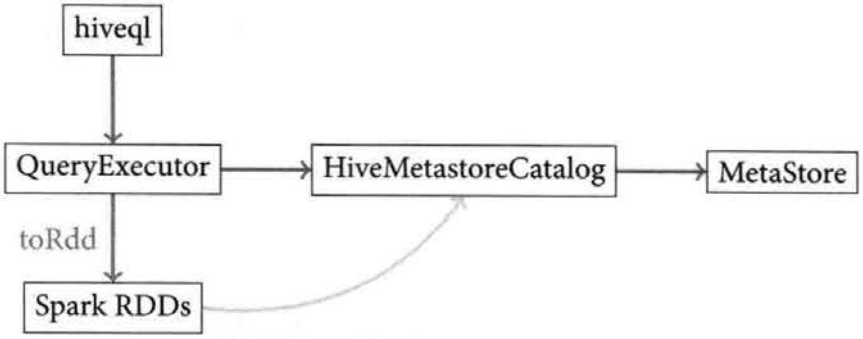


图 7.18 HiveMetastoreCatalog

以createTable函数为例说明对Hive Library的依赖。

代码清单 7.43 createTable

```
def createTable(
      databaseName: String,
      tableName: String,
      schema: Seq[Attribute],
      allowExisting: Boolean = false): Unit = {
    val table = new Table(databaseName, tableName)
    val hiveSchema =
      schema.map(attr => new FieldSchema(attr.name,
       toMetastoreType(attr.dataType), ""))
    table.setFields(hiveSchema)

    val sd = new StorageDescriptor()
    table.getTTable.setSd(sd)
    sd.setCols(hiveSchema)

    // TODO: THESE ARE ALL DEFAULTS, WE NEED TO PARSE / UNDERSTAND
    // the output specs.
    sd.setCompressed(false)
    sd.setParameters(Map[String, String]())
    sd.setInputFormat("org.apache.hadoop.mapred.TextInputFormat")
    sd.setOutputFormat("org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
    ")
    val serDeInfo = new SerDeInfo()
    serDeInfo.setName(tableName)
```

```
    serDeInfo.setSerializationLib("org.apache.hadoop.hive.serde2.lazy.
    LazySimpleSerDe")
    serDeInfo.setParameters(Map[String, String]())
    sd.setSerdeInfo(serDeInfo)

    try client.createTable(table) catch {
      case e: org.apache.hadoop.hive.ql.metadata.HiveException
        if e.getCause.isInstanceOf[org.apache.hadoop.hive.metastore.api.
    AlreadyExistsException] &&
          allowExisting => // Do nothing.
    }
  }
```

7.5.1 Hive On Spark环境搭建

安装概述

整体的安装过程分为以下几步：

- 搭建Hadoop集群（整个Cluster由3台机器组成，一台作为Master，另两台作为Slave）。
- 编译Spark 1.0，使其支持Hadoop 2.4.0和Hive。
- 运行Hive On Spark的测试用例（Spark和Hadoop Namenode运行在同一台机器）。

Hadoop集群搭建

步骤1：IP规划。

创建基于KVM的虚拟机，利用libvirt提供的图形管理界面，创建3台虚拟机，非常方便。内存和IP地址分配如下：

- master 2G 192.168.122.102
- slave1 4G 192.168.122.103
- slave2 4G 192.168.122.104

在虚拟机上安装OS的过程就略过了，OS安装完成之后，确保以下软件也已经安装：

- JDK
- OpenSSH

步骤2：创建用户组和用户。

在每台机器上创建名为hadoop的用户组，添加名为hduser的用户，具体bash命令如下所示。

```
groupadd hadoop
useradd -b /home -m -g hadoop hduser
passwd hduser
```

步骤3：无密码登录。

在启动Slave机器上的datanode或nodemanager的时候需要输入用户名和密码。为了避免每次都要输入密码，可以利用如下指令创建无密码登录。注意是从Master到Slave机器的单向无密码。

代码清单 7.44 生成ssh key

```
cd $HOME/.ssh
ssh-keygen -t dsa
```

将id_dsa.pub复制为authorized_keys，然后上传到slave1和slave2中的$HOME/.ssh目录。

代码清单 7.45 上传ssh key

```
cp id_dsa.pub authorized_keys
#确保在slave1和slave2机器中，hduser的$HOME目录下已经创建好了.ssh目录
scp authorized_keys slave1:$HOME/.ssh
scp authorized_keys slave2:$HOME/.ssh
```

步骤4：更改每台机器上的/etc/hosts。

在组成集群的master、slave1和slave2中，向/etc/hosts文件添加如下内容。

代码清单 7.46 更改hosts文件

```
192.168.122.102 master
192.168.122.103 slave1
192.168.122.104 slave2
```

更改完成之后，可以在master上执行ssh slave1来进行测试；如果没有输入密码的过程就直接登录入slave1，则说明上述的配置成功。

步骤5：下载Hadoop 2.4.0。

以hduser身份登录master，执行如下指令。

代码清单 7.47 下载hadoop2.4.0

```
cd /home/hduser
wget http://mirror.esocc.com/apache/hadoop/common/hadoop-2.4.0/hadoop-2.4.0.tar.gz
```

```
mkdir yarn
tar zvxf hadoop-2.4.0.tar.gz -C yarn
```

步骤6：修改Hadoop配置文件，添加如下内容到.bashrc。

代码清单 7.48 修改.bashrc

```
export HADOOP_HOME=/home/hduser/yarn/hadoop-2.4.0
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

步骤7：修改 $HADOOP_HOME/libexec/hadoop-config.sh。

在hadoop-config.sh文件开头处添加如下内容。

代码清单 7.49 设置JAVA_HOME

```
export JAVA_HOME=/opt/java
```

步骤8：修改$HADOOP_CONF_DIR/yarn-env.sh。

在yarn-env.sh开头添加如下内容。

代码清单 7.50 yarn-env.sh

```
export JAVA_HOME=/opt/java
export HADOOP_HOME=/home/hduser/yarn/hadoop-2.4.0
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

步骤9：xml配置文件修改。

文件1：$HADOOP_CONF_DIR/core-site.xml

代码清单 7.51 core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://master:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/hduser/yarn/hadoop-2.4.0/tmp</value>
  </property>
</configuration>
```

文件2：$HADOOP_CONF_DIR/hdfs-site.xml

代码清单 7.52 hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
 <configuration>
   <property>
     <name>dfs.replication</name>
     <value>2</value>
   </property>
   <property>
     <name>dfs.permissions</name>
     <value>false</value>
   </property>
 </configuration>
```

文件3：$HADOOP_CONF_DIR/mapred-site.xml

代码清单 7.53 mapred-site.xml

```
<?xml version="1.0"?>
<configuration>
 <property>
```

```
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

文件4：$HADOOP_CONF_DIR/yarn-site.xml

代码清单 7.54 yarn-site.xml

```
<?xml version="1.0"?>
 <configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>master:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>master:8030</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>master:8040</value>
  </property>
 </configuration>
```

文件5：$HADOOP_CONF_DIR/slaves

在文件中添加如下内容。

代码清单 7.55 slaves

```
slave1
```

```
slave2
```

步骤10：创建tmp目录。

在$HADOOP_HOME下创建tmp目录。

代码清单 7.56 创建tmp目录

```
mkdir $HADOOP_HOME/tmp
```

步骤11：复制YARN目录到slave1和slave2。

刚才所做的配置文件更改发生在master机器上，将整个更改过的内容全部复制到slave1和slave2。

代码清单 7.57 上传YARN的配置文件到目标机

```
for target in slave1 slave2
do
    scp -r yarn $target:~/
    scp $HOME/.bashrc $target:~/
done
```

步骤12：格式化namenode。

在master机器上对namenode进行格式化。

代码清单 7.58 格式化namenode

```
bin/hadoop namenode -format
```

步骤13：启动cluster集群。

代码清单 7.59 启动cluster

```
sbin/hadoop-daemon.sh start namenode
sbin/hadoop-daemons.sh start datanode
sbin/yarn-daemon.sh start resourcemanager
sbin/yarn-daemons.sh start nodemanager
sbin/mr-jobhistory-daemon.sh start historyserver
```

注意：daemon.sh表示只在本机运行，daemons.sh表示在所有的cluster节点上运行。

7.5.2 编译支持Hadoop 2.x的Spark

（接7.5.1节步骤）步骤14：编译Spark 1.0。

Spark的编译还是很简单的，失败的原因大部分可以归结于所依赖的Jar包无法正常下载。

为了让Spark 1.0支持Hadoop 2.4.0和Hive，请使用如下指令编译。

代码清单 7.60 编译支持Hive的Spark

```
SPARK_HADOOP_VERSION=2.4.0 SPARK_YARN=true  SPARK_HIVE=true sbt/sbt assembly
```

如果一切顺利，则将会在assembly目录下生成spark-assembly-1.0.0-SNAPSHOT-hadoop2.4.0.jar。

步骤15：创建运行包。

编译之后整个$SPARK_HOME目录下所有的文件体积还是很大的，大概有两个多GB。有哪些是运行的时候真正需要的呢？下面将会列出这些目录和文件。

代码清单 7.61 Spark运行所需的目录列表

```
$SPARK_HOME/bin
$SPARK_HOME/sbin
$SPARK_HOME/lib_managed
$SPARK_HOME/conf
$SPARK_HOME/assembly/target/scala-2.10
```

将上述目录的内容复制到/tmp/spark-dist，然后创建压缩包。

代码清单 7.62 生成Spark运行包

```
mkdir /tmp/spark-dist
for i in $SPARK_HOME/{bin,sbin,lib_managed,conf,assembly/target/scala-2.10}
do
  cp -r $i /tmp/spark-dist
done
cd /tmp/
tar czvf spark-1.0-dist.tar.gz spark-dist
```

步骤16：上传运行包到master机器。

将生成的运行包上传到master（192.168.122.102）。

代码清单 7.63 上传运行包到目标机

```
scp spark-1.0-dist.tar.gz hduser@192.168.122.102:~/
```

7.5.3 运行Hive On Spark测试用例

经过上述重重折磨，终于到了最为紧张的时刻了。

以hduser身份登录master机，解压spark-1.0-dist.tar.gz。

代码清单 7.64 解压Spark

```
#after login into the master as hduser
tar zxvf spark-1.0-dist.tar.gz
cd spark-dist
```

更改conf/spark-env.sh。

代码清单 7.65 更改spark-env.sh

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

运行最简单的example。

用bin/spark-shell指令启动shell之后，运行如下Scala代码。

代码清单 7.66 运行Hive On Spark

```
val sc: SparkContext // An existing SparkContext.
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

// Importing the SQL context gives access to all the public SQL functions
// and implicit conversions.
import hiveContext._

hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
hql("FROM src SELECT key, value").collect().foreach(println)
```

第8章

GraphX

“在天成象，在地成形，变化现矣。”

《易经》

图（Graph）是对网络的结构抽象，通常表示为$G = (V, E)$。它是由顶点集合V和边的集合E构成的一种数据结构，用来描述网络内各个实体的特征和关联关系。它用顶点来记录实体的特征属性，用边来表征实体之间各类复杂的关联关系。关于图的最早研究可以追溯至“哥尼斯堡七桥问题”，欧拉通过利用图论和拓扑知识，圆满地解决和回答了哥尼斯堡居民提出的问题，并由此而开辟了数学中重要的一个分支——图论。图因为具有直观、清晰、表达能力强等特点，被广泛应用于社交网络、生物数据分析、推荐系统、复杂对象识别和软件代码剽窃检测等领域。在我们身边，就存在着各种各样的网络结构的图数据，如Web Link、社交网络、交通网络、通信网络、神经网络等。

8.1 GraphX简介

GraphX是Spark推出的专门处理图数据（如Web Graphs、Social Networks）的计算平台，支持各种常见的图算法，包括PageRank、Triangle Counting等。GraphX 基于Spark核心模块 Core 实

现，其内含各类常见的图操作，并提供Pregel API，一些现成的Case样例也简化了一些图计算处理任务的设计。

8.1.1 主要特点

主要特点如下。

（1） 基于Spark核心模块实现。

GraphX中图的基本要素（包括EdgeRDD、VertexRDD）都是基于RDD扩展实现的，它们都具有RDD的一般特征，包括可划分、可计算、持久存储等。同时还包括了图的一些特殊操作，包括边变换、边过滤、内连接等。

（2） 统一的数据模型抽象和高效的数据操作。

GraphX是基于属性图模型的高层抽象，并提供了相应的数据操作接口，便于开发者对图进行统一高效的变换操作。在Graph类中，定义包含了对属性图的一些基本操作，包括属性变换、结构操作、连接操作等抽象的接口函数定义，并由GraphImpl显式实现。这为一般用户读取数据、构建图模型以及操作图数据提供了高度抽象、一致和便捷的开发接口，能够满足常见的图数据处理任务。

（3） 支持Pregel编程接口。

无论是Bagel还是GraphX，Spark都是基于Pregel这个基本的图计算处理模型实现的，因此Spark原生态地支持类Pregel接口及操作。在Pregel中，图处理任务都是由一系列的迭代步骤，也就是超步（SuperStep）构成的。在每一个超步中，用户都可以通过定义顶点的操作接口函数，并通过消息更新与vertex关联的顶点状态，作为下一个迭代的输入；如此反复，直到迭代结束结果输出。在Spark中，这其实是对RDD进行操作和计算。

（4） 具有良好的可扩展性。

GraphX采用了良好的接口设计，设计与实现相分离，函数实现简洁清晰，非常便于用户对所需功能进行扩展，如在VertexRDD和EdgeRDD的参数中，加入StorageLevel就可控制存储资源的占用，也可通过对Graph 引入新操作来丰富Graph的属性图操作接口等。

8.1.2 版本演化

Spark最早处理图数据的模块名为Bagel，Bagel是Google Pregel图处理框架的Spark轻量级实现，在最新的GraphX 1.0.2版本中，其核心代码也只有几百行。

Bagel最早出现在Spark的0.5版本中，是一个基本的Pregel 原型实现。与Pregel 类似，在Bagel中，所有图处理任务都是由一系列的超步构成的。在每一个SuperStep 中，用户可以通过Bagel 的接口指定顶点更新和处理的函数，以及与其他顶点之间的消息通信。

在0.7.3版本中，官方文档就提供了关于Bagel 的两个样例实现，Pagerank和Shortest Path，用户可以通过run脚本命令运行上述程序。

而当Spark更新至0.8时，鉴于业界对分布式图计算的需求比较突出，特别是GraphLab、Powergraph 等新兴图计算平台的出现，开始促使Spark考虑开创GraphX 分支，并将其作为Spark独立的图计算处理平台。在Spark 0.9.0中，GraphX的Alpha版本出现并融入Master 主分支，Spark将其定位为新的面向图数据和并行图计算的Spark API 接口，并通过一系列丰富的属性图操作接口，扩展和改善了Bagel的处理性能和应用范围。

在Spark 1.0.0 版本中，Spark对GrapX 做了性能优化和改善。

最新发布的Spark 1.0.2，GraphX包含了一个由0.9.1 演进而来的用户定义接口，特别地，EdgeRDD 也可以通过添加邻接点属性而构建边点三元组Triplet对象。而在最新的Master主分支，EdgeRDD 和VertexRDD 正准备考虑加入StorageLevel参数，从而能够灵活控制图结构数据占用的存储资源。

8.1.3 应用场景

Spark的应用场景与图数据挖掘算法及应用是紧密相关的，目前常用的应用场景如下。

（1） 节点影响力计算。

PageRank最早用于网页排名搜索。实际上，这与图结构数据的节点影响力计算机理和实现是一致的。我们可以利用社交网络中的信息，如提及关系、转发关系等，进行PageRank计算，从而可以对具体的用户或信息进行影响力排名评估。

（2） 图数据搜索。

图搜索是一种“大数据”时代适应社会计算的搜索方式。所有的社会活动构成了社会网络，本质上这是图的一种表现形式，所以图搜索自然而然地就成了工业界和学术界的共同关注点。比如知识图谱（knowledge graph）被引入Google搜索引擎来提高搜索结果的质量。此外，Facebook于2013年初推出了Graph Search，它的功能是让用户能搜索到社交链接上的信息，例如“我朋友都喜欢哪些纽约的餐馆”、“我去年和某某人的合照”、“我朋友去过的国家公园”等。这标志着图搜索在社会计算“大数据”时代将很有可能成为一种统一的面向社会网络的搜索模式。

（3） 社交圈子识别。

社交圈子识别是图数据挖掘中的一类典型应用，其根本思想就是对图中的节点根据不同的兴趣标签或影响力进行分组，从而使得关联紧密的节点形成“圈子”，从而体现网络中节点的社团结构和属性。例如，微博上对具有Spark研究兴趣的用户或者具有相同消费习惯的用户进行分组，以发现社团或圈子中有影响力的人或信息。

（4） 标签传播。

近些年，随着知识图谱相关研究的兴起，用户兴趣图谱开始凸显其重要的数据价值。

我们可以通过构建用户兴趣图谱为用户推荐感兴趣的信息：包括新闻、图片、群组、广告等。“标签传播”就是基于图数据结构实现的一种数据挖掘算法，它能够通过构建的社交网络，利用标签传播算法发现网络中密集连接的子图。这其中就蕴含了许多与图数据处理和图计算相关的内容，包括图的构建、标签信息传播、算法的迭代执行等，这些都可用图数据处理的相关知识加以解决，并应用到个性化推荐系统中。

8.2 分布式图计算处理技术介绍

下面分别从图数据模型、图数据分割、图数据存储几个方面对常见的图数据分布与并行处理技术做一个简单的介绍。

8.2.1 属性图

当前的大规模图数据处理，采用的数据模型有多种，按照图中节点的复杂程度可以分为简单节点图模型和复杂节点图模型；按照一条边可以连接的顶点数据，可以分为简单图模型和超图模型。不论是简单图模型、超图模型、简单节点模型还是复杂节点模型，其顶点和边都可以带有属性，GraphX的数据处理模型就是属性图Property Graph。

属性图（Property Graph）是由带有属性信息的节点和边构成的图，这些属性主要用来描述节点和边的特征。一个属性图，主要由顶点和边构成，它具体包括：

- 顶点与顶点的属性集合。
- 边与边的属性集合。

以社交网络平台微博为例，如图8.1所示，图中有3个顶点$Vertex = \{1,3,4\}$和3条有向边$Edge = \{follow, creat, retweet\}$，顶点1和2 分别表示用户Peter和Tom，顶点3 表示发布的某条微博。顶点可以具有自己的属性信息，如姓名和年龄；或者是微博信息，如微博的具体内容和发布时间。顶点与顶点之间的关系由有向边表达，包括follow粉丝关系、create创建关系及retweet转发关系。边的属性可以是Label，也可以是具体的数值。

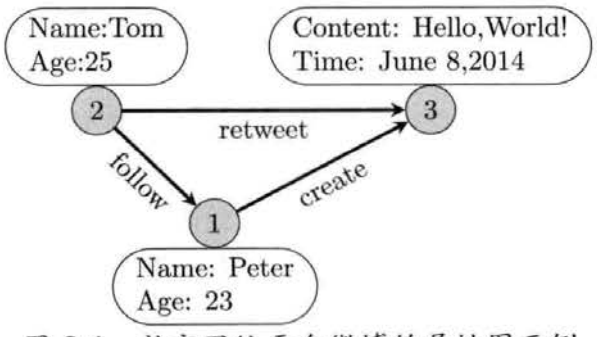


图 8.1 社交网络平台微博的属性图示例

8.2.2 图数据的存储与分割

图具有多种类型的存储结构，常见的包括邻接矩阵（Adjacency Matrix）、邻接表（Adjacency List）和十字链表（Orthogonal List）。在Spark中，采用的是基于RDD的数据存储和设计，其数据存储特性遵循RDD数据的一些基本特性，包括持久化、可分割性、依赖性、容错性等。

在分布式环境下处理大规模图数据，必须进行有效的图分割。由于图数据本身固有的连通性和图计算表现出来的强耦合性等特点，因此为了实现高效的并行处理，尽可能地降低分布式处理的各子图之间的耦合度是非常重要的。有效的图分割就是实现解耦的重要手段。将一个大图分割为若干子图，有两个主要原则：

（1） 提高子图内部的连通性，降低子图之间的连通性，这尤其适合分布式的并行处理机制。

（2） 考虑子图规模的均衡性，尽量保证各子图的数据规模均衡，不要出现较大的偏斜，从数据规模方面防止各并行任务的执行时间相差过大，降低任务同步控制的影响。

当前，最常用的两种图分割策略主要是边分割和点分割，并在GraphLab和 PowerGraph中得以运用和实践。以图8.2（a）为例，边分割就是保持顶点在各个节点均匀分布，采用了顶点复制策略保证跨节点的边数目最小，这里3个节点分别存储了2、1、1个顶点，而每个节点也都通过顶点复制策略存储了相应顶点的邻接点和边的信息。可以看出，边分割增加了邻接顶点和边的存储开销，而任何顶点和边的更新都要进行数据同步和通信，这也增加了相应的网络开销。

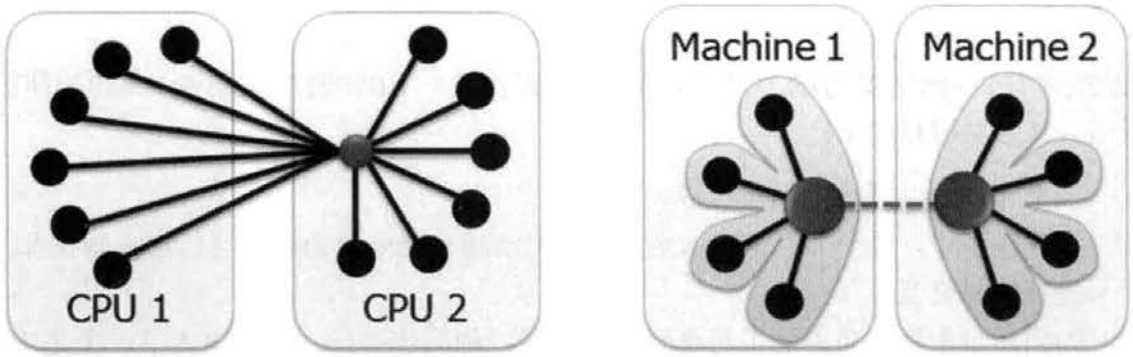
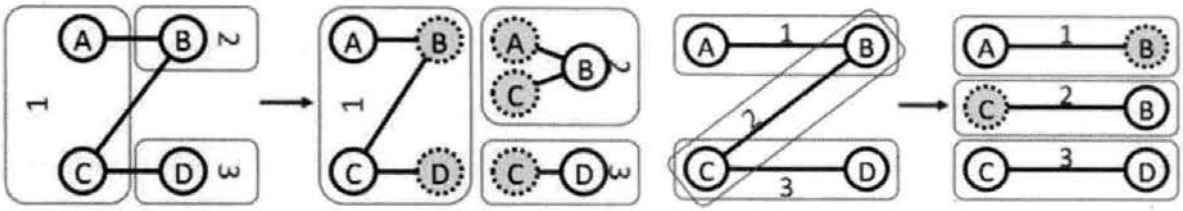


图 8.2 PowerGraph中的边分割与点分割

与之相反，点分割就是保持各个节点边的均匀分布，而采用顶点复制策略维持各个边的邻接顶点信息。以图8.3（b）为例，由{a,b,c,d}构成的图有3条边，点分割就是要将3条边均匀分布在3个计算节点上，而点与点之间的邻接信息是通过顶点复制记录的，这样做能够保证每个边只存储一次。这种策略在解决超级节点时能够起到较好的作用，它能够减少跨节点之间的数据通信，这也是PowerGraph比GraphLab在解决超级节点问题时具有较好性能的原因之一。在分割

超级节点时，相较于边分割，PowerGraph 提倡点分割，优先考虑超级节点邻接边分布的均匀性，从而有效地减少节点之间的通信，提高算法的均衡性和效率。



（a）边分割　　（b）点分割

图 8.3　边分割与点分割

8.3 Pregel计算模型

Pregel是由Google在2010年推出的专门处理图数据的分布式处理平台，相关成果发表在SIGMOD2010。Pregel是第一个用来处理图数据的分布式处理平台，具有里程碑的意义。Pregel基于消息通信的节点式编程实现，它采用随机分割策略对图数据进行分割，节点之间通过消息通信完成操作，其数据同步机制采用的是Bulk Synchoronous Parallel Model（BSP 同步栅栏模型）。

8.3.1 BSP

在继续讲述Pregel更多的细节之前，先来简要了解一下BSP的处理思想。BSP处理模型如图 8.4所示。其含义解释如下：

（1）一个作业由一系列的超步组成。

（2）在各超步中，每个节点执行本地计算，在该超步结束前通过消息传递机制在各节点间进行消息交互。

（3）如果超步中每个节点的消息交互尚未完成，则同步等待；否则进入下一个超步。

8.3.2 像顶点一样思考

利用BSP计算模型，Pregel实现了对图的并行处理框架。Pregel这个图处理框架的鲜明特征就是以“顶点”为中心进行操作上的抽象。Vertex作为一等公民在Pregel中享有至高的地位。如想很好地掌握Pregel这样一个处理框架，就必须要学会像图中的顶点一样思考。

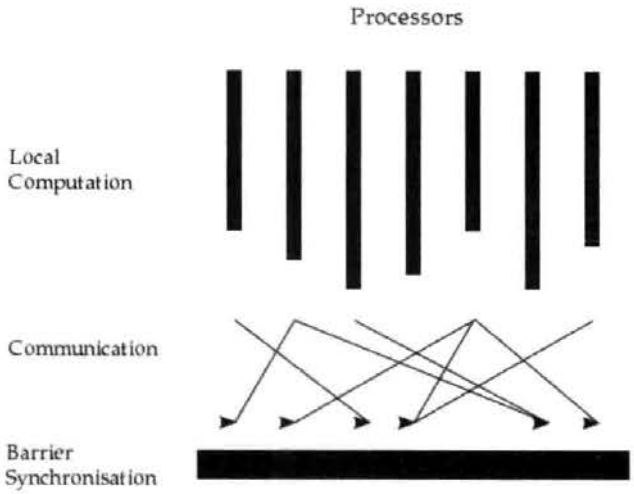


图 8.4 BSP同步栅栏模型

图8.5形象表述了以顶点为中心的计算模型。

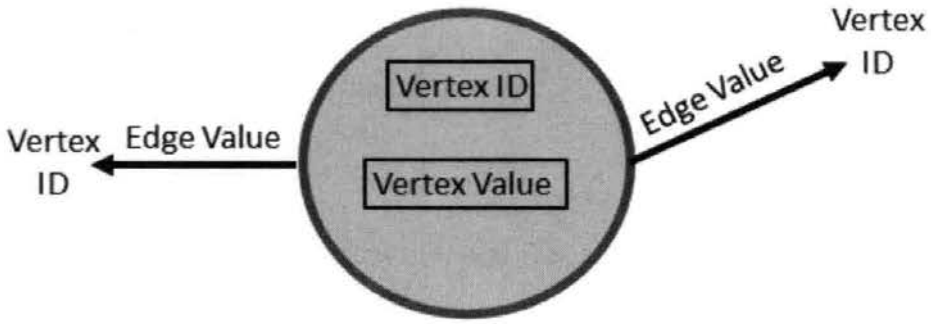


图 8.5 以顶点为中心的计算模型

一个典型的Pregel计算过程会经历如下步骤：

（1） 读取输入初始化图。

（2） 图初始化完成后，运行一系列超步，直到整个计算结束，超步间通过全局的同步点进行分隔。

（3） 输出计算结果。

在每一个超步中，针对参与计算的顶点执行相同的处理函数。处理函数由用户定义，在该函数内的处理逻辑大致如下所述：

（1） 读取在超步$S-1$发给自己的消息。

（2） 更改顶点自己的状态和以自己为源点的边。

（3） 发送消息给相邻顶点，这些消息将会在超步$S+1$相邻顶点接收。

那么如何判断整个计算都已经完成呢？条件是所有顶点都表决可以停止工作，即所有顶点都已经进入“非活跃状态”。图8.6是顶点的状态机转换示意图。

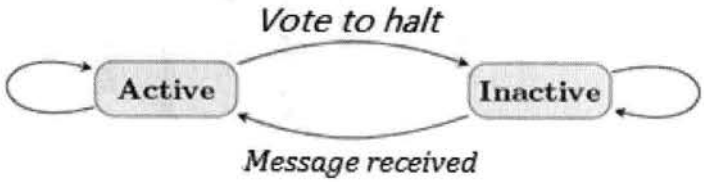


图 8.6 Vertex状态机

图8.7体现了在求取最大值的过程中，每个超步的变化过程。

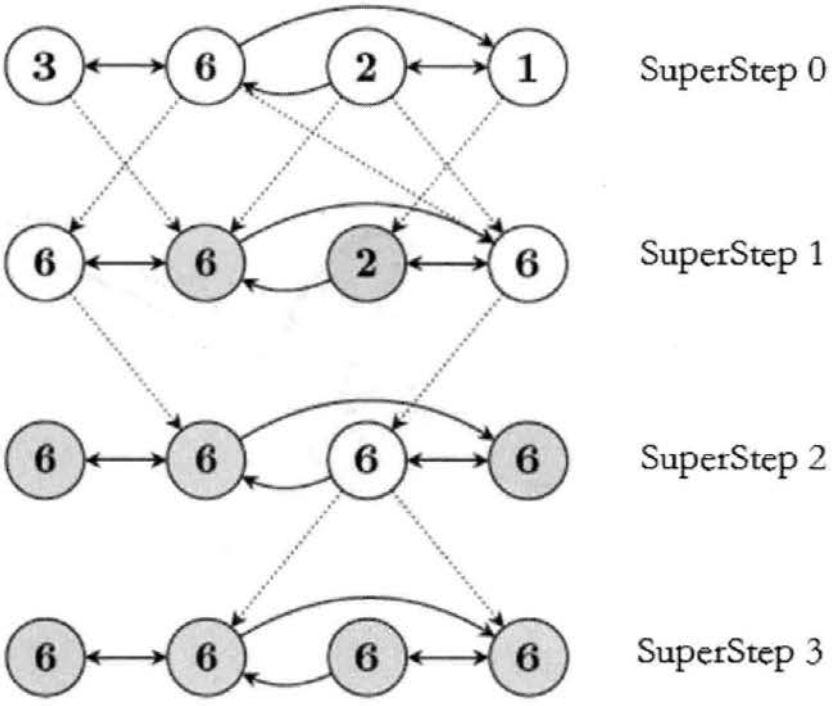


图 8.7 最大值计算

为了减少数据通信和同步的开销，Pregel还设计了Combiner和 Aggregator。Combiner能够在每次消息传递前进行判断，如果有多条消息从一个节点发送至另一节点，那么就可以将这些信息进行合并批处理，从而减少消息传递的通信开销。

比如有多个顶点要向顶点V发送消息，消息中只包含一个整数，顶点V拿到这些数值之后进行求和操作，那么完全可以先进行局部求和操作，然后再发送给顶点V，这样就大大降低了通信开销。

而Aggregator能够对各工作节点提供全局统一的数据视图，各个节点在计算过程中可以直接将数据更新至主节点的Aggregator，需要时可以直接查询使用，降低了数据同步和维护的代价。

8.4 GraphX图计算框架实现分析

除了GraphX之外，Apache Hama也是基于Pregel论文的开源实现，但目前Hama无论在成熟度还是整个生态圈方面还不是很成熟。

接下来具体分析一下Pregel在GraphX中的源码实现。先看一幅图，显示Pregel使用到的类之间的主要关系，具体内容如图 8.8所示。

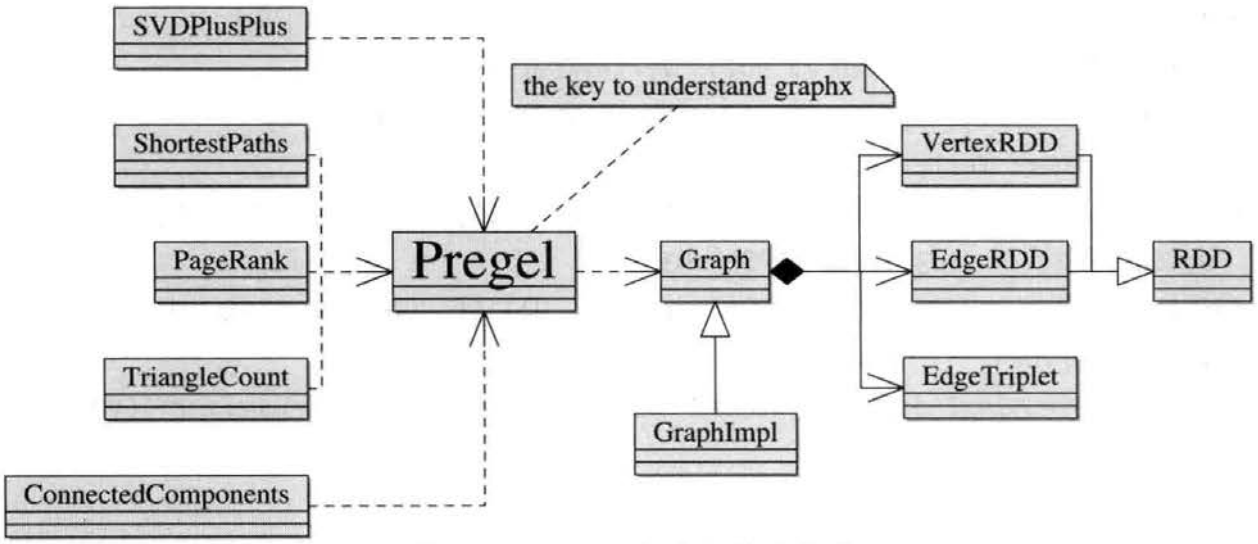


图 8.8 GraphX主要组件关系图

在GraphX，基于Pregel计算模型，实现了如下与图相关的算法：

（1） SVD++。
（2） ShortestPaths 最短路径问题。
（3） ConnectedComponents 连通子图。
（4） StronglyConnectedComponents 强连通图。
（5） LabelPropagation LPA算法。
（6） TriangleCount。

8.4.1 基本概念

为了能够对GraphX有足够深入的了解，我们先对GraphX 的基本数据类型进行简单的学习。

VertexRDD

VertexRDD是对RDD[(VertexID, A)]的继承和扩展，并加入了一些顶点属性信息，如VertexID表示顶点的ID且具有唯一性。我们可以通过ID，快速索引到所需要的顶点。

而VertexRDD[T]表示由类型T构成属性的顶点集合，本质上它是一个HashMap。

VertexRDD对象主要有Filter、mapValues、diff、leftJoin、innerJoin、aggregateUsingIndex接口，其具体接口定义如下所示。

代码清单 8.1 VertexRDD接口函数定义

```
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 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]
}
```

Filter能够取出满足指定条件的顶点集合。Filter操作保留了原始RDD的索引结构（Partition信息），本质上调用的是RDD的mapPartition操作，对RDD的每个Partition进行函数运算，并返回new RDD。

mapValues是对VertexRDD的属性进行函数操作，用户通过接口传入函数f，f再通过属性映射map((vid, attr) => f(attr)实现对顶点属性的操作。

diff接口主要用来比对两个VertexRDD的不同，并返回两个VertexRDD中非共同元素构成的VertexRDD。

innerJoin和leftJoin类似于数据库中的连接操作，对具有相同VertexId键值的VertexRDD进行连接合并操作。

aggregateUsingIndex主要用来对具有相同ID的VertexRDD进行reduceFunc操作。

EdgeRdd

EdgeRDD[ED, VD]是对RDD[Edge[ED]]的继承扩展。EdgeRDD内的边是通过分块组织来进行管理的，具体分割策略由PartitionStrategy来指定。在每个分块中，边的属性信息和邻接信息是分别存储的，这样的结构设计能够较好地支持动态属性的更新。除RDD基本的功能操作之外，EdgeRDD还具有以下3个接口函数，包括mapValues、reverse和innerJoin。

代码清单 8.2 EdgeRDD接口函数定义

```
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2, VD]
def reverse: EdgeRDD[ED, VD]
def innerJoin[ED2, ED3](other: EdgeRDD[ED2, VD])(f: (VertexId, VertexId, ED, ED2)
    => ED3): EdgeRDD[ED3, VD]
```

mapValues主要对EdgeRDD的属性进行函数变换操作，其变换操作的内容由传入的函数f决定，mapValues只改变EdgeRDD的属性值和类型，而不会改变本身的划分结构。

reverse是对EdgeRDD中所有的边进行反转操作，改变边的原有源节点和目的节点的方向。

innerJoin 主要对两个 EdgeRDD a 和 b 进行连接操作，这里要求 a 和 b 必须具有相同的PartitionStrategy。

代码清单 8.3 EdgeRDD示例

```
val edgeArray = Array(Edge(2L, 1L, 7),Edge(2L, 4L, 2),Edge(3L, 2L, 4),Edge(3L, 6L,
    3),Edge(4L, 1L, 1),Edge(5L, 2L, 2),Edge(5L, 3L, 8),Edge(5L, 6L, 3))
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
```

EdgeTriplet

边点三元体Triplet是GraphX提出的一种特有类型，它是除了顶点和边之外的属性图第三视角，为后续的数据计算和查找提供了便利。边点三元体是顶点和边的结合，它通过扩展edge类并对其加入源顶点属性和目标顶点属性来创建新类，其结构组成如图8.9 所示。

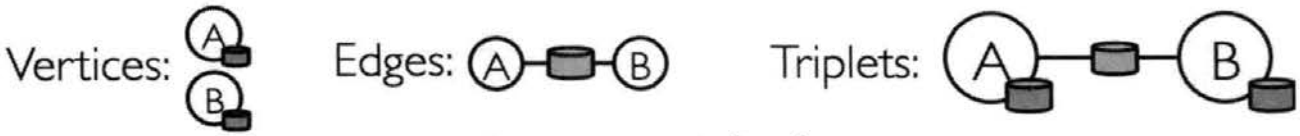


图 8.9 Triplets组成示意

EdgeTriplet接口定义如代码清单8.4所示。

代码清单 8.4 EdgeTriplet接口定义

```
class EdgeTriplet[VD, ED] extends Edge[ED] {
 var srcAttr: VD = _
 var dstAttr: VD = _
}
```

我们可以看出，EdgeTriplet实质上就是实现了对Edge[ED]的一层封装，并加入了边的源节点属性和目标节点的属性信息。EdgeTriplet这种新的数据结构为图的一些数据遍历和操作提供了方便。

比如，我们可以直接利用EdgeTriplet遍历边的源节点和目标节点。

代码清单 8.5 利用EdgeTriplet遍历源节点和目标节点

```
for (triplet <- graph.triplets.collect) {
  println(s"source ${triplet.srcAttr._1} dst ${triplet.dstAttr._1}")
}
```

再有，我们也可以利用triplet中的filter接口选择满足指定条件的边。

代码清单 8.6 利用EdgeTriplet选择满足指定条件的边

```
for (triplet <- graph.triplets.filter(t => t.attr > 3).collect) {
  println(s"source ${triplet.srcAttr._1} dst ${triplet.dstAttr._1}")
}
```

8.4.2 图的加载与构建

GraphX提供了几类方式来构建图结构数据。

- GraphLoader.edgeListFile

  GraphLoader的edgeListFile接口提供了一种可以直接从二维边结构数据创建图的方法。一般来说，加载的是邻接表，每一行表示一条边，行中的数字表示边的两个顶点。

  代码清单 8.7 邻接表数据格式

  ```
  This is a adjacency list
  1 2
  3 4
  5 6
  ```

  对于上述结构的邻接表格式数据，我们可以采用如下方法直接进行图数据的构建。

代码清单 8.8 基于邻接表构建图结构数据

```
val sc = new SparkContext(master,fileinput)
val graph = GraphLoader.edgeListFile(sc, args(1),true)
```

- Graph.apply方法

我们可以通过Graph伴生对象中的apply方法，通过传入VertexRDD和EdgeRDD来直接构建Graph，如下所示。

代码清单 8.9 图结构数据的构建

```
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

- Graph.fromEdges

由边集合构建图，这里的边类型为RDD[Edge[ED]]。

- Graph.fromEdgeTuples

通过由顶点ID标示的边（edges）来构建生成图，这里rawEdges的参数类型为RDD[(VertexId, VertexId)]。与 Graph.fromEdges有所不同。

8.4.3 图数据存储与分割

GraphX中采用顶点分割的方法对图数据进行分割，旨在减少通信和存储代价。我们在前面提到关于顶点分割的一些知识，所谓顶点分割就是把图中的边分散在各个节点，而顶点可能会跨节点存在。具体的边分割策略是由PartitionStrategy决定的，用户可以通过Graph.partitionBy接口指定图的具体分割策略。默认地，图是基于边初始的划分方法进行分割的。GraphX中现有以下分割策略。

- EdgePartition1D

EdgePartition1D划分方法主要是根据源的VertexID和partitionID进行随机分配，具有相同源节点的边会被分配在同一个块内。

其具体分配算法如下所示。

val mixingPrime: VertexId = 1125899906842597L

(math.abs(src) * mixingPrime).toInt % numParts

其中，mixingPrime是给定的一个固定Long Int值。mixingPrime主要用来增加分配过程的随机性和均匀性。最后分配的Partition就是源顶点的绝对值与mixingPrime的乘积与partitionID的余数。

- EdgePartition2D

EdgePartition2D是一种二维的划分方法，它能够将边划分至N个节点，保证顶点的复制份

数小于$2^{sqrt(numParts)}$，同时尽可能地保持边分布的均匀性，从而保证计算负载的平衡。其主要分割思路如下。

（1） 获取分配矩阵的平方根因子。

val ceilSqrtNumParts: PartitionID = math.ceil(math.sqrt(numParts)).toInt

ceilSqrtNumParts是2的整数次幂，它能够保证在对源顶点和目标顶点随机化的过程中，其分配的复制份数不会超过$2^{sqrt(numParts)}$。

（2） 将源顶点和目标顶点的ID随机化。

这里，依然是通过引入一个mixingPrime常量增加对col和row分配的随机性和均匀性，具体过程如下。

val mixingPrime: VertexId = 1125899906842597L

val col: PartitionID = (math.abs(src * mixingPrime) % ceilSqrtNumParts).toInt

val row: PartitionID = (math.abs(dst * mixingPrime) % ceilSqrtNumParts).toInt

（3） 均匀分配至numParts个数据块。

(col * ceilSqrtNumParts + row) % numParts

- RandomVertexCut

RandomVertexCut的策略非常简单，就是根据源顶点和目标顶点构成的边进行hash编码，再对分割数numParts进行除余计算，具体计算过程如下。

math.abs((src, dst).hashCode())

- CanonicalRandomVertexCut

CanonicalRandomVertexCut与RandomVertexCut类似，只不过在hash编码前，进行了一次顶点值排序。在源顶点和目标顶点中，ID小的排在前面，ID大的放在后面。然后再将由递增顺序构成的边对分割数numParts进行除余计算，得到相应的 Partition，具体计算过程如下。

val lower = math.min(src, dst)

val higher = math.max(src, dst)

math.abs((lower, higher).hashCode()) % numParts

8.4.4 操作接口

与Spark中的RDD具有许多基础操作算子一样（如count、map、filter 等），GraphX中的属性图graph也具有一些基本的操作集合，为开发使用提供便捷的API接口。这些操作接口可以分为属性变化、结构操作、连接操作、邻接聚合、缓存操作等。

属性变换

属性变换（Property Operators）能够通过map函数对graph的顶点和边进行属性变换操作，从而产生变换后的新图。属性操作具体包括3个操作接口，即mapVertices、mapEdges和mapTriplets。

代码清单 8.10 属性变换相关的API

```
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]
}
```

结构操作

结构操作（Structural Operators），主要是对图的顶点和边进行比对和过滤。在现有的Graph接口中，主要包括4类结构操作：reverse、subgraph、mask、groupEdges。

代码清单 8.11 结构操作相关的API

```
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]
}
```

reverse操作，主要用来对图中所有的有向边进行翻转操作，同时保留原图的顶点和边的所有属性信息，同时也不会更改顶点和边的数量。reverse 操作，可以用来进行反向PageRank计算。

subgraph操作，能够对graph进行信息过滤和提取，并返回满足指定条件的子图。从功能上看，其比较类似于rdd中的filter操作。

mask操作，假设有a、b两个图，以b.mask(a)为例，b.mask(a)就是对两个图a和b进行比对，并返回与a 具有相同节点和边的子图。

groupEdges主要用来对边进行合并操作，如合并或去除两点之间的重复边、两点之间边上权值的合并计算。

连接操作

连接操作（Join Operators），同数据库中的连接操作功能类似，主要通过关联属性获取其他有用的属性信息。在现有的GraphX连接操作中，主要有joinVertices、outerJoinVertices两类。

joinVertices利用外部数据与原有数据进行合并，可以理解为新旧数据的求和操作。

代码清单 8.12 joinVertices使用举例

```
import org.apache.spark.graphx.util.GraphGenerators
val rawGraph: Graph[Int, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 10, numEParts=3).mapVertices( (
    id, _) => id )
val outDeg: VertexRDD[Int] = rawGraph.outDegrees
val graph = rawGraph.leftJoinVertices[Int,Int](outDeg,
  (v, deg) => deg )
```

使用foreach来检查最新的(vid,data)。

代码清单 8.13 打印vid和data

```
  rawGraph.vertices.foreach(println)
```

outerJoinVertices是利用外部的数据源来更新当前顶点的属性，是对已有属性的替换。下面举个例子来说明。

代码清单 8.14 outerJoinVertices示例

```
import org.apache.spark.graphx.util.GraphGenerators
val rawGraph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 10, numEParts=3).mapVertices( (
    id, _) => id.toDouble )
val outDeg: VertexRDD[Int] = rawGraph.outDegrees()
val newGraph = rawGraph.outerJoinVertices(outDeg) {
  (vid, data, optDeg) => optDeg.getOrElse(0)
}
```

在rawGraph中，顶点1的键值对是$(vid, data) = (1, 1.0)$，在newGraph中用每个顶点的出度替换了原来的值，则顶点1的新键值对是$(vid, data) = (1, 10)$。

8.4.5 Pregel在GraphX中的源码实现

Pregel在GraphX中的实现定义于文件Pregel.scala，其处理逻辑在apply函数中得到体现。

代码清单 8.15 Pregel.apply

```
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
```

整个Pregel作业结束的前提有两种可能：（1）达到最大的迭代次数，（2）所有的顶点都已进入非活跃状态。

在Pregel.apply函数中最重要和最难以理解的函数是mapReduceTriplets。从抽象的角度来说，它实现了Pregel基本模型中的sendMessage和combiner。

mapFunc用来决定是否需要向相邻顶点发送消息及发送什么样的消息；reduceFunc则实现了combiner功能，用以减少不必要的通信开销。

代码清单 8.16 mapReduceTriplets

```
override def mapReduceTriplets[A: ClassTag](
      mapFunc: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
      reduceFunc: (A, A) => A,
      activeSetOpt: Option[(VertexRDD[_], EdgeDirection)] = None): VertexRDD[A] =
    {

    vertices.cache()

    // For each vertex, replicate its attribute only to partitions where it is
    // in the relevant position in an edge.
    val mapUsesSrcAttr = accessesVertexAttr(mapFunc, "srcAttr")
    val mapUsesDstAttr = accessesVertexAttr(mapFunc, "dstAttr")
    replicatedVertexView.upgrade(vertices, mapUsesSrcAttr, mapUsesDstAttr)
    val view = activeSetOpt match {
      case Some((activeSet, _)) =>
        replicatedVertexView.withActiveSet(activeSet)
```

```
  case None =>
    replicatedVertexView
}
val activeDirectionOpt = activeSetOpt.map(_._2)

// Map and combine.
val preAgg = view.edges.partitionsRDD.mapPartitions(_.flatMap {
  case (pid, edgePartition) =>
    // Choose scan method
    val activeFraction = edgePartition.numActives.getOrElse(0) / edgePartition
.indexSize.toFloat
    val edgeIter = activeDirectionOpt match {
      case Some(EdgeDirection.Both) =>
        if (activeFraction < 0.8) {
          edgePartition.indexIterator(srcVertexId => edgePartition.isActive(
srcVertexId))
            .filter(e => edgePartition.isActive(e.dstId))
        } else {
          edgePartition.iterator.filter(e =>
            edgePartition.isActive(e.srcId) && edgePartition.isActive(e.dstId)
)
        }
      case Some(EdgeDirection.Either) =>
        // TODO: Because we only have a clustered index on the source
        // vertex ID, we can't filter the index here. Instead we have
        // to scan all edges and then do the filter.
        edgePartition.iterator.filter(e =>
          edgePartition.isActive(e.srcId) || edgePartition.isActive(e.dstId))
      case Some(EdgeDirection.Out) =>
        if (activeFraction < 0.8) {
          edgePartition.indexIterator(srcVertexId => edgePartition.isActive(
srcVertexId))
        } else {
          edgePartition.iterator.filter(e => edgePartition.isActive(e.srcId))
        }
```

```
          case Some(EdgeDirection.In) =>
            edgePartition.iterator.filter(e => edgePartition.isActive(e.dstId))
          case _ => // None
            edgePartition.iterator
        }

        // Scan edges and run the map function
        val mapOutputs = edgePartition.upgradeIterator(edgeIter, mapUsesSrcAttr,
    mapUsesDstAttr)
          .flatMap(mapFunc(_))
        // Note: This doesn't allow users to send messages to arbitrary vertices.
        edgePartition.vertices.aggregateUsingIndex(mapOutputs, reduceFunc).
    iterator
    }).setName("GraphImpl.mapReduceTriplets - preAgg")

    // do the final reduction reusing the index map
    vertices.aggregateUsingIndex(preAgg, reduceFunc)
  } // end of mapReduceTriplets
```

有些读者可能会觉得上面的代码过于枯燥、难以理解，下面举一个实际的例子来说明mapReduceTriplets的使用，通过使用示例来进一步理解mapReduceTriplets的功能。

代码清单 8.17 mapReduceTriplets应用示例

```
import org.apache.spark.graphx.util.GraphGenerators

val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100,numEparts=10).mapVertices(
    (id, _) => id.toDouble )

val olderFollowers: VertexRDD[(Int, Double)] = graph.mapReduceTriplets[(Int,
    Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      //如果粉丝年龄比自己大，则加入计数
      Iterator((triplet.dstId, (1, triplet.srcAttr)))
    } else {
```

```
      //否则忽略
      Iterator.empty
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)

val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match { case (count, totalAge) =>
     totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

上述代码用以求解比自己年龄大的粉丝的平均年龄。

8.5 PageRank

8.5.1 什么是PageRank

PageRank是Google专有的算法，用于衡量特定网页相对于搜索引擎索引中的其他网页而言的重要程度。它由Larry Page 和 Sergey Brin在20世纪90年代后期发明。PageRank实现了将链接价值概念作为排名因素。

PageRank将对页面的链接看成投票，指示了重要性。

8.5.2 PageRank核心思想

“在互联网上，如果一个网页被很多其他网页所链接，就说明它受到普遍的承认和依赖，那么它的排名就很高。”（摘自《数学之美》第10章。）

“你说得这也太简单了吧，不是跟没说一个样嘛，怎么用数学来表示呢?”许多人会这么说。

呵呵，起初笔者也是这么想的，后来多看了几遍之后，明白了一点点。分析步骤用文字表述如下：

（1） 网页和网页之间的关系用图来表示。

（2） 网页A和网页B之间的链接关系表示任意一个用户从网页A转到网页B的可能性（概率）。

（3） 所有网页的排名用一维向量 $\boldsymbol{B}$ 来表示。

所有网页之间的链接用矩阵 $\boldsymbol{A}$ 来表示，所有网页排名用 $\boldsymbol{B}$ 来表示。

$$\boldsymbol{B}_i = \boldsymbol{A} \times \boldsymbol{B}_{i-1}$$

初始假设，所有的网页排名都是$1/N$，即 $\boldsymbol{B}_0 = (\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})$，显然通过简单的矩阵运算，可以得到$\boldsymbol{B}_1, \boldsymbol{B}_2, \ldots$，可以证明$\boldsymbol{B}_i$最终会收敛，即$\boldsymbol{B}_i$无限趋近于$\boldsymbol{B}$，此时$\boldsymbol{B} = \boldsymbol{B} \times \boldsymbol{A}$。因此，当两次迭代的结果$\boldsymbol{B}_i$和$\boldsymbol{B}_{i-1}$之间的差异非常小，接近于0时，迭代运算结束。

由于网页之间链接的数量相比互联网规模非常稀疏，因此计算网页的网页排名也需要对零概率或者小概率事件进行平滑处理。网页的排名是个一维向量，对它的平滑处理只能利用一个小的常数α，上述的计算公式演化为

$$\boldsymbol{B}_i = [\frac{\alpha}{N} \bullet \boldsymbol{I} + (1-\alpha)\boldsymbol{A}] \bullet \boldsymbol{B}_{i-1}$$

理论分析，数学建模完成之后剩下的就是工程实现了。由于成功地将问题转化成了图相关的计算，因此利用Pregel计算框架就可以轻松地将问题彻底解决掉了。

在 8.3节中已经提到，在Pregel提供给上层应用使用的编程接口中，需要用户自定义3个函数：

- vertexProgram
- sendMessage
- messageCombiner

根据PageRank的数学建模不难实现这3个函数，其源码如代码清单 8.18所示。

代码清单 8.18 PageRank.run

```
def run[VD: ClassTag, ED: ClassTag](
    graph: Graph[VD, ED], numIter: Int, resetProb: Double = 0.15): Graph[Double,
   Double] =
{
  // Initialize the pagerankGraph with each edge attribute having
  // weight 1/outDegree and each vertex with attribute 1.0.
  val pagerankGraph: Graph[Double, Double] = graph
    // Associate the degree with each vertex
    .outerJoinVertices(graph.outDegrees) { (vid, vdata, deg) => deg.getOrElse(0)
   }
    // Set the weight on the edges based on the degree
    .mapTriplets( e => 1.0 / e.srcAttr )
```

```
    // Set the vertex attributes to the initial pagerank values
    .mapVertices( (id, attr) => 1.0 )
    .cache()

  // Define the three functions needed to implement PageRank in the GraphX
  // version of Pregel
  def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
    resetProb + (1.0 - resetProb) * msgSum
  def sendMessage(edge: EdgeTriplet[Double, Double]) =
    Iterator((edge.dstId, edge.srcAttr * edge.attr))
  def messageCombiner(a: Double, b: Double): Double = a + b
  // The initial message received by all vertices in PageRank
  val initialMessage = 0.0

  // Execute pregel for a fixed number of iterations.
  Pregel(pagerankGraph, initialMessage, numIter, activeDirection = EdgeDirection
  .Out)(
    vertexProgram, sendMessage, messageCombiner)
}
```

最后以一个完整例子来结束本章内容。

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
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)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

第9章

MLLib

“不愤不启，不悱不发。举一隅不以三隅反，则不复也。”

《论语·述而篇》

好的机器学习算法不仅能极大地减轻人工操作，甚至能够以此创业建立一个科技巨头，如PageRank之于传说中的Google。

机器学习算法牵涉到大量的并行化计算，Spark就为这些机器学习算法提供了一个不错的实现环境。

本章将对一些常见的机器学习算法在Spark MLLib中的实现做一些简要的分析。

9.1 线性回归

线性回归（Linear Regression）是利用被称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个被称为回归系数的模型参数的线性组合。只有一个自变量的情况被称为简单回归，大于一个自变量情况的则为多元回归。

给定一个随机样本$(Y_i, X_{i1}, \ldots, X_{ip}), i = 1, \ldots, n$，一个线性回归模型假设回归因子$Y$和回归变量$X_{i1}$，……，$X_{ip}$之间的关系是除了$X$的影响以外，还有其他的变量存在。我们加入一个误差项ε_i（也是一个随机变量来捕获），表示除了X_{i1}，……，X_{ip}之外任何对Y_i的影响。所以，一个多变量线性回归模型表示为以下的形式：

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \ldots + \beta_p X_{ip} + \varepsilon_i, \quad i = 1, \ldots, n$$

其他模型可能被认定成非线性模型。一个线性回归模型不需要是自变量的线性函数。线性在这里表示Y_i的条件均值在参数β里是线性的。例如模型

$$Y_i = \beta_1 X_i + \beta_2 X_i^2 + \varepsilon_i$$

在β_1和β_2里是线性的，但在X_i^2里是非线性的，它是X_i的非线性函数。

9.1.1 数据和估计

用矩阵表示多变量线性回归模型为下式：

$$\boldsymbol{Y} = \boldsymbol{X}\beta + \boldsymbol{\varepsilon}$$

其中，$\boldsymbol{Y}$是一个包括了观测值$Y_1, \ldots, Y_n$的列向量，$\boldsymbol{\varepsilon}$包括了未观测的随机向量$\varepsilon_1, \ldots, \varepsilon_n$，另外还有回归向量的观测矩阵$\boldsymbol{X}$。

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}$$

9.1.2 线性回归参数求解方法

回归分析的最初目的是估计模型的参数，以便达到对数据的最佳拟合。

那么如何对计算出来的模型进行评估呢？

最小二乘法是一种数学优化技术。它通过最小化误差的平方和寻找数据的最佳函数匹配。换句话说：计算得到一个函数$\mu(x)$=$\mu(xi; \alpha_1, \alpha_2, \ldots, \alpha_k)$，使得观察值$Y$和函数值的误差平方和最小，即求得一组参数$\alpha_1, \alpha_2, \ldots, \alpha_k$使得下式的值最小：

$$S = \sum_{i=1}^{n} [y_i - \mu(x_i; \alpha_1, \alpha_2, \ldots, \alpha_k)]^2$$

分别求S对$\alpha_1, \alpha_2, \ldots, \alpha_k$的偏导数，并让它们等于零。

求解上述方程组得到$\alpha_1, \alpha_2, \ldots, \alpha_k$的估计值，将得到的估计值代入函数$\mu(x) = \mu(x_i; \alpha_1, \alpha_2, \ldots, \alpha_k)$中就可以得到我们想要的函数（最佳函数匹配）。

在决定一个最佳拟合的不同标准之中，最小二乘法是非常优越的。最小二乘法的拟合标准：

- 用函数表示为$\min\limits_{\vec{i}} \sum\limits_{i=1}^{n} (y_m - Y_i)^2$。

- 用欧几里得度量表示为 $\min_{\vec{i}} \|\vec{y}_m(\vec{x}) - \vec{y}\|_2$。

直接求取参数β

参数向量β的估计值为 $\hat{\beta} = (X^T X)^{-1} X^T y$。

为了达到演示的目的，可以使用octave来计算线性回归，以下是具体步骤。

步骤1: 启动octave。

代码清单 9.1 运行octave

```
octave
```

步骤2: 为octave安装strut和optim。

代码清单 9.2 安装struct和optim

```
octave:1>pkg install -forge struct
octave:1>pkg install -forge optim
```

步骤3: 加载struct和optim。

代码清单 9.3 加载struct和optim

```
octave:1>pkg load struct
octave:1>pkg load optim
```

步骤4: 使用LinearRegression函数，在octave环境内运行如下脚本。

代码清单 9.4 在octave中实现线性回归示例

```
n = 100;
x = sort(rand(n,1)*5-1);
y = 1+0.05*x + 0.1*randn(size(x));
F = [ones(n,1),x(:)];
[p,e_var,r,p_var,y_var] = LinearRegression(F,y);
yFit = F*p;
figure()
plot(x,y,'+b',x,yFit,'-g',x,yFit+1.96*sqrt(y_var),'--r',x,yFit-1.96*sqrt(y_var)
   ,'--r')
title('straight line by linear regression')
legend('data','fit','+/-95%')
```

```
grid on
```

结果如图9.1所示。

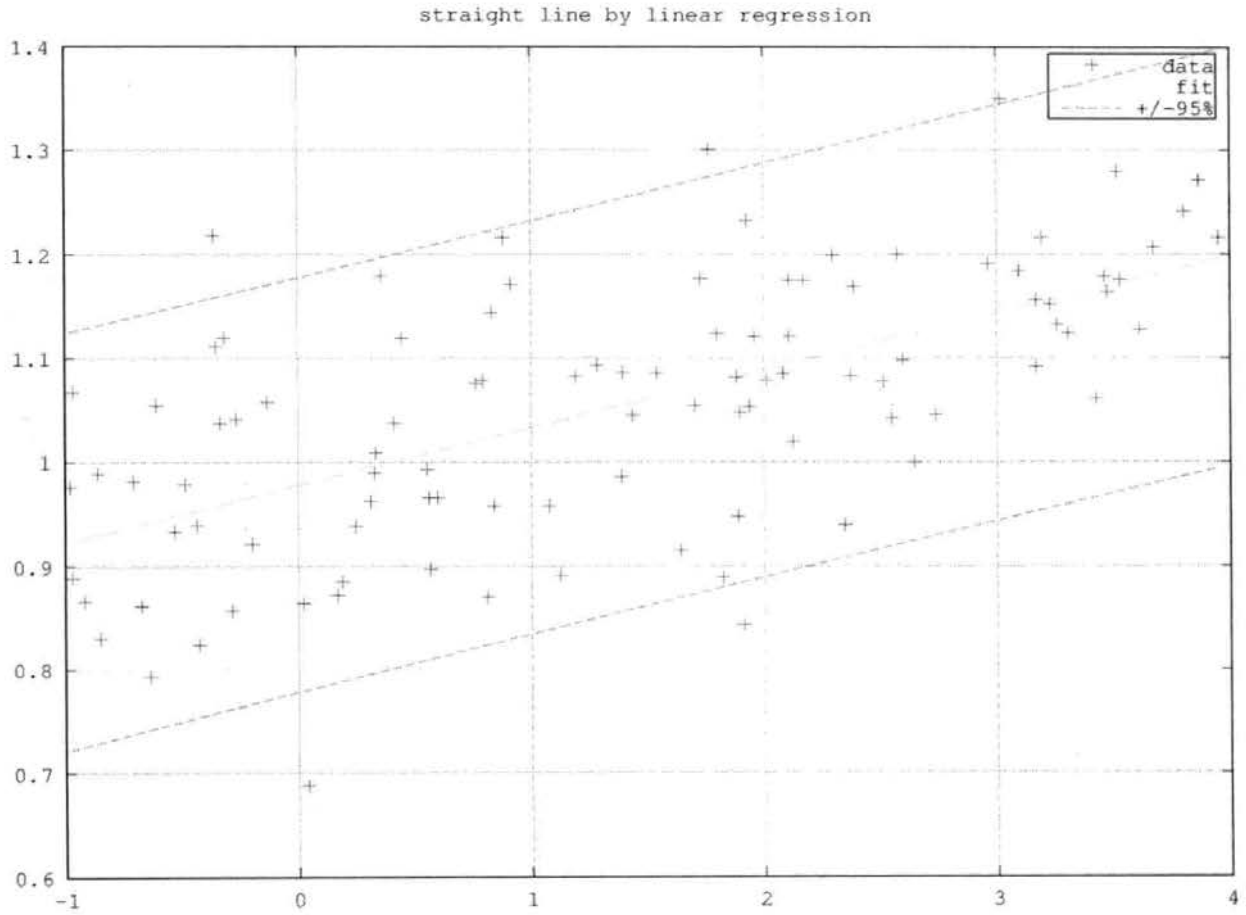


图 9.1 在octave中实现线性回归示例

梯度下降法

梯度下降法（gradient descent）基于这样一种观察：如果实值函数$F(X)$在点a处可微且有定义，那么函数$F(X)$在a点沿着梯度相反的方向$-\bigtriangledown F(a)$下降最快。

因而，如果$b = a - \gamma \bigtriangledown F(a)$对于$\gamma > 0$为一个极小数值时成立，那么$F(a) \geqslant F(b)$。

考虑到这一点，我们可以从函数F的局部极小值的初始估计X_0出发，并考虑如下序列$X_0, X_1, X_2 \ldots$使得 $X_{n+1} = X_n - \gamma_n \bigtriangledown F(X_n), n \geqslant 0$。

因此，可得到 $F(x_0) \geqslant F(X_1) \geqslant F(X_2) \geqslant \ldots$

如果顺利的话（X_n）收敛到期望的极值，注意每次迭代的步长γ可变。

图9.2显示了这一过程，这里假设F定义在平面上，并且函数图像是一个碗形。其中，曲线是等高线（水平集），即函数F为常数的集合构成的曲线。箭头指向该点梯度的反方向（一点处的梯度方向与通过该点的等高线垂直）。沿着梯度下降方向，将最终到达碗底，即函数F值最小的点。

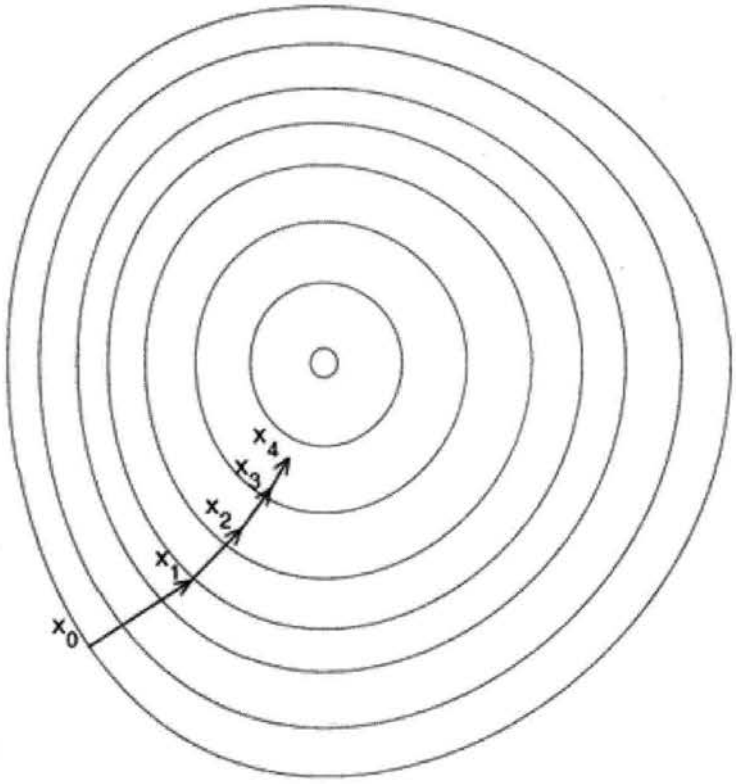


图 9.2 梯度下降过程演示

算法实现

对于输入样本集$x^{(i)}, y^{(i)}; i = 1, \ldots, m$，其中$x^i = [x_0^{(i)}, \ldots, x_n^i]^T$。上标为样本序号，共$m$个样本；下标为每个样本的特征数，共$n$个特征。

设拟合函数（hypothesis）为（其中$X_0 = 1$）

$$h(x) = \sum_{i=0}^{n} \theta_i X_i = \theta^T x$$

设目标函数为

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

则拟合函数为$\min(J(\theta))$，算法仅需找出达到此标准的参数向量$\theta = [\theta_0, \ldots, \theta_n]^T$的值。

其演算步骤如下所述。

- 随机选取参数向量的初始值。

- 对于每个参数θ_j（$j \in [0,n]$）的值，做如下更新：

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

将$J(\theta)$带入上式，则有

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)} \quad (for\ every\ j)$$

由于该算法每次更新θ中的一个元素，需要处理整个输入样本集，所以该算法也叫批量梯度下降（batch gradient descent）。当将参数向量θ中的所有元素更新后带入$J(\theta)$，通过两次迭代值来判读其是否收敛，以终止迭代。

更新θ时$h(x)$的值，由上一次完全更新的θ值带入后求解。

梯度下降法的特点有三：

- 有可能会收敛于局部最小值。
- 靠近极小值时速度减慢。
- 由于每次更新都需要处理整个样本集的数据，因此对于大量数据来说，速度较慢。

随机梯度下降

随机梯度下降（stochastic gradient descent）针对梯度下降法的不足做出了改进，其基本算法与梯度下降类似，只不过在每次更新参数向量时，只按照其中一个参数的梯度方向，更新参数向量中的一个参数。更新θ_j的表达式如下：

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

讲到这里你可能会问，列了一大堆公式之后，我还是不懂啊，最小二乘法和随机梯度下降算法有什么关系呢？

二者的联系是这样的：最小二乘法能够很好地评估线性回归的拟合度，而利用梯度下降法可找到最能满足最小二乘法的权重向量。

最小二乘法是判定线性回归拟合度最好的成本函数（cost-function），而梯度下降是用来帮助找到成本函数中对应向量的方法。于是问题就演变成用随机梯度下降法来找到cost-function的最小值。

随机梯度下降法更为精简的表示如下：

$$w := w - \alpha \bigtriangledown Q_i(w)$$

其中，$\bigtriangledown Q_i(w)$表示对$Q_i(w)$求导。

不同的成本函数，代入上述公式即可求得对应的具体梯度计算方法。以线性回归为例来看一看使用随机梯度下降法的具体过程。

假设有一组两维的训练样本$(x_1,y_1),\ldots,(x_n,y_n)$，准备找到拟合度最好的线性回归方程$y = w_1 + w_2 x$，拟合度好坏的评判标准采用最小二乘法（least square error），那么将拟合函数

代入随机梯度下降计算公式之后得到的目标函数如下所示：

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (w_1 + w_2 x_i - y_i)^2$$

将上述表达式展开得到：

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \alpha \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}$$

看到这里基本上能够想到为什么线性回归算法可以使用Spark来实现了，原因有二：

（1） 有迭代处理的算法步骤。

（2） 求解方程可以用矩阵来表示，而矩阵运算可以并行处理。

9.1.3 正则化

对较复杂的数据建模的时候，普通线性回归会有一些问题:

- 稳定性与可靠性。当X列满秩的时候，使用线性回归有解析解。如果观测数量n和预测变量m比较接近，则容易产生过拟合；如果nX^TX是奇异的，则最小二乘回归得不到有意义的结果。因此，线性回归缺少稳定性与可靠性。
- 模型解释能力的问题。包括在一个多元线性回归模型里的很多变量可能是和响应变量无关的；也有可能产生多重共线性的现象：即多个预测变量之间明显相关。这些情况都会增加模型的复杂程度，削弱模型的解释能力。这时候需要进行变量选择（特征选择）。

针对OLS的问题，在变量选择方面有3种扩展的方法：

- 子集选择。这是传统的方法，包括逐步回归和最优子集法等，对可能的部分子集拟合线性模型，利用判别准则（如AIC、BIC、Cp、调整R2 等）决定最优的模型。
- 收缩方法（shrinkage method）。收缩方法又被称为正则化（regularization）。主要是岭回归（ridge regression）和lasso回归。通过对最小二乘估计加入惩罚约束，使某些系数的估计为0。
- 维数缩减。主成分回归（PCR）和偏最小二乘回归（PLS）的方法。把p个预测变量投影到m维空间（$m < p$），利用投影得到的不相关的组合建立线性模型。

下边就岭回归和lasso法做一些具体的阐述。

岭回归或脊回归（ridge regression）

即在线性回归的代价函数后面加上一个惩罚项，对拟合函数进行平滑，防止过拟合。同样求导可得

$$\theta = (X^TX + \lambda I)^{-1} X^T y$$

（其中， I是单位矩阵，λ 是岭参数，0<λ <1，λ =0是最小二乘法估计。）

岭回归是对最小二乘回归的一种补充。它损失了无偏性，来换取高的数值稳定性，从而得到较高的计算精度。

lasso回归

lasso是在RSS最小化的计算中加入一个L_1范数作为惩罚约束：

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p}|\beta_j| = RSS + \lambda \sum_{j=1}^{p}|\beta_j|$$

L_1范数的好处是，当λ充分大时，可以把某些待估系数精确地收缩到0。

关于岭回归和lasso，当然也可以把它们看作一个以RSS为目标函数、以惩罚项为约束的优化问题。

9.2 线性回归的代码实现

9.2.1 简单示例

线性回归算法的简单示例程序如下（见图9.3）。

代码清单 9.5 线性回归示例

```
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors

// Load and parse the data
val data = sc.textFile("mllib/data/ridge-data/lpsa.data")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split(' ').map(_.toDouble
    )))
}

// Building the model
val numIterations = 100
val model = LinearRegressionWithSGD.train(parsedData, numIterations)

// Evaluate model on training examples and compute training error
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)
```

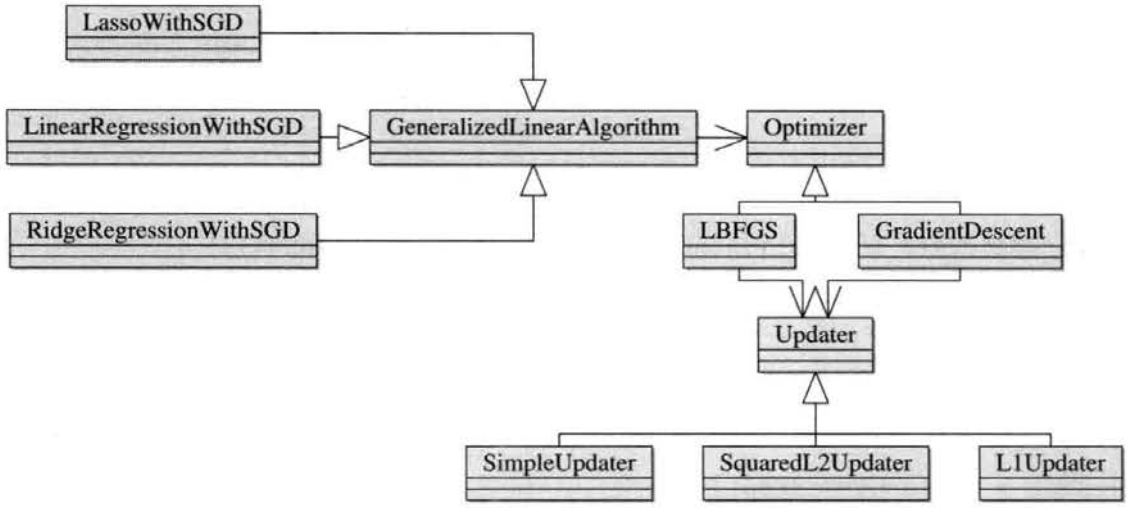


图 9.3 线性回归实现的类图概览

在第 9.1.2节的数学基础分析中讲到的随机梯度下降法及正则化规则与图 9.3中的类是什么关系呢？

可以从如下两个方面来看其内在联系。

- **Optimizer**：最优化算法。Optimizer表示用什么方法来使得成本函数（cost-function）效果最好。随机梯度下降（stochastic gradient descent）就是方法之一，还可以采用L-BFGS。
- **Updater**：正则化规则。岭回归使用SquaredL2Updater，lasso使用 L1Updater，没有正则化地使用SimpleUpdater。

只有将两者有机地联系起来，才会明白代码为什么需要这样安排。将理论与工程实践关联起来。

9.2.2 入口函数 train

现在开始分析一下函数的调用途径。

train→run，run函数的处理逻辑如下：

- 利用最优化算法来求得最优解，optimizer.optimize。
- 根据最优解创建相应的回归模型，createModel。

LinearRegressionWithSGD.train
└— LinearRegressionWithSGD.train
└— optimizer.optimize
└— GradientDescent.runMiniBatchSGD

代码清单 9.6 GeneralizedLinearRegression.run

```
def run(input: RDD[LabeledPoint], initialWeights: Vector): M = {

    // Check the data properties before running the optimizer
    if (validateData && !validators.forall(func => func(input))) {
      throw new SparkException("Input validation failed.")
    }

    // Prepend an extra variable consisting of all 1.0's for the intercept.
    val data = if (addIntercept) {
      input.map(labeledPoint => (labeledPoint.label, appendBias(labeledPoint.
    features)))
    } else {
      input.map(labeledPoint => (labeledPoint.label, labeledPoint.features))
    }

    val initialWeightsWithIntercept = if (addIntercept) {
      appendBias(initialWeights)
    } else {
      initialWeights
    }

    val weightsWithIntercept = optimizer.optimize(data,
    initialWeightsWithIntercept)

    val intercept = if (addIntercept) weightsWithIntercept(weightsWithIntercept.
    size - 1) else 0.0
    val weights =
      if (addIntercept) {
```

```
      Vectors.dense(weightsWithIntercept.toArray.slice(0, weightsWithIntercept.
  size - 1))
    } else {
      weightsWithIntercept
    }

  createModel(weights, intercept)
}
```

假设当前使用的最优化算法是GradientDescent，相应的优化代码如下所示。

代码清单 9.7 optimize

```
def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {
  val (weights, _) = GradientDescent.runMiniBatchSGD(
    data,
    gradient,
    updater,
    stepSize,
    numIterations,
    regParam,
    miniBatchFraction,
    initialWeights)
  weights
}
```

9.2.3 最优化算法 optimizer

runMiniBatchSGD是真正计算gradient及loss的地方。

代码清单 9.8 runMiniBatchSGD

```
def runMiniBatchSGD(
    data: RDD[(Double, Vector)],
    gradient: Gradient,
    updater: Updater,
    stepSize: Double,
    numIterations: Int,
```

```
    regParam: Double,
    miniBatchFraction: Double,
    initialWeights: Vector): (Vector, Array[Double]) = {

  val stochasticLossHistory = new ArrayBuffer[Double](numIterations)

  val numExamples = data.count()
  val miniBatchSize = numExamples * miniBatchFraction

  // if no data, return initial weights to avoid NaNs
  if (numExamples == 0) {

    logInfo("GradientDescent.runMiniBatchSGD returning initial weights, no data
found")
    return (initialWeights, stochasticLossHistory.toArray)

  }

  // Initialize weights as a column vector
  var weights = Vectors.dense(initialWeights.toArray)
  val n = weights.size

  /**
   * For the first iteration, the regVal will be initialized as sum of weight
   * squares if it's L2 updater; for L1 updater, the same logic is followed.
   */
  var regVal = updater.compute(
    weights, Vectors.dense(new Array[Double](weights.size)), 0, 1, regParam)._2

  for (i <- 1 to numIterations) {
    val bcWeights = data.context.broadcast(weights)
    // Sample a subset (fraction miniBatchFraction) of the total data compute
    // and sum up the subgradients on this subset (this is one map-reduce)
    val (gradientSum, lossSum) = data.sample(false, miniBatchFraction, 42 + i)
      .treeAggregate((BDV.zeros[Double](n), 0.0))(
```

```
      seqOp = (c, v) => (c, v) match { case ((grad, loss), (label, features))
=>
        val l = gradient.compute(features, label, bcWeights.value, Vectors.
fromBreeze(grad))
        (grad, loss + l)
      },
      combOp = (c1, c2) => (c1, c2) match { case ((grad1, loss1), (grad2,
loss2)) =>
        (grad1 += grad2, loss1 + loss2)
      })

  /**
   * NOTE(Xinghao): lossSum is computed using the weights from the previous
   * iteration and regVal is the regularization value computed in the previous
   * iteration as well.
   */
  stochasticLossHistory.append(lossSum / miniBatchSize + regVal)
  val update = updater.compute(
    weights, Vectors.fromBreeze(gradientSum / miniBatchSize), stepSize, i,
regParam)
  weights = update._1
  regVal = update._2
}

logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses %
s".format(
  stochasticLossHistory.takeRight(10).mkString(", ")))

(weights, stochasticLossHistory.toArray)

}
```

上述代码中最需要引起重视的部分是aggregate函数的使用。

代码清单 9.9 aggregate in runMiniBatchSGD

```
val (gradientSum, lossSum) = data.sample(false, miniBatchFraction, 42 + i)
```

```
  .treeAggregate((BDV.zeros[Double](n), 0.0))(
  seqOp = (c, v) => (c, v) match { case ((grad, loss), (label, features)) =>
    val l = gradient.compute(features, label, bcWeights.value, Vectors.fromBreeze
    (grad))
    (grad, loss + l)
  },
  combOp = (c1, c2) => (c1, c2) match { case ((grad1, loss1), (grad2, loss2)) =>
    (grad1 += grad2, loss1 + loss2)
  })
```

先看一看aggregate函数在RDD中的定义。

代码清单 9.10 RDD.aggregate

```
def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U):
    U = {
    // Clone the zero value since we will also be serializing it as part of tasks
    var jobResult = Utils.clone(zeroValue, sc.env.closureSerializer.newInstance())
    val cleanSeqOp = sc.clean(seqOp)
    val cleanCombOp = sc.clean(combOp)
    val aggregatePartition = (it: Iterator[T]) => it.aggregate(zeroValue)(
    cleanSeqOp, cleanCombOp)
    val mergeResult = (index: Int, taskResult: U) => jobResult = combOp(jobResult,
     taskResult)
    sc.runJob(this, aggregatePartition, mergeResult)
    jobResult
  }
```

aggregate函数有3个入参：一是初始值zeroValue，二是seqOp，三是combOp。其中，**seqOp**会被并行执行，具体由各个Executor上的Task来完成计算。**combOp**则是串行执行，在此，combOp操作在JobWaiter的taskSucceeded函数中被调用。

代码清单 9.11 JobWaiter.taskSucceeded

```
  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()
    }
  }
```

为了进一步加深对aggregate函数的理解，现在举一个小例子。

启动spark-shell后，运行如下代码。

代码清单 9.12 aggregate example

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.aggregate(0)(math.max(_, _), _ + _)
//运行结果为9
res0: Int = 9
```

仔细观察一下运行时的日志输出。aggregate提交的Job由一个stage（stage0）组成。由于整个数据集被分成两个Partition，所以为stage0创建了两个Task并行处理。

代码清单 9.13 aggregate函数运行日志

```
SparkContext: Starting job: aggregate at <console>:15
DAGScheduler: Got job 0 (aggregate at <console>:15) with 2 output partitions
 (allowLocal=false)
DAGScheduler: Final stage: Stage 0(aggregate at <console>:15)
DAGScheduler: Parents of final stage: List()
DAGScheduler: Missing parents: List()
DAGScheduler: Submitting Stage 0 (ParallelCollectionRDD[0] at parallelize at <
   console>:12), which has no missing parents
DAGScheduler: Submitting 2 missing tasks from Stage 0 (ParallelCollectionRDD[0]
   at parallelize at <console>:12)
TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
TaskSetManager: Re-computing pending task lists.
TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, PROCESS_LOCAL,
   1153 bytes)
```

```
TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, PROCESS_LOCAL,
   1153 bytes)
Executor: Running task 0.0 in stage 0.0 (TID 0)
Executor: Running task 1.0 in stage 0.0 (TID 1)
Executor: Finished task 1.0 in stage 0.0 (TID 1). 622 bytes result sent to driver
Executor: Finished task 0.0 in stage 0.0 (TID 0). 622 bytes result sent to driver
TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 84 ms on localhost
   (1/2)
TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 95 ms on localhost
   (2/2)
DAGScheduler: Stage 0 (aggregate at <console>:15) finished in 0.107 s
TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
SparkContext: Job finished: aggregate at <console>:15, took 0.248419409 s
```

讲完了aggregate函数的执行过程，回过头来继续介绍组成seqOp的gradient.compute函数。

LeastSquareGradient用来计算梯度和误差，注意cmopute中cumGraident会返回改变后的结果。这里计算公式依据的就是cost-function中的$\bigtriangledown Q(w)$。

代码清单 9.14 LeastSquareGradient

```
class LeastSquaresGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vector): (Vector,
    Double) = {
    val brzData = data.toBreeze
    val brzWeights = weights.toBreeze
    val diff = brzWeights.dot(brzData) - label
    val loss = diff * diff
    val gradient = brzData * (2.0 * diff)

    (Vectors.fromBreeze(gradient), loss)
  }

  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
```

```
    cumGradient: Vector): Double = {
  val brzData = data.toBreeze
  val brzWeights = weights.toBreeze
  //"•"表示点积，是接受在实数R上的两个向量
  //并返回一个实数标量的二元运算，它的结果
  //是欧几里得空间的标准内积。两个向量的点积
  //写作a•b，点乘的结果叫作点积，也称作数量积
  val diff = brzWeights.dot(brzData) - label

  //下面这句话完成y += a*x
  brzAxpy(2.0 * diff, brzData, cumGradient.toBreeze)

  diff * diff
 }
}
```

在上述代码中频繁出现Breeze相关的函数。你一定会很好奇，这是个什么新鲜玩意儿。

说开了其实一点也不稀奇：由于计算中有大量的矩阵（Matrix）和向量（Vector）计算，因此，为了更好地支持和封装这些计算引入了Breeze库。

Breeze、Epic及Puck是scalanlp中的三大支柱性项目，具体可参考www.scalanlp.org，看一看fromBreeze的源码实现。

代码清单 9.15 fromBreeze

```
private[mllib] def fromBreeze(breezeVector: BV[Double]): Vector = {
  breezeVector match {
    case v: BDV[Double] =>
      if (v.offset == 0 && v.stride == 1) {
        new DenseVector(v.data)
      } else {
        new DenseVector(v.toArray)
         // Can't use underlying array directly, so make a new one
      }
    case v: BSV[Double] =>
      if (v.index.length == v.used) {
        new SparseVector(v.length, v.index, v.data)
      } else {
```

```
          new SparseVector(v.length, v.index.slice(0, v.used), v.data.slice(0, v.
    used))
        }
      case v: BV[_] =>
        sys.error("Unsupported Breeze vector type: " + v.getClass.getName)
    }
  }
```

9.2.4 权重更新 update

根据本次迭代出来的梯度和误差对权重系数进行更新，这个时候就需要用到正则化规则了。也就是下述语句会触发权重系数的更新。

代码清单 9.16 update

```
  val update = updater.compute(
     weights, Vectors.fromBreeze(gradientSum / miniBatchSize), stepSize, i,
    regParam)
```

以岭回归为例，看看其更新过程的代码实现。

代码清单 9.17 SquareL2Updater

```
class SquaredL2Updater extends Updater {
  override def compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {
    // add up both updates from the gradient of the loss (= step) as well as
    // the gradient of the regularizer (= regParam * weightsOld)
    // w' = w - thisIterStepSize * (gradient + regParam * w)
    // w' = (1 - thisIterStepSize * regParam) * w - thisIterStepSize * gradient
    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.toBreeze.toDenseVector
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    brzAxpy(-thisIterStepSize, gradient.toBreeze, brzWeights)
```

```
    val norm = brzNorm(brzWeights, 2.0)

    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * norm)
  }
}
```

9.2.5 结果预测 predict

计算出权重系数(weights)和截距intecept,就可以用来创建线性回归模型LinearRegressionModel,利用模型的predict函数来对观测值进行预测。

代码清单 9.18 predictPoint

```
class LinearRegressionModel (
    override val weights: Vector,
    override val intercept: Double)
  extends GeneralizedLinearModel(weights, intercept) with RegressionModel with
    Serializable {

  override protected def predictPoint(
      dataMatrix: Vector,
      weightMatrix: Vector,
      intercept: Double): Double = {
    weightMatrix.toBreeze.dot(dataMatrix.toBreeze) + intercept
  }
}
```

9.3 分类算法

除了线性回归中的目标函数可以使用随机梯度下降算法来求得最优解外，逻辑回归和支持向量机的目标函数同样也可以使用随机梯度下降法来求解。

GradientDescent类的继承体系如图 9.4所示，LeastSquareGradient已经在前面讲到，另外两个子类会在接下来的内容中详细讲解。

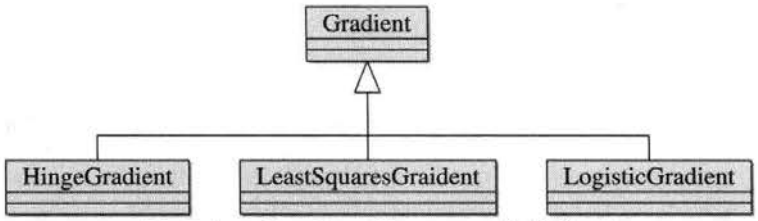


图 9.4 GradientDescent继承体系

9.3.1 逻辑回归

逻辑回归（Logical Regression）的假设函数：

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

逻辑回归用于分类中0/1问题的判断，即预测结果要么是0，要么是1的二值分类问题。前提条件是二值得满足伯努利分布：

$$P(y=1|x;\theta) = h_\theta(x)$$
$$P(y=0|x;\theta) = 1 - h_\theta(x)$$

逻辑回归的损失函数采用对数损失函数或对数似然损失函数：

$$L(Y, P(Y|X)) = -\log P(Y|X)$$
$$L(h_\theta(x), y) = \begin{cases} y=1, -\log(h_\theta(x)) \\ y=0, -\log(1-h_\theta(x)) \end{cases}$$
$$J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} y^i \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h(x^{(i)}))]$$

对损失函数求偏导数，仍然能得到类似于线性回归假设函数求偏导得到的梯度公式如下：

$$\frac{\delta}{\delta\theta_i} = \frac{1}{m}\sum_{i=1}^{m} L(h_\theta(x^{(i)} - y^{(i)}))x^{(i)}$$

可以看出其与线性回归的迭代求解公式在形式上完全一样，唯一的区别在于线性回归的假设函数是

$$y^i = \theta^T x^i + \varepsilon^i$$

而逻辑回归的假设函数是

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

搞清楚逻辑回归的目标函数可以采用随机梯度下降法来求解之后，再来看其源码实现就很容易了。

代码清单 9.19 LogisticGradient

```
class LogisticGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vector): (Vector,
    Double) = {
```

```
    val brzData = data.toBreeze
    val brzWeights = weights.toBreeze
    val margin: Double = -1.0 * brzWeights.dot(brzData)
    val gradientMultiplier = (1.0 / (1.0 + math.exp(margin))) - label
    val gradient = brzData * gradientMultiplier
    val loss =
      if (label > 0) {
        math.log1p(math.exp(margin))
        // log1p is log(1+p) but more accurate for small p
      } else {
        math.log1p(math.exp(margin)) - margin
      }

    (Vectors.fromBreeze(gradient), loss)
  }

  %% 代入逻辑回归的假设函数
  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val brzData = data.toBreeze
    val brzWeights = weights.toBreeze
    val margin: Double = -1.0 * brzWeights.dot(brzData)
    val gradientMultiplier = (1.0 / (1.0 + math.exp(margin))) - label

    brzAxpy(gradientMultiplier, brzData, cumGradient.toBreeze)

    if (label > 0) {
      math.log1p(math.exp(margin))
    } else {
      math.log1p(math.exp(margin)) - margin
    }
  }
}
```

9.3.2 支持向量机

支持向量机（Supported Vector Machine）的目标函数如下：

$$\min_{w}\left\{\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\varepsilon_i\right\}$$

其中，ε_i是对变量X_i根据hinge损失函数计算的结果：

$$\varepsilon_i = \begin{cases} 0 & if\ y_i(w_o + \sum_j y_j\alpha_j K(x_j, x_i)) \geqslant 1 \\ 1 - y_i(w_o + \sum_j y_j\alpha_j K(x_j, x_i) & otherwise \end{cases}$$

将ε代入之后，目标函数转换为

$$\min_{m}\sum_{1}^{N} L(y_i, f(x_i; w)) + \frac{\lambda}{2}\|W\|^2$$

该目标函数的最优化解可以使用随机梯度下降法求得。

至此可以来看一下HingeGradient的源码实现了。

代码清单 9.20 HingeGradient

```
class HingeGradient extends Gradient {
  override def compute(data: Vector, label: Double, weights: Vector): (Vector,
    Double) = {
    val brzData = data.toBreeze
    val brzWeights = weights.toBreeze
    val dotProduct = brzWeights.dot(brzData)

    // Our loss function with {0, 1} labels is max(0, 1 - (2y - 1) (f_w(x)))
    // Therefore the gradient is -(2y - 1)*x
    val labelScaled = 2 * label - 1.0

    if (1.0 > labelScaled * dotProduct) {
      (Vectors.fromBreeze(brzData * (-labelScaled)), 1.0 - labelScaled *
    dotProduct)
    } else {
      (Vectors.dense(new Array[Double](weights.size)), 0.0)
    }
  }

  override def compute(
      data: Vector,
```

```
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val brzData = data.toBreeze
    val brzWeights = weights.toBreeze
    val dotProduct = brzWeights.dot(brzData)

    // Our loss function with {0, 1} labels is max(0, 1 - (2y - 1) (f_w(x)))
    // Therefore the gradient is -(2y - 1)*x
    val labelScaled = 2 * label - 1.0

    if (1.0 > labelScaled * dotProduct) {
      brzAxpy(-labelScaled, brzData, cumGradient.toBreeze)
      1.0 - labelScaled * dotProduct
    } else {
      0.0
    }
  }
}
```

9.4 拟牛顿法

9.4.1 数学原理

L-BFGS是拟牛顿法（Qausi-Newton Method）的多个变种之一。

拟牛顿法是求解非线性优化问题最有效的方法之一。在详细说明L-BFGS之前，先回顾一下牛顿法及BFGS算法。

牛顿法的基本思想是在极小点附近通过对目标函数做二阶泰勒（Taylor）展开，进而得到极小点的下一个估计值。

设x_k为当前的极小点估计值，则

$$\varphi(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$$

表示$f(x)$在x_k附近的二阶泰勒展开式，由极值必要条件可知$\varphi(x)$应满足$\varphi'(x) = 0$。即

$$f'(x_k) + f''(x_k)(x - x_k) = 0$$

从而求得

$$x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

若给定初值x_0，则可以构造出如下的迭代格式：

$$x_{k+1} = x_k + \frac{f'(x_k)}{f''(x_k)},\ k = 0, 1, \ldots$$

产生的序列x_k来逼近$f(x)$的极小点，在一定条件下x_k可以收敛到$f(x)$的极小值。

$\boldsymbol{X}$是N维向量，对于$N > 1$的情况，二阶泰勒展式可以做推广，此时

$$\varphi(x) = f(x_k) + \bigtriangledown f(x_k) \cdot (x - x_k) + \frac{1}{2} \cdot (x - x_k)^T \cdot \bigtriangledown^2 f(x_k) \cdot (x - x_k)$$

其中，$\bigtriangledown f$为f的梯度向量，$\bigtriangledown^2 f$为f的海森矩阵。将其分别简记为$\boldsymbol{g}$和$\boldsymbol{H}$。特别地，当f的混合偏导数可交换次序，则海森矩阵$\boldsymbol{H}$为对称矩阵;而$\bigtriangledown f(x_k)$和$\bigtriangledown^2 f(x_k)$则表示将$x$取为$x_k$后得到的实值向量和矩阵，分别简记为$\boldsymbol{g}_k$（$g$表示gradient）和$\boldsymbol{H}_k$（$\boldsymbol{H}$表示Hessian）。

推广后，求极值的条件演变为

$$\boldsymbol{g}_k + \boldsymbol{H}_k \cdot (x - x_k) = 0$$

如果矩阵$\boldsymbol{H}_k$非奇异，则可求得

$$x = x_k - \boldsymbol{H}_k^{-1} \cdot \boldsymbol{g}_k$$

若给定初始值x_0，则同样可以构造出迭代格式：

$$x_{k+1} = x_k - \boldsymbol{H}_k^{-1} \cdot \boldsymbol{g}_k \qquad k = 0, 1, \ldots$$

这就是经典的牛顿迭代法，其迭代格式中的搜索方向$d_k = -\boldsymbol{H}_k^{-1} \cdot \boldsymbol{g}_k$称为牛顿方向。

Algorithm 1: 经典牛顿法

1 给定初值x_0和精度阈值ε，并令 $k := 0$。
2 计算$\boldsymbol{g}_k$和$\boldsymbol{H}_k$。
3 若$\|\boldsymbol{g}_k\| < \varepsilon$，则停止迭代；否则确定搜索方向$d_k = -\boldsymbol{H}_k^{-1} \cdot \boldsymbol{g}_k$。
4 计算新的迭代点$x_k + 1 := x_k + d_k$ 。
5 令 $k := k + 1$，转至步骤2。

牛顿法的优点体现在以下方面：

- 目标函数是二次函数时，由于二次泰勒展开函数与原目标函数是完全相同的二次式，海森矩阵退化为一个常数矩阵，只需要一次迭代就可达到$f(x)$的极小点x^*，因此牛顿法是具有二次收敛性的算法。

- 对于非二次函数，如果函数的二次性态较强，或迭代点已进入极小点的邻域，则其收敛速度也会很快。

经典牛顿法虽然具有二次收敛性，但是要求初始点需要尽量靠近极小点，否则有可能不收敛。计算过程中需要计算目标函数的二阶偏导数，难度较大。更为复杂的是目标函数的海森（Hesse）矩阵无法保持正定，会导致算法产生的方向不能保证是f在X_k处的下降方向，从而令牛顿法失效；特别地，如果海森矩阵奇异，牛顿方向可能根本是不存在的。

拟牛顿法

牛顿法收敛速度快，但是计算过程中需要计算目标函数的二阶偏导数，难度较大；目标函数的海森矩阵无法保持正定，从而令牛顿法失效。

为了解决这两个问题，人们提出了拟牛顿法，即“模拟”牛顿法的改进型算法。基本思想是不用二阶偏导数而构造出可以近似海森矩阵的逆的正定对称阵，从而在“拟牛顿”的条件下优化目标函数。海森矩阵构造方法不同决定了不同的拟牛顿法。

满足什么样条件的矩阵可以被当作$\boldsymbol{H}$的近似矩阵呢？这个判断的依据就是**拟牛顿方程**，用$\boldsymbol{B}$表示对海森矩阵$\boldsymbol{H}$本身的近似；而用$\boldsymbol{D}$表示对海森矩阵的逆$\boldsymbol{H}^{-1}$的近似，即$\boldsymbol{B} \approx \boldsymbol{H}, \boldsymbol{D} \approx \boldsymbol{H}^{-1}$。

推导过程如下。

设经过$k+1$次迭代后得到x_{k+1}，此时将目标函数$f(x)$在x_{k+1}附近作泰勒展开，取二阶近似得到

$$f(x) \approx f(x_{k+1}) + \bigtriangledown f(x_{k+1}) \cdot (x - x_{k+1}) + \frac{1}{2} \cdot (x - x_{k+1})^T \cdot \bigtriangledown^2 f(x_{k+1}) \cdot (x - x_{k+1})$$

在两边同时作用一个梯度算子，可得

$$\bigtriangledown f(x) \approx \bigtriangledown f(x_{k+1}) + \boldsymbol{H}_{k+1} \cdot (x - x_{k+1})$$

令$x = x_k$，整理可得

$$\boldsymbol{g}_{k+1} - \boldsymbol{g}_k \approx \boldsymbol{H}_{k+1} \cdot (x_{k+1} - x_k)$$

引入符号

$$s_k = x_{k+1} - x_k, y_k = \boldsymbol{g}_{k+1} - \boldsymbol{g}_k$$

则条件可写成

$$y_k \approx \boldsymbol{H}_{k+1} \cdot s_k \text{或者} s_k \approx \boldsymbol{H}_{k+1}^{-1} \cdot y_k$$

这就是拟牛顿条件，用记号$\boldsymbol{B}$和$\boldsymbol{D}$来表示就是

$$y_k = \boldsymbol{B}_{k+1} \cdot s_k \text{或} s_k = \boldsymbol{D}_{k+1} \cdot y_k$$

BFGS算法

BFGS算法取$\bigtriangledown_k$为秩2的对称矩阵，即令$\bigtriangledown_k = \alpha_k \mu_k \mu_k^T + \beta_k v_k v_k^T$，推导可得BFGS修正公式如下：

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k S_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

BFGS算法描述见Algorithm 2。

Algorithm 2: BFGS算法

1 给定初始值X_0和精度阈值ϵ，并令$D_0 = I,\ k := 0$。

2 确定搜索方向$d_k = -D_k \cdot g_k$。

3 计算步长λ_k，令$s_k = \lambda_k d_k, x_{k+1} := x_k + s_k$。

4 若$\|g_{k+1}\| < \epsilon$，则算法结束。

5 计算$y_k = g_{k+1} - g_k$。

6 计算$D_{k+1} = (I - \frac{s_k y_k^T}{y_k^T s_k} D_k (I - \frac{y_k s_k^T}{y_k^T s_k})) + \frac{s_k s_k^T}{y_k^T s_k}$。

7 令$k := k + 1$，转到步骤2。

L-BFGS算法

有了前面这么多的基础，终于可以提L-BFGS算法本身了。

在BFGS算法中，需要一个$N \times N$的矩阵$\boldsymbol{D}_k$。当N很大时，存储整个矩阵几乎变得不可能。即便意识到$\boldsymbol{D}_k$的对称性，将其整个加载到内存中还是不可沉受之重。

L-BFGS的基本思想就是只存储计算过程中的向量序列S_i,y_i，根据S_i和y_i来计算出$\boldsymbol{D}_k$。同时只存储最近的m个S_i,y_i。

算法描述如下所示。

Algorithm 3: L-BFGS算法

1 初始化

$$\delta = \begin{cases} 0 & if\ k \leqslant m \\ k - m & if\ k > m \end{cases} ; L = \begin{cases} k & if\ k \leqslant m \\ m & if\ k > m \end{cases} \quad q_l = g_k$$

2 后向循环

for *i = L -1, L-2, ..., 1, 0* **do**

3
$$j = i + \delta$$
$$\alpha_i = \rho_j s_j^T q_i + 1$$
$$q_i = q_{i+1} - \alpha_i Y_j$$

4 前向循环

$r_0 = D_0 \cdot q_0$

for *i =0, 1, ..., L-2, L-1* **do**

5
$$j = i + \delta$$
$$\beta_j = \rho_j y_j^T r_i$$
$$r_{i+1} = r_i + (\alpha_i - \beta_j) s_j$$

算出的r_L即$\boldsymbol{H}_k \cdot \boldsymbol{g}_k$的值。

9.4.2 代码实现

L-BFGS算法中使用到的正则化方法是SquaredL2Updater。

函数调用关系如图9.5所示。

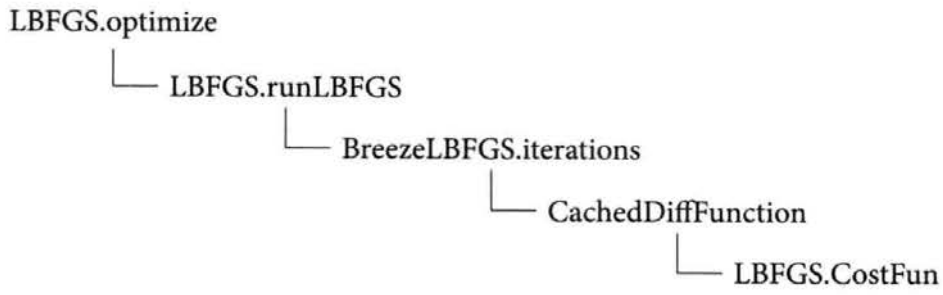


图 9.5 L-BFGS调用关系图

算法实现上使用到了由scalanlp的成员项目Breeze库提供的BreezeLBFGS函数，mllib中只是定义了BreezeLBFGS所需要的DiffFunctions。

代码清单 9.21 runLBFGS

```
def runLBFGS(
      data: RDD[(Double, Vector)],
      gradient: Gradient,
      updater: Updater,
      numCorrections: Int,
      convergenceTol: Double,
      maxNumIterations: Int,
      regParam: Double,
      initialWeights: Vector): (Vector, Array[Double]) = {

    val lossHistory = new ArrayBuffer[Double](maxNumIterations)

    val numExamples = data.count()

    val costFun =
      new CostFun(data, gradient, updater, regParam, numExamples)

    val lbfgs = new BreezeLBFGS[BDV[Double]](maxNumIterations, numCorrections,
```

```
    convergenceTol)

    val states =
      lbfgs.iterations(new CachedDiffFunction(costFun), initialWeights.toBreeze.
    toDenseVector)

    var state = states.next()
    while(states.hasNext) {
      lossHistory.append(state.value)
      state = states.next()
    }
    lossHistory.append(state.value)
    val weights = Vectors.fromBreeze(state.x)

    logInfo("LBFGS.runLBFGS finished. Last 10 losses %s".format(
      lossHistory.takeRight(10).mkString(", ")))

    (weights, lossHistory.toArray)
  }
```

CostFun的定义如下。

代码清单 9.22 CostFun

```
private class CostFun(
    data: RDD[(Double, Vector)],
    gradient: Gradient,
    updater: Updater,
    regParam: Double,
    numExamples: Long) extends DiffFunction[BDV[Double]] {

    private var i = 0

    override def calculate(weights: BDV[Double]) = {
      // Have a local copy to avoid the serialization of CostFun object
      // which is not serializable.
      val localGradient = gradient
```

```
    val n = weights.length
    val bcWeights = data.context.broadcast(weights)

    val (gradientSum, lossSum) = data.aggregate((BDV.zeros[Double](n), 0.0))
        (seqOp = (c, v) => (c, v) match { case ((grad, loss), (label, features))
  =>
          val l = localGradient.compute(
            features, label, Vectors.fromBreeze(bcWeights.value), Vectors.
  fromBreeze(grad))
          (grad, loss + l)
        },
        combOp = (c1, c2) => (c1, c2) match { case ((grad1, loss1), (grad2, loss2
  )) =>
          (grad1 += grad2, loss1 + loss2)
        })

     val regVal = updater.compute(
      Vectors.fromBreeze(weights),
      Vectors.dense(new Array[Double](weights.size)), 0, 1, regParam)._2

    val loss = lossSum / numExamples + regVal

    val gradientTotal = weights - updater.compute(
      Vectors.fromBreeze(weights),
      Vectors.dense(new Array[Double](weights.size)), 1, 1, regParam)._1.toBreeze


    axpy(1.0 / numExamples, gradientSum, gradientTotal)

    i += 1

    (loss, gradientTotal)
  }
}
```

9.5 MLLib与其他应用模块间的整合

Spark号称是一个一站式的解决方案，可以将streaming,sql,graph及mllib很好地结合在一起。本节通过几个具体实例来说明mllib与其他应用模块的结合。

示例1：sql和mllib相结合，利用sql接口从外部读取数据。

代码清单 9.23 使用sql作为mllib的数据源

```
val trainingTable=sql("""
  SELECT e.action,
       u.age,
           u.latitude,
           u.longitude
   FROM Users u
   JOIN Events e
   ON u.userId = e.userId""")

 val training = trainingTable.map { row =>
    val features = Vectors.dense(row(1),row(2),row(3))
        LabeledPoint(row(0),features)
 }

 val model = SVMWithSGD.train(training)
```

示例2：graphx和mllib相结合，使用线性回归算法对PageRank的结果进行分析。

代码清单 9.24 mllib和graphx相结合应用示例

```
val graph = Graph(pages, links)
val pageRank: RDD[(Long, Double)] = graph.staticPageRank(10).vertices
val labelAndFeatures: RDD[(Long, (Double, Seq(Int,Double)))] = ..
val training: RDD[LabeledPoint] =
   labelAndFeatures.join(pageRank).map {
     case (id, ((label, features), pageRank)) =>
```

```
        LabeledPoint(label, Vectors.sparse(features ++ (1000,
        pageRank)
}

val model = LogisticRegressionWithSGD.train(training)
```

第四部分

附录

附录 A　Spark 源码调试
附录 B　源码阅读技巧

附录A

Spark源码调试

在Spark源码走读过程中，难免会对一些函数执行及调用关系不甚明了，这时就期望能够借助于调试来解除疑惑。

Intellij IDEA是对Scala语言提供最好支持的IDE。借助于Intellij的调试工具，能够很好地对Spark的具体运行做深入了解。

本节将会详细介绍如何使用Intellij IDEA来调试Spark。

A.1 前提

本文假设开发环境是在Linux平台，并且已经安装下列软件，我个人使用的是Arch Linux：

- JDK
- Scala
- Sbt
- intellij-idea-community-edition

A.2　安装Scala插件

为IDEA安装Scala插件，具体步骤如下。

步骤1：启动Intellij IDEA之后，选择File→Setting.

步骤2：选择右侧的Install Jetbrains Plugin，在弹出窗口的左侧输入Scala，如图 A.1所示。

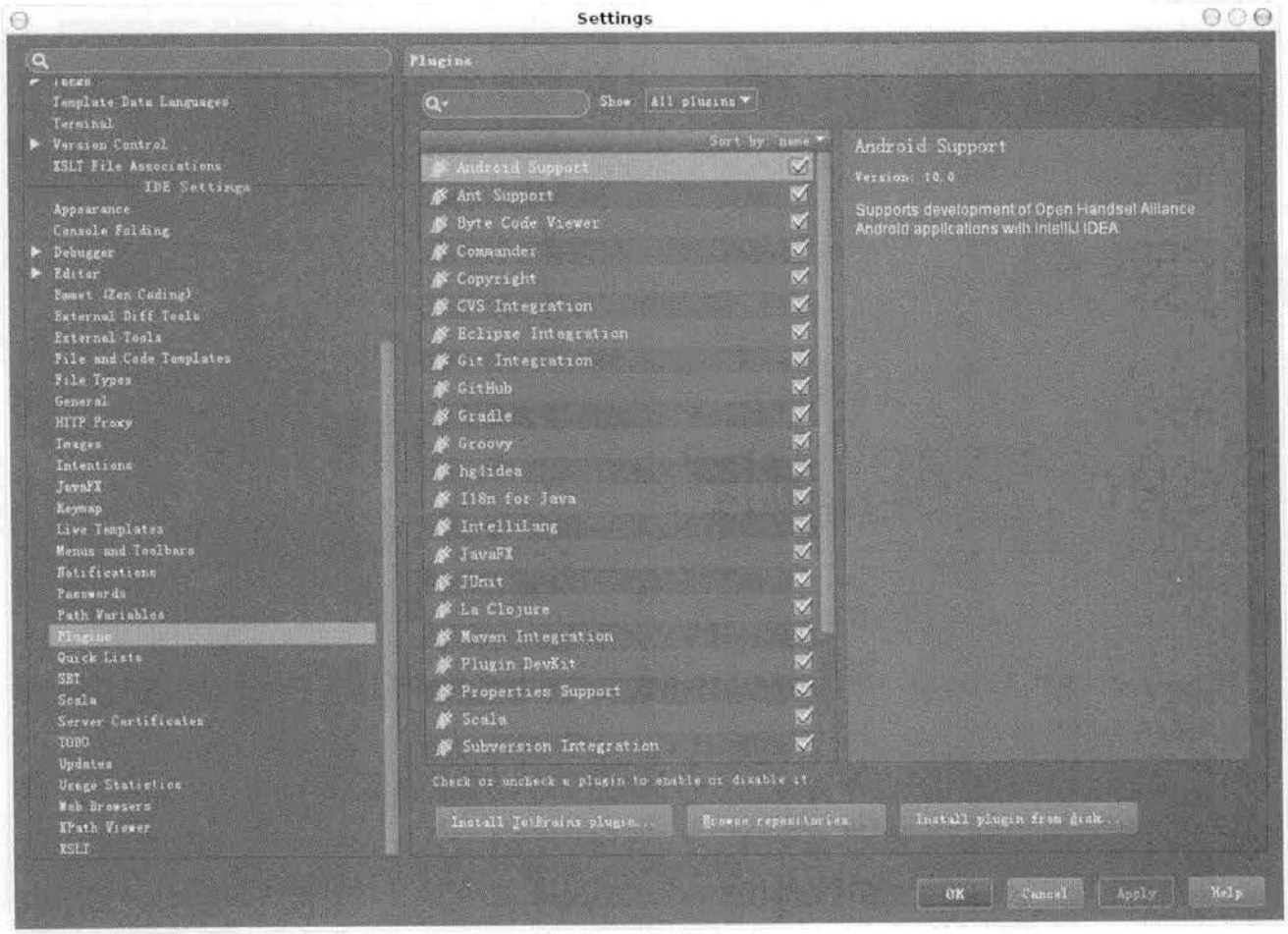


图 A.1　安装plugin

然后单击Update plugin或Install plugin，如图 A.2所示。

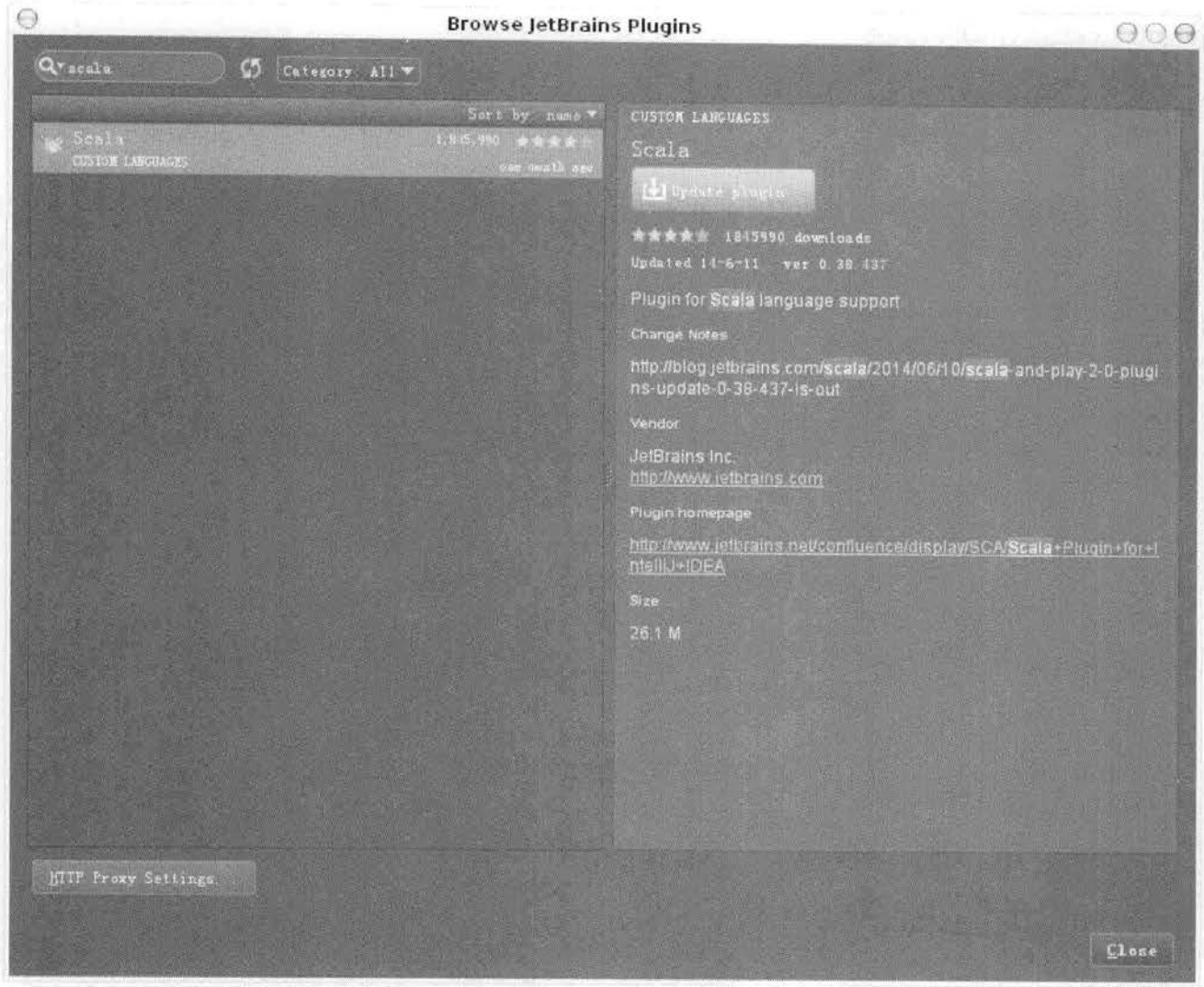


图 A.2 安装Scala插件

步骤3: Scala插件安装结束之后需要重启IDEA，这样Scala插件才能生效。由于在IDEA 13中已经原生支持 Sbt，所以已无须为IDEA安装单独的Sbt插件。

A.3 源码下载及导入

下载源码，假设使用git同步最新的源码。

代码清单 A.1 git同步源码

```
git clone https://github.com/apache/spark.git
```

A.3.1　导入Spark源码

步骤1: 选择File→Import Project，在弹出的窗口中指定Spark源码目录，如图 A.3所示。

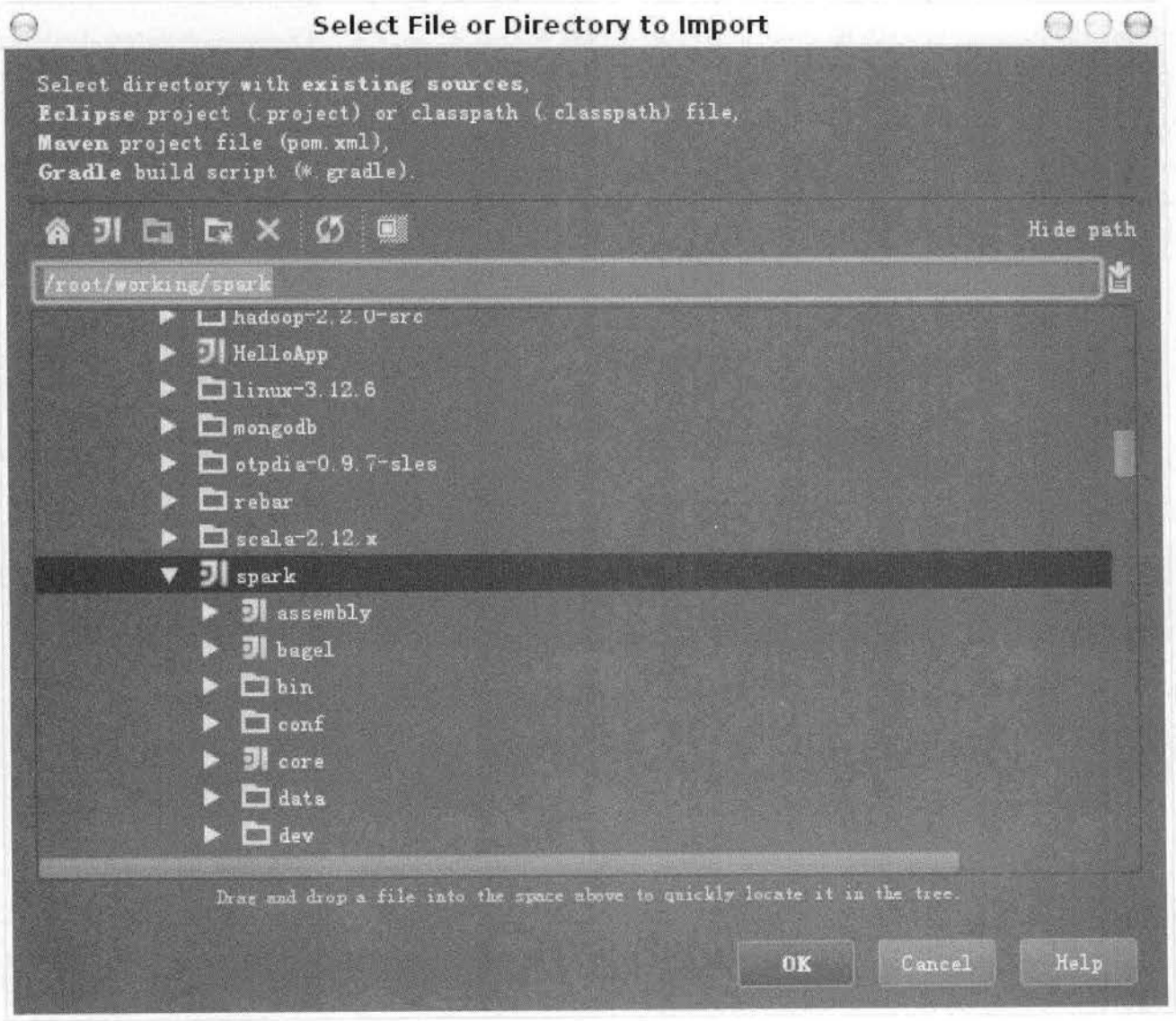


图 A.3　导入Spark源码

步骤2: 选择项目类型为Sbt Project，单击Next按钮，如图 A.4所示。

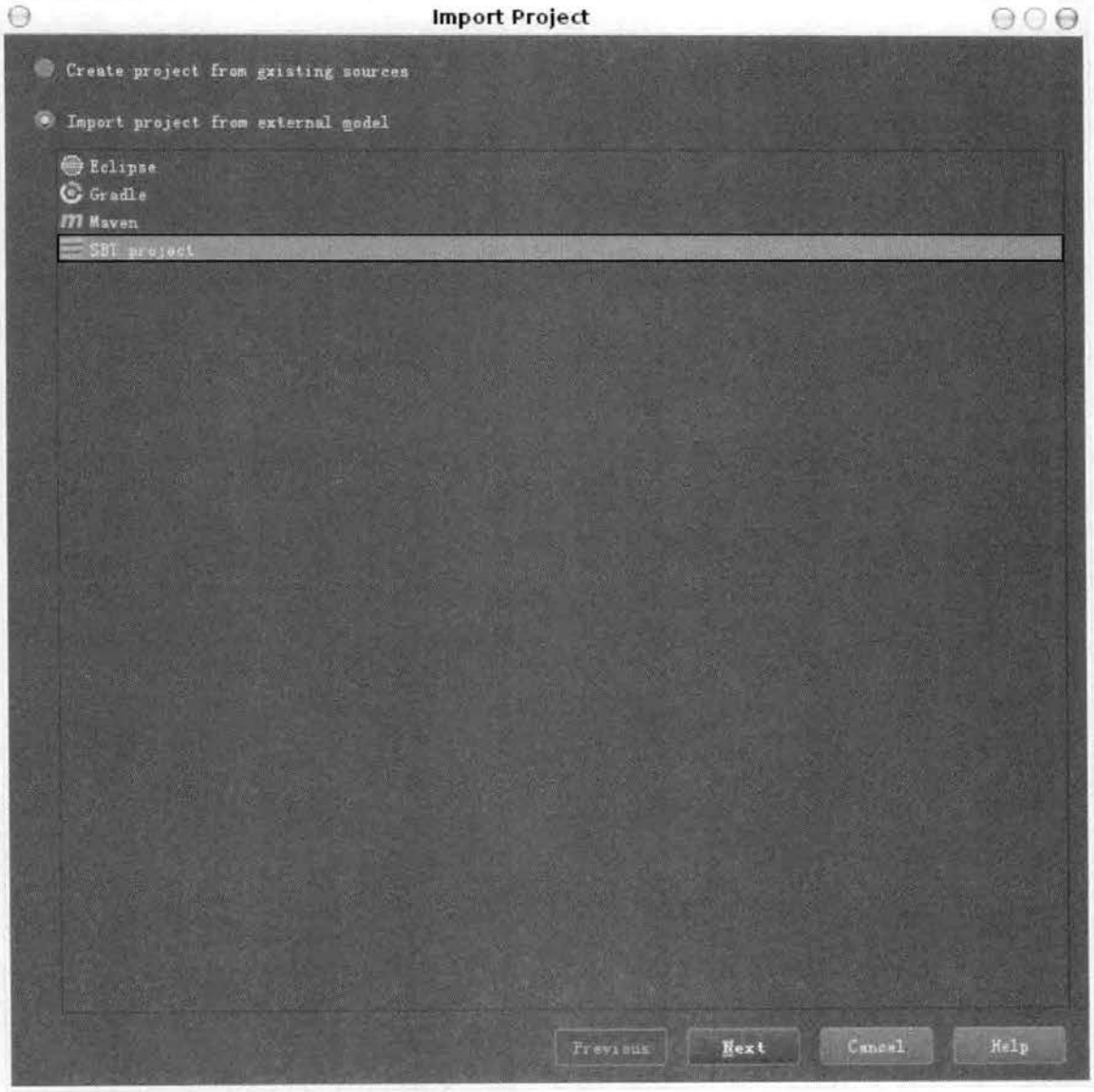


图 A.4 指定工程类型

步骤3: 在新弹出的窗口中先选中Use auto-import，然后单击Finish按钮，如图 A.5所示。

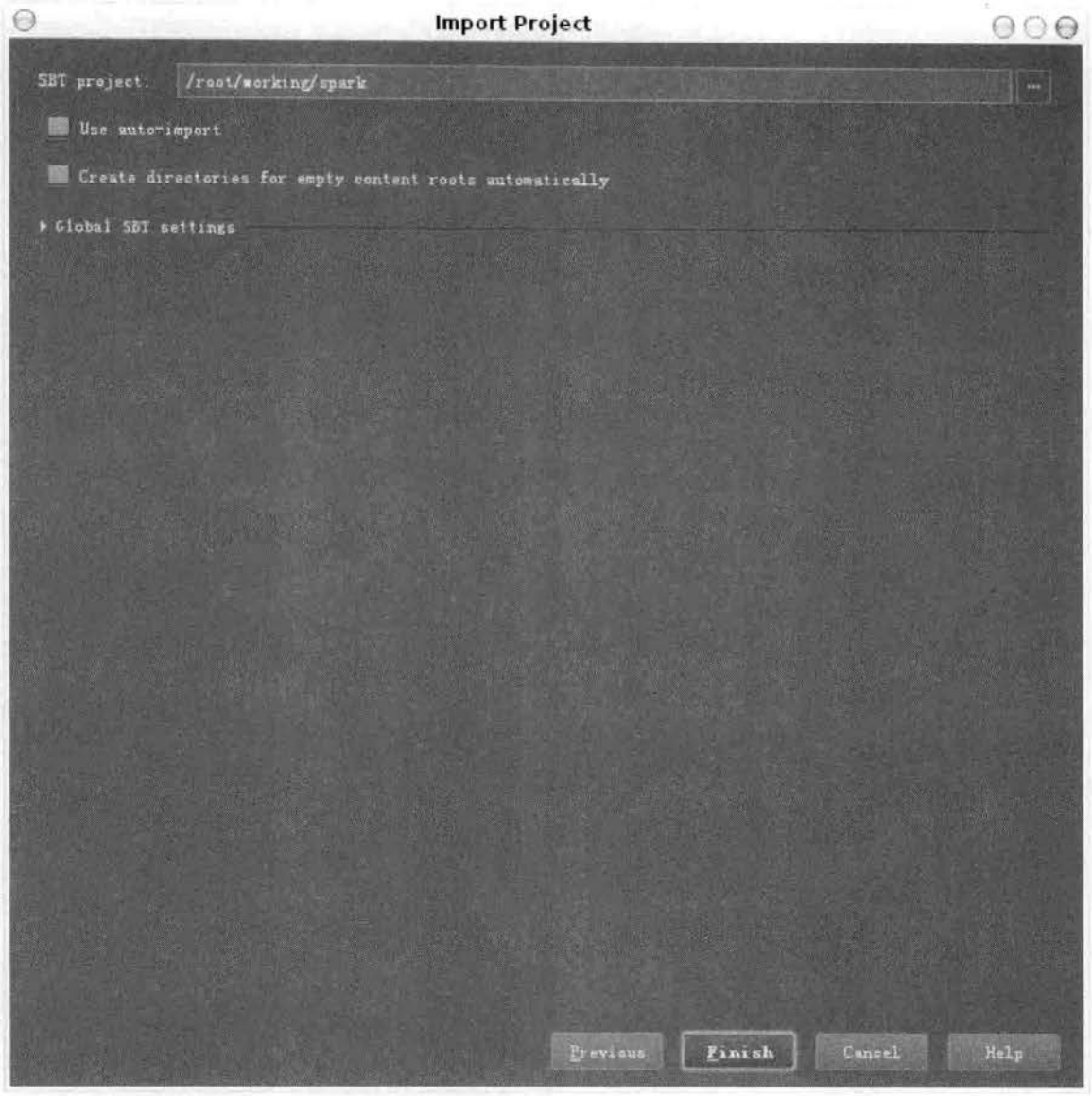


图 A.5　使用auto-import

导入设置完成，进入漫长的等待，IDEA会对导入的源码进行编译，同时会生成文件索引。

如果在提示栏出现如下的提示内容：“is waiting for .sbt.ivy.lock”，就说明该 lock文件无法创建，需要手工删除，具体操作如下。

代码清单 A.2　删除lock文件

```
cd $HOME/.ivy2
rm *.lock
```

手工删除掉lock之后，重启IDEA，重启后会继续上次没有完成的Sbt过程。

A.4 源码编译

在使用IDEA来编译Spark的示例程序时，中间会多次出错，出现错误的原因是没有很好地解决包之间的依赖关系。

解决办法如下。

步骤1: 选择File→Project Structures。

步骤2: 在右侧dependencies中添加新的module，如图 A.6所示。

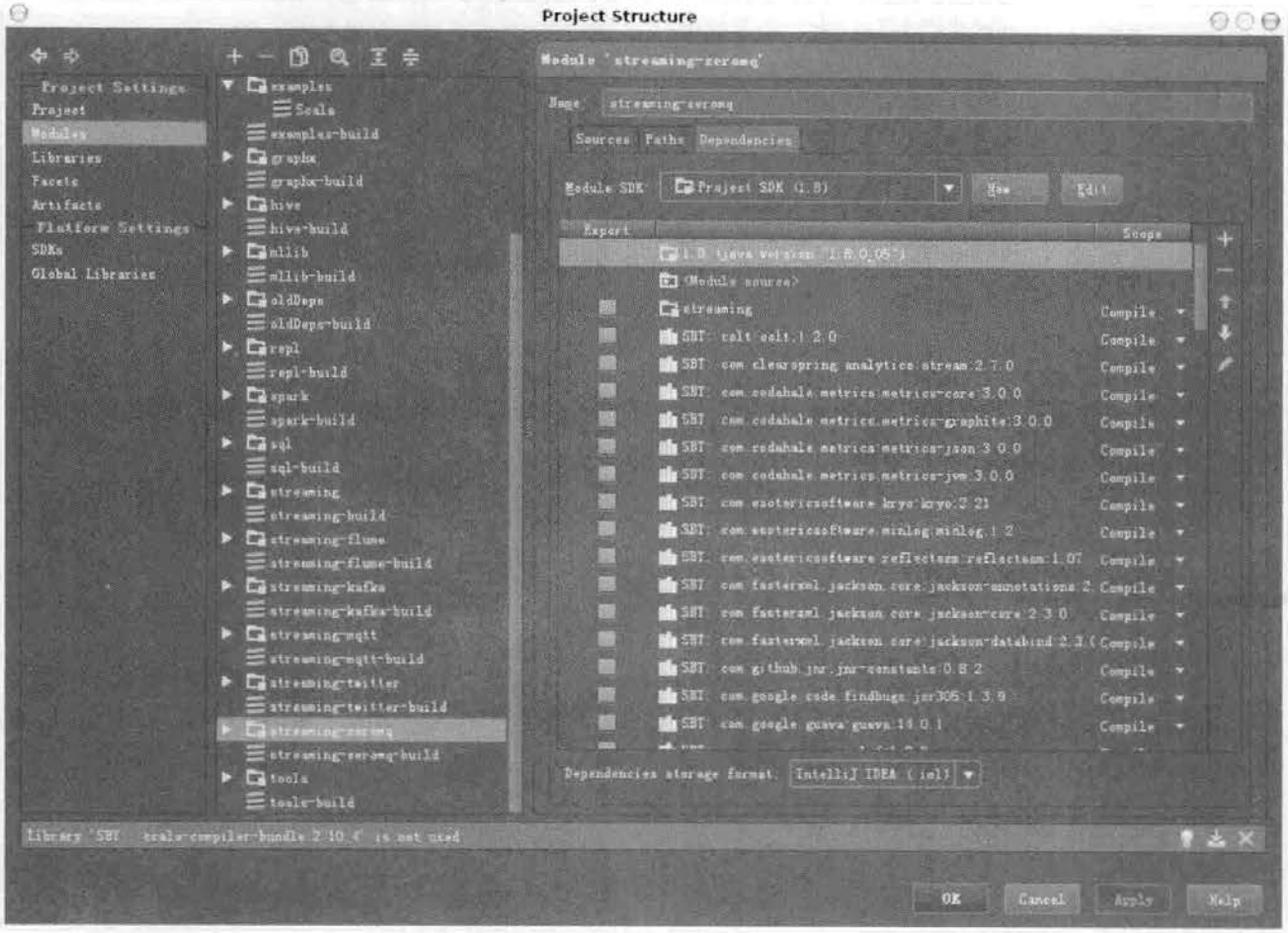


图 A.6 添加新module

步骤3: 选择spark-core。其他模块，如streaming-twitter、streaming-kafka、streaming-flume、streaming-mqtt出错的情况，解决方案与此类似（见图A.7）。

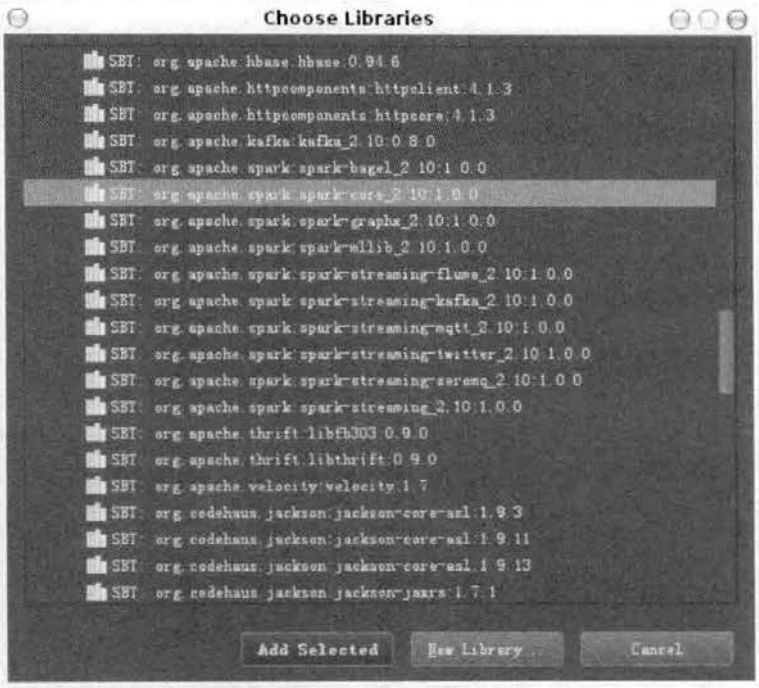


图 A.7　选择spark-core

注意Example编译报错时的处理稍有不同，在指定Dependencies的时候，不是选择Library而是选择Module dependency，在弹出的窗口中选择SQL。

A.5　调试LogQuery

步骤1: 选择Run→Edit configurations。

步骤2: 添加Application。注意右侧窗口中配置项内容的填写，分别为Main class、vm options、working directory、use classpath of module。

图 A.8显示了如何进行参数设置。

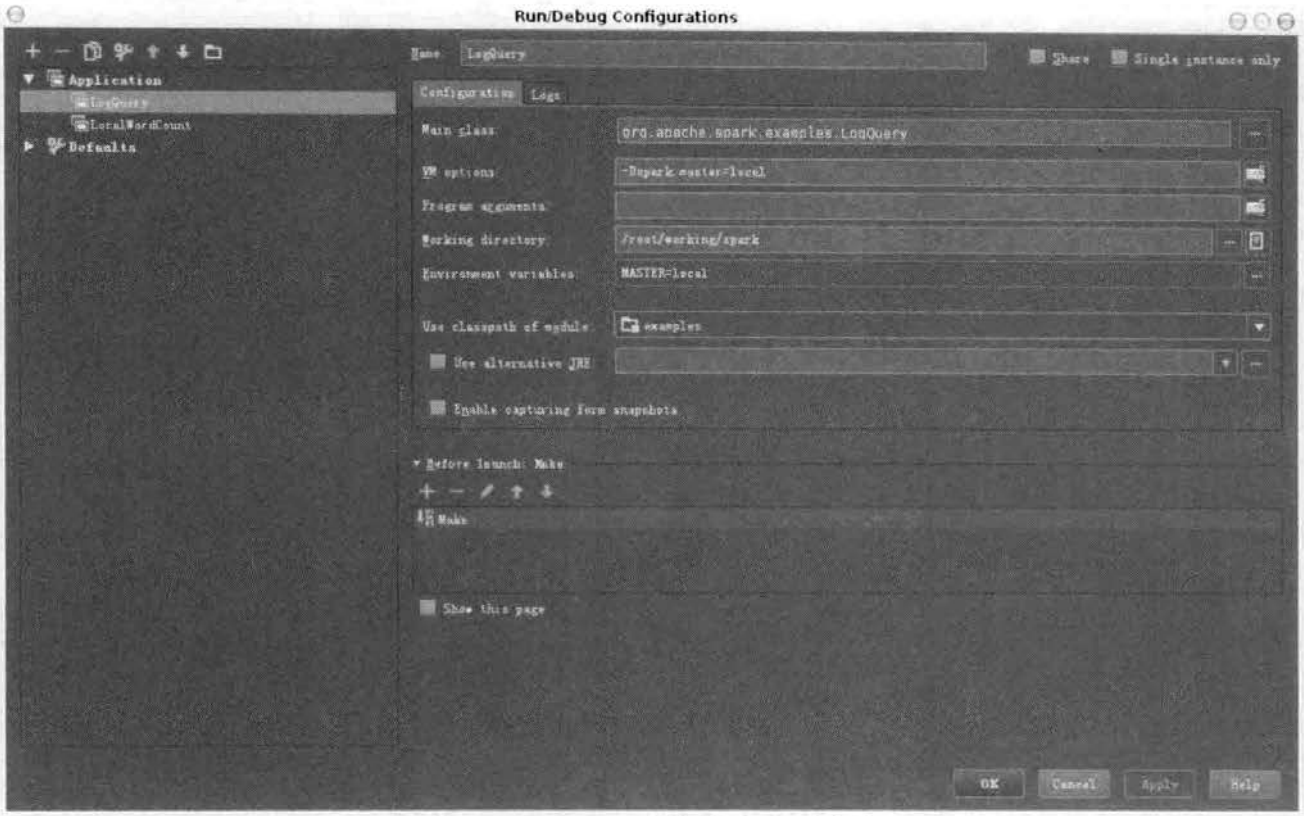


图 A.8 设置环境变量和启动参数

-Dspark.master=local 指定Spark的运行模式，可根据需要做适当修改。

步骤3: 至此，在Run菜单中可以发现有“Run LogQuery”一项存在，尝试运行，保证编译成功。

步骤4: 断点设置。在源文件的左侧双击即可打上断点标记，然后单击 Run→Debug LogQuery，大功告成，如图 A.9所示，可以查看变量和调用堆栈了。

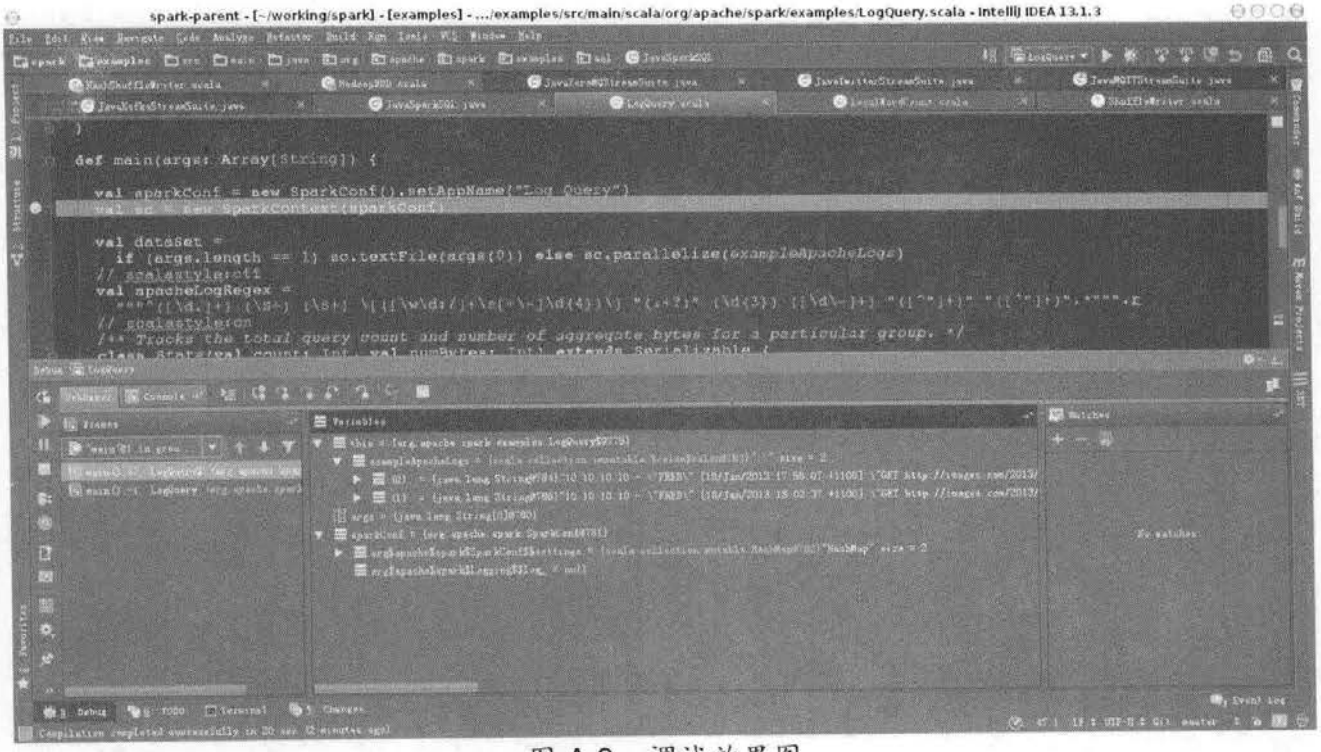

spark-parent - [~/working/spark] - [examples] - .../examples/src/main/scala/org/apache/spark/examples/LogQuery.scala - IntelliJ IDEA 13.1.3
def main(args: Array[String]) {
val sparkConf = new SparkConf().setAppName("Log Query")
val sc = new SparkContext(sparkConf)
val dataSet =
if (args.length == 1) sc.textFile(args(0)) else sc.parallelize(exampleApacheLogs)
// scalastyle:off
val apacheLogRegex =
// scalastyle:on
/** Tracks the total query count and number of aggregate bytes for a particular group. */

图 A.9　调试效果图

附录B

源码阅读技巧

B.1 思维模式

源码阅读其实是一个逆向的工程，这期间必然会遇到种种问题。一般来说，我会遵循这样一个思维范式，如图B.1所示。

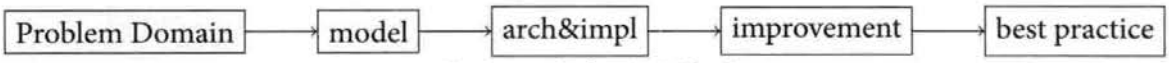


图 B.1 分析思路导图

（1） 首先搞清楚要分析的产品解决的问题是什么，这个问题在哪个大的范畴里，也就是要搞清楚Problem Domain。一个著名的开源产品必定在Wikipedia上有相应的条目，所以一开始去查找Wikipedia是破题的一种极好方式。

（2） 清楚要分析产品的大体框架和关键性的概念，也就是理解清楚 architecture和key concept。如果在这个时候有一些问题搞不定，可以多去相应的用户论坛和stackoverflow网站上提问。另外一个非常值得光顾的站点是slidesshare.com。

（3） 将分析的产品实实在在地运行起来。我一般选择Debian或Arch Linux作为工作平台，

它们提供了丰富的软件包，可以很快地将东西安装并运行。熟悉Linux本身对于开源项目的源码阅读还是大有裨益的。

（4） 修改日志级别，得到丰富的日志信息。有了这个为基础，再来开始真正的源码阅读和分析。

（5） 源码分析的时候，要始终关注以下几个问题：

- 进程以及线程的启动顺序。
- 搞清楚调用关系call flow。
  - 这一部分代码是在同一个进程中吗？在同一个线程中吗？运行在同一台机器中吗？
  - 每一个线程都要问清楚，什么时候启动、什么时候停止。
- 消息传递的路径。针对每一个函数，搞清楚input是谁传给我的、output要传给谁、由哪个来传。
- 搞清楚上述问题之后，就将最开始提到的对architecture的了解做到具体而微了。有了这个基础之后，再继续往下问。
  - 当前实现的性能如何，比如I/O、CPU、Network。这个需要做相应的测试方面的实验。
  - 当前的解决方案还有优化空间吗？比如针对Spark中的scheduling问题，就有sparrow的优化机制提出。

（6） 碰到具体的问题一时解决不了怎么办？

- 用好Google，用好stackoverflow。
- 将碰到的问题模型化，写一些验证性的代码，或者是写一个小的demo来验证。我在解决许多很“妖”的bug时，也是采用类似的思路。
- 找到相应的用户论坛，发帖虚心请教。
- 如果还是不行，就先搁一搁，去看能看懂的地方。

分布式应用还有几个共同的问题，分别是：

- 消息的编解码方式。
- 消息的传递机制，如可以使用ZeroMQ或是RabbitMQ。
- 分布式环境下的同步机制，如基于Paxos的Zookeeper。

B.2 框架为应用而生

本书着重于从源码实现的角度来谈Spark这个项目。其实任何一个项目都有三重属性，分别是业务应用、项目管理和技术实现，如图 B.2所示。

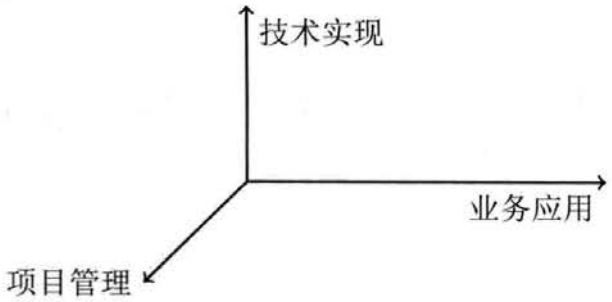


图 B.2 项目的三重属性

而其中业务应用是最为关键的，找到合适的业务，将技术合理地应用于其中，为技术与市场结合找到适当的衔接点。任何技术都需要与市场相联系，只有经历了市场检验的技术才能得到推广和应用。

就Spark而言，其是数据处理中的一个环节，如果要从整体解决方案的角度来考虑的话，还需要掌握数据入口端的相关技术，如Web Server及相关的数据分发队列；另外，数据的存储也是要考虑到的地方，是存储到NoSQL还是直接放入到 HDFS中，这些都是在整体的解决方案中需要做出的选择。

目前Apache网站上的项目代表着互联网行业中非常热门的技术，选择一个项目认真深入地研究下去，一定会有所收获。

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Apache Spark 源码剖析

与Hadoop、Hive、Storm等老牌大数据系统相比，Spark的代码体积要小得多。然而这样一套精简的系统却同时承载了批处理、流处理、迭代计算、关系查询、图计算等多种计算范式，再加上Scala和函数式编程并不为普通程序员所熟悉，阅读和分析Spark源码并不是一件特别轻松的事情。本书记录了一系列分析Spark源码的实用技巧，并给出了一个合理的阅读顺序，相信可以令学习Spark的读者们事半功倍。

Spark Contributor，Databricks工程师 连城

介绍Spark的书籍很多，但一般不够全面，而这本书非常系统、全面地介绍了Spark源码，深入浅出、细致入微，把Spark的由来、Spark整体框架、Spark各软件栈、Spark环境搭建、Spark部署模式等从源码角度一步步剖析得非常清楚。作者有很强的系统设计、软件工程功底，读者不仅可以从书中学到Spark知识，还可以学到作者对新技术研究、源码研究很多好的方法和技巧。授人以鱼不如授人以渔，对在校大学生、Spark初学者、大数据开发工程师来说，这本书非常值得拥有。

华为大数据平台开发部部长 陈亮

难以置信，薄薄的一本书可以兼具如此的广度与深度。除了Spark核心系统，本书还介绍了Streaming、SQL、GraphX、MLLib等扩展库，内容相当全面。但更“赞”的是本书对Spark及各扩展库的运行机理，无不提纲挈领，一一阐明，让读者不但知其然，还能知其所以然。如果想在生产环境中用好Spark，本书值得细读。

网易杭州研究院副院长 汪源

Spark目前正在蓬勃发展，越来越多的公司把大数据计算任务迁移到Spark平台上来。Spark开发的学习曲线并不陡峭，但是处理大数据，需要的不仅是逻辑正确的程序，还需要高性能的程序。如果想把Spark的性能挖掘到极致，那就需要深入了解Spark的设计思想和运行机制，而要了解这些，没有比读源代码更直接的了。许鹏老师的这本书，对于那些没有时间、精力直接“啃”源代码或者对Scala语言还不太精通的读者来说是一个福音。

TalkingData 首席数据科学家 张夏天

上架建议：计算机/Spark

博文视点Broadvview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：付　睿 @Winnie说说
责任编辑：李云静
封面设计：李　玲

ISBN 978-7-121-25420-8
9 787121 254208 >
定价：68.00元