## 1.2　Spark 初体验

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

### 1.2.1　运行 spark-shell

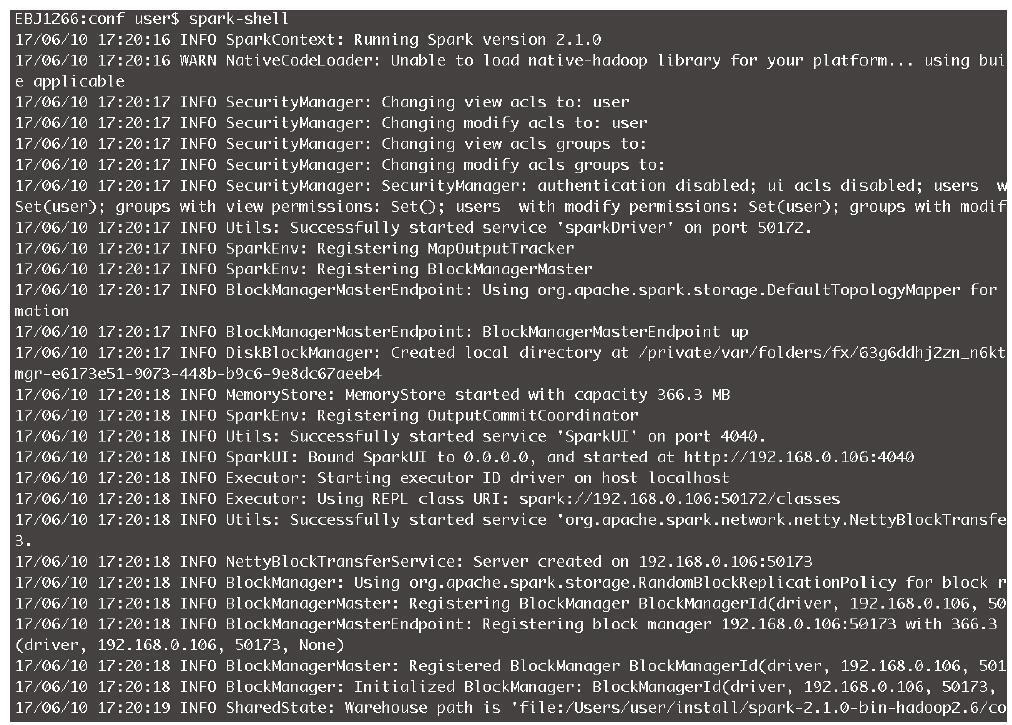
图 1-3 中显示了很多信息，这里进行一些说明。

* 安装完 Spark 2.1.0 后，如果没有明确指定 log4j 的配置，那么 Spark 会使用 core 模块的 org/apache/spark/目录下的 log4j-defaults.properties 作为 log4j 的默认配置。log4j-defaults.properties 指定的 Spark 日志级别为 WARN。用户可以到 Spark 安装目录的 conf 文件夹下，从 log4j.properties.template 复制一份 log4j.properties 文件，并在其中增加自己想要的配置。
* 除了指定 log4j.properties 文件外，还可以在 spark-shell 命令行中通过 sc.setLog-Level（newLevel）语句指定日志级别。
* SparkContext 的 Web UI 的地址是：[**http://192.168.0.106:4040**](http://192.168.0.106:4040/)。192.168.0.106 是作者本地安装 Spark 的机器的 IP 地址，4040 是 SparkContext 的 Web UI 的默认监听端口。
* 指定的部署模式（即 master）为 local[\*]。当前应用（Application）的 ID 为 local-1497084620457。
* 可以在 spark-shell 命令行通过 sc 使用 SparkContext，通过 spark 使用 SparkSession。sc 和 spark 实际分别是 SparkContext 和 SparkSession 在 Spark REPL 中的变量名，具体细节将在 1.2.3 节分析。

由于 Spark Core 的默认日志级别是 WARN，所以看到的信息不是很多。现在我们将 Spark 安装目录的 conf 文件夹下的 log4j.properties.template 通过如下命令复制出一份：

cp log4j.properties.template log4j.properties

并将 log4j.properties 中的 log4j.logger.org.apache.spark.repl.Main=WARN 修改为 log4j.logger.org.apache.spark.repl.Main=INFO，然后我们再次运行 spark-shell，将打印出更丰富的信息，如图 1-4 所示。

图 1-4　Spark 启动过程打印的部分信息

从图 1-4 展示的启动日志中我们可以看到 SecurityManager、SparkEnv、BlockManager-MasterEndpoint、DiskBlockManager、MemoryStore、SparkUI、Executor、NettyBlock-TransferService、BlockManager、BlockManagerMaster 等信息。它们是做什么的？刚刚接触 Spark 的读者只需要知道这些信息即可，具体内容将在后边进行详细的介绍。

### 1.2.2　执行 word count

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

1）输入 val lines=sc.textFile（「../README.md」，2），以 Spark 安装目录下的 README.md 文件内容作为 word count 例子的数据源，执行结果如图 1-5 所示。

图 1-5　步骤 1 执行结果

图 1-5 告诉我们，lines 的实际类型是 MapPartitionsRDD㊟。

2）textFile 方法对文本文件是逐行读取的，我们需要输入 val words=lines.flatMap(line=>line.split（「」）)，将每行文本按照空格分隔以得到每个单词，执行结果如图 1-6 所示。

图 1-6　步骤 2 执行结果

图 1-6 告诉我们，lines 在经过 flatMap 方法的转换后，得到的 words 的实际类型也是 MapPartitionsRDD。

3）对于得到的每个单词，通过输入 val ones=words.map(w=>（w，1）)，将每个单词的计数初始化为 1，执行结果如图 1-7 所示。

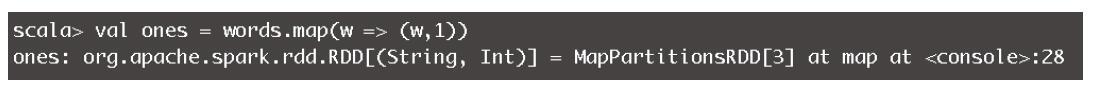
图 1-7　步骤 3 执行结果

图 1-7 告诉我们，words 在经过 map 方法的转换后，得到的 ones 的实际类型也是 MapPartitionsRDD。

4）输入 val counts=ones.reduceByKey（\_+\_），对单词进行计数值的聚合，执行结果如图 1-8 所示。

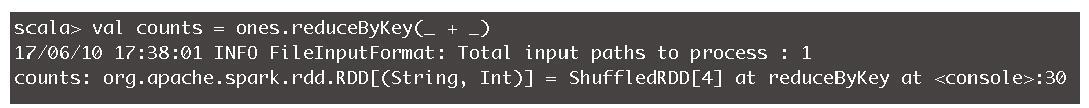
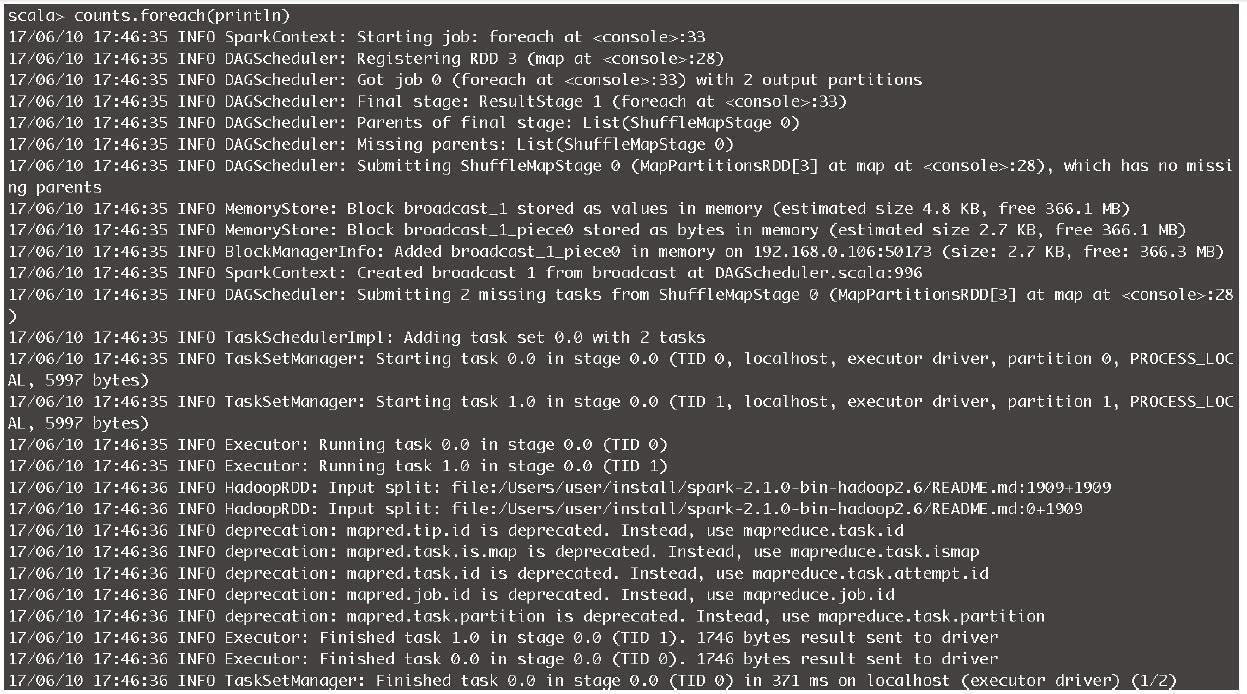
图 1-8　步骤 4 执行结果

图 1-8 告诉我们，ones 在经过 reduceByKey 方法的转换后，得到的 counts 的实际类型是 ShuffledRDD。

5）输入 counts.foreach（println），将每个单词的计数值打印出来，作业的执行过程如图 1-9 和图 1-10 所示。作业的输出结果如图 1-11 所示。

图 1-9　步骤 5 执行过程第一部分

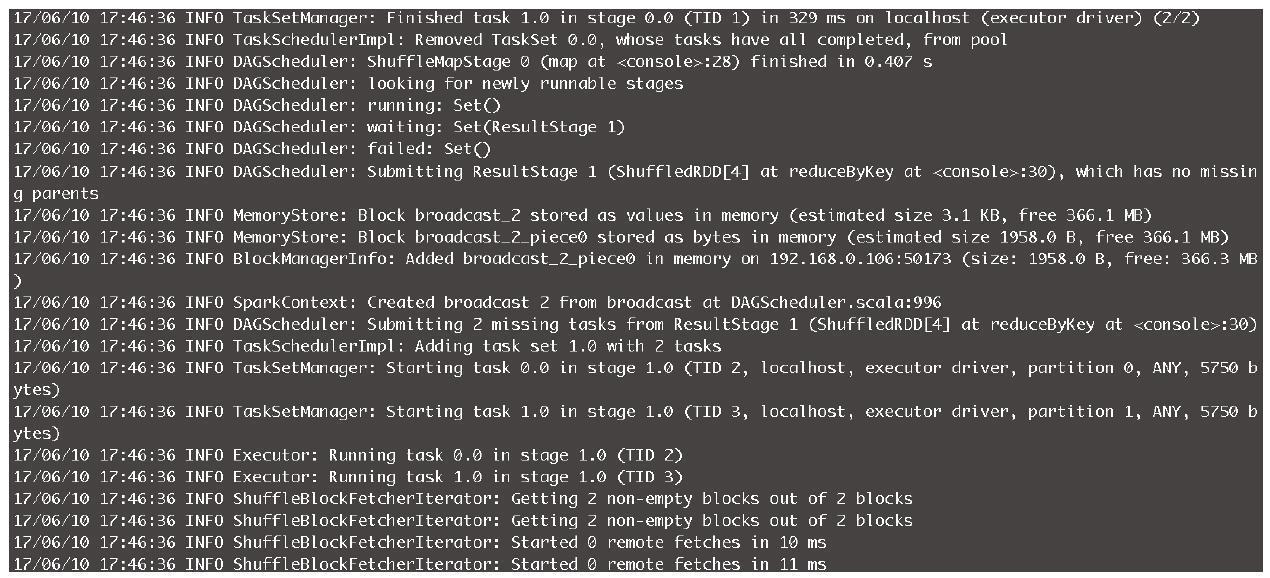
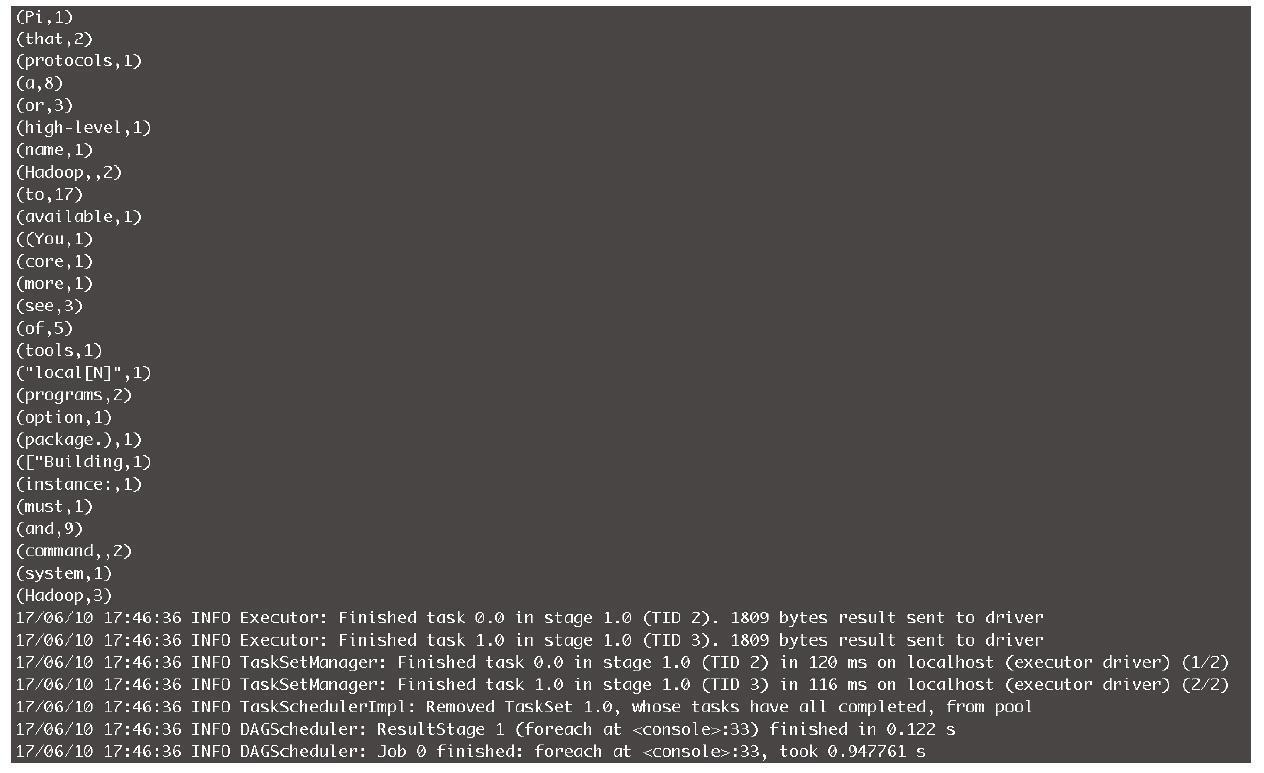
图 1-10　步骤 5 执行过程第二部分

图 1-9 和图 1-10 展示了很多作业提交、执行的信息，这里挑选关键的内容进行介绍。

图 1-11　步骤 5 输出结果

* SparkContext 为提交的 Job 生成的 ID 是 0。
* word count 例子一共产生了 4 个 RDD㊟，被划分为 ResultStage 和 ShuffleMapStage。ShuffleMapStage 的 ID 为 0，尝试号为 0。ResultStage 的 ID 为 1，尝试号也为 0。在 Spark 中，如果 Stage 没有执行完成，就会进行多次重试。Stage 无论是首次执行还是重试，都被视为是一次 Stage 尝试（stage attempt），每次尝试都有一个唯一的尝试号（attempt number）。
* 由于 Job 有两个分区，所以 ShuffleMapStage 和 ResultStage 都有两个 Task 被提交。每个 Task 也会有多次尝试，因而也有属于 Task 的尝试号。从图中看出，Shuffle-MapStage 中的两个 Task 和 ResultStage 中的两个 Task 的尝试号也都是 0。
* HadoopRDD 则用于读取文件内容。

图 1-11 展示了单词计数的输出结果和最后打印的任务结束的日志信息。

笔者在本节介绍的 word count 例子是以 SparkContext 的 API 来实现的，读者也可以选择在 spark-shell 中通过运用 SparkSession 的 API 来实现。本书在第 10 章将介绍 Spark 源码自带的用 SparkSession 的 API 来实现的 word count 的 Java 应用程序。

### 1.2.3　剖析 spark-shell

通过在 spark-shell 中执行 word count 的过程，让读者了解到可以使用 spark-shell 提交 Spark 作业。现在读者应该很想知道 spark-shell 究竟做了什么呢？

**1.脚本分析**

在 Spark 安装目录的 bin 文件夹下可以找到 spark-shell，其中有代码清单 1-1 所示的一段脚本。

代码清单 1-1　spark-shell 脚本

function main() {  
 if $cygwin; then  
 stty -icanon min 1 -echo > /dev/null 2>&1  
 export SPARK\_SUBMIT\_OPTS="$SPARK\_SUBMIT\_OPTS -Djline.terminal=unix"  
 "${SPARK\_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name "Spark shell" "$@"  
 stty icanon echo > /dev/null 2>&1  
 else  
 export SPARK\_SUBMIT\_OPTS  
 "${SPARK\_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name "Spark shell" "$@"  
 fi  
}

我们看到脚本 spark-shell 里执行了 spark-submit 脚本，那么打开 spark-submit 脚本，发现代码清单 1-2 中所示的脚本。

代码清单 1-2　spark-submit 脚本

if [ -z "${SPARK\_HOME}" ]; then  
 source "$(dirname "$0")"/find-spark-home  
fi  
# disable randomized hash for string in Python 3.3+  
export PYTHONHASHSEED=0  
exec "${SPARK\_HOME}"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"

可以看到 spark-submit 中又执行了脚本 spark-class。打开脚本 spark-class，首先发现以下一段脚本：

# 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

上面的脚本是为了找到 Java 命令。在 spark-class 脚本中还会找到以下内容：

build\_command() {  
 "$RUNNER" -Xmx128m -cp "$LAUNCH\_CLASSPATH" org.apache.spark.launcher.Main "$@"  
 printf "%d\0" $?  
}  
  
CMD=()  
while IFS= read -d '' -r ARG; do  
 CMD+=("$ARG")  
done < <(build\_command "$@")

根据代码清单 1-2，脚本 spark-submit 在执行 spark-class 脚本时，给它增加了参数 SparkSubmit。所以读到这里，应该知道 Spark 启动了以 SparkSubmit 为主类的 JVM 进程。

**2.远程监控**

为便于在本地对 Spark 进程进行远程监控，在 spark-shell 脚本中找到以下配置：

SPARK\_SUBMIT\_OPTS="$SPARK\_SUBMIT\_OPTS -Dscala.usejavacp=true"

并追加以下 jmx 配置：

-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=10207  
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote. ssl=false

如果 Spark 安装在其他机器，那么在本地打开 Java VisualVM 后需要添加远程主机，如图 1-12 所示。

图 1-12　添加远程主机

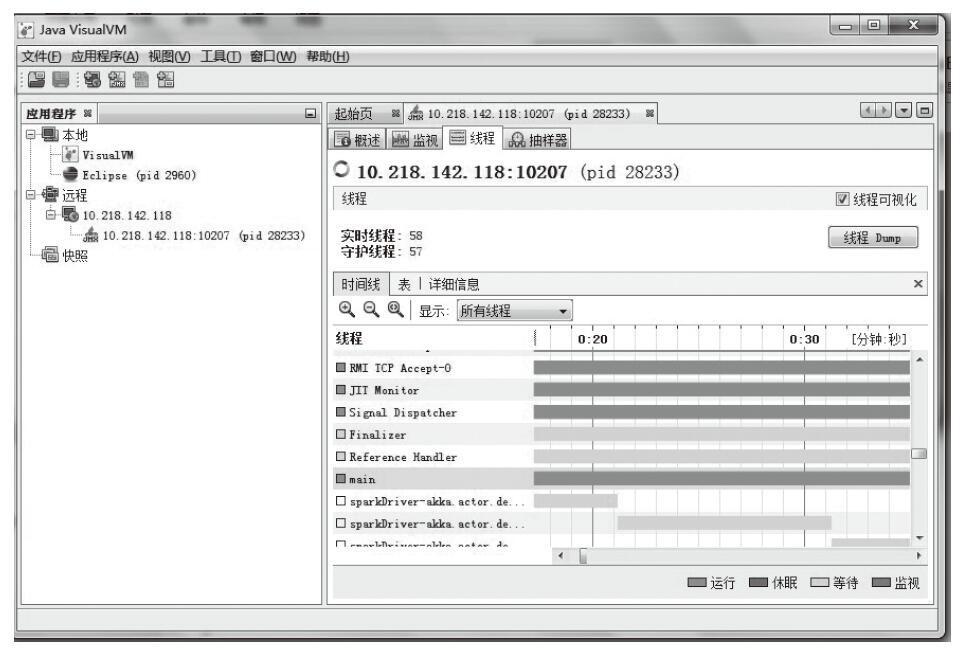
图 1-13　添加 JMX 连接

右击已添加的远程主机，添加 JMX 连接，如图 1-13 所示。

如果 Spark 安装在本地，那么打开 Java VisualVM 后就会在应用程序窗口中看到 org.apache.spark.deploy.SparkSubmit 进程，只需双击即可。

选择右侧的「线程」选项卡，选择 main 线程，然后点击「线程 Dump」按钮，如图 1-14 所示。

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

图 1-14　查看 Spark 线程

代码清单 1-3　main 线程的 Dump 信息

"main" #1 prio=5 os\_prio=31 tid=0x00007fa012802000 nid=0x1303 runnable [0x0000000-10d11c000]  
 java.lang.Thread.State: RUNNABLE  
 at java.io.FileInputStream.read0(Native Method)  
 at java.io.FileInputStream.read(FileInputStream.java:207)  
 at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:169)  
 - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)  
 at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:137)  
 at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:246)  
 at jline.internal.InputStreamReader.read(InputStreamReader.java:261)  
 - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)  
 at jline.internal.InputStreamReader.read(InputStreamReader.java:198)  
 - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)  
 at jline.console.ConsoleReader.readCharacter(ConsoleReader.java:2145)  
 at jline.console.ConsoleReader.readLine(ConsoleReader.java:2349)  
 at jline.console.ConsoleReader.readLine(ConsoleReader.java:2269)  
 at scala.tools.nsc.interpreter.jline.InteractiveReader.readOneLine(JLineReader.scala:57)  
 at scala.tools.nsc.interpreter.InteractiveReader$$anonfun$readLine$2.apply(InteractiveReader.scala:37)  
 at scala.tools.nsc.interpreter.InteractiveReader$$anonfun$readLine$2.apply(InteractiveReader.scala:37)  
 at scala.tools.nsc.interpreter.InteractiveReader$.restartSysCalls(InteractiveReader.scala:44)  
 at scala.tools.nsc.interpreter.InteractiveReader$class.readLine(Interactive-Reader.scala:37)  
 at scala.tools.nsc.interpreter.jline.InteractiveReader.readLine(JLineReader.scala:28)  
 at scala.tools.nsc.interpreter.ILoop.readOneLine(ILoop.scala:404)  
 at scala.tools.nsc.interpreter.ILoop.loop(ILoop.scala:413)  
 at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply$mcZ$sp(ILoop.scala:923)  
 at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:909)  
 at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:909)  
 at scala.reflect.internal.util.ScalaClassLoader$.savingContextLoader(ScalaClassLoader.scala:97)  
 at scala.tools.nsc.interpreter.ILoop.process(ILoop.scala:909)  
 at org.apache.spark.repl.Main$.doMain(Main.scala:68)  
 at org.apache.spark.repl.Main$.main(Main.scala:51)  
 at org.apache.spark.repl.Main.main(Main.scala)  
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)  
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)  
 at java.lang.reflect.Method.invoke(Method.java:498)  
 at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:738)  
 at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)  
 at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)  
 at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)  
 at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)

从 main 线程的栈信息中可以看出程序的调用顺序：SparkSubmit.main→repl.Main→Iloop.process。

**3.源码分析**

我们根据上面的线索，直接阅读 Iloop 的 process 方法的源码㊟，如代码清单 1-4 所示。

代码清单 1-4　process 的实现

def process(settings: Settings): Boolean = savingContextLoader {  
 this.settings = settings  
 createInterpreter()  
 // sets in to some kind of reader depending on environmental cues  
 in = in0.fold(chooseReader(settings))(r => SimpleReader(r, out, interactive = true))  
 globalFuture = future {  
 intp.initializeSynchronous()  
 loopPostInit()  
 !intp.reporter.hasErrors  
 }  
 loadFiles(settings)  
 printWelcome()  
 try loop() match {  
 case LineResults.EOF => out print Properties.shellInterruptedString  
 case \_ =>  
 }  
 catch AbstractOrMissingHandler()  
 finally closeInterpreter()  
 true  
}

根据代码清单 1-4，Iloop 的 process 方法调用了 loadFiles 方法。Spark 中的 SparkILoop 继承了 Iloop 并重写了 loadFiles 方法，其实现如下：

override def loadFiles(settings: Settings): Unit = {  
 initializeSpark()  
 super.loadFiles(settings)  
}

根据上面展示的代码，loadFiles 方法调用了 SparkILoop 的 initializeSpark 方法，initialize Spark 的实现如代码清单 1-5 所示。

代码清单 1-5　initializeSpark 的实现

def initializeSpark() {  
 intp.beQuietDuring {  
 processLine("""  
 @transient val spark = if (org.apache.spark.repl.Main.sparkSession != null) {  
 org.apache.spark.repl.Main.sparkSession  
 } else {  
 org.apache.spark.repl.Main.createSparkSession()  
 }  
 @transient val sc = {  
 val \_sc = spark.sparkContext  
 if (\_sc.getConf.getBoolean("spark.ui.reverseProxy", false)) {  
 val proxyUrl = \_sc.getConf.get("spark.ui.reverseProxyUrl", null)  
 if (proxyUrl != null) {  
 println(s"Spark Context Web UI is available at ${proxyUrl}/proxy/${\_sc.applicationId}")  
 } else {  
 println(s"Spark Context Web UI is available at Spark Master Public URL")  
 }  
 } else {  
 \_sc.uiWebUrl.foreach {  
 webUrl => println(s"Spark context Web UI available at ${webUrl}")  
 }  
 }  
 println("Spark context available as 'sc' " +  
 s"(master = ${\_sc.master}, app id = ${\_sc.applicationId}).")  
 println("Spark session available as 'spark'.")  
 \_sc  
 }  
 """)  
 processLine("import org.apache.spark.SparkContext.\_")  
 processLine("import spark.implicits.\_")  
 processLine("import spark.sql")  
 processLine("import org.apache.spark.sql.functions.\_")  
 replayCommandStack = Nil // remove above commands from session history.  
 }  
 }

我们看到，initializeSpark 向交互式 shell 发送了一大串代码，Scala 的交互式 shell 将调用 org.apache.spark.repl.Main 的 createSparkSession 方法（见代码清单 1-6），创建 Spark-Session。我们看到常量 spark 将持有 SparkSession 的引用，并且 sc 持有 SparkSession 内部初始化好的 SparkContext。所以我们才能够在 spark-shell 的交互式 shell 中使用 sc 和 spark。

代码清单 1-6　createSparkSession 的实现

def createSparkSession(): SparkSession = {  
 val execUri = System.getenv("SPARK\_EXECUTOR\_URI")  
 conf.setIfMissing("spark.app.name", "Spark shell")  
 conf.set("spark.repl.class.outputDir", outputDir.getAbsolutePath())  
 if (execUri != null) {  
 conf.set("spark.executor.uri", execUri)  
 }  
 if (System.getenv("SPARK\_HOME") != null) {  
 conf.setSparkHome(System.getenv("SPARK\_HOME"))  
 }  
  
 val builder = SparkSession.builder.config(conf)  
 if (conf.get(CATALOG\_IMPLEMENTATION.key, "hive").toLowerCase == "hive") {  
 if (SparkSession.hiveClassesArePresent) {  
 sparkSession = builder.enableHiveSupport().getOrCreate()  
 logInfo("Created Spark session with Hive support")  
 } else {  
 builder.config(CATALOG\_IMPLEMENTATION.key, "in-memory")  
 sparkSession = builder.getOrCreate()  
 logInfo("Created Spark session")  
 }  
 } else {  
 sparkSession = builder.getOrCreate()  
 logInfo("Created Spark session")  
 }  
 sparkContext = sparkSession.sparkContext  
 sparkSession  
}

根据代码清单 1-6 所示，createSparkSession 方法通过 SparkSession 的 API 创建 Spark-Session 实例。本书将有关 SparkSession 等 API 的内容放在第 10 章进行讲解，初次接触 Spark 的读者现在只需要了解即可。

## 1.3　阅读环境准备

准备 Spark 阅读环境，同样需要一台好机器。笔者调试源码的机器内存是 8GB。源码阅读的前提是首先在 IDE 环境中打包、编译通过。常用的 IDE 有 IntelliJ IDEA 和 Eclipse，笔者选择用 Eclipse 编译和阅读 Spark 源码，原因有两个：一是由于使用多年，对它比较熟悉；二是社区中使用 Eclipse 编译 Spark 的资料太少，在这里可以做个补充。笔者在 Mac OS 系统中编译 Spark 源码，除了安装 JDK 和 Scala 外，还需要安装以下工具。

**1.3.1　安装 SBT**

由于 Scala 使用 SBT 作为构建工具，所以需要下载 SBT。下载地址：[**http://www.scala-sbt.org/**](http://www.scala-sbt.org/)，下载安装包并安装，本书以 sbt-0.13.12.tgz 为例。

1）移动到选好的安装目录，例如：

mv sbt-0.13.12.tgz ~/install/

2）进入安装目录，执行以下命令：

chmod 755 sbt-0.13.12.tgz  
tar -xzvf sbt-0.13.12.tgz

3）配置环境：

cd ~  
vim .bash\_profile

4）添加如下配置：

export SBT\_HOME=$HOME/install/sbt  
export PATH=$SBT\_HOME/bin:$PATH

输入以下命令，使环境变量快速生效：

source .bash\_profile

安装完毕后，使用 sbt about 命令查看，确认安装正常，如图 1-15 所示。

图 1-15　查看 SBT 安装是否正常

**1.3.2　安装 Git**

由于 Spark 源码使用 Git 作为版本控制工具，所以需要下载 Git 的客户端工具。下载地址：[**https://git-scm.com**](https://git-scm.com/)，下载最新的版本并安装。

安装完毕后可使用 git--version 命令来查看安装是否正常，如图 1-16 所示。

图 1-16　查看 Git 是否安装成功

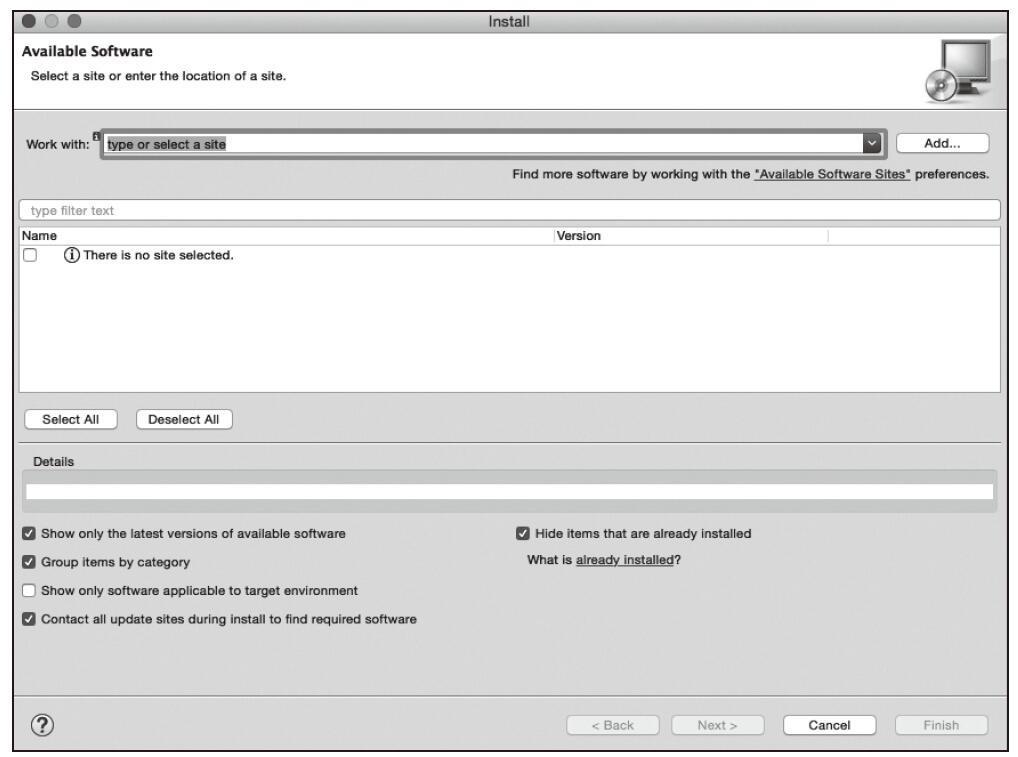
**1.3.3　安装 Eclipse Scala IDE 插件**

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

由于笔者本地的 Eclipse 版本是 Eclipse Mars.2Release（4.5.2），所以选择安装插件 [**http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site**](http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site)，如图 1-17 所示。

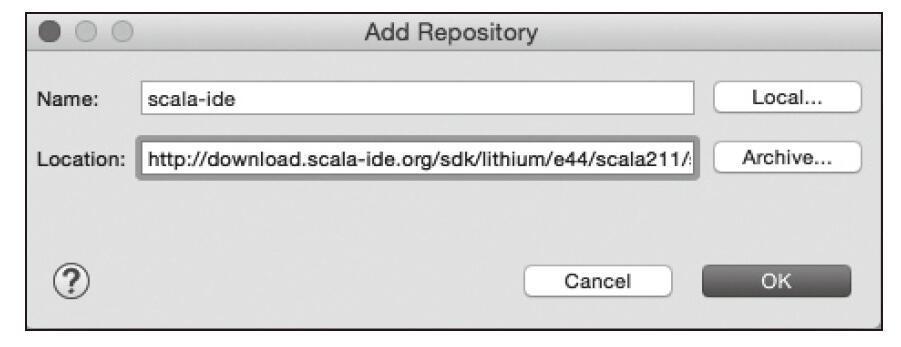
图 1-17　Eclipse Scala IDE 插件安装地址

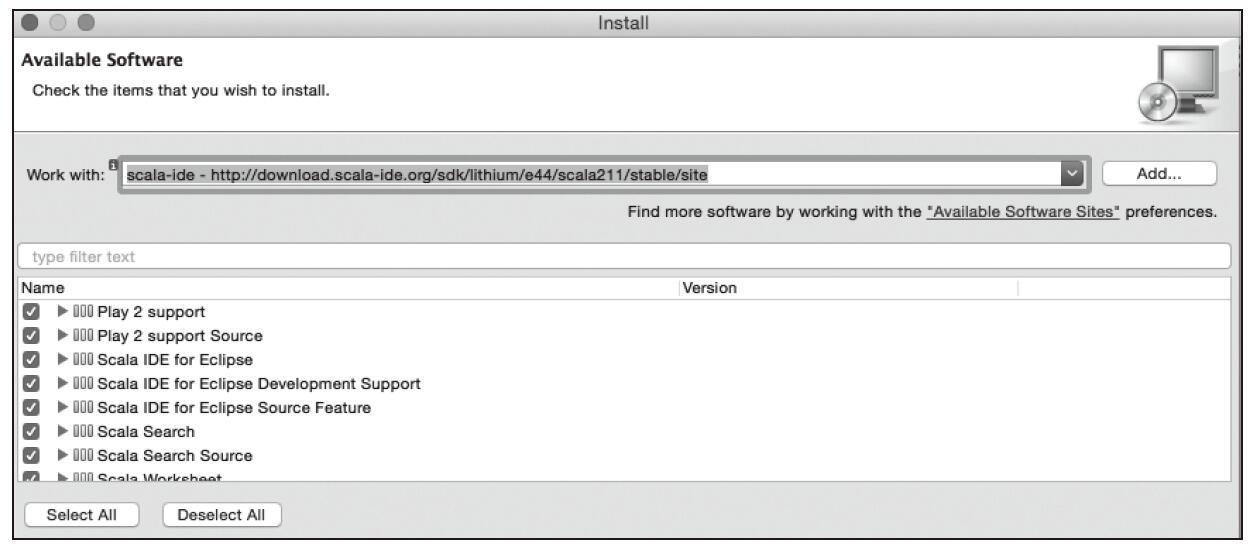
在 Eclipse 中选择 Help 菜单，然后选择 Install New Software 选项，打开 Install 对话框，如图 1-18 所示。

图 1-18　打开 Install 对话框

点击 Add 按钮，打开 Add Repository 对话框，输入插件地址，如图 1-19 所示。

全选插件的内容，完成安装，如图 1-20 所示。

图 1-19　添加 Scala IDE 插件地址

图 1-20　完成安装 Scala IDE 插件

## 1.4　Spark 源码编译与调试

**1.下载 Spark 源码**

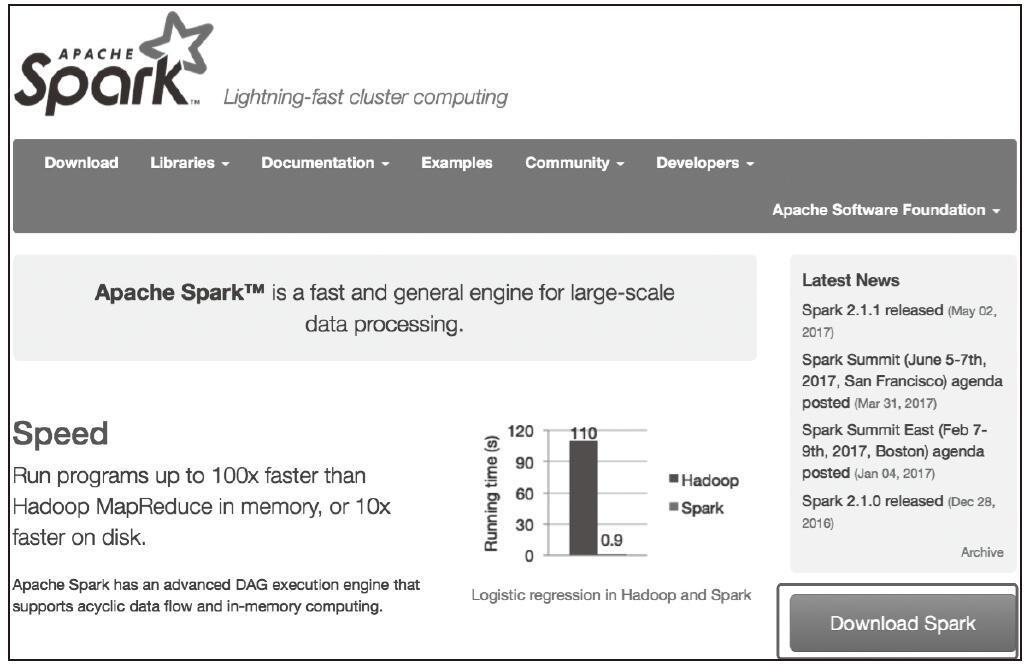
首先，访问 Spark 官网 [**http://spark.apache.org/**](http://spark.apache.org/)，如图 1-21 所示。

点击 Download Spark 按钮，在下一个页面找到 Git 地址，如图 1-22 所示。

笔者在当前用户目录下创建 Source 文件夹作为放置 Spark 源码的地方，进入此文件夹并输入 git clone [**git://github.com/apache/spark.git**](git://github.com/apache/spark.git) 命令将源码下载到本地，如图 1-23 所示。

**2.构建 Scala 应用**

进入 Spark 根目录，执行 sbt 命令。然后会下载和解析很多 Jar 包，要等待一段时间，笔者大概花费了一个多小时才执行完，如图 1-24 所示。

图 1-21　Spark 官网

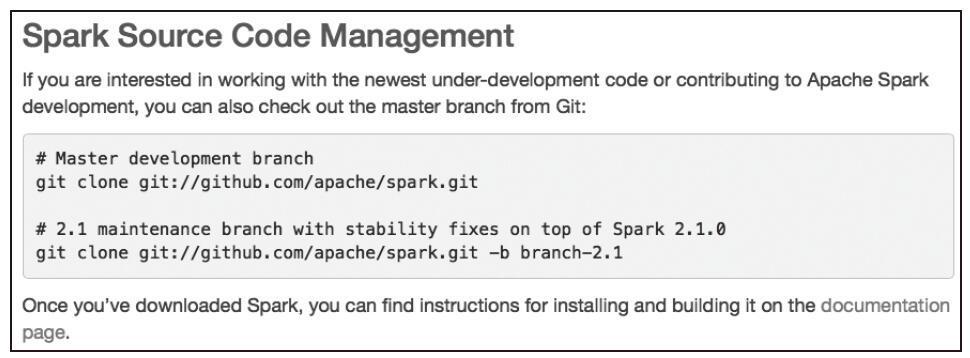
图 1-22　Spark 官方 Git 地址

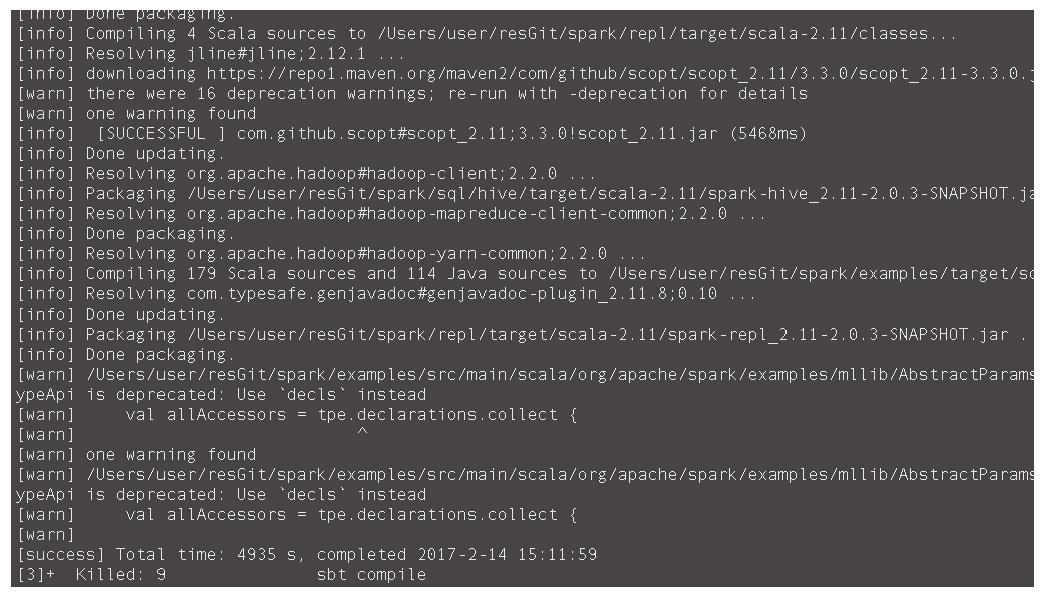
图 1-23　下载 Spark 源码

图 1-24　构建 Scala 应用

从图 1-24 中可以看出，sbt 构建完毕时会出现提示符「>」。

**3.使用 sbt 生成 Eclipse 工程文件**

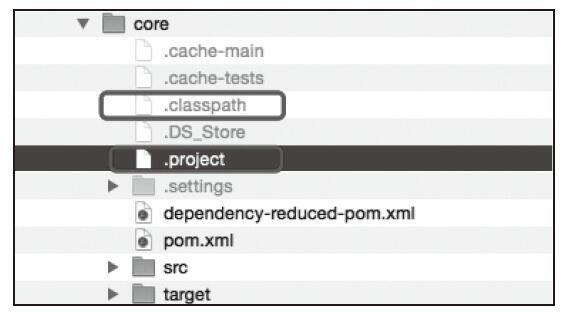
在 sbt 命令出现提示符「>」后，输入 eclipse 命令，开始生成 Eclipse 工程文件，也需要花费很长的时间，笔者大致花费了 40 分钟。完成时的状况如图 1-25 所示。

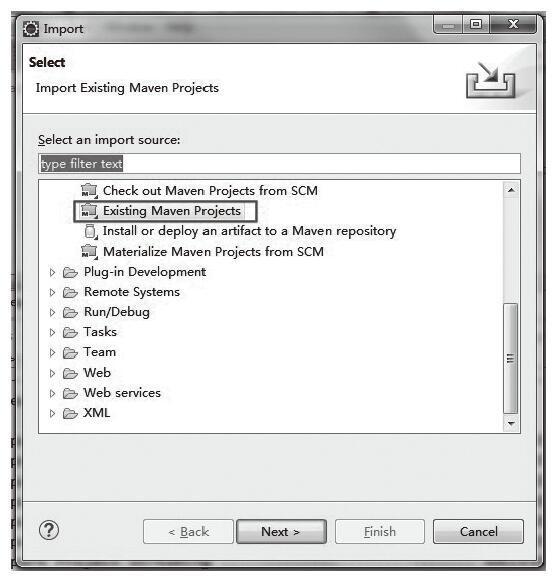
图 1-25　sbt 编译过程

现在我们查看 Spark 下的子文件夹，发现其中都生成了.project 和.classpath 文件。比如 mllib 项目下就生成了.project 和.classpath 文件，如图 1-26 所示。

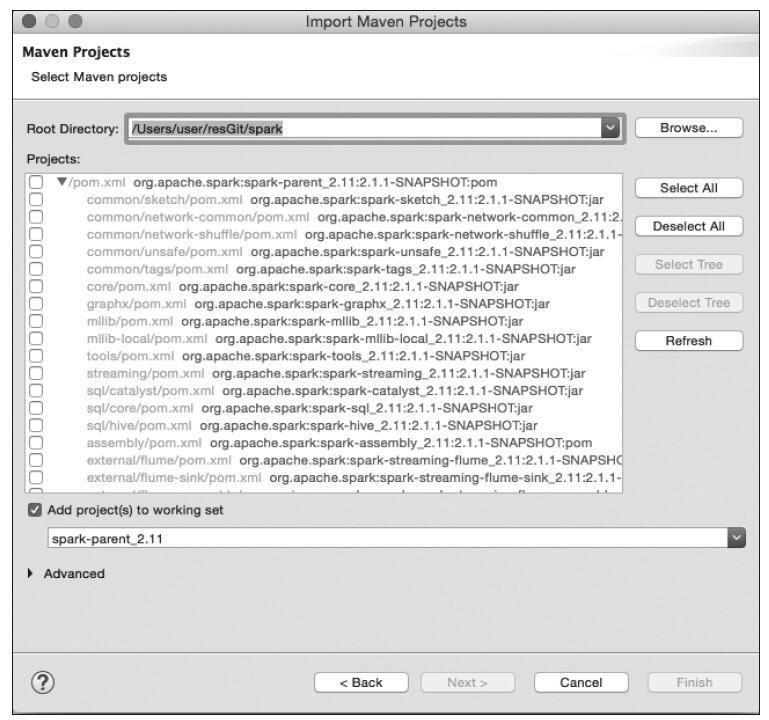
**4.编译 Spark 源码**

由于 Spark 使用 Maven 作为项目管理工具，所以需要将 Spark 项目作为 Maven 项目导入到 Eclipse 中，如图 1-27 所示。

图 1-26　sbt 生成的项目文件

图 1-27　导入 Maven 项目

点击 Next 按钮进入下一个对话框，如图 1-28 所示。

图 1-28　选择 Maven 项目

全选所有项目，点击 finish 按钮，这样就完成了导入，如图 1-29 所示。

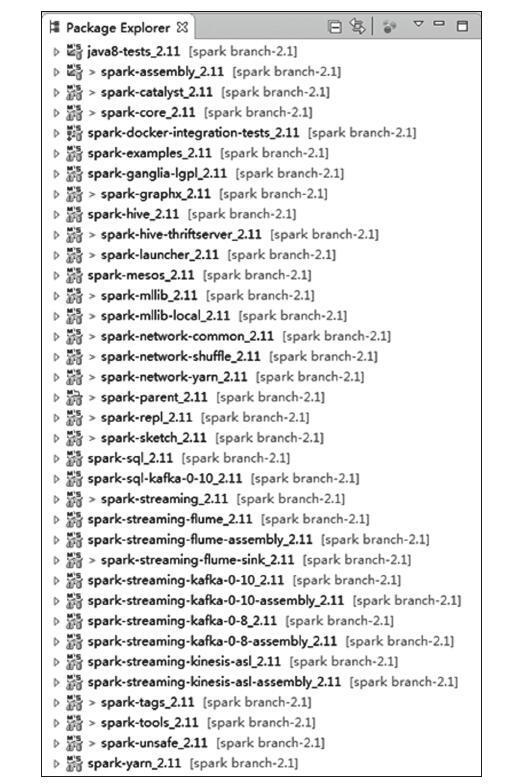
导入完成后，需要设置每个子项目的 Build Path。右击每个项目，选择 Build Path→Configure Build Path 命令，打开 Java Build Path 对话框，如图 1-30 所示。

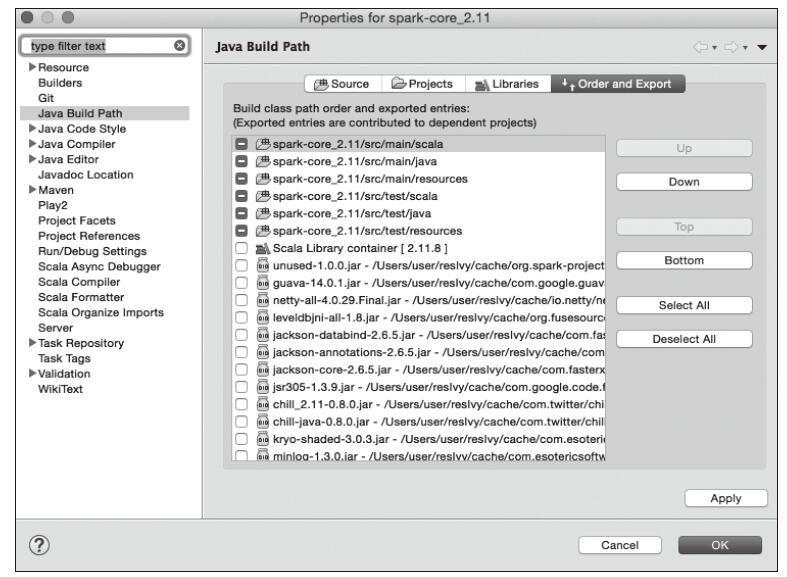
Eclipse 在对项目编译时，可能会出现很多错误，只要仔细分析报错原因就能一一排除。所有错误解决后运行 mvn clean install，如图 1-31 所示。

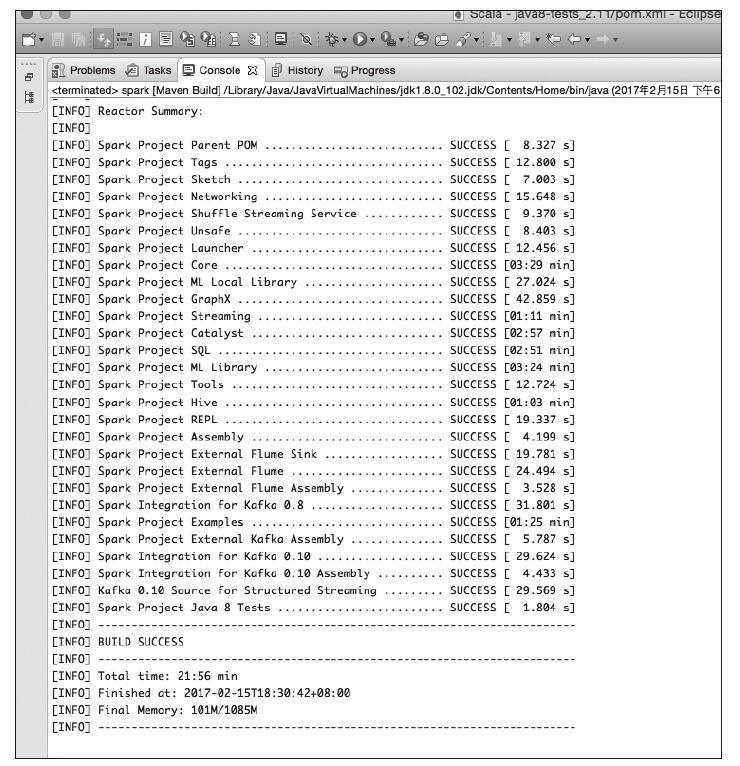
**5.调试 Spark 源码**

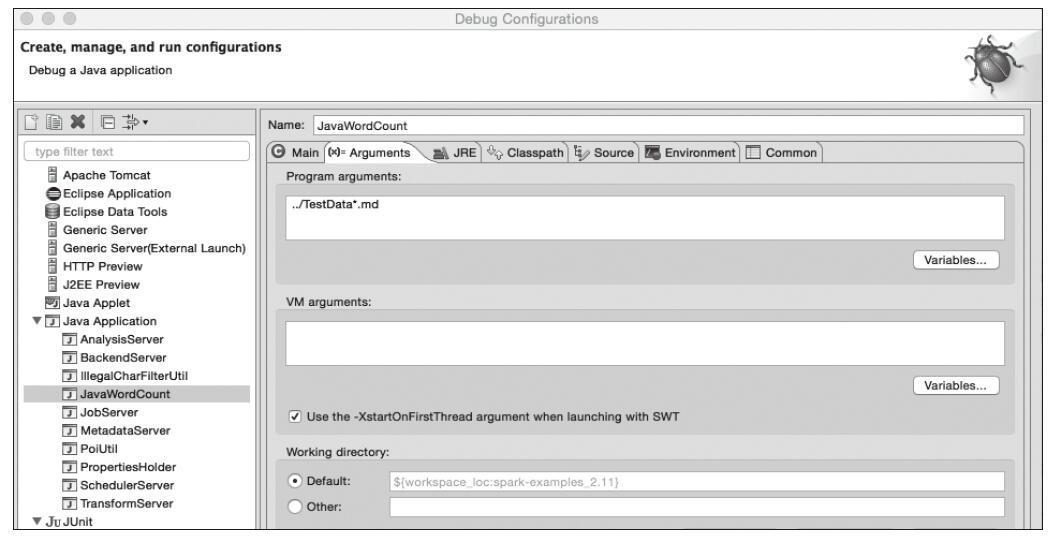
以 Spark 源码自带的 JavaWordCount 为例，介绍如何调试 Spark 源码。右击 JavaWordCount.java，选择 Debug As→Java Application 命令即可。如果想修改配置参数，右击 JavaWordCount.java，选择 Debug As→Debug Configurations 命令，从打开的对话框中选择 JavaWordCount，在右侧区域可以修改 Java 执行参数、JRE、classpath、环境变量等配置，如图 1-32 所示。

读者也可以在 Spark 源码中设置断点，进行跟踪调试。

图 1-29　导入完成的项目

图 1-30　Java 构建路径

图 1-31　编译成功

图 1-32　源码调试

## 1.5　小结

本章通过引导大家在 Linux 操作系统下搭建基本的执行环境，并且介绍 spark-shell 等脚本的执行，帮助读者由浅入深地进行 Spark 源码的学习。由于笔者的多数开发工作都在 Mac 系统下，所以本书介绍的搭建源码阅读环境是在 Mac 环境下。Eclipse 有广大的用户群，即便是一些开始使用 IntelliJ 的用户，对 Eclipse 也不陌生，所以本章选择 Eclipse 作为导入、编译、阅读、调试 Spark 源码的工具，希望能降低读者的学习门槛

# 第 2 章　设计理念与基本架构

「若夫乘天地之正，而御六气之辩，以游无穷者，彼且恶乎待哉？」

——《庄子·逍遥游》

### 本章导读

随着近十年互联网的迅猛发展，越来越多的人融入了互联网——利用搜索引擎查询词条或问题；社交圈子从现实搬到了 Facebook、Twitter、微信等社交平台上；女孩子们现在少了逛街，多了在各大电商平台上的购买；喜欢棋牌的人能够在对战平台上找到世界各地的玩家对弈。在国内随着网民数量的持续增加，互联网公司的数据在体量、产生速度、多样性等方面呈现出巨大的变化。

互联网产生的数据相比传统软件产生的数据而言，有着数据挖掘的巨大潜力。通过对数据的挖掘，可以统计出 PV、UV，计算出不同设备与注册率、促销与下单率之间的关系，甚至构建热点分析、人群画像等算法模型，产生一系列报表、图形、离线统计、实时计算的产品。互联网公司如果能有效利用这些数据，将对决策和战略发展起到至关重要的作用。

在大数据的大势之下，Hadoop、Spark、Flink、Storm、Dremel、Impala、Tez 等一系列大数据技术如雨后春笋般不断涌现。工程师们正在使用这些工具摸索前行。

上一章介绍了 Spark 运行环境和源码阅读环境的搭建，为读者学习 Spark 做好准备。本章首先从 Spark 产生的背景开始，介绍 Spark 的主要特点、基本概念、版本变迁，然后简要说明 Spark 的主要模块和编程模型。最后从 Spark 的设计理念和基本架构入手，使读者能够对 Spark 有一个宏观的认识，为之后的内容打好基础。

## 2.1　初识 Spark

Spark 是一个通用的并行计算框架，由加州伯克利大学（UC Berkeley）的 AMP 实验室开发于 2009 年，并于 2010 年开源，2013 年成长为 Apache 旗下在大数据领域最活跃的开源项目之一。

Spark 目前已经走过了 0.x 和 1.x 两个时代，现在正在 2.x 时代稳步发展。Spark 从 2012 年 10 月 15 日发布 Spark 0.6 到 2016 年 1 月 4 日发布 Spark 1.6 只经过了三年时间，那时候差不多每个月都会有新的版本发布，平均每个季度会发布一个新的二级版本。

自从 2016 年 7 月发布了 Spark 2.0.0 版本以来，只在当年 12 月发布了 Spark 2.1.0 版本，到目前为止还没有新的二级版本发布。Spark 发布新版本的节奏明显慢了下来，当然这也跟 Spark 团队过于激进的决策（比如很多 API 不能向前兼容）有关。

Spark 也是基于 map reduce 算法模型实现的分布式计算框架，拥有 Hadoop MapReduce 所具有的优点，并且解决了 Hadoop MapReduce 中的诸多缺陷。

**2.1.1　Hadoop MRv1 的局限**

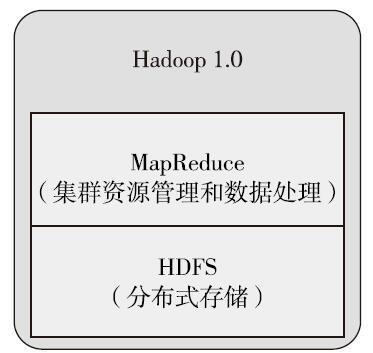
早在 Hadoop 1.0 版本，当时采用的是 MRv1 版本的 MapReduce 编程模型。MRv1 版本的实现都封装在 org.apache.hadoop.mapred 包中，MRv1 的 Map 和 Reduce 是通过接口实现的。MRv1 包括以下三个部分。

* 运行时环境（JobTracker 和 TaskTracker）。
* 编程模型（MapReduce）。
* 数据处理引擎（Map 任务和 Reduce 任务）。

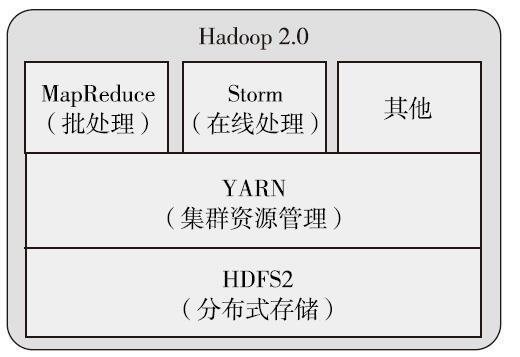
MRv1 存在以下不足。

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

MRv1 的示意如图 2-1 所示。

图 2-1　MRv1 示意图 ㊟

Apache 为了解决以上问题，对 Hadoop 进行升级改造，于是 MRv2 最终诞生了。在 MRv2 中，重用了 MRv1 中的编程模型和数据处理引擎。但是运行时环境被重构了。JobTracker 被拆分成通用的资源调度平台 ResourceManager（简称 RM），节点管理器 NodeManager 和负责各个计算框架的任务调度模型 ApplicationMaster（简称 AM）。ResourceManager 依然负责对整个集群的资源管理，但是在任务资源的调度方面只负责将资源封装为 Container 分配给 ApplicationMaster 的一级调度，二级调度的细节将交给 ApplicationMaster 去完成，这大大减轻了 ResourceManager 的压力，使得 ResourceManager 更加轻量。NodeManager 负责对单个节点的资源管理，并将资源信息、Container 运行状态、健康状况等信息上报给 ResourceManager。ResourceManager 为了保证 Container 的利用率，会监控 Container，如果 Container 未在有限的时间内使用，ResourceManager 将命令 NodeManager「杀死」Container，以便将资源分配给其他任务。MRv2 中 MapReduce 的核心不再是 MapReduce 框架，而是 YARN。在以 YARN 为核心的 MRv2 中，MapReduce 框架是可插拔的，完全可以替换为其他 MapReduce 实现，比如 Spark、Storm 等。MRv2 的示意如图 2-2 所示。

图 2-2　MRv2 示意图

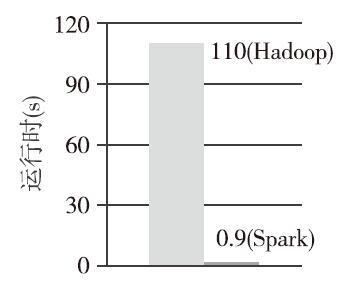
Hadoop MRv2 虽然解决了 MRv1 中的一些问题，但是由于对 HDFS 的频繁操作（包括计算结果持久化、数据备份、资源下载及 Shuffle 等），导致磁盘 I/O 成为系统性能的瓶颈，因此只适用于离线数据处理或批处理，而不能支持对迭代式、流式数据的处理。

**2.1.2　Spark 的特点**

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

1. **减少磁盘 I/O：**随着实时大数据应用越来越多，Hadoop 作为离线的高吞吐、低响应框架已不能满足这类需求。Hadoop MapReduce 的 map 端将中间输出和结果存储在磁盘中，reduce 端又需要从磁盘读写中间结果，势必造成磁盘 I/O 成为瓶颈。Spark 允许将 map 端的中间输出和结果存储在内存中，reduce 端在拉取中间结果时避免了大量的磁盘 I/O。Hadoop YARN 中的 ApplicationMaster 申请到 Container 后，具体任务需要利用 NodeManager 从 HDFS 的不同节点下载任务所需的资源（如 Jar 包），这也增加了磁盘 I/O。Spark 将应用程序上传的资源文件缓冲到 Driver 本地文件服务的内存中，当 Executor 执行任务时直接从 Driver 的内存中读取，也节省了大量的磁盘 I/O。
2. **增加并行度：**由于将中间结果写到磁盘与从磁盘读取中间结果属于不同的环节，Hadoop 将它们简单地通过串行执行衔接起来。Spark 把不同的环节抽象为 Stage，允许多个 Stage 既可以串行执行，又可以并行执行。
3. **避免重新计算：**当 Stage 中某个分区的 Task 执行失败后，会重新对此 Stage 调度，但在重新调度的时候会过滤已经执行成功的分区任务，所以不会造成重复计算和资源浪费。
4. **可选的 Shuffle 排序：**Hadoop MapReduce 在 Shuffle 之前有着固定的排序操作，而 Spark 则可以根据不同场景选择在 map 端排序还是 reduce 端排序。
5. **灵活的内存管理策略：**Spark 将内存分为堆上的存储内存、堆外的存储内存、堆上的执行内存、堆外的执行内存 4 个部分。Spark 既提供了执行内存和存储内存之间固定边界的实现，又提供了执行内存和存储内存之间「软」边界的实现。Spark 默认使用「软」边界的实现，执行内存或存储内存中的任意一方在资源不足时都可以借用另一方的内存，最大限度地提高资源的利用率，减少对资源的浪费。Spark 由于对内存使用的偏好，内存资源的多寡和使用率就显得尤为重要，为此 Spark 的内存管理器提供的 Tungsten 实现了一种与操作系统的内存 Page 非常相似的数据结构，用于直接操作操作系统内存，节省了创建的 Java 对象在堆中占用的内存，使得 Spark 对内存的使用效率更加接近硬件。Spark 会给每个 Task 分配一个配套的任务内存管理器，对 Task 粒度的内存进行管理。Task 的内存可以被多个内部的消费者消费，任务内存管理器对每个消费者进行 Task 内存的分配与管理，因此 Spark 对内存有着更细粒度的管理。

基于以上所列举的优化，Spark 官网声称其性能比 Hadoop 快 100 倍，如图 2-3 所示。即便是内存不足需要磁盘 I/O 时，其速度也是 Hadoop 的 10 倍以上。

图 2-3　Hadoop 与 Spark 执行逻辑回归时间比较

Spark 还有其他一些特点。

1. **检查点支持：**Spark 的 RDD 之间维护了血缘关系（lineage），一旦某个 RDD 失败了，则可以由父 RDD 重建。虽然 lineage 可用于错误后 RDD 的恢复，但对于很长的 lineage 来说，恢复过程非常耗时。如果应用启用了检查点，那么在 Stage 中的 Task 都执行成功后，SparkContext 将把 RDD 计算的结果保存到检查点，这样当某个 RDD 执行失败后，再由父 RDD 重建时就不需要重新计算，而直接从检查点恢复数据。
2. **易于使用。**Spark 现在支持 Java、Scala、Python 和 R 等语言编写应用程序，大大降低了使用者的门槛。除此之外，还自带了 80 多个高等级操作符，允许在 Scala、Python、R 的 shell 中进行交互式查询。
3. **支持交互式：**Spark 使用 Scala 开发，并借助于 Scala 类库中的 Iloop 实现交互式 shell，提供对 REPL（Read-eval-print-loop）的实现。
4. **支持 SQL 查询。**在数据查询方面，Spark 支持 SQL 及 Hive SQL，这极大地方便了传统 SQL 开发和数据仓库的使用者。
5. **支持流式计算：**与 MapReduce 只能处理离线数据相比，Spark 还支持实时的流计算。Spark 依赖 Spark Streaming 对数据进行实时的处理，其流式处理能力还要强于 Storm。
6. **可用性高。**Spark 自身实现了 Standalone 部署模式，此模式下的 Master 可以有多个，解决了单点故障问题。Spark 也完全支持使用外部的部署模式，比如 YARN、Mesos、EC2 等。
7. **丰富的数据源支持：**Spark 除了可以访问操作系统自身的文件系统和 HDFS 之外，还可以访问 Kafka、Socket、Cassandra、HBase、Hive、Alluxio（Tachyon）及任何 Hadoop 的数据源。这极大地方便了已经使用 HDFS、HBase 的用户顺利迁移到 Spark。
8. **丰富的文件格式支持：**Spark 支持文本文件格式、CSV 文件格式、JSON 文件格式、ORC 文件格式、Parquet 文件格式、Libsvm 文件格式，也有利于 Spark 与其他数据处理平台的对接。

**2.1.3　Spark 使用场景**

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

笔者就目前所知道的 Spark 应用场景进行介绍。

**1.医疗健康**

看病是一个非常典型的分析过程——医生根据患者的一些征兆、检验结果，结合医生本人的经验得出结论，最后给出相应的治疗方案。各地区医疗水平参差不齐，特别是高水平医生更为紧缺，好医院的地区分布很不均衡。

大数据根据患者的患病征兆、检验报告，通过病理分析模型找出病因并给出具体的治疗方案。即便是医疗水平落后的地区，只需要输入患者的患病征兆和病例数据，即可体验高水平医师的服务。通过 Spark 从海量数据中实时计算出病因，各个地区的医疗水平和效率将获得大幅度提升，同时也能很好地降低因为医生水平而导致误诊的概率。

实施医疗健康的必然措施是监测和预测。通过监测不断更新整个医疗基础库的知识，并通过医疗健康模型预测出疾病易发的地区和人群。

**2.电商**

通过对用户的消费习惯、季节、产品使用周期等数据的收集，建立算法模型来判断消费者未来一个月、几个月甚至一年的消费需求（不是简单地根据你已经消费的产品，显示推荐广告位），进而提高订单转化率。

在市场营销方面，通过给买家打标签，构建人群画像，进而针对不同的人群，精准投放广告、红包或优惠券。

**3.安全领域**

面对日益复杂的网络安全，通过检测和数据分析区分出不同的安全类型，并针对不同的安全类型，实施不同的防御和打击措施。

* 端安全：使用安全卫士、云查杀对经过大数据分析得到的病毒、木马等进行防御。
* 电商安全：反刷单、反欺诈、合规。
* 金融安全：风险控制。
* 企业安全：反入侵。
* 国家安全：舆情监测，打击罪犯。

**4.金融领域**

**构建金融云，**通过对巨量的计量数据进行收集，然后通过 Spark 实时处理分析，利用低延迟的数据处理能力，应对急迫的业务需求和数据增长。

**量化投资，**收集大宗商品的价格，以及黄金，石油等各种数据，分析黄金、股票等指数趋势，支持投资决策。

除了以上领域外，Spark 在搜索引擎、生态圈异常检测、生物计算等诸多领域都有广泛的应用场景。

## 2.2　Spark 基础知识

Spark 在 2009 年诞生于 UC Berkeley 的 AMP 实验室，2010 年正式对外开源。经过 5 年多的发展，Spark 目前的最新版本是 2.2.0。本书在写作之初选择了 Spark 2.1.0 作为分析的主要版本，相信绝大多数内容依然会满足读者朋友的需要。本节将从 Spark 的基本概念和开发语言两个角度来介绍 Spark 的基础知识。

**1.基本概念**

要想对 Spark 有整体性的了解，推荐读者阅读 Matei Zaharia 的 Spark 论文。此处笔者先介绍 Spark 中的一些概念。

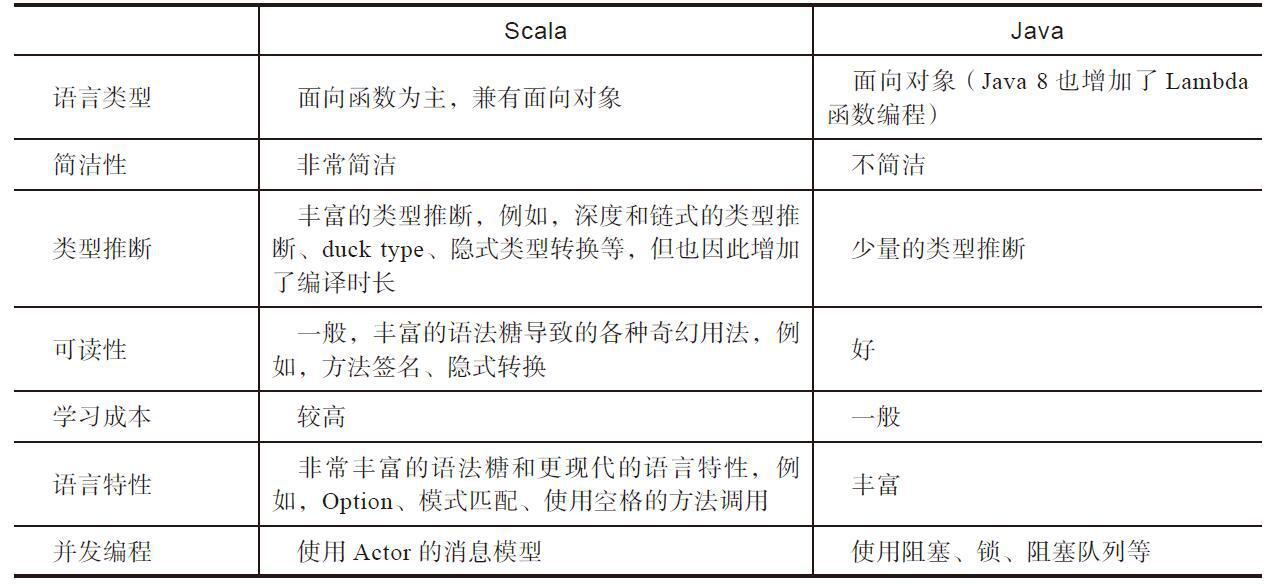
* RDD（resillient distributed dataset）：弹性分布式数据集。Spark 应用程序通过使用 Spark 的转换 API，可以将 RDD 封装为一系列具有血缘关系的 RDD，也就是 DAG。只有通过 Spark 的动作 API 才会将 RDD 及其 DAG 提交到 DAGScheduler。RDD 的祖先一定是一个跟数据源相关的 RDD，负责从数据源迭代读取数据。
* DAG（Directed Acycle graph）：有向无环图。在图论中，如果一个有向图无法从某个顶点出发经过若干条边回到该点，则这个图是一个有向无环图（DAG 图）。Spark 使用 DAG 来反映各 RDD 之间的依赖或血缘关系。
* Partition：数据分区，即一个 RDD 的数据可以划分为多少个分区。Spark 根据 Partition 的数量来确定 Task 的数量。
* NarrowDependency：窄依赖，即子 RDD 依赖于父 RDD 中固定的 Partition。Narrow-Dependency 分为 OneToOneDependency 和 RangeDependency 两种。
* ShuffleDependency：Shuffle 依赖，也称为宽依赖，即子 RDD 对父 RDD 中的所有 Partition 都可能产生依赖。子 RDD 对父 RDD 各个 Partition 的依赖将取决于分区计算器（Partitioner）的算法。
* Job：用户提交的作业。当 RDD 及其 DAG 被提交给 DAGScheduler 调度后，DAGScheduler 会将所有 RDD 中的转换及动作视为一个 Job。一个 Job 由一到多个 Task 组成。
* Stage：Job 的执行阶段。DAGScheduler 按照 ShuffleDependency 作为 Stage 的划分节点对 RDD 的 DAG 进行 Stage 划分（上游的 Stage 将为 ShuffleMapStage）。因此一个 Job 可能被划分为一到多个 Stage。Stage 分为 ShuffleMapStage 和 ResultStage 两种。
* Task：具体执行任务。一个 Job 在每个 Stage 内都会按照 RDD 的 Partition 数量，创建多个 Task。Task 分为 ShuffleMapTask 和 ResultTask 两种。ShuffleMapStage 中的 Task 为 ShuffleMapTask，而 ResultStage 中的 Task 为 ResultTask。ShuffleMapTask 和 ResultTask 类似于 Hadoop 中的 Map 任务和 Reduce 任务。
* Shuffle：Shuffle 是所有 MapReduce 计算框架的核心执行阶段，Shuffle 用于打通 map 任务（在 Spark 中就是 ShuffleMapTask）的输出与 reduce 任务（在 Spark 中就是 ResultTask）的输入，map 任务的中间输出结果按照指定的分区策略（例如，按照 key 值哈希）分配给处理某一个分区的 reduce 任务。

**2.开发语言**

目前越来越多的语言可以运行在 Java 虚拟机上，Java 平台上的多语言混合编程正成为一种潮流。在混合编程模式下可以充分利用每种语言的特点和优势，以便更好地完成功能。Spark 同时选择了 Scala 和 Java 作为开发语言，也是为了充分利用二者各自的优势。表 2-1 所示是对这两种语言进行的比较。

https://pic4.zhimg.com/v2-9f34416ca7114ee3b65a969f664f4e76_r.jpg

注意　虽然 Actor 是 Scala 语言最初进行推广时最吸引人的特性之一，但是随着 Akka 更加强大的 Actor 类库的出现，Scala 已经在官方网站宣布废弃 Scala 自身的 Actor 编程模型，转而全面拥抱 Akka 提供的 Actor 编程模型。与此同时，从 Spark 2.0.0 版本开始，Spark 却放弃了使用 Akka，转而使用 Netty 实现了自己的 RPC 框架。遥想当年 Scala「鼓吹」Actor 编程模型优于 Java 的同步编程模型时，又有谁会想到如今这种场面呢？

表 2-1　Scala 与 Java 的比较

Scala 作为函数式编程的代表，天生适合并行运行，如果用 Java 语言实现相同的功能，会显得非常臃肿。很多介绍 Spark 的新闻或文章经常以 Spark 内核代码行数少或 API 精炼等内容作为宣传的「法器」，这应该也是选择 Scala 的原因之一。另一方面，由于函数式编程更接近计算机思维，因此便于通过算法从大数据中建模，这也更符合 Spark 作为大数据框架的理念。

由于 Java 适合服务器、中间件开发，所以 Spark 使用 Java 更多的是开发底层的基础设施或中间件。

## 2.3　Spark 基本设计思想

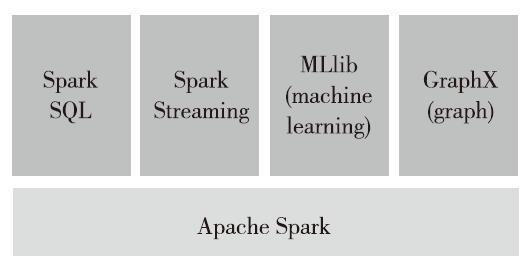
Spark 是一个功能丰富的大数据计算平台，与所有的大型系统一样，Spark 从设计到开发，也根据功能的不同进行模块的拆分。Spark 包含很多子模块，理解这些模块有助于我们掌握 Spark 的设计思想。Spark 的模块按照重要程度可分为核心功能和扩展功能。核心功能是 Spark 设计理念的核心实现，也是 Spark 陆续加入新功能的基础。在核心功能之上，通过不断地将丰富的扩展功能持续集成到 Spark 中，使得 Spark 满足更多市场、应用和用户的需求，促使 Spark 生态圈更加繁荣。

Spark 在编程模型方面没有采用 Hadoop 的 MapReduce 编程模型，而是封装了丰富的转换与执行的 API。开发人员可以充分利用这些 API 编写自己的 Spark 应用程序，而不用拘泥于如何套用 MapReduce 编程模型中的 Mapper 或 Reducer。Spark 提供的转换 API 将会在底层将数据封装为 RDD，并对这些 RDD 进行转换、构建血缘、构建 DAG、分片划分等处理，这些过程都是透明的，使得开发人员能更加专注于自己的业务实现。

下面就 Spark 的模块设计与模型设计，进行更详细的介绍。

**2.3.1　Spark 模块设计**

整个 Spark 主要由 Spark Coke、Spark SQL、Spark Streaming、GraphX、MLlib 组成，而后四项的能力都是建立在核心引擎之上，如图 2-4 所示。

图 2-4　Spark 各模块的依赖关系

我们先来看看 Spark 的核心功能 Spark Core 的介绍。

**1.Spark 核心功能**

Spark Core 中提供了 Spark 最基础与最核心的功能，主要包括以下几项。

（1）基础设施

在 Spark 中有很多基础设施，被 Spark 中的各种组件广泛使用。这些基础设施包括 Spark 配置（SparkConf）、Spark 内置的 RPC 框架（在早期 Spark 版本中 Spark 使用的是 Akka）、事件总线（ListenerBus）、度量系统。

1. SparkConf 用于管理 Spark 应用程序的各种配置信息。
2. Spark 内置的 RPC 框架使用 Netty 实现，有同步和异步的多种实现，Spark 各个组件间的通信都依赖于此 RPC 框架。
3. 如果说 RPC 框架是跨机器节点不同组件间的通信设施，那么事件总线就是 Spark Context 内部各个组件间使用事件——监听器模式异步调用的实现。
4. 度量系统由 Spark 中的多种度量源（Source）和多种度量输出（Sink）构成，完成对整个 Spark 集群中各个组件运行期状态的监控。

（2）SparkContext

通常而言，用户开发的 Spark 应用程序的提交与执行都离不开 SparkContext 的支持。在正式提交应用程序之前，首先需要初始化 SparkContext。SparkContext 隐藏了网络通信、分布式部署、消息通信、存储体系、计算引擎、度量系统、文件服务、Web UI 等内容，应用程序开发者只需要使用 SparkContext 提供的 API 完成功能开发。

（3）SparkEnv

Spark 执行环境 SparkEnv 是 Spark 中的 Task 运行所必需的组件。SparkEnv 内部封装了 RPC 环境（RpcEnv）、序列化管理器、广播管理器（BroadcastManager）、map 任务输出跟踪器（MapOutputTracker）、存储体系、度量系统（MetricsSystem）、输出提交协调器（OutputCommitCoordinator）等 Task 运行所需的各种组件。

（4）存储体系

Spark 优先考虑使用各节点的内存作为存储，当内存不足时才会考虑使用磁盘，这极大地减少了磁盘 I/O，提升了任务执行的效率，使得 Spark 适用于实时计算、迭代计算、流式计算等场景。在实际场景中，有些 Task 是存储密集型的，有些则是计算密集型的，所以有时候会造成存储空间很空闲，而计算空间的资源又很紧张。Spark 的内存存储空间与执行存储空间之间的边界可以是「软」边界，因此资源紧张的一方可以借用另一方的空间，这既可以有效利用资源，又可以提高 Task 的执行效率。此外，Spark 的内存空间还提供了 Tungsten 的实现，直接操作操作系统的内存。由于 Tungsten 省去了在堆内分配 Java 对象，因此能更加有效地利用系统的内存资源，并且因为直接操作系统内存，空间的分配和释放也更迅速。在 Spark 早期版本还使用了以内存为中心的高容错的分布式文件系统 Alluxio（Tachyon）供用户进行选择。Alluxio 能够为 Spark 提供可靠的内存级的文件共享服务。

（5）调度系统

调度系统主要由 DAGScheduler 和 TaskScheduler 组成，它们都内置在 SparkContext 中。DAGScheduler 负责创建 Job、将 DAG 中的 RDD 划分到不同的 Stage、给 Stage 创建对应的 Task、批量提交 Task 等功能。TaskScheduler 负责按照 FIFO 或者 FAIR 等调度算法对批量 Task 进行调度；为 Task 分配资源；将 Task 发送到集群管理器的当前应用的 Executor 上，由 Executor 负责执行等工作。现如今，Spark 增加了 SparkSession 和 DataFrame 等新的 API，SparkSession 底层实际依然依赖于 SparkContext。

（6）计算引擎

计算引擎由内存管理器（MemoryManager）、Tungsten、任务内存管理器（TaskMemory-Manager）、Task、外部排序器（ExternalSorter）、Shuffle 管理器（ShuffleManager）等组成。MemoryManager 除了对存储体系中的存储内存提供支持和管理外，还为计算引擎中的执行内存提供支持和管理。Tungsten 除用于存储外，也可以用于计算或执行。TaskMemoryManager 对分配给单个 Task 的内存资源进行更细粒度的管理和控制。ExternalSorter 用于在 map 端或 reduce 端对 ShuffleMapTask 计算得到的中间结果进行排序、聚合等操作。ShuffleManager 用于将各个分区对应的 ShuffleMapTask 产生的中间结果持久化到磁盘，并在 reduce 端按照分区远程拉取 ShuffleMapTask 产生的中间结果。

**2.Spark 扩展功能**

为了扩大应用范围，Spark 陆续增加了一些扩展功能，主要包括以下几项。

（1）Spark SQL

由于 SQL 具有普及率高、学习成本低等特点，为了扩大 Spark 的应用面，还增加了对 SQL 及 Hive 的支持。Spark SQL 的过程可以总结为：首先使用 SQL 语句解析器（SqlParser）将 SQL 转换为语法树（Tree），并且使用规则执行器（RuleExecutor）将一系列规则（Rule）应用到语法树，最终生成物理执行计划并执行的过程。其中，规则包括语法分析器（Analyzer）和优化器（Optimizer）。Hive 的执行过程与 SQL 类似。

（2）Spark Streaming

Spark Streaming 与 Apache Storm 类似，也用于流式计算。Spark Streaming 支持 Kafka、Flume、Kinesis 和简单的 TCP 套接字等多种数据输入源。输入流接收器（Receiver）负责接入数据，是接入数据流的接口规范。Dstream 是 Spark Streaming 中所有数据流的抽象，Dstream 可以被组织为 DStream Graph。Dstream 本质上由一系列连续的 RDD 组成。

（3）GraphX

Spark 提供的分布式图计算框架。GraphX 主要遵循整体同步并行计算模式（Bulk Synchronous Parallell，BSP）下的 Pregel 模型实现。GraphX 提供了对图 Graph 的抽象，Graph 由顶点（Vertex）、边（Edge）及继承了 Edge 的 EdgeTriplet（添加了 srcAttr 和 dstAttr，用来保存源顶点和目的顶点的属性）三种结构组成。GraphX 目前已经封装了最短路径、网页排名、连接组件、三角关系统计等算法的实现，用户可以选择使用。

（4）MLlib

Spark 提供的机器学习框架。机器学习是一门涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多领域的交叉学科。MLlib 目前已经提供了基础统计、分类、回归、决策树、随机森林、朴素贝叶斯、保序回归、协同过滤、聚类、维数缩减、特征提取与转型、频繁模式挖掘、预言模型标记语言、管道等多种数理统计、概率论、数据挖掘方面的数学算法。

**小贴士：**由于本书旨在着重分析 Spark Core 的原理，因此不会对 Spark 的扩展功能进行分析。对 Spark SQL、Spark Streaming、GraphX、MLlib 等内容感兴趣的读者，可以查阅其他文章或书籍，也可以选择阅读笔者所著的《深入理解 Spark：核心思想与源码分析》一书对上述内容的介绍。

**2.3.2　Spark 模型设计（重要）**

**1.Spark 编程模型**

正如 Hadoop 在介绍 MapReduce 编程模型时选择 word count 的例子，并且使用图形来说明一样，笔者也选择用图形展现 Spark 编程模型。

Spark 应用程序从编写到提交、执行、输出的整个过程如图 2-5 所示。

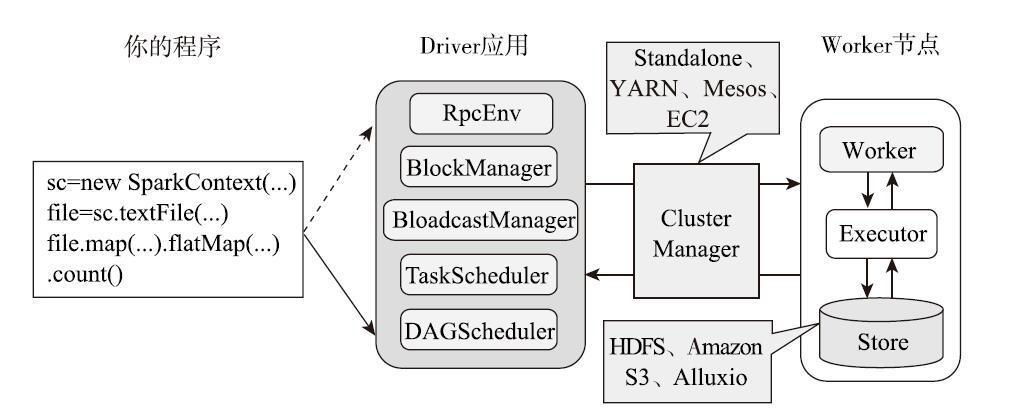
图 2-5　代码执行过程

图 2-5 中描述了 Spark 编程模型的关键环节，其步骤如下。

1）用户使用 SparkContext 提供的 API㊟编写 Driver 应用程序。此外，SparkSession、DataFrame、SQLContext、HiveContext 及 StreamingContext 都对 SparkContext 进行了封装，并提供了 DataFrame、SQL、Hive 及流式计算相关的 API。

2）使用 SparkContext 提交的用户应用程序，首先会通过 RpcEnv 向集群管理器（Cluster Manager）注册应用（Application）并且告知集群管理器需要的资源数量。集群管理器根据 Application 的需求，给 Application 分配 Executor 资源，并在 Worker 上启动 CoarseGrainedExecutorBackend 进程（该进程内部将创建 Executor）。Executor 所在的 CoarseGrainedExecutorBackend 进程在启动的过程中将通过 RpcEnv 直接向 Driver 注册 Executor 的资源信息，TaskScheduler 将保存已经分配给应用的 Executor 资源的地址、大小等相关信息。然后，SparkContext 根据各种转换 API，构建 RDD 之间的血缘关系和 DAG，RDD 构成的 DAG 将最终提交给 DAGScheduler。DAGScheduler 给提交的 DAG 创建 Job，并根据 RDD 的依赖性质将 DAG 划分为不同的 Stage。DAGScheduler 根据 Stage 内 RDD 的 Partition 数量创建多个 Task 并批量提交给 TaskScheduler。TaskScheduler 对批量的 Task 按照 FIFO 或 FAIR 调度算法进行调度，然后给 Task 分配 Executor 资源，最后将 Task 发送给 Executor 由 Executor 执行。此外，SparkContext 还会在 RDD 转换开始之前使用 BlockManager 和 BroadcastManager 将任务的 Hadoop 配置进行广播。

3）集群管理器（Cluster Manager）会根据应用的需求，给应用分配资源，即将具体任务分配到不同 Worker 节点上的多个 Executor 来处理任务的运行。Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

4）Task 在运行的过程中需要对一些数据（如中间结果、检查点等）进行持久化，Spark 支持选择 HDFS、Amazon S3、Alluxio（原名叫 Tachyon）等作为存储。

**2.RDD 计算模型**

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

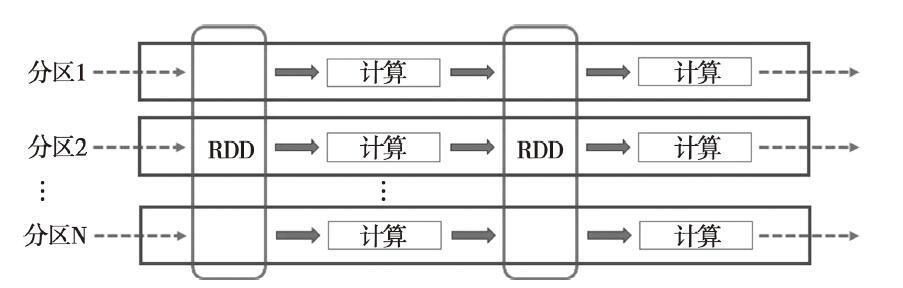
图 2-6　RDD 计算模型

图 2-6 只是简单地从分区的角度将 RDD 的计算看作是管道，如果从 RDD 的血缘关系、Stage 划分的角度来看，由 RDD 构成的 DAG 经过 DAGScheduler 调度后，将变成如图 2-7 所示的样子。

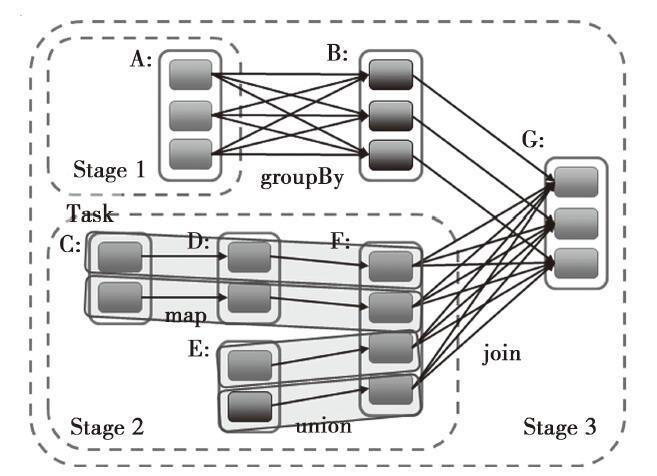
图 2-7　DAGScheduler 对由 RDD 构成的 DAG 进行调度

图 2-7 中共展示了 A、B、C、D、E、F、G 一共 7 个 RDD。每个 RDD 中的小方块代表一个分区，将会有一个 Task 处理此分区的数据。RDD A 经过 groupByKey 转换后得到 RDD B。RDD C 经过 map 转换后得到 RDD D。RDD D 和 RDD E 经过 union 转换后得到 RDD F。RDD B 和 RDD F 经过 join 转换后得到 RDD G。从图 2-7 中可以看到，map 和 union 生成的 RDD 与其上游 RDD 之间的依赖是 NarrowDependency，而 groupByKey 和 join 生成的 RDD 与其上游的 RDD 之间的依赖是 ShuffleDependency。由于 DAGScheduler 按照 ShuffleDependency 作为 Stage 的划分的依据，因此 A 被划入了 ShuffleMapStage 1；C、D、E、F 被划入了 ShuffleMapStage 2；B 和 G 被划入了 ResultStage 3。

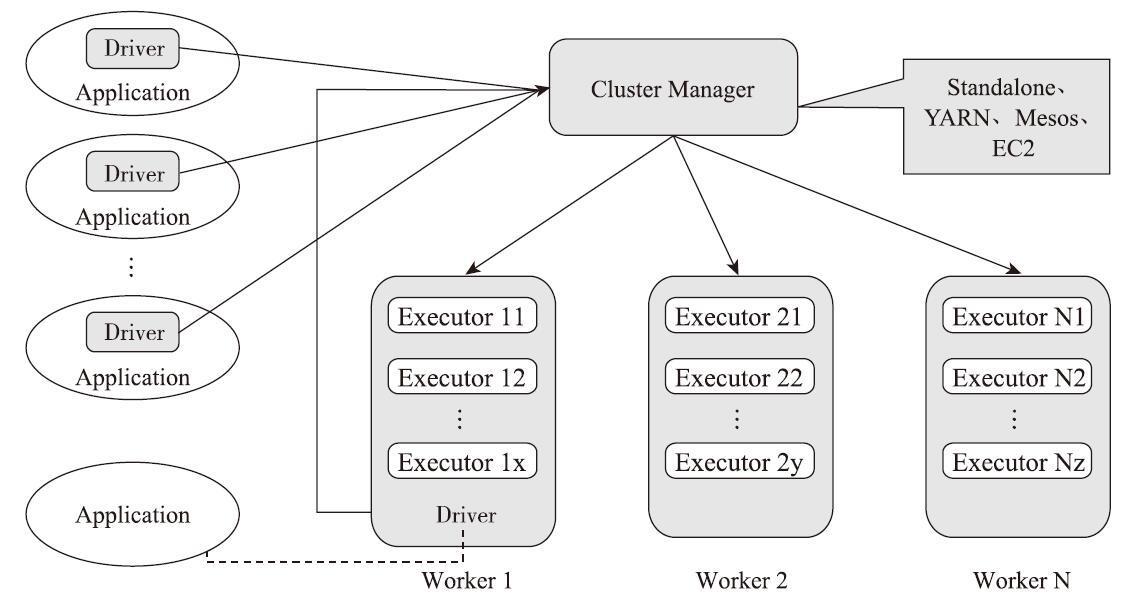
## 2.4　Spark 基本架构

从集群部署的角度来看，Spark 集群由集群管理器（Cluster Manager）、工作节点（Worker）、执行器（Executor）、驱动器（Driver）、应用程序（Application）等部分组成，它们之间的整体关系如图 2-8 所示。

下面结合图 2-8 对这些组成部分及它们之间的关系进行介绍。

（1）Cluster Manager

Spark 的集群管理器，主要负责对整个集群资源的分配与管理。Cluster Manager 在 YARN 部署模式下为 ResourceManager；在 Mesos 部署模式下为 Mesos Master；在 Standalone 部署模式下为 Master。Cluster Manager 分配的资源属于一级分配，它将各个 Worker 上的内存、CPU 等资源分配给 Application，但是并不负责对 Executor 的资源分配。Standalone 部署模式下的 Master 会直接给 Application 分配内存、CPU 及 Executor 等资源。目前，Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

图 2-8　Spark 基本架构图

　这里提到了部署模式中的 Standalone、YARN、Mesos 等模式，读者暂时知道这些内容即可，第 9 章将对它们进行详细介绍。

https://pic3.zhimg.com/v2-88cd7e0676935c4475a7b4e9a65610b9_r.jpg注意

（2）Worker

Spark 的工作节点。在 YARN 部署模式下实际由 NodeManager 替代。Worker 节点主要负责以下工作：将自己的内存、CPU 等资源通过注册机制告知 Cluster Manager；创建 Executor；将资源和任务进一步分配给 Executor；同步资源信息、Executor 状态信息给 Cluster Manager 等。在 Standalone 部署模式下，Master 将 Worker 上的内存、CPU 及 Executor 等资源分配给 Application 后，将命令 Worker 启动 CoarseGrainedExecutorBackend 进程（此进程会创建 Executor 实例）。

（3）Executor

执行计算任务的一线组件。主要负责任务的执行及与 Worker、Driver 的信息同步。

（4）Driver

Application 的驱动程序，Application 通过 Driver 与 Cluster Manager、Executor 进行通信。Driver 可以运行在 Application 中，也可以由 Application 提交给 Cluster Manager 并由 Cluster Manager 安排 Worker 运行。

（5）Application

用户使用 Spark 提供的 API 编写的应用程序，Application 通过 Spark API 将进行 RDD 的转换和 DAG 的构建，并通过 Driver 将 Application 注册到 Cluster Manager。Cluster Manager 将会根据 Application 的资源需求，通过一级分配将 Executor、内存、CPU 等资源分配给 Application。Driver 通过二级分配将 Executor 等资源分配给每一个任务，Application 最后通过 Driver 告诉 Executor 运行任务。

## 2.5　小结

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

# 第 3 章　Spark 基础设施

「道生一，一生二，二生三，三生万物。」

——《道德经》

**本章导读**

我们生活的地球，是一个大千世界。地理上，既有寸草不生的撒哈拉大沙漠，又有河水泛滥的亚马逊森林；古诗中，既有王维所高歌的「大漠孤烟直，长河落日圆」，又有张继所低吟的「月落乌啼霜满天，江枫渔火对愁眠」；音乐中，既有古筝弹奏的典雅《高山流水》，又有笛子吹奏的民俗《小放牛》。道家认为道包含阴阳二气，阴或阳既有对立的一面，又有交汇的一面。阴阳交汇就产生了世间万物。用现代科学来讲，道就是整个宇宙的规律或法则，它包含一系列的自然法则，如阴阳、正负、矛盾等。这些法则将生成各种元素，元素经过化学的、物理的反应创造出天地万物。

同理，Spark 无论多么引人注目，究其根本，也脱离不了最基本、最简单的基础设施。Spark 的任何核心功能都离不开这些基础设施的构建，因此阅读本章内容将有助于我们理解 Spark 最底层的原理与实现。本书的核心篇以本章为始，也是秉着「深入浅出」的原则，让大家从基础知识慢慢进入核心内容的理解。

本章主要讲解的内容如下。

* Spark 配置。
* RPC 框架。
* 事件总线。
* 度量系统。

## 3.1　Spark 配置

任何优秀的软件或服务都会提供一些配置参数，这些配置参数有些是内置的，有些则是可以由用户配置的。对于熟悉 Java 的开发人员来说，对 JVM 进行性能调优是一个经常需要面对的工作，这个过程常常伴随着各种 JVM 参数的调整与测试。之所以将这些参数交给具体的开发人员去调整，是因为软件或服务的提供者也无法保证给定的默认参数是最符合用户应用场景与软硬件环境的。举一个简单的例子：当用户的 QPS 发生变化时，对于 Web 服务的 JVM 来说也应当相应调整内存的大小或限制。

Spark 作为一款优秀的计算框架，也配备了各种各样的系统配置参数。SparkConf 是 Spark 的配置类，这个类在 Spark 的历史版本中已经存在很久了，Spark 中的每一个组件都直接或者间接地使用着它所存储的属性，这些属性都存储在如下的数据结构中。

private val settings = new ConcurrentHashMap[String, String]()

由以上代码的泛型㊟可以看出，Spark 的所有配置，无论是 key 还是 value 都是 String 类型。Spark 的配置通过以下 3 种方式获取。

* 来源于系统参数（即使用 System.getProperties 获取的属性）中以 spark.作为前缀的那部分属性；
* 使用 SparkConf 的 API 进行设置；
* 从其他 SparkConf 中克隆。

下面将具体说明这 3 种方式的实现。

**3.1.1　系统属性中的配置**

在 SparkConf 中有一个 Boolean 类型的构造器属性 loadDefaults，当 loadDefaults 为 true 时，将会从系统属性中加载 Spark 配置，代码如下：

if (loadDefaults) {  
 loadFromSystemProperties(false)  
}  
  
private[spark] def loadFromSystemProperties(silent: Boolean): SparkConf = {  
 // 加载以spark.开头的系统属性  
 for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {  
 set(key, value, silent)  
 }  
 this  
}

以上代码调用了 Utils 工具类㊟的 getSystemProperties 方法，其作用为获取系统的键值对属性。loadFromSystemProperties 方法在获取了系统属性后，使用 Scala 守卫过滤出其中以「spark.」字符串为前缀的 key 和 value 并且调用 set 方法（见代码清单 3-1），最终设置到 settings 中。

代码清单 3-1　SparkConf 中 set 方法的实现

private[spark] def set(key: String, value: String, silent: Boolean): SparkConf = {  
 if (key == null) {  
 throw new NullPointerException("null key")  
 }  
 if (value == null) {  
 throw new NullPointerException("null value for " + key)  
 }  
 if (!silent) {  
 logDeprecationWarning(key)  
 }  
 settings.put(key, value)  
 this  
}

**3.1.2　使用 SparkConf 配置的 API**

给 SparkConf 添加配置的一种常见方式是使用 SparkConf 中提供的 API。其中有些 API 最终实际调用了 set 的重载方法，如代码清单 3-2 所示。

代码清单 3-2　SparkConf 中重载的 set 方法

def set(key: String, value: String): SparkConf = {  
 set(key, value, false)  
}

可以看到代码清单 3-2 中的 set 方法实际也是调用了代码清单 3-1 中的 set 方法。

SparkConf 中的 setMaster、setAppName、setJars、setExecutorEnv、setSparkHome、setAll 等方法最终都是通过代码清单 3-2 中的 set 方法完成 Spark 配置的，本书以其中最为常用的 setMaster 和 setAppName 为例，用代码清单 3-3 和代码清单 3-4 来展示它们的实现。

代码清单 3-3　设置 Spark 的部署模式的配置方法 setMaster

def setMaster(master: String): SparkConf = {  
 set("spark.master", master)  
}

代码清单 3-4　设置 Spark 的应用名称的配置方法 setAppName

def setAppName(name: String): SparkConf = {  
 set("spark.app.name", name)  
}

**3.1.3　克隆 SparkConf 配置**

有些情况下，同一个 SparkConf 实例中的配置信息需要被 Spark 中的多个组件共用，例如，组件 A 中存在一个 SparkConf 实例 a，组件 B 中也需要实例 a 中的配置信息，这时该如何处理？我们往往首先想到的方法是将 SparkConf 实例定义为全局变量或者通过参数传递给其他组件，但是这会引入并发问题。虽然 settings 是线程安全的 ConcurrentHashMap 类，而且 ConcurrentHashMap 也被证明是高并发下性能表现不错的数据结构，但是只要存在并发，就一定会有性能的损失问题。我们可以新建一个 SparkConf 实例 b，并将 a 中的配置信息全部拷贝到 b 中，这种方式显然不是最优雅的，复制代码会散落在程序的各个角落。现在是时候阅读下 SparkConf 的构造器了，代码如下所示：

class SparkConf(loadDefaults: Boolean) extends Cloneable with Logging with Serializable {  
//省略无关代码  
def this() = this(true)

SparkConf 继承了 Cloneable 特质并实现了 clone 方法，clone 方法（见代码清单 3-5）的实现跟我们所讨论的方式是一样的，并且通过 Cloneable 特质提高了代码的可复用性。

代码清单 3-5　克隆 SparkConf 配置

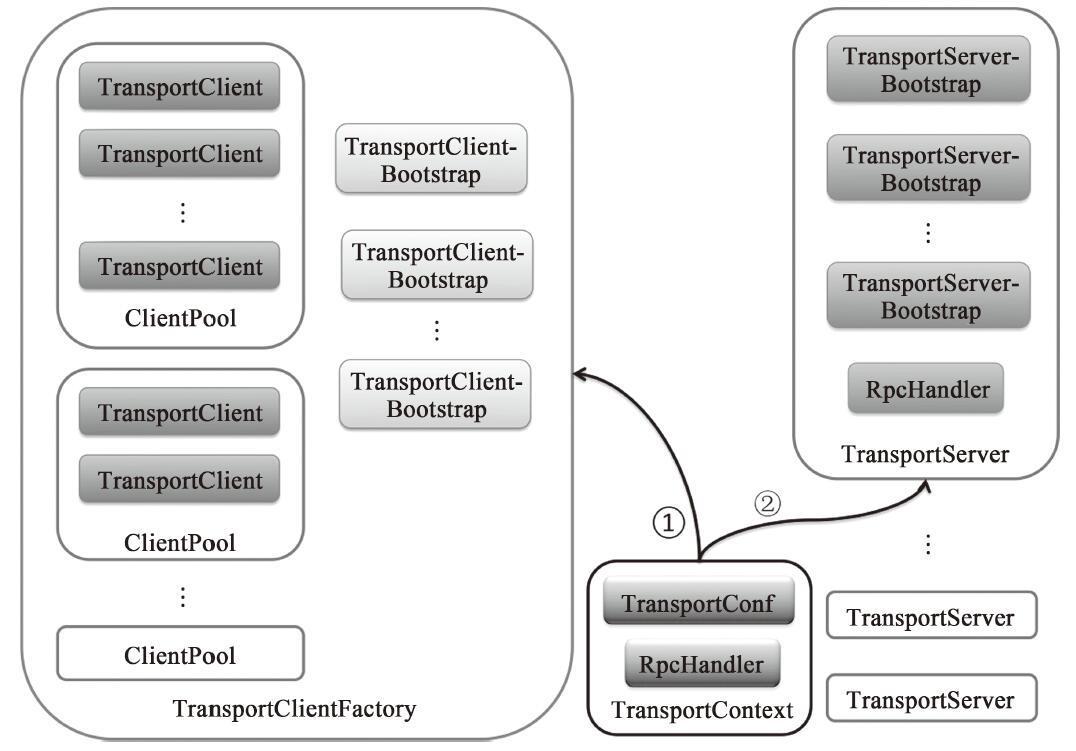
override def clone: SparkConf = {  
 val cloned = new SparkConf(false)  
 settings.entrySet().asScala.foreach { e =>  
 cloned.set(e.getKey(), e.getValue(), true)  
 }  
 cloned  
}

这样我们就可以在任何想要使用 SparkConf 的地方使用克隆方式来优雅地编程了。

## 3.2　Spark 内置 RPC 框架

在 Spark 中很多地方都涉及网络通信，比如 Spark 各个组件间的消息互通、用户文件与 Jar 包的上传、节点间的 Shuffle 过程、Block 数据的复制与备份等。在 Spark 0.x.x 与 Spark 1.x.x 版本中，组件间的消息通信主要借助于 Akka㊟，使用 Akka 可以轻松地构建强有力的高并发与分布式应用。但是 Akka 在 Spark 2.0.0 版本中被移除了，Spark 官网文档对此的描述为：「Akka 的依赖被移除了，因此用户可以使用任何版本的 Akka 来编程了。」Spark 团队的决策者或许认为对 Akka 具体版本的依赖，限制了用户对 Akka 不同版本的使用。尽管如此，笔者依然认为 Akka 是一款非常优秀的开源分布式系统，我参与的一些 Java Application 或者 Java Web 就利用 Akka 的丰富特性实现了分布式一致性、最终一致性及分布式事务等分布式环境面对的问题。在 Spark 1.x.x 版本中，用户文件与 Jar 包的上传采用了由 Jetty㊟实现的 HttpFileServer，但在 Spark 2.0.0 版本中它也被废弃了，现在使用的是基于 Spark 内置 RPC 框架的 NettyStreamManager。节点间的 Shuffle 过程和 Block 数据的复制与备份这两个部分在 Spark 2.0.0 版本中依然沿用了 Netty㊟，通过对接口和程序进行重新设计，将各个组件间的消息互通、用户文件与 Jar 包的上传等内容统一纳入 Spark 的 RPC 框架体系中。

我们先来看看 RPC 框架的基本架构，如图 3-1 所示。

图 3-1　Spark 内置 RPC 框架的基本架构

TransportContext 内部包含传输上下文的配置信息 TransportConf 和对客户端请求消息进行处理的 RpcHandler。TransportConf 在创建 TransportClientFactory 和 TransportServer 时都是必需的，而 RpcHandler 只用于创建 TransportServer。TransportClientFactory 是 RPC 客户端的工厂类。TransportServer 是 RPC 服务端的实现。图中记号的含义如下。

* 记号 ① 表示通过调用 TransportContext 的 createClientFactory 方法创建传输客户端工厂 TransportClientFactory 的实例。在构造 TransportClientFactory 的实例时，还会传递客户端引导程序 TransportClientBootstrap 的列表。此外，TransportClientFactory 内部还存在针对每个 Socket 地址的连接池 ClientPool，这个连接池缓存的定义如下：

private final ConcurrentHashMap<SocketAddress, ClientPool> connectionPool;

ClientPool 的类型定义如下：

private static class ClientPool {  
 TransportClient[] clients;  
 Object[] locks;  
  
 ClientPool(int size) {  
 clients = new TransportClient[size];  
 locks = new Object[size];  
 for (int i = 0; i < size; i++) {  
 locks[i] = new Object();  
 }  
 }  
}

由此可见，ClientPool 实际是由 TransportClient 的数组构成的，而 locks 数组中的 Object 与 clients 数组中的 TransportClient 按照数组索引一一对应，通过对每个 TransportClient 分别采用不同的锁，降低并发情况下线程间对锁的争用，进而减少阻塞，提高并发度。

* 记号 ② 表示通过调用 TransportContext 的 createServer 方法创建传输服务端 TransportServer 的实例。在构造 TransportServer 的实例时，需要传递 TransportContext、host、port、RpcHandler 及服务端引导程序 TransportServerBootstrap 的列表。

有了对 Spark 内置 RPC 框架的基本架构的了解，现在正式介绍 Spark 的 RPC 框架所包含的各个组件。

* TransportContext：传输上下文，包含了用于创建传输服务端（TransportServer）和传输客户端工厂（TransportClientFactory）的上下文信息，并支持使用 Transport-ChannelHandler 设置 Netty 提供的 SocketChannel 的 Pipeline 的实现。
* TransportConf：传输上下文的配置信息。
* RpcHandler：对调用传输客户端（TransportClient）的 sendRPC 方法发送的消息进行处理的程序。
* MessageEncoder：在将消息放入管道前，先对消息内容进行编码，防止管道另一端读取时丢包和解析错误。
* MessageDecoder：对从管道中读取的 ByteBuf 进行解析，防止丢包和解析错误。
* TransportFrameDecoder：对从管道中读取的 ByteBuf 按照数据帧进行解析。
* RpcResponseCallback：RpcHandler 对请求的消息处理完毕后进行回调的接口。
* TransportClientFactory：创建 TransportClient 的传输客户端工厂类。
* ClientPool：在两个对等节点间维护的关于 TransportClient 的池子。ClientPool 是 TransportClientFactory 的内部组件。
* TransportClient：RPC 框架的客户端，用于获取预先协商好的流中的连续块。TransportClient 旨在允许有效传输大量数据，这些数据将被拆分成几百 KB 到几 MB 的块。当 TransportClient 处理从流中获取的块时，实际的设置是在传输层之外完成的。sendRPC 方法能够在客户端和服务端的同一水平线的通信进行这些设置。
* TransportClientBootstrap：当服务端响应客户端连接时在客户端执行一次的引导程序。
* TransportRequestHandler：用于处理客户端的请求并在写完块数据后返回的处理程序。
* TransportResponseHandler：用于处理服务端的响应，并且对发出请求的客户端进行响应的处理程序。
* TransportChannelHandler：代理由 TransportRequestHandler 处理的请求和由 Transport-ResponseHandler 处理的响应，并加入传输层的处理。
* TransportServerBootstrap：当客户端连接到服务端时在服务端执行一次的引导程序。
* TransportServer：RPC 框架的服务端，提供高效的、低级别的流服务。

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

### 3.2.1　RPC 配置 TransportConf

上文提到 TransportContext 中的 TransportConf 给 Spark 的 RPC 框架提供配置信息，它有两个成员属性——配置提供者 conf 和配置的模块名称 module。这两个属性的定义如下：

private final ConfigProvider conf;  
private final String module;

其中 conf 是真正的配置提供者，其类型 ConfigProvider 是一个抽象类，如代码清单 3-6 所示。

代码清单 3-6　ConfigProvider 的实现

public abstract class ConfigProvider {  
public abstract String get(String name);  
  
public String get(String name, String defaultValue) {  
 try {  
 return get(name);  
 } catch (NoSuchElementException e) {  
 return defaultValue;  
 }  
}  
  
public int getInt(String name, int defaultValue) {  
 return Integer.parseInt(get(name, Integer.toString(defaultValue)));  
}  
  
public long getLong(String name, long defaultValue) {  
 return Long.parseLong(get(name, Long.toString(defaultValue)));  
}  
  
public double getDouble(String name, double defaultValue) {  
 return Double.parseDouble(get(name, Double.toString(defaultValue)));  
}  
  
public boolean getBoolean(String name, boolean defaultValue) {  
 return Boolean.parseBoolean(get(name, Boolean.toString(defaultValue)));  
 }  
}

从代码清单 3-6 可以看到，ConfigProvider 中包括 get、getInt、getLong、getDouble、getBoolean 等方法，这些方法都是基于抽象方法 get 获取值，经过一次类型转换而实现。这个抽象的 get 方法将需要子类去实现。

Spark 通常使用 SparkTransportConf 创建 TransportConf，其实现如代码清单 3-7 所示。

代码清单 3-7　SparkTransportConf 的实现

object SparkTransportConf {  
 private val MAX\_DEFAULT\_NETTY\_THREADS = 8  
 def fromSparkConf(\_conf: SparkConf, module: String, numUsableCores: Int = 0): TransportConf = {  
 val conf = \_conf.clone  
 val numThreads = defaultNumThreads(numUsableCores)  
 conf.setIfMissing(s"spark.$module.io.serverThreads", numThreads.toString)  
 conf.setIfMissing(s"spark.$module.io.clientThreads", numThreads.toString)  
  
 new TransportConf(module, new ConfigProvider {  
 override def get(name: String): String = conf.get(name)  
 })  
 }  
 private def defaultNumThreads(numUsableCores: Int): Int = {  
 val availableCores =  
 if (numUsableCores > 0) numUsableCores else Runtime.getRuntime.availableProcessors()  
 math.min(availableCores, MAX\_DEFAULT\_NETTY\_THREADS)  
 }  
}

从代码清单 3-7 看到，可以使用 SparkTransportConf 的 fromSparkConf 方法来构造 TransportConf。传递的三个参数分别为 SparkConf、模块名 module 及可用的内核数 num-UsableCores。如果 numUsableCores 小于等于 0，那么线程数是系统可用处理器的数量，不过系统的内核数不可能全部用于网络传输，所以这里将分配给网络传输的内核数量最多限制在 8 个。最终确定的线程数将用于设置客户端传输线程数（spark.$module.io.clientThreads 属性）和服务端传输线程数（spark.$module.io.serverThreads 属性）。from-SparkConf 最终构造 TransportConf 对象时传递的 ConfigProvider 为实现了 get 方法的匿名的内部类，get 的实现实际是代理了 SparkConf 的 get 方法。

### 3.2.2　RPC 客户端工厂 TransportClientFactory

TransportClientFactory 是创建 TransportClient 的工厂类。在说明图 3-1 中的记号 ① 时提到过，TransportContext 的 createClientFactory 方法可以创建 TransportClientFactory 的实例，其实现如代码清单 3-8 所示。

代码清单 3-8　创建传输客户端工厂

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

可以看到，TransportContext 中有两个重载的 createClientFactory 方法，它们最终在构造 TransportClientFactory 时都会传递两个参数：TransportContext 和 TransportClientBootstrap 列表。TransportClientFactory 构造器的实现如代码清单 3-9 所示。

代码清单 3-9　TransportClientFactory 的构造器

public TransportClientFactory(  
 TransportContext context,  
 List<TransportClientBootstrap> clientBootstraps) {  
 this.context = Preconditions.checkNotNull(context);  
 this.conf = context.getConf();  
 this.clientBootstraps = Lists.newArrayList(Preconditions.checkNotNull(clientBootstraps));  
 this.connectionPool = new ConcurrentHashMap<>();  
 this.numConnectionsPerPeer = conf.numConnectionsPerPeer();  
 this.rand = new Random();  
  
 IOMode ioMode = IOMode.valueOf(conf.ioMode());  
 this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);  
 this.workerGroup = NettyUtils.createEventLoop(  
 ioMode,  
 conf.clientThreads(),  
 conf.getModuleName() + "-client");  
 this.pooledAllocator = NettyUtils.createPooledByteBufAllocator(  
 conf.preferDirectBufs(), false /\* allowCache \*/, conf.clientThreads());  
}

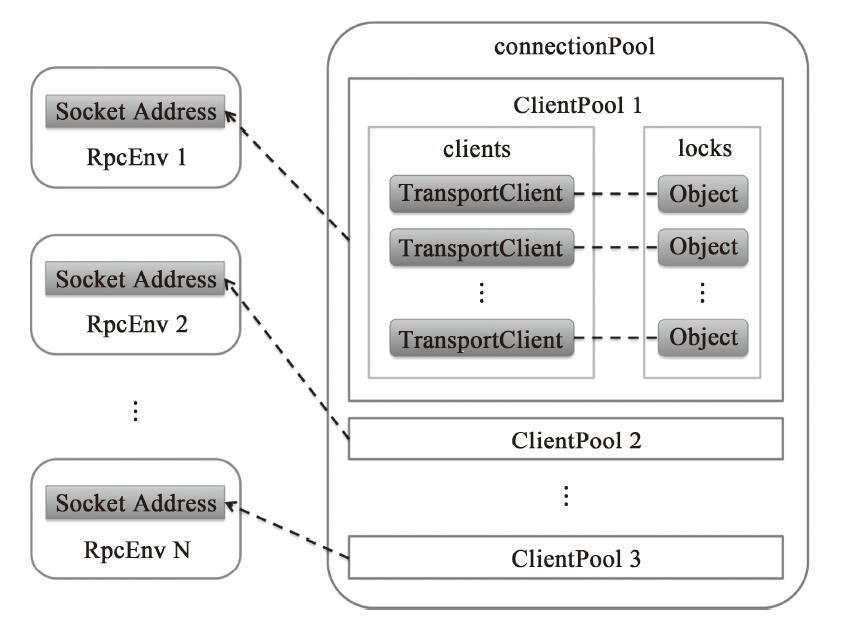
TransportClientFactory 构造器中的各个变量如下。

·context：参数传递的 TransportContext 的引用。

·conf：指 TransportConf，这里通过调用 TransportContext 的 getConf 获取。

·clientBootstraps：参数传递的 TransportClientBootstrap 列表。

·connectionPool：针对每个 Socket 地址的连接池 ClientPool 的缓存。connectionPool 的数据结构较为复杂，为便于读者理解，这里以图 3-2 来表示 connectionPool 的数据结构。

图 3-2　TransportClientFactory 的 connectionPool

* numConnectionsPerPeer：从 TransportConf 获取的 key 为「spark.+ 模块名 +.io.num-ConnectionsPerPeer」的属性值。此属性值用于指定对等节点间的连接数。这里的模块名实际为 TransportConf 的 module 字段。Spark 的很多组件都利用 RPC 框架构建，它们之间按照模块名区分，例如，RPC 模块的 key 为「spark.rpc.io.num ConnectionsPerPeer」。
* rand：对 Socket 地址对应的连接池 ClientPool 中缓存的 TransportClient 进行随机选择，对每个连接做负载均衡。
* ·ioMode：IO 模式，即从 TransportConf 获取 key 为「spark.+ 模块名 +.io.mode」的属性值。默认值为 NIO，Spark 还支持 EPOLL。
* socketChannelClass：客户端 Channel 被创建时使用的类，通过 ioMode 来匹配，默认为 NioSocketChannel，Spark 还支持 EpollEventLoopGroup。
* workerGroup：根据 Netty 的规范，客户端只有 worker 组，所以此处创建 worker-Group。workerGroup 的实际类型是 NioEventLoopGroup。
* pooledAllocator：汇集 ByteBuf 但对本地线程缓存禁用的分配器。
* TransportClientFactory 里大量使用了 NettyUtils，关于 NettyUtils 的具体实现，请参看附录 G㊟。

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

https://pic4.zhimg.com/v2-4a621c3ee3020211641a6f65c0de7a59_r.jpg

##### 提示

**1.客户端引导程序 TransportClientBootstrap**

TransportClientFactory 的 clientBootstraps 属性是 TransportClientBootstrap 的列表。Transport ClientBootstrap 是在 TransportClient 上执行的客户端引导程序，主要对连接建立时进行一些初始化的准备（例如验证、加密）。TransportClientBootstrap 所做的操作往往是昂贵的，好在建立的连接可以重用。TransportClientBootstrap 的接口定义如代码清单 3-10 所示。

代码清单 3-10　TransportClientBootstrap 的定义

public interface TransportClientBootstrap {  
void doBootstrap(TransportClient client, Channel channel) throws RuntimeException;  
}

TransportClientBootstrap 有两个实现类：EncryptionDisablerBootstrap 和 SaslClientBootstrap。为了对 TransportClientBootstrap 的作用能有更深的了解，这里以 EncryptionDisablerBootstrap 为例，EncryptionDisablerBootstrap 的实现如代码清单 3-11 所示。

代码清单 3-11　EncryptionDisablerBootstrap 的实现

private static class EncryptionDisablerBootstrap implements Transport-ClientBootstrap {  
 @Override  
 public void doBootstrap(TransportClient client, Channel channel) {  
 channel.pipeline().remove(SaslEncryption.ENCRYPTION\_HANDLER\_NAME);  
 }  
}

根据代码清单 3-11 可以看到，EncryptionDisablerBootstrap 的作用是移除客户端管道中的 SASL 加密。

**2.创建 RPC 客户端 TransportClient**

有了 TransportClientFactory，Spark 的各个模块就可以使用它创建 RPC 客户端 Transport-Client 了。每个 TransportClient 实例只能和一个远端的 RPC 服务通信，所以 Spark 中的组件如果想要和多个 RPC 服务通信，就需要持有多个 TransportClient 实例。创建 Transport-Client 的方法如代码清单 3-12 所示（实际为从缓存中获取 TransportClient）。

代码清单 3-12　从缓存中获取 TransportClient

public TransportClient createClient(String remoteHost, int remotePort)  
 throws IOException, InterruptedException {  
 // 创建InetSocketAddress  
 final InetSocketAddress unresolvedAddress =  
 InetSocketAddress.createUnresolved(remoteHost, remotePort);  
  
 ClientPool clientPool = connectionPool.get(unresolvedAddress);  
 if (clientPool == null) {  
 connectionPool.putIfAbsent(unresolvedAddress, new ClientPool(numConnectionsPerPeer));  
 clientPool = connectionPool.get(unresolvedAddress);  
 }  
   
 // 随机选择一个TransportClient  
 int clientIndex = rand.nextInt(numConnectionsPerPeer);   
 TransportClient cachedClient = clientPool.clients[clientIndex];  
  
 if (cachedClient != null && cachedClient.isActive()) {// 获取并返回激活的  
 //TransportClient  
 TransportChannelHandler handler = cachedClient.getChannel().pipeline()  
 .get(TransportChannelHandler.class);  
 synchronized (handler) {  
 handler.getResponseHandler().updateTimeOfLastRequest();  
 }  
  
 if (cachedClient.isActive()) {  
 logger.trace("Returning cached connection to {}: {}",  
 cachedClient.getSocketAddress(), cachedClient);  
 return cachedClient;  
 }  
 }  
  
 final long preResolveHost = System.nanoTime();  
 final InetSocketAddress resolvedAddress = new InetSocketAddress(remoteHost, remotePort);  
 final long hostResolveTimeMs = (System.nanoTime() - preResolveHost) / 1000000;  
 if (hostResolveTimeMs > 2000) {  
 logger.warn("DNS resolution for {} took {} ms", resolvedAddress, host ResolveTimeMs);  
 } else {  
 logger.trace("DNS resolution for {} took {} ms", resolvedAddress, host ResolveTimeMs);  
 }  
 // 创建并返回TransportClient对象  
 synchronized (clientPool.locks[clientIndex]) {  
 cachedClient = clientPool.clients[clientIndex];  
  
 if (cachedClient != null) {  
 if (cachedClient.isActive()) {  
 logger.trace("Returning cached connection to {}: {}", resolvedAddress, cachedClient);  
 return cachedClient;  
 } else {  
 logger.info("Found inactive connection to {}, creating a new one.", resolvedAddress);  
 }  
 }  
 clientPool.clients[clientIndex] = createClient(resolvedAddress);   
 return clientPool.clients[clientIndex];  
 }  
}

从代码清单 3-12 得知，创建 TransportClient 的步骤如下。

1. 调用 InetSocketAddress 的静态方法 createUnresolved 构建 InetSocketAddress（这种方式创建 InetSocketAddress，可以在缓存中已经有 TransportClient 时避免不必要的域名解析），然后从 connectionPool 中获取与此地址对应的 ClientPool，如果没有，则需要新建 ClientPool，并放入缓存 connectionPool 中。
2. 根据 numConnectionsPerPeer 的大小（使用「spark.+ 模块名 +.io.numConnections-PerPeer」属性配置），从 ClientPool 中随机选择一个 TransportClient。
3. 如果 ClientPool 的 clients 数组中在随机产生的索引位置不存在 TransportClient 或者 TransportClient 没有激活，则进入第 5 步，否则对此 TransportClient 进行第 4 步的检查。
4. 更新 TransportClient 的 channel 中配置的 TransportChannelHandler 的最后一次使用时间，确保 channel 没有超时，然后检查 TransportClient 是否是激活状态，最后返回此 TransportClient 给调用方。
5. 由于缓存中没有 TransportClient 可用，于是调用 InetSocketAddress 的构造器创建 InetSocketAddress 对象（直接使用 InetSocketAddress 的构造器创建 InetSocketAddress 会进行域名解析），在这一步骤多个线程可能会产生竞态条件（由于没有同步处理，所以多个线程极有可能同时执行到此处，都发现缓存中没有 TransportClient 可用，于是都使用 InetSocketAddress 的构造器创建 InetSocketAddress）。
6. 第 5 步创建 InetSocketAddress 的过程中产生的竞态条件如果不妥善处理，会产生线程安全问题，所以到了 ClientPool 的 locks 数组发挥作用的时候了。按照随机产生的数组索引，locks 数组中的锁对象可以对 clients 数组中的 TransportClient 一对一进行同步。即便之前产生了竞态条件，但是在这一步只能有一个线程进入临界区。在临界区内，先进入的线程调用重载的 createClient 方法创建 TransportClient 对象并放入 ClientPool 的 clients 数组中。当率先进入临界区的线程退出临界区后，其他线程才能进入，此时发现 ClientPool 的 clients 数组中已经存在了 TransportClient 对象，那么将不再创建 TransportClient，而是直接使用它。

代码清单 3-12 的整个执行过程实际解决了 TransportClient 缓存的使用及 createClient 方法的线程安全问题，并没有涉及创建 TransportClient 的实现。TransportClient 的创建过程在重载的 createClient 方法（见代码清单 3-13）中实现。

代码清单 3-13　创建 TransportClient

private TransportClient createClient(InetSocketAddress address)  
 throws IOException, InterruptedException {  
 logger.debug("Creating new connection to {}", address);  
 // 构建根引导程序Bootstrap并对其进行配置  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(workerGroup)  
 .channel(socketChannelClass)  
 .option(ChannelOption.TCP\_NODELAY, true)  
 .option(ChannelOption.SO\_KEEPALIVE, true)  
 .option(ChannelOption.CONNECT\_TIMEOUT\_MILLIS, conf.connectionTimeoutMs())  
 .option(ChannelOption.ALLOCATOR, pooledAllocator);  
  
 final AtomicReference<TransportClient> clientRef = new AtomicReference<>();  
 final AtomicReference<Channel> channelRef = new AtomicReference<>();  
 // 为根引导程序设置管道初始化回调函数  
 bootstrap.handler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 public void initChannel(SocketChannel ch) {  
 TransportChannelHandler clientHandler = context.initializePipeline(ch);  
 clientRef.set(clientHandler.getClient());  
 channelRef.set(ch);  
 }  
 });  
  
 long preConnect = System.nanoTime();  
 ChannelFuture cf = bootstrap.connect(address);// 使用根引导程序连接远程服务器  
 if (!cf.await(conf.connectionTimeoutMs())) {  
 throw new IOException(  
 String.format("Connecting to %s timed out (%s ms)", address, conf.connectionTimeoutMs()));  
 } else if (cf.cause() != null) {  
 throw new IOException(String.format("Failed to connect to %s", address), cf.cause());  
 }  
  
 TransportClient client = clientRef.get();  
 Channel channel = channelRef.get();  
 assert client != null : "Channel future completed successfully with null client";  
  
 // Execute any client bootstraps synchronously before marking the Client as successful.  
 long preBootstrap = System.nanoTime();  
 logger.debug("Connection to {} successful, running bootstraps...", address);  
 try {  
 for (TransportClientBootstrap clientBootstrap : clientBootstraps) {  
 clientBootstrap.doBootstrap(client, channel);// 给TransportClient设置客户端引导程序  
 }  
 } catch (Exception e) { // catch non-RuntimeExceptions too as bootstrap may be written in Scala  
 long bootstrapTimeMs = (System.nanoTime() - preBootstrap) / 1000000;  
 logger.error("Exception while bootstrapping client after " + bootstrapTimeMs + " ms", e);  
 client.close();  
 throw Throwables.propagate(e);  
 }  
 long postBootstrap = System.nanoTime();  
  
 logger.info("Successfully created connection to {} after {} ms ({} ms spent in bootstraps)",  
 address, (postBootstrap - preConnect) / 1000000, (postBootstrap - preBootstrap) / 1000000);  
  
 return client;  
}

从代码清单 3-13 得知，真正创建 TransportClient 的步骤如下。

1. 构建根引导程序 Bootstrap 并对其进行配置。
2. 为根引导程序设置管道初始化回调函数，此回调函数将调用 TransportContext 的 initializePipeline 方法初始化 Channel 的 pipeline。
3. 使用根引导程序连接远程服务器，当连接成功对管道初始化时会回调初始化回调函数，将 TransportClient 和 Channel 对象分别设置到原子引用 clientRef 与 channelRef 中。
4. 给 TransportClient 设置客户端引导程序，即设置 TransportClientFactory 中的 Transport-ClientBootstrap 列表。
5. 返回此 TransportClient 对象。

### 3.2.3　RPC 服务端 TransportServer

TransportServer 是 RPC 框架的服务端，可提供高效、低级别的流服务。在说明图 3-1 中的记号 ② 时提到过，TransportContext 的 createServer 方法用于创建 TransportServer，其实现如代码清单 3-14 所示。

代码清单 3-14　创建 RPC 服务端

public TransportServer createServer(int port, List<TransportServerBootstrap> bootstraps) {  
 return new TransportServer(this, null, port, rpcHandler, bootstraps);  
}  
  
public TransportServer createServer(  
 String host, int port, List<TransportServerBootstrap> bootstraps) {  
 return new TransportServer(this, host, port, rpcHandler, bootstraps);  
}  
  
public TransportServer createServer(List<TransportServerBootstrap> bootstraps) {  
 return createServer(0, bootstraps);  
}  
  
public TransportServer createServer() {  
 return createServer(0, Lists.<TransportServerBootstrap>newArrayList());  
}

代码清单 3-14 中列出了 4 个名为 createServer 的重载方法，但是它们最终调用了 TransportServer 的构造器（见代码清单 3-15）来创建 TransportServer 实例。

代码清单 3-15　TransportServer 的构造器

public TransportServer(  
 TransportContext context,  
 String hostToBind,  
 int portToBind,  
 RpcHandler appRpcHandler,  
 List<TransportServerBootstrap> bootstraps) {  
 this.context = context;  
 this.conf = context.getConf();  
 this.appRpcHandler = appRpcHandler;  
 this.bootstraps = Lists.newArrayList(Preconditions.checkNotNull(bootstraps));  
  
 try {  
 init(hostToBind, portToBind);  
 } catch (RuntimeException e) {  
 JavaUtils.closeQuietly(this);  
 throw e;  
 }  
}

TransportServer 的构造器中的各个变量如下。

* context：参数传递的 TransportContext 的引用。
* conf：指 TransportConf，这里通过调用 TransportContext 的 getConf 获取。
* appRpcHandler：RPC 请求处理器 RpcHandler。
* bootstraps：参数传递的 TransportServerBootstrap 列表。

TransportServer 的构造器（见代码清单 3-15）中调用了 init 方法，init 方法用于对 TransportServer 进行初始化，如代码清单 3-16 所示。

代码清单 3-16　TransportServer 的初始化

private void init(String hostToBind, int portToBind) {  
 // 根据Netty的API文档，Netty服务端需同时创建bossGroup和workerGroup  
 IOMode ioMode = IOMode.valueOf(conf.ioMode());  
 EventLoopGroup bossGroup =  
 NettyUtils.createEventLoop(ioMode, conf.serverThreads(), conf.getModuleName() + "-server");  
 EventLoopGroup workerGroup = bossGroup;  
 // 创建一个汇集ByteBuf但对本地线程缓存禁用的分配器  
 PooledByteBufAllocator allocator = NettyUtils.createPooledByteBufAllocator(  
 conf.preferDirectBufs(), true /\* allowCache \*/, conf.serverThreads());  
 // 创建Netty的服务端根引导程序并对其进行配置  
 bootstrap = new ServerBootstrap()  
 .group(bossGroup, workerGroup)  
 .channel(NettyUtils.getServerChannelClass(ioMode))  
 .option(ChannelOption.ALLOCATOR, allocator)  
 .childOption(ChannelOption.ALLOCATOR, allocator);  
  
 if (conf.backLog() > 0) {  
 bootstrap.option(ChannelOption.SO\_BACKLOG, conf.backLog());  
 }  
 if (conf.receiveBuf() > 0) {  
 bootstrap.childOption(ChannelOption.SO\_RCVBUF, conf.receiveBuf());  
 }  
 if (conf.sendBuf() > 0) {  
 bootstrap.childOption(ChannelOption.SO\_SNDBUF, conf.sendBuf());  
 }  
 // 为根引导程序设置管道初始化回调函数  
 bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception {  
 RpcHandler rpcHandler = appRpcHandler;  
 for (TransportServerBootstrap bootstrap : bootstraps) {  
 rpcHandler = bootstrap.doBootstrap(ch, rpcHandler);  
 }  
 context.initializePipeline(ch, rpcHandler);  
 }  
 });  
 // 给根引导程序绑定Socket的监听端口  
 InetSocketAddress address = hostToBind == null ?  
 new InetSocketAddress(portToBind): new InetSocketAddress(hostToBind, portToBind);  
 channelFuture = bootstrap.bind(address);  
 channelFuture.syncUninterruptibly();  
  
 port = ((InetSocketAddress) channelFuture.channel().localAddress()).getPort();  
 logger.debug("Shuffle server started on port: {}", port);  
}

代码清单 3-16 中 TransportServer 初始化的步骤如下。

**小贴士：**根据 Netty 的 API 文档，Netty 服务端需同时创建 bossGroup 和 workerGroup。

1. 创建 bossGroup 和 workerGroup。
2. 创建一个汇集 ByteBuf 但对本地线程缓存禁用的分配器。
3. 调用 Netty 的 API 创建 Netty 的服务端根引导程序并对其进行配置。
4. 为根引导程序设置管道初始化回调函数，此回调函数首先设置 TransportServer-Bootstrap 到根引导程序中，然后调用 TransportContext 的 initializePipeline 方法初始化 Channel 的 pipeline。
5. 给根引导程序绑定 Socket 的监听端口，最后返回监听的端口。

https://pic2.zhimg.com/v2-bc03cc5b7cff02382e9bccbbb3a58521_r.jpg

##### 提示

代码清单 3-16 中使用了 NettyUtils 工具类的很多方法，在附录 G 中有对它们的详细介绍。EventLoopGroup、PooledByteBufAllocator、ServerBootstrap 都是 Netty 提供的 API，对于它们的更多介绍，请访问。[**http://netty.io/**](http://netty.io/)

### 3.2.4　管道初始化

在代码清单 3-13 创建 TransportClient 和代码清单 3-16 对 TransportServer 初始化的实现中，都在管道初始化回调函数中调用了 TransportContext 的 initializePipeline 方法，initializePipeline 方法（见代码清单 3-17）将调用 Netty 的 API 对管道初始化。

代码清单 3-17　管道初始化

public TransportChannelHandler initializePipeline(  
 SocketChannel channel,  
 RpcHandler channelRpcHandler) {  
 try {  
 TransportChannelHandler channelHandler = createChannelHandler(channel, channelRpcHandler);  
 channel.pipeline()  
 .addLast("encoder", ENCODER)  
 .addLast(TransportFrameDecoder.HANDLER\_NAME, NettyUtils.createFrame Decoder())  
 .addLast("decoder", DECODER)  
 .addLast("idleStateHandler", new IdleStateHandler(0, 0, conf.connection TimeoutMs() / 1000))  
 .addLast("handler", channelHandler);  
 return channelHandler;  
 } catch (RuntimeException e) {  
 logger.error("Error while initializing Netty pipeline", e);  
 throw e;  
 }  
}

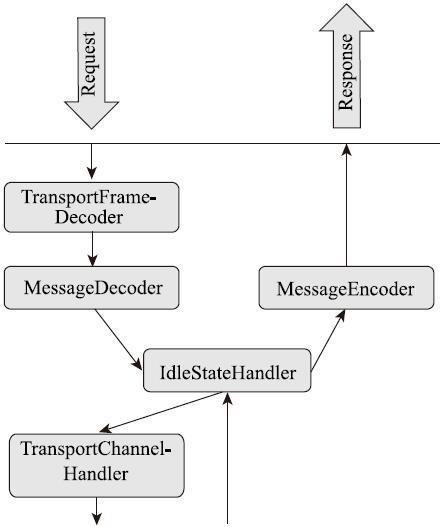
根据代码清单 3-17，initializePipeline 方法的执行步骤如下。

1）调用 createChannelHandler 方法创建 TransportChannelHandler，从 createChannel-Handler 的实现（见代码清单 3-18）中可以看到，真正创建 TransportClient 是在这里发生的。通过 TransportClient 的构造过程看到 RpcHandler 与 TransportClient 毫无关系，Transport-Client 只使用了 TransportResponseHandler。TransportChannelHandler 在服务端将代理 Transport-RequestHandler 对请求消息进行处理，并在客户端代理 TransportResponseHandler 对响应消息进行处理。

代码清单 3-18　创建 TransportChannelHandler

private TransportChannelHandler createChannelHandler(Channel channel, RpcHandler rpcHandler) {  
 TransportResponseHandler responseHandler = new TransportResponseHandler(channel);  
 TransportClient client = new TransportClient(channel, responseHandler);  
 TransportRequestHandler requestHandler = new TransportRequestHandler(channel, client,  
 rpcHandler);  
 return new TransportChannelHandler(client, responseHandler, requestHandler,  
 conf.connectionTimeoutMs(), closeIdleConnections);  
}

2）对管道进行设置，这里的 ENCODER（即 MessageEncoder）派生自 Netty 的 ChannelOutboundHandler 接口；DECODER（即 MessageDecoder）、TransportChannelHandler 及 TransportFrameDecoder（由工具类 NettyUtils 的静态方法 createFrameDecoder 创建）派生自 Netty 的 ChannelInboundHandler 接口；IdleState Handler 同时实现了 ChannelOutbound-Handler 和 ChannelInboundHandler 接口。根据 Netty 的 API 行为，通过 addLast 方法注册多个 Handler 时，ChannelInboundHandler 按照注册的先后顺序执行，ChannelOutboundHandler 按照注册的先后顺序逆序执行，因此在管道两端（无论是服务端还是客户端）处理请求和响应的流程如图 3-3 所示。

图 3-3　管道处理请求和响应的流程图

### 3.2.5　TransportChannelHandler 详解

TransportChannelHandler 实现了 Netty 的 ChannelInboundHandler㊟，以便对 Netty 管道中的消息进行处理。图 3-3 中的这些 Handler（除了 MessageEncoder）由于都实现了 ChannelInbound-Handler 接口，作为自定义的 ChannelInbound-Handler，所以都要重写 channelRead 方法。Netty 框架使用工作链模式来对每个 ChannelInboundHandler 的实现类的 channelRead 方法进行链式调用。TransportChannelHandler 实现的 channelRead 方法如代码清单 3-19 所示。

代码清单 3-19　TransportChannelHandler 的 channelRead 实现

@Override  
public void channelRead(ChannelHandlerContext ctx, Object request) throws Exception {  
 if (request instanceof RequestMessage) {  
 requestHandler.handle((RequestMessage) request);  
 } else if (request instanceof ResponseMessage) {  
 responseHandler.handle((ResponseMessage) request);  
 } else {  
 ctx.fireChannelRead(request);  
 }  
}

从代码清单 3-19 看到，当 TransportChannelHandler 读取的 request 是 RequestMessage 时，则将此消息的处理进一步交给 TransportRequestHandler，当读取的 request 是 Response-Message 时，则将此消息的处理进一步交给 TransportResponseHandler。

**1.MessageHandler 的继承体系**

TransportRequestHandler 与 TransportResponseHandler 都继承自抽象类 MessageHandler，MessageHandler 定义了子类的规范，详细定义如代码清单 3-20 所示。

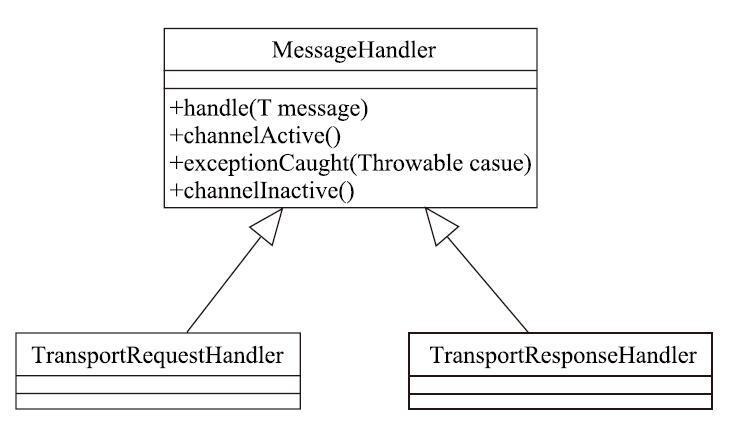
代码清单 3-20　MessageHandler 规范

public abstract class MessageHandler<T extends Message> {  
 public abstract void handle(T message) throws Exception;  
 public abstract void channelActive();  
 public abstract void exceptionCaught(Throwable cause);  
 public abstract void channelInactive();  
}

MessageHandler 中定义的各个方法的作用分别如下。

* handle：用于对接收到的单个消息进行处理。
* channelActive：当 channel 激活时调用。
* exceptionCaught：当捕获到 channel 发生异常时调用。
* channelInactive：当 channel 非激活时调用。

Spark 中 MessageHandler 类的继承体系如图 3-4 所示。

图 3-4　MessageHandler 类的继承体系

**2.Message 的继承体系**

根据代码清单 3-20 我们知道，MessageHandler 同时也是一个 Java 泛型类，其子类能处理的消息都派生自接口 Message。Message 的定义如代码清单 3-21 所示。

代码清单 3-21　Message 的定义

public interface Message extends Encodable {  
 Type type();  
 ManagedBuffer body();  
 boolean isBodyInFrame();

Message 中定义的三个接口方法的作用分别如下。

* type：返回消息的类型。
* body：返回消息中可选的内容体。
* isBodyInFrame：用于判断消息的主体是否包含在消息的同一帧中。

Message 接口继承了 Encodable 接口，Encodable 的定义如代码清单 3-22 所示。

代码清单 3-22　Encodable 的定义

public interface Encodable {  
 int encodedLength();  
 void encode(ByteBuf buf);  
}

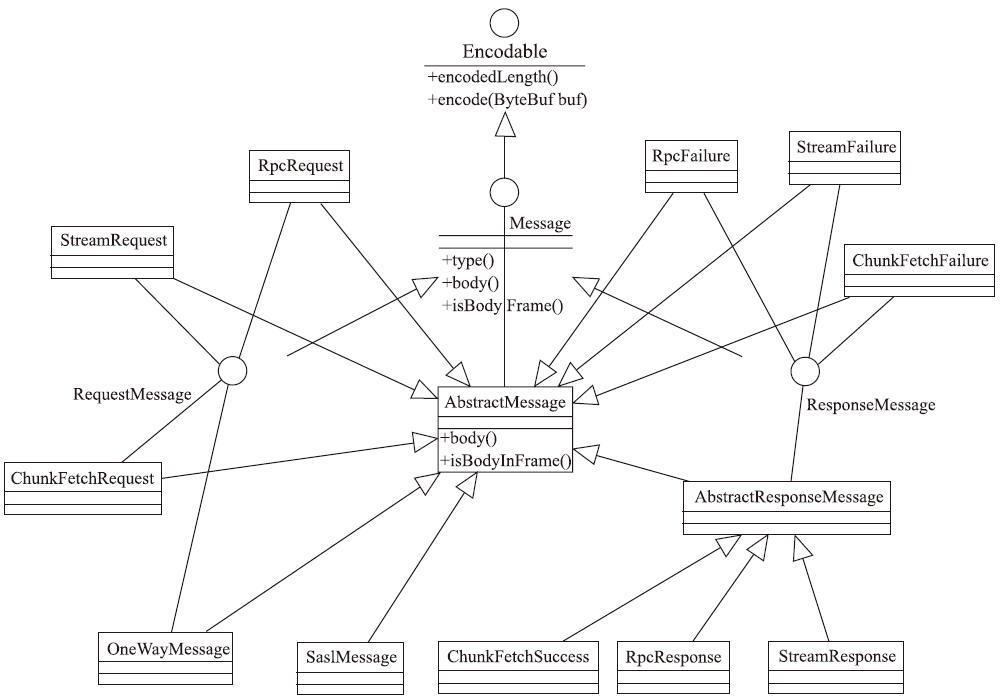
实现 Encodable 接口的类将可以转换到一个 ByteBuf 中，多个对象将被存储到预先分配的单个 ByteBuf，所以这里的 encodedLength 用于返回转换的对象数量。下面一起来看看 Message 的类继承体系，如图 3-5 所示。

从图 3-5 看到，最终的消息实现类都直接或间接地实现了 RequestMessage 或 Response-Message 接口，其中 RequestMessage 的具体实现有 4 种，分别如下。

* ChunkFetchRequest：请求获取流的单个块的序列。
* RpcRequest：此消息类型由远程的 RPC 服务端进行处理，是一种需要服务端向客户端回复的 RPC 请求信息类型。
* OneWayMessage：此消息也需要由远程的 RPC 服务端进行处理，与 RpcRequest 不同的是，不需要服务端向客户端回复。
* StreamRequest：此消息表示向远程的服务发起请求，以获取流式数据。

由于 OneWayMessage 不需要响应，所以 ResponseMessage 对于成功或失败状态的实现各有 3 种，分别如下。

* ChunkFetchSuccess：处理 ChunkFetchRequest 成功后返回的消息。
* ChunkFetchFailure：处理 ChunkFetchRequest 失败后返回的消息。
* RpcResponse：处理 RpcRequest 成功后返回的消息。
* RpcFailure：处理 RpcRequest 失败后返回的消息。

图 3-5　Message 的类继承体系

* StreamResponse：处理 StreamRequest 成功后返回的消息。
* StreamFailure：处理 StreamRequest 失败后返回的消息。

**3.ManagedBuffer 的继承体系**

回头再看看代码清单 3-21 中对 body 接口的定义，可以看到，其返回内容体的类型为 ManagedBuffer。ManagedBuffer 提供了由字节构成数据的不可变视图（也就是说，ManagedBuffer 并不存储数据，也不是数据的实际来源，这同关系型数据库的视图类似）。我们先来看看抽象类 ManagedBuffer 中对行为的定义，如代码清单 3-23 所示。

代码清单 3-23　ManagedBuffer 的定义

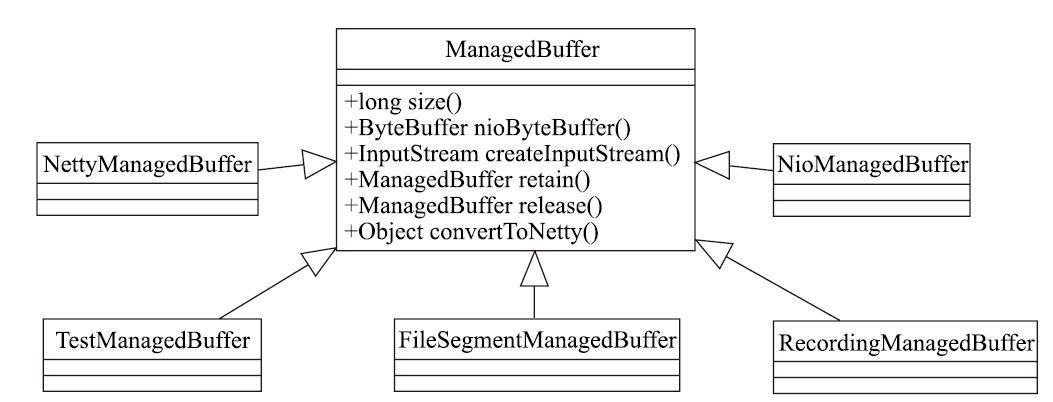
public abstract class ManagedBuffer {  
 public abstract long size();  
 public abstract ByteBuffer nioByteBuffer() throws IOException;  
 public abstract InputStream createInputStream() throws IOException;  
 public abstract ManagedBuffer retain();  
 public abstract ManagedBuffer release();  
 public abstract Object convertToNetty() throws IOException;  
}

ManagedBuffer 中定义了 6 个方法，分别如下。

* size：返回数据的字节数。
* nioByteBuffer：将数据按照 NIO 的 ByteBuffer 类型返回。
* createInputStream：将数据按照 InputStream 返回。
* retain：当有新的使用者使用此视图时，增加引用此视图的引用数。
* release：当有使用者不再使用此视图时，减少引用此视图的引用数。当引用数为 0 时释放缓冲区。
* convertToNetty：将缓冲区的数据转换为 Netty 的对象，用来将数据写到外部。此方法返回的数据类型要么是 io.netty.buffer.ByteBuf，要么是 io.netty.channel.File-Region。

ManagedBuffer 的具体实现有很多，我们可以通过图 3-6 来了解。

图 3-6 中列出了 ManagedBuffer 的 5 个实现类，其中 TestManagedBuffer 和 Recording-ManagedBuffer 用于测试。NettyManagedBuffer 中的缓冲为 io.netty.buffer.ByteBuf，NioManaged-Buffer 中的缓冲为 java.nio.ByteBuffer。NettyManagedBuffer 和 NioManagedBuffer 的实现都非常简单，留给读者自行阅读。本节挑选 FileSegmentManagedBuffer 作为 ManagedBuffer 具体实现的例子进行介绍。

图 3-6　ManagedBuffer 的继承体系

FileSegmentManagedBuffer 的作用为获取一个文件中的一段，它一共有 4 个由 final 修饰的属性，全部都通过 FileSegmentManagedBuffer 的构造器传入属性值，这 4 个属性分别如下。

* conf：也就是 TransportConf。
* file：所要读取的文件。
* offset：所要读取文件的偏移量。
* length：所要读取的长度。

下面将逐个介绍 FileSegmentManagedBuffer 对于 ManagedBuffer 的实现。

* NIO 方式读取文件。FileSegmentManagedBuffer 实现的 nioByteBuffer 方法如代码清单 3-24 所示。

代码清单 3-24　nioByteBuffer 方法的实现

@Override  
public ByteBuffer nioByteBuffer() throws IOException {  
 FileChannel channel = null;  
 try {  
 channel = new RandomAccessFile(file, "r").getChannel();  
 if (length < conf.memoryMapBytes()) {  
 ByteBuffer buf = ByteBuffer.allocate((int) length);  
 channel.position(offset);  
 while (buf.remaining() != 0) {  
 if (channel.read(buf) == -1) {  
 throw new IOException(String.format("Reached EOF before filling buffer\n" +  
 "offset=%s\nfile=%s\nbuf.remaining=%s",  
 offset, file.getAbsoluteFile(), buf.remaining()));  
 }  
 }  
 buf.flip();  
 return buf;  
 } else {  
 return channel.map(FileChannel.MapMode.READ\_ONLY, offset, length);  
 }  
 } catch (IOException e) {  
 try {  
 if (channel != null) {  
 long size = channel.size();  
 throw new IOException("Error in reading " + this + " (actual file length " + size + ")",  
 e);  
 }  
 } catch (IOException ignored) {  
 // ignore  
 }  
 throw new IOException("Error in opening " + this, e);  
 } finally {  
 JavaUtils.closeQuietly(channel);  
 }  
}

nioByteBuffer 的实现还是很简单的，主要利用 RandomAccessFile 获取 FileChannel，然后使用 java.nio.ByteBuffer 和 FileChannel 的 API 将数据写入缓冲区 java.nio.ByteBuffer 中。

* 文件流方式读取文件。FileSegmentManagedBuffer 实现的 createInputStream 方法如代码清单 3-25 所示。

代码清单 3-25　createInputStream 的实现

@Override  
public InputStream createInputStream() throws IOException {  
 FileInputStream is = null;  
 try {  
 is = new FileInputStream(file);  
 ByteStreams.skipFully(is, offset);  
 return new LimitedInputStream(is, length);  
 } catch (IOException e) {  
 try {  
 if (is != null) {  
 long size = file.length();  
 throw new IOException("Error in reading " + this + " (actual file length " + size + ")",  
 e);  
 }  
 } catch (IOException ignored) {  
 // ignore  
 } finally {  
 JavaUtils.closeQuietly(is);  
 }  
 throw new IOException("Error in opening " + this, e);  
 } catch (RuntimeException e) {  
 JavaUtils.closeQuietly(is);  
 throw e;  
 }  
}

createInputStream 的实现还是很简单的，这里不过多介绍。

* 将数据转换为 Netty 对象。FileSegmentManagedBuffer 实现的 convertToNetty 方法如代码清单 3-26 所示。

代码清单 3-26　convertToNetty 的实现

@Override  
public Object convertToNetty() throws IOException {  
 if (conf.lazyFileDescriptor()) {  
 return new DefaultFileRegion(file, offset, length);  
 } else {  
 FileChannel fileChannel = new FileInputStream(file).getChannel();  
 return new DefaultFileRegion(fileChannel, offset, length);  
 }  
}

* 其他方法的实现。其他方法由于实现非常简单，所以这里就不一一列出了，感兴趣的读者可以自行查阅。

### 3.2.6　服务端 RpcHandler 详解

由于 TransportRequestHandler 实际是把请求消息交给 RpcHandler 进行进一步处理，所以这里对 RpcHandler 首先做个介绍。RpcHandler 是一个抽象类，定义了一些 RPC 处理器的规范，其主要实现如代码清单 3-27 所示。

代码清单 3-27　RpcHandler 的实现

public abstract class RpcHandler {  
private static final RpcResponseCallback ONE\_WAY\_CALLBACK = new OneWayRpc-Callback();  
public abstract void receive(  
 TransportClient client,  
 ByteBuffer message,  
 RpcResponseCallback callback);  
public abstract StreamManager getStreamManager();  
public void receive(TransportClient client, ByteBuffer message) {  
 receive(client, message, ONE\_WAY\_CALLBACK);  
}  
public void channelActive(TransportClient client) { }  
public void channelInactive(TransportClient client) { }  
public void exceptionCaught(Throwable cause, TransportClient client) { }  
private static class OneWayRpcCallback implements RpcResponseCallback {  
 private static final Logger logger = LoggerFactory.getLogger(OneWayRpcCallback.class);  
 @Override  
 public void onSuccess(ByteBuffer response) {  
 logger.warn("Response provided for one-way RPC.");  
 }  
 @Override  
 public void onFailure(Throwable e) {  
 logger.error("Error response provided for one-way RPC.", e);  
 }  
 }  
}

代码清单 3-27 中 RpcHandler 的各个方法的作用如下。

* receive：这是一个抽象方法，用来接收单一的 RPC 消息，具体处理逻辑需要子类去实现。receive 接收 3 个参数，分别是 TransportClient、ByteBuffer 和 RpcResponse-Callback。RpcResponseCallback 用于对请求处理结束后进行回调，无论处理结果是成功还是失败，RpcResponseCallback 都会被调用一次。RpcResponseCallback 的接口定义如下。

public interface RpcResponseCallback {  
 void onSuccess(ByteBuffer response);  
 void onFailure(Throwable e);  
}

* 重载的 receive：只接收 TransportClient 和 ByteBuffer 两个参数，RpcResponse-Callback 为默认的 ONE\_WAY\_CALLBACK，其类型为 OneWayRpcCallback，从代码清单 3-27 中 OneWayRpcCallback 的实现可以看出，其 onSuccess 和 onFailure 只是打印日志，并没有针对客户端做回复处理。
* channelActive：当与给定客户端相关联的 channel 处于活动状态时调用。
* channelInactive：当与给定客户端相关联的 channel 处于非活动状态时调用。
* exceptionCaught：当 channel 产生异常时调用。
* getStreamManager：获取 StreamManager，StreamManager 可以从流中获取单个的块，因此它也包含着当前正在被 TransportClient 获取的流的状态。

介绍完 RpcHandler，现在回到 TransportRequestHandler 的处理过程。Transport-RequestHandler 处理以上 4 种 RequestMessage 的实现如代码清单 3-28 所示。

代码清单 3-28　TransportRequestHandler 的 handle 方法

@Override  
public void handle(RequestMessage request) {  
 if (request instanceof ChunkFetchRequest) {  
 processFetchRequest((ChunkFetchRequest) request);  
 } else if (request instanceof RpcRequest) {  
 processRpcRequest((RpcRequest) request);  
 } else if (request instanceof OneWayMessage) {  
 processOneWayMessage((OneWayMessage) request);  
 } else if (request instanceof StreamRequest) {  
 processStreamRequest((StreamRequest) request);  
 } else {  
 throw new IllegalArgumentException("Unknown request type: " + request);  
 }  
}

结合代码清单 3-28，下面详细分析这 4 种类型请求的处理过程。

**1.处理块获取请求**

processFetchRequest 方法用于处理 ChunkFetchRequest 类型的消息，其实现如代码清单 3-29 所示。

代码清单 3-29　processFetchRequest 的实现

private void processFetchRequest(final ChunkFetchRequest req) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Received req from {} to fetch block {}", getRemoteAddress (channel),  
 req.streamChunkId);  
 }  
 ManagedBuffer buf;  
 try {  
 streamManager.checkAuthorization(reverseClient, req.streamChunkId.streamId);  
 streamManager.registerChannel(channel, req.streamChunkId.streamId);  
 buf = streamManager.getChunk(req.streamChunkId.streamId, req.streamChunkId.chunkIndex);  
 } catch (Exception e) {  
 logger.error(String.format("Error opening block %s for request from %s",  
 req.streamChunkId, getRemoteAddress(channel)), e);  
 respond(new ChunkFetchFailure(req.streamChunkId, Throwables.getStackTraceAsString(e)));  
 return;  
 }  
 respond(new ChunkFetchSuccess(req.streamChunkId, buf));  
}

代码清单 3-29 中的 streamManager 是通过调用 RpcHandler 的 getStreamManager 方法获取的 StreamManager。processFetchRequest 的处理都依托于 RpcHandler 的 Stream-Manager，其处理步骤如下。

1. 调用 StreamManager 的 checkAuthorization 方法，校验客户端是否有权限从给定的流中读取。
2. 调用 StreamManager 的 registerChannel 方法，将一个流和一条（只能是一条）客户端的 TCP 连接关联起来，这可以保证对于单个的流只会有一个客户端读取。流关闭之后就永远不能够重用了。
3. 调用 StreamManager 的 getChunk 方法，获取单个的块（块被封装为 Managed-Buffer）。由于单个的流只能与单个的 TCP 连接相关联，因此 getChunk 方法不能为了某个特殊的流而并行调用。
4. 将 ManagedBuffer 和流的块 Id 封装为 ChunkFetchSuccess 后，调用 respond 方法返回给客户端。

有关 StreamManager 的具体实现，读者可以参考 5.3.5 节介绍的 NettyStreamManager 和 6.9.2 节介绍的 NettyBlockRpcServer 中的 OneForOneStreamManager。

**2.处理 RPC 请求**

processRpcRequest 方法用于处理 RpcRequest 类型的消息，其实现如代码清单 3-30 所示。

代码清单 3-30　processRpcRequest 的实现

private void processRpcRequest(final RpcRequest req) {  
 try {  
 rpcHandler.receive(reverseClient, req.body().nioByteBuffer(), new RpcResponse Callback() {  
 @Override  
 public void onSuccess(ByteBuffer response) {  
 respond(new RpcResponse(req.requestId, new NioManagedBuffer(response)));  
 }  
  
 @Override  
 public void onFailure(Throwable e) {  
 respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));  
 }  
 });  
 } catch (Exception e) {  
 logger.error("Error while invoking RpcHandler#receive() on RPC id " + req.requestId, e);  
 respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));  
 } finally {  
 req.body().release();  
 }  
}

代码清单 3-30 中将 RpcRequest 消息的内容体、发送消息的客户端及一个 RpcResponse-Callback 类型的匿名内部类作为参数传递给了 RpcHandler 的 receive 方法。这就是说真正用于处理 RpcRequest 消息的是 RpcHandler，而非 TransportRequestHandler。由于 RpcHandler 是抽象类（见代码清单 3-27），其 receive 方法也是抽象方法，所以具体的操作将由 RpcHandler 的实现了 receive 方法的子类来完成。所有继承 RpcHandler 的子类都需要在其 receive 方法的具体实现中回调 RpcResponseCallback 的 onSuccess（处理成功时）或者 onFailure（处理失败时）方法。从 RpcResponseCallback 的实现来看，无论处理结果成功还是失败，都将调用 respond 方法对客户端进行响应。

**3.处理流请求**

processStreamRequest 方法用于处理 StreamRequest 类型的消息，其实现如代码清单 3-31 所示。

代码清单 3-31　processStreamRequest 的实现

private void processStreamRequest(final StreamRequest req) {  
 ManagedBuffer buf;  
 try {  
 buf = streamManager.openStream(req.streamId);// 将获取到的流数据封装为ManagedBuffer  
 } catch (Exception e) {  
 logger.error(String.format(  
 "Error opening stream %s for request from %s", req.streamId, getRemoteAddress(channel)), e);  
 respond(new StreamFailure(req.streamId, Throwables.getStackTrace-AsString(e)));  
 return;  
 }  
 if (buf != null) {  
 respond(new StreamResponse(req.streamId, buf.size(), buf));  
 } else {  
 respond(new StreamFailure(req.streamId, String.format(  
 "Stream '%s' was not found.", req.streamId)));  
 }  
}

代码清单 3-31 中也使用了 RpcHandler 的 StreamManager，其处理步骤如下。

1. 调用 StreamManager 的 openStream 方法将获取到的流数据封装为 ManagedBuffer。
2. 当成功或失败时调用 respond 方法向客户端响应。

**4.处理无需回复的 RPC 请求**

processOneWayMessage 方法用于处理 StreamRequest 类型的消息，其实现如代码清单 3-32 所示。

代码清单 3-32　processOneWayMessage 的实现

private void processOneWayMessage(OneWayMessage req) {  
 try {  
 rpcHandler.receive(reverseClient, req.body().nioByteBuffer());  
 } catch (Exception e) {  
 logger.error("Error while invoking RpcHandler#receive() for one-way message.", e);  
 } finally {  
 req.body().release();  
 }  
}

processOneWayMessage 方法的实现与 processRpcRequest 非常相似，区别在于 processOneWayMessage 调用了代码清单 3-27 中 ONE\_WAY\_CALLBACK 的 receive 方法，因而 processOneWayMessage 在处理完 RPC 请求后不会对客户端作出响应。

从以上 4 种处理的分析可以看出，最终的处理都由 RpcHandler 及其内部组件完成。除了 OneWayMessage 的消息外，其余三种消息都是最终调用 respond 方法响应客户端，其实现如代码清单 3-33 所示。

代码清单 3-33　respond 的实现

private void respond(final Encodable result) {  
 final SocketAddress remoteAddress = channel.remoteAddress();  
 channel.writeAndFlush(result).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 logger.trace("Sent result {} to client {}", result, remoteAddress);  
 } else {  
 logger.error(String.format("Error sending result %s to %s; closing connection",  
 result, remoteAddress), future.cause());  
 channel.close();  
 }  
 }  
 }  
 );  
}

可以看到 respond 方法中实际调用了 Channel 的 writeAndFlush 方法㊟来响应客户端。

### 3.2.7　服务端引导程序 TransportServerBootstrap

TransportServer 的构造器（见代码清单 3-15）中的 bootstraps 是 TransportServerBootstrap 的列表。接口 TransportServerBootstrap 定义了服务端引导程序的规范，服务端引导程序旨在当客户端与服务端建立连接之后，在服务端持有的客户端管道上执行的引导程序。TransportServerBootstrap 的定义如代码清单 3-34 所示。

代码清单 3-34　TransportServerBootstrap 的定义

public interface TransportServerBootstrap {  
 RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler);  
}

TransportServerBootstrap 的 doBootstrap 方法将对服务端的 RpcHandler 进行代理，接收客户端的请求。TransportServerBootstrap 有 SaslServerBootstrap 和 EncryptionChecker-Bootstrap 两个实现类。为了更清楚地说明 TransportServerBootstrap 的意义，我们以 SaslServer-Bootstrap 为例，来讲解其实现（见代码清单 3-35）。

代码清单 3-35　SaslServerBootstrap 的 doBootstrap 实现

public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {  
 return new SaslRpcHandler(conf, channel, rpcHandler, secretKeyHolder);  
}

根据代码清单 3-35 我们知道，SaslServerBootstrap 的 doBootstrap 方法实际创建了 SaslRpcHandler，SaslRpcHandler 负责对管道进行 SASL（Simple Authentication and Security Layer）加密。SaslRpcHandler 本身也继承了 RpcHandler，所以我们重点来看其 receive 方法的实现，如代码清单 3-36 所示。

代码清单 3-36　SaslRpcHandler 的 receive 方法

@Override  
public void receive(TransportClient client, ByteBuffer message, RpcResponseCallback callback) {  
 if (isComplete) {  
 // 将消息传递给SaslRpcHandler所代理的下游RpcHandler并返回  
 delegate.receive(client, message, callback);  
 return;  
 }  
  
 ByteBuf nettyBuf = Unpooled.wrappedBuffer(message);  
 SaslMessage saslMessage;  
 try {  
 saslMessage = SaslMessage.decode(nettyBuf);// 对客户端发送的消息进行SASL解密  
 } finally {  
 nettyBuf.release();  
 }  
  
 if (saslServer == null) {  
 // 如果saslServer还未创建，则需要创建SparkSaslServer  
 client.setClientId(saslMessage.appId);  
 saslServer = new SparkSaslServer(saslMessage.appId, secretKeyHolder,  
 conf.saslServerAlwaysEncrypt());  
 }  
 byte[] response;  
 try {  
 response = saslServer.response(JavaUtils.bufferToArray(// 使用saslServer处理已解密的消息  
 saslMessage.body().nioByteBuffer()));  
 } catch (IOException ioe) {  
 throw new RuntimeException(ioe);  
 }  
 callback.onSuccess(ByteBuffer.wrap(response));  
 if (saslServer.isComplete()) {  
 logger.debug("SASL authentication successful for channel {}", client);  
 isComplete = true;// SASL认证交换已经完成  
 if (SparkSaslServer.QOP\_AUTH\_CONF.equals(saslServer.getNegotiatedProperty(Sasl.QOP))) {  
 logger.debug("Enabling encryption for channel {}", client);  
 // 对管道进行SASL加密  
 SaslEncryption.addToChannel(channel, saslServer, conf.maxSaslEncrypted-BlockSize());  
 saslServer = null;  
 } else {  
 saslServer.dispose();  
 saslServer = null;  
 }  
 }  
}

根据代码清单 3-36，SaslRpcHandler 处理客户端消息的步骤如下。

1. 如果 SASL 认证交换已经完成（isComplete 等于 true），则将消息传递给 SaslRpc　Handler 所代理的下游 RpcHandler 并返回。
2. 如果 SASL 认证交换未完成（isComplete 等于 false），则对客户端发送的消息进行 SASL 解密。
3. 如果 saslServer 还未创建，则需要创建 SparkSaslServer。当 SaslRpcHandler 接收到客户端的第一条消息时会做此操作。
4. 使用 saslServer 处理已解密的消息，并将处理结果通过 RpcResponseCallback 的回调方法返回给客户端。
5. 如果 SASL 认证交换已经完成，则将 isComplete 置为 true。
6. 对管道进行 SASL 加密。

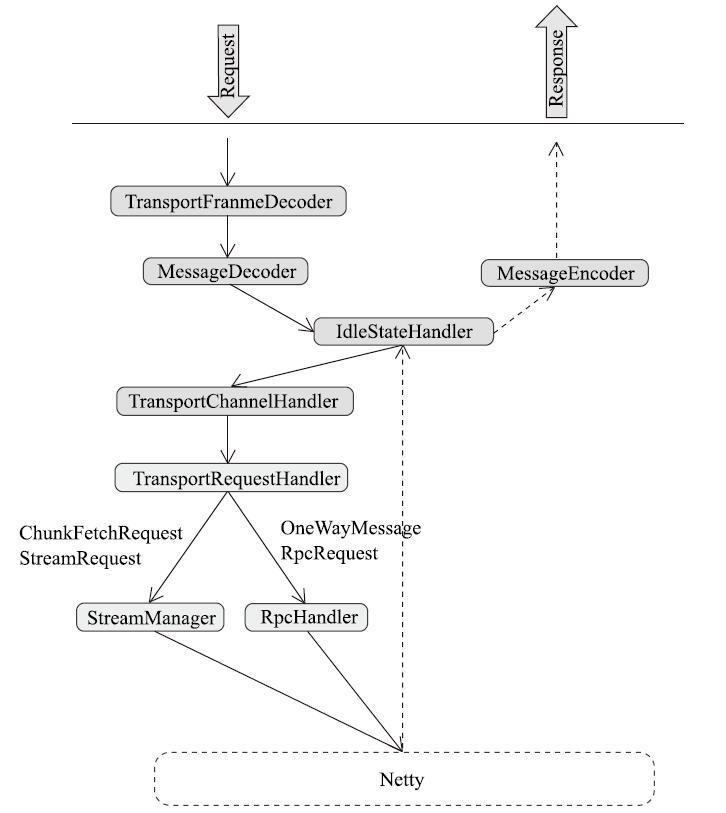
SaslServerBootstrap 是通过 SaslRpcHandler 对下游 RpcHandler 进行代理的一种 Tran sportServerBootstrap。EncryptionCheckerBootstrap 是另一种 TransportServerBootstrap 的实现，它通过将自身加入 Netty 的管道中实现引导，EncryptionCheckerBootstrap 的 doBootstrap 方法的实现如代码清单 3-37 所示。

代码清单 3-37　EncryptionCheckerBootstrap 的 doBootstrap 实现

@Override  
public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {  
 channel.pipeline().addFirst("encryptionChecker", this);  
 return rpcHandler;  
}

在详细介绍了 TransportChannelHandler 之后，我们就可以对图 3-3 进行扩展，把 Transport RequestHandler、TransportServerBootstrap 及 RpcHandler 的处理流程增加进来，如图 3-7 所示。

有读者可能会问，图 3-7 中并未见 TransportServerBootstrap 的身影。根据对 Transport-ServerBootstrap 的两种实现的举例，我们知道 TransportServerBootstrap 将可能存在于图中任何两个组件的箭头连线中间，起到引导、包装、代理的作用。

图 3-7　RPC 框架服务端处理请求、响应流程图

### 3.2.8　客户端 TransportClient 详解

在介绍完服务端 RpcHandler 对请求消息的处理之后，现在来看看客户端发送 RPC 请求的原理。我们在分析代码清单 3-18 中的 createChannelHandler 方法时，看到调用了 TransportClient 的构造器（见代码清单 3-38），其中 TransportResponseHandler 的引用将赋给 handler 属性。

代码清单 3-38　TransportClient 的构造器

public TransportClient(Channel channel, TransportResponseHandler handler) {  
 this.channel = Preconditions.checkNotNull(channel);  
 this.handler = Preconditions.checkNotNull(handler);  
 this.timedOut = false;  
}

TransportClient 一共有 5 个方法用于发送请求，分别如下。

1. fetchChunk：从远端协商好的流中请求单个块。
2. stream：使用流的 ID，从远端获取流数据。
3. sendRpc：向服务端发送 RPC 的请求，通过 At least Once Delivery 原则保证请求不会丢失。
4. sendRpcSync：向服务端发送异步的 RPC 的请求，并根据指定的超时时间等待响应。
5. send：向服务端发送 RPC 的请求，但是并不期望能获取响应，因而不能保证投递的可靠性。

本节只选择最常用的 sendRpc 和 fetchChunk 进行分析，其余实现都可以触类旁通。

**1.发送 RPC 请求**

sendRpc 方法的实现如代码清单 3-39 所示。

代码清单 3-39　sendRpc 的实现

public long sendRpc(ByteBuffer message, final RpcResponseCallback callback) {  
 final long startTime = System.currentTimeMillis();  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending RPC to {}", getRemoteAddress(channel));  
 }  
 // 使用UUID生成请求主键requestId  
 final long requestId = Math.abs(UUID.randomUUID().getLeastSignificantBits());  
 handler.addRpcRequest(requestId, callback);// 添加requestId与RpcResponseCallback的引用之间的关系  
 // 发送RPC请求  
 channel.writeAndFlush(new RpcRequest(requestId, new NioManagedBuffer　(message))).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 long timeTaken = System.currentTimeMillis() - startTime;  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending request {} to {} took {} ms", requestId,  
 getRemoteAddress(channel), timeTaken);  
 }  
 } else {  
 String errorMsg = String.format("Failed to send RPC %s to %s: %s", requestId,  
 getRemoteAddress(channel), future.cause());  
 logger.error(errorMsg, future.cause());  
 handler.removeRpcRequest(requestId);  
 channel.close();  
 try {  
 callback.onFailure(new IOException(errorMsg, future.cause()));  
 } catch (Exception e) {  
 logger.error("Uncaught exception in RPC response callback handler!", e);  
 }  
 }  
 }  
 });  
 return requestId;  
}

结合代码清单 3-39，我们知道 sendRpc 方法的实现步骤如下。

1）使用 UUID 生成请求主键 requestId。

2）调用 addRpcRequest 向 handler（特别提醒下读者，这里的 handler 不是 RpcHandler，而是通过 TransportClient 构造器传入的 TransportResponseHandler）添加 requestId 与回调类 RpcResponseCallback 的引用之间的关系。TransportResponseHandler 的 addRpcRequest 方法（见代码清单 3-40）将更新最后一次请求的时间为当前系统时间，然后将 requestId 与 RpcResponseCallback 之间的映射加入到 outstandingRpcs 缓存中。outstandingRpcs 专门用于缓存发出的 RPC 请求信息。

代码清单 3-40　添加 RPC 请求到缓存

public void addRpcRequest(long requestId, RpcResponseCallback callback) {  
 updateTimeOfLastRequest();  
 outstandingRpcs.put(requestId, callback);  
}

3）调用 Channel 的 writeAndFlush 方法将 RPC 请求发送出去，这和在代码清单 3-33 中服务端调用的 respond 方法响应客户端的一样，都是使用 Channel 的 writeAndFlush 方法。当发送成功或者失败时会回调 ChannelFutureListener 的 operationComplete 方法。如果发送成功，那么只会打印 requestId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用 TransportResponseHandler 的 removeRpcRequest 方法（见代码清单 3-41），将此次请求从 outstandingRpcs 缓存中移除。

代码清单 3-41　从缓存中删除 RPC 请求

public void removeRpcRequest(long requestId) {  
 outstandingRpcs.remove(requestId);  
}

请求发送成功后，客户端将等待接收服务端的响应。根据图 3-3，返回的消息也会传递给 TransportChannelHandler 的 channelRead 方法（见代码清单 3-19），根据之前的分析，消息的分析将最后交给 TransportResponseHandler 的 handle 方法来处理。TransportResponseHandler 的 handle 方法分别对图 3-5 中的 6 种 ResponseMessage 进行处理，由于服务端使用 processRpcRequest 方法（见代码清单 3-30）处理 RpcRequest 类型的消息后，返回给客户端的消息为 RpcResponse 或 RpcFailure，所以我们来看看客户端的 TransportResponseHandler 的 handle 方法如何处理 RpcResponse 和 RpcFailure，如代码清单 3-42 所示。

代码清单 3-42　RpcResponse 和 RpcFailure 消息的处理

} else if (message instanceof RpcResponse) {  
 RpcResponse resp = (RpcResponse) message;  
 RpcResponseCallback listener = outstandingRpcs.get(resp.requestId);// 获取RpcResponseCallback  
 if (listener == null) {  
 logger.warn("Ignoring response for RPC {} from {} ({} bytes) since it is not outstanding",  
 resp.requestId, getRemoteAddress(channel), resp.body().size());  
 } else {  
 outstandingRpcs.remove(resp.requestId);  
 try {  
 listener.onSuccess(resp.body().nioByteBuffer());  
 } finally {  
 resp.body().release();  
 }  
 }  
} else if (message instanceof RpcFailure) {  
 RpcFailure resp = (RpcFailure) message;  
 RpcResponseCallback listener = outstandingRpcs.get(resp.requestId); // 获取RpcResponseCallback  
 if (listener == null) {  
 logger.warn("Ignoring response for RPC {} from {} ({}) since it is not outstanding",  
 resp.requestId, getRemoteAddress(channel), resp.errorString);  
 } else {  
 outstandingRpcs.remove(resp.requestId);  
 listener.onFailure(new RuntimeException(resp.errorString));  
 }

从代码清单 3-42 看到，处理 RpcResponse 的逻辑如下。

1. 使用 RpcResponse 对应的 RpcRequest 的主键 requestId，从 outstandingRpcs 缓存中获取注册的 RpcResponseCallback，此处的 RpcResponseCallback 即为代码清单 3-39 中传递给 sendRpc 方法的 RpcResponseCallback。
2. 移除 outstandingRpcs 缓存中 requestId 和 RpcResponseCallback 的注册信息。
3. 调用 RpcResponseCallback 的 onSuccess 方法，处理成功响应后的具体逻辑。这里的 RpcResponseCallback 需要在各个使用 TransportClient 的 sendRpc 方法的场景中分别实现。
4. 释放 RpcResponse 的 body，回收资源。

处理 RpcFailure 的逻辑如下。

1. 使用 RpcFailure 对应的 RpcRequest 的主键 requestId，从 outstandingRpcs 缓存中获取注册的 RpcResponseCallback，此处的 RpcResponseCallback 即为代码清单 3-39 中传递给 sendRpc 方法的 RpcResponseCallback。
2. 移除 outstandingRpcs 缓存中 requestId 和 RpcResponseCallback 的注册信息。
3. 调用 RpcResponseCallback 的 onFailure 方法，处理失败响应后的具体逻辑。这里的 RpcResponseCallback 需要在使用 TransportClient 的 sendRpc 方法时指定或实现。

**2.发送获取块请求**

fetchChunk 的实现如代码清单 3-43 所示。

代码清单 3-43　fetchChunk 的实现

public void fetchChunk(  
 long streamId,  
 final int chunkIndex,  
 final ChunkReceivedCallback callback) {  
 final long startTime = System.currentTimeMillis();  
 if (logger.isDebugEnabled()) {  
 logger.debug("Sending fetch chunk request {} to {}", chunkIndex, getRemoteAddress (channel));  
 }  
 final StreamChunkId streamChunkId = new StreamChunkId(streamId, chunkIndex);// 创建StreamChunkId  
 // 添加StreamChunkId与ChunkReceivedCallback之间的对应关系  
 handler.addFetchRequest(streamChunkId, callback);  
 // 发送块请求  
 channel.writeAndFlush(new ChunkFetchRequest(streamChunkId)).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 long timeTaken = System.currentTimeMillis() - startTime;  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending request {} to {} took {} ms", streamChunkId,  
 getRemoteAddress(channel), timeTaken);  
 }  
 } else {  
 String errorMsg = String.format("Failed to send request %s to %s: %s", streamChunkId,  
 getRemoteAddress(channel), future.cause());  
 logger.error(errorMsg, future.cause());  
 handler.removeFetchRequest(streamChunkId);  
 channel.close();  
 try {  
 callback.onFailure(chunkIndex, new IOException(errorMsg, future.cause()));  
 } catch (Exception e) {  
 logger.error("Uncaught exception in RPC response callback handler!", e);  
 }  
 }  
 }  
 });  
}

结合代码清单 3-43，我们知道 fetchChunk 方法的实现步骤如下。

1）使用流的标记 streamId 和块的索引 chunkIndex 创建 StreamChunkId。

2）调用 addFetchRequest 向 handler（特别提醒下读者，这里的 handler 不是 RpcHandler，而是通过 TransportClient 构造器传入的 TransportResponseHandler）添加 StreamChunkId 与回调类 ChunkReceivedCallback 的引用之间的关系。TransportResponseHandler 的 addFetchRequest 方法（见代码清单 3-44）将更新最后一次请求的时间为当前系统时间，然后将 StreamChunkId 与 ChunkReceivedCallback 之间的映射加入到 outstandingFetches 缓存中。outstandingFetches 专门用于缓存发出的块请求信息。

代码清单 3-44　添加块请求到缓存

public void addFetchRequest(StreamChunkId streamChunkId, ChunkReceivedCallback callback) {  
 updateTimeOfLastRequest();  
 outstandingFetches.put(streamChunkId, callback);  
}

3）调用 Channel 的 writeAndFlush 方法将块请求发送出去，这和在代码清单 3-33 中服务端调用的 respond 方法响应客户端的一样，都是使用 Channel 的 writeAndFlush 方法。当发送成功或者失败时会回调 ChannelFutureListener 的 operationComplete 方法。如果发送成功，那么只会打印 StreamChunkId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用 TransportResponseHandler 的 removeFetchRequest 方法（见代码清单 3-45），将此次请求从 outstandingFetches 缓存中移除。

代码清单 3-45　从缓存中删除 RPC 请求

public void removeRpcRequest(long requestId) {  
 outstandingRpcs.remove(requestId);  
}

请求发送成功后，客户端将等待接收服务端的响应。根据图 3-3，返回的消息也会传递给 TransportChannelHandler 的 channelRead 方法（见代码清单 3-19），根据之前的分析，消息的分析将最后交给 TransportResponseHandler 的 handle 方法来处理。TransportResponseHandler 的 handle 方法分别对图 3-5 中的 6 种处理结果进行处理，由于服务端使用 processFetchRequest 方法（见代码清单 3-29）处理 ChunkFetchRequest 类型的消息后返回给客户端的消息为 ChunkFetchSuccess 或 ChunkFetchFailure，所以我们来看看客户端的 TransportResponseHandler 的 handle 方法如何处理 ChunkFetchSuccess 和 ChunkFetchFailure，如代码清单 3-46 所示。

代码清单 3-46　ChunkFetchSuccess 和 ChunkFetchFailure 消息的处理

if (message instanceof ChunkFetchSuccess) {  
 ChunkFetchSuccess resp = (ChunkFetchSuccess) message;  
 ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);  
 if (listener == null) {  
 logger.warn("Ignoring response for block {} from {} since it is not outstanding",  
 resp.streamChunkId, getRemoteAddress(channel));  
 resp.body().release();  
 } else {  
 outstandingFetches.remove(resp.streamChunkId);  
 listener.onSuccess(resp.streamChunkId.chunkIndex, resp.body());  
 resp.body().release();  
 }  
} else if (message instanceof ChunkFetchFailure) {  
 ChunkFetchFailure resp = (ChunkFetchFailure) message;  
 ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);  
 if (listener == null) {  
 logger.warn("Ignoring response for block {} from {} ({}) since it is not outstanding",  
 resp.streamChunkId, getRemoteAddress(channel), resp.errorString);  
 } else {  
 outstandingFetches.remove(resp.streamChunkId);  
 listener.onFailure(resp.streamChunkId.chunkIndex, new ChunkFetchFailure-Exception(  
 "Failure while fetching " + resp.streamChunkId + ": " + resp.errorString));  
 }  
}

从代码清单 3-46 看到，处理 ChunkFetchSuccess 的逻辑如下。

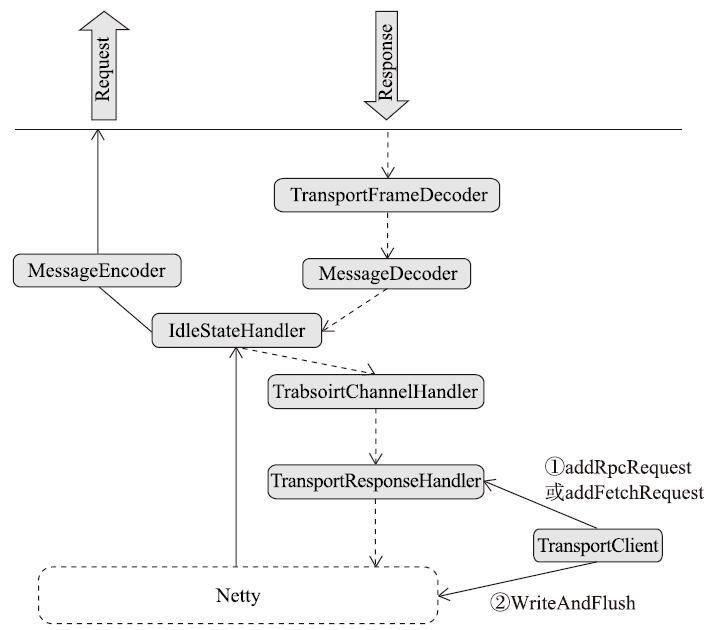
1. 使用 ChunkFetchSuccess 对应的 StreamChunkId，从 outstandingFetches 缓存中获取注册的 ChunkReceivedCallback，此处的 ChunkReceivedCallback 即为代码清单 3-43 中传递给 fetchChunk 方法的 ChunkReceivedCallback。
2. 移除 outstandingFetches 缓存中 StreamChunkId 和 ChunkReceivedCallback 的注册信息。
3. 调用 ChunkReceivedCallback 的 onSuccess 方法，处理成功响应后的具体逻辑。这里的 ChunkReceivedCallback 需要在各个使用 TransportClient 的 fetchChunk 方法的场景中分别实现。
4. 释放 ChunkFetchSuccess 的 body，回收资源。

处理 ChunkFetchFailure 的逻辑如下。

1. 使用 ChunkFetchFailure 对应的 StreamChunkId，从 outstandingFetches 缓存中获取注册的 ChunkReceivedCallback，此处的 ChunkReceivedCallback 即为代码清单 3-43 中传递给 fetchChunk 方法的 ChunkReceivedCallback。
2. 移除 outstandingFetches 缓存中 StreamChunkId 和 ChunkReceivedCallback 的注册信息。
3. 调用 ChunkReceivedCallback 的 onFailure 方法，处理失败响应后的具体逻辑。这里的 ChunkReceivedCallback 需要在各个使用 TransportClient 的 fetchChunk 方法的场景中分别实现。

在详细介绍了 TransportClient 和 TransportResponseHandler 之后，对于客户端我们就可以扩展图 3-3，把 TransportResponseHandler 及 TransportClient 的处理流程增加进来，如图 3-8 所示。

图 3-8 中的序号 ① 表示调用 TransportResponseHandler 的 addRpcRequest 方法（或 addFetchRequest 方法），将更新最后一次请求的时间为当前系统时间，然后将 requestId 与 RpcResponseCallback 之间的映射加入到 outstandingRpcs 缓存中（或将 StreamChunkId 与 ChunkReceivedCallback 之间的映射加入到 outstandingFetches 缓存中）。序号 ② 表示调用 Channel 的 writeAndFlush 方法将 RPC 请求发送出去。图中的虚线表示当 TransportResponseHandler 处理 RpcResponse 和 RpcFailure 时，将从 outstandingRpcs 缓存中获取此请求对应的 RpcResponseCallback（或处理 ChunkFetchSuccess 和 ChunkFetchFailure 时，将从 outstandingFetches 缓存中获取 StreamChunkId 对应的 ChunkReceivedCallback），并执行回调。此外，TransportClientBootstrap 将可能存在于图 3-8 中任何两个组件的箭头连线中间。

图 3-8　客户端请求、响应流程图

[继续阅读 3.3 事件总线](https://www.zhihu.com/pub/reader/119559409/chapter/931554611209330688)

## 3.3　事件总线

Spark 定义了一个特质㊟ListenerBus，可以接收事件并且将事件提交到对应事件的监听器。为了对 ListenerBus 有个直观的理解，我们先来看看它的代码实现，如代码清单 3-47 所示。

代码清单 3-47　ListenerBus 的定义

private[spark] trait ListenerBus[L <: AnyRef, E] extends Logging {  
  
private[spark] val listeners = new CopyOnWriteArrayList[L]  
  
final def addListener(listener: L): Unit = {  
 listeners.add(listener)  
}  
  
final def removeListener(listener: L): Unit = {  
 listeners.remove(listener)  
}  
final def postToAll(event: E): Unit = {  
 val iter = listeners.iterator  
 while (iter.hasNext) {  
 val listener = iter.next()  
 try {  
 doPostEvent(listener, event)  
 } catch {  
 case NonFatal(e) =>  
 logError(s"Listener ${Utils.getFormattedClassName(listener)} threw an exception", e)  
 }  
 }  
}  
  
protected def doPostEvent(listener: L, event: E): Unit  
  
private[spark] def findListenersByClass[T <: L : ClassTag](): Seq[T] = {  
 val c = implicitly[ClassTag[T]].runtimeClass  
 listeners.asScala.filter(\_.getClass == c).map(\_.asInstanceOf[T]).toSeq  
}  
  
}

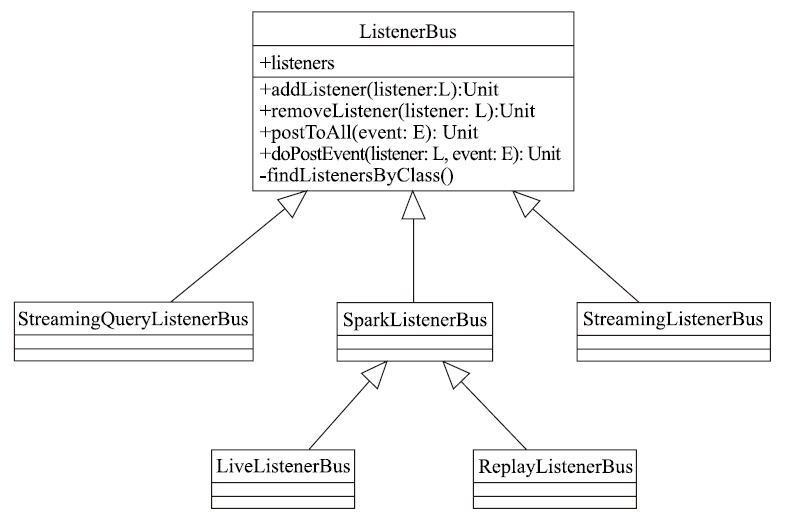
代码清单 3-47 中展示了 ListenerBus 是个泛型特质，其泛型参数为[L<:AnyRef，E]，其中 L 是代表监听器的泛型参数，可以看到 ListenerBus 支持任何类型的监听器，E 是代表事件的泛型参数。ListenerBus 中各个成员的作用如下。

* listeners：用于维护所有注册的监听器，其数据结构为 CopyOnWriteArrayList[L]。
* addListener：向 listeners 中添加监听器的方法，由于 listeners 采用 CopyOnWrite-ArrayList 来实现，所以 addListener 方法是线程安全的。
* removeListener：从 listeners 中移除监听器的方法，由于 listeners 采用 CopyOn-WriteArrayList 来实现，所以 removeListener 方法是线程安全的。
* postToAll：此方法的作用是将事件投递给所有的监听器。虽然 CopyOnWrite ArrayList 本身是线程安全的，但是由于 postToAll 方法内部引入了「先检查后执行」的逻辑，因而 postToAll 方法不是线程安全的，所以所有对 postToAll 方法的调用应当保证在同一个线程中。
* doPostEvent：用于将事件投递给指定的监听器，此方法只提供了接口定义，具体实现需要子类提供。
* findListenersByClass：查找与指定类型相同的监听器列表。

**3.3.1　ListenerBus 的继承体系**

理解了 ListenerBus 的定义后，本节一起来看看有哪些类继承了它。ListenerBus 的类继承体系如图 3-9 所示。

从图 3-9 中可以看到有 3 种 ListenerBus 的具体实现，分别如下。

图 3-9　ListenerBus 的类继承体系

* SparkListenerBus：用于将 SparkListenerEvent 类型的事件投递到 SparkListener-Interface 类型的监听器。
* StreamingQueryListenerBus：用于将 StreamingQueryListener.Event 类型的事件投递到 StreamingQueryListener 类型的监听器，此外还会将 StreamingQueryListener.Event 类型的事件交给 SparkListenerBus。
* StreamingListenerBus：用于将 StreamingListenerEvent 类型的事件投递到 Streaming Listener 类型的监听器，此外还会将 StreamingListenerEvent 类型的事件交给 Spark ListenerBus。

SparkListenerBus 也有如下两种实现。

* LiveListenerBus：采用异步线程将 SparkListenerEvent 类型的事件投递到 SparkListener 类型的监听器。
* ReplayListenerBus：用于从序列化的事件数据中重播事件。

有了对事件总线的这些介绍，读者已经在宏观上对其有所认识，但是如果没有具体的实现，ListenerBus 本身也无法发挥作用。下一小节我们将针对 SparkListenerBus 从更加微观的角度说明如何使用事件总线。

**3.3.2　SparkListenerBus 详解**

有了之前对 ListenerBus 的定义及 ListenerBus 类继承体系的准备，本小节将详细介绍 SparkListenerBus 的实现，如代码清单 3-48 所示。

代码清单 3-48　SparkListenerBus 的实现

private[spark] trait SparkListenerBus  
extends ListenerBus[SparkListenerInterface, SparkListenerEvent] {  
protected override def doPostEvent(  
 listener: SparkListenerInterface,  
 event: SparkListenerEvent): Unit = {  
 event match {  
 case stageSubmitted: SparkListenerStageSubmitted =>  
 listener.onStageSubmitted(stageSubmitted)  
 case stageCompleted: SparkListenerStageCompleted =>  
 listener.onStageCompleted(stageCompleted)  
 case jobStart: SparkListenerJobStart =>  
 listener.onJobStart(jobStart)  
 case jobEnd: SparkListenerJobEnd =>  
 listener.onJobEnd(jobEnd)  
 case taskStart: SparkListenerTaskStart =>  
 listener.onTaskStart(taskStart)  
 case taskGettingResult: SparkListenerTaskGettingResult =>  
 listener.onTaskGettingResult(taskGettingResult)  
 case taskEnd: SparkListenerTaskEnd =>  
 listener.onTaskEnd(taskEnd)  
 case environmentUpdate: SparkListenerEnvironmentUpdate =>  
 listener.onEnvironmentUpdate(environmentUpdate)  
 case blockManagerAdded: SparkListenerBlockManagerAdded =>  
 listener.onBlockManagerAdded(blockManagerAdded)  
 case blockManagerRemoved: SparkListenerBlockManagerRemoved =>  
 listener.onBlockManagerRemoved(blockManagerRemoved)  
 case unpersistRDD: SparkListenerUnpersistRDD =>  
 listener.onUnpersistRDD(unpersistRDD)  
 case applicationStart: SparkListenerApplicationStart =>  
 listener.onApplicationStart(applicationStart)  
 case applicationEnd: SparkListenerApplicationEnd =>  
 listener.onApplicationEnd(applicationEnd)  
 case metricsUpdate: SparkListenerExecutorMetricsUpdate =>  
 listener.onExecutorMetricsUpdate(metricsUpdate)  
 case executorAdded: SparkListenerExecutorAdded =>  
 listener.onExecutorAdded(executorAdded)  
 case executorRemoved: SparkListenerExecutorRemoved =>  
 listener.onExecutorRemoved(executorRemoved)  
 case blockUpdated: SparkListenerBlockUpdated =>  
 listener.onBlockUpdated(blockUpdated)  
 case logStart: SparkListenerLogStart => // ignore event log metadata  
 case \_ => listener.onOtherEvent(event)  
 }  
}  
}

我们看到 SparkListenerBus 已经实现了 ListenerBus 的 doPostEvent 方法，通过对 Spark ListenerEvent 事件的匹配，执行 SparkListenerInterface 监听器的相应方法。

这里的 SparkListenerEvent 其实是个特质，代码清单 3-48 中列出的 SparkListener-StageSubmitted、SparkListenerStageCompleted 等都是继承了 SparkListenerEvent 特质的样例类㊟。为了说明问题，这里仅仅摘选 SparkListenerEvent 及部分 SparkListenerEvent 子类，其实现如下。

@DeveloperApi  
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, include = JsonTypeInfo.As.PROPERTY, property = "Event")  
trait SparkListenerEvent {  
 protected[spark] def logEvent: Boolean = true  
}  
@DeveloperApi  
case class SparkListenerStageSubmitted(stageInfo: StageInfo, properties: Properties = null)  
 extends SparkListenerEvent  
@DeveloperApi  
case class SparkListenerStageCompleted(stageInfo: StageInfo)  
 extends SparkListenerEvent  
@DeveloperApi  
case class SparkListenerTaskStart(stageId: Int, stageAttemptId: Int, taskInfo: TaskInfo)  
 extends SparkListenerEvent  
// 省略其他SparkListenerEvent的实现  
private[spark] case class SparkListenerLogStart(sparkVersion: String) extends SparkListenerEvent

SparkListenerInterface 也是一个特质，其中定义了所有 SparkListener 应当遵守的接口规范。由于 SparkListenerInterface 中定义了很多接口，为说明问题，只摘抄 SparkListener-Interface 中的部分接口定义，其代码如下。

private[spark] trait SparkListenerInterface {  
 def onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit  
 def onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit  
 // 省略其他接口方法  
 def onOtherEvent(event: SparkListenerEvent): Unit  
}

结合代码清单 3-48，我们知道以上代码片段中的 onStageCompleted 和 onStage-Submitted 将在 SparkListenerBus 的 doPostEvent 方法中分别匹配到 SparkListenerStage Completed 和 SparkListenerStageSubmitted 事件时执行，而对于 doPostEvent 中无法匹配的事件，都将执行 onOtherEvent 方法。

在详细介绍了 ListenerBus 及 SparkListenerBus 后，我们知道当有事件需要通知监听器的时候，可以调用 ListenerBus 的 postToAll 方法，postToAll 方法遍历所有监听器并调用 SparkListenerBus 实现的 doPostEvent 方法，doPostEvent 方法对事件类型进行匹配后调用监听器的不同方法。整个投递事件的过程是通过方法调用实现的，所以这是一个同步调用。在监听器比较多的时候，这个过程会相对比较耗时，在 Spark UI（将在第 4 章中详细介绍）中为了达到页面的即时刷新，实现了 SparkListenerBus 的子类 LiveListenerBus。下一小节将围绕 LiveListenerBus 来详细说明异步投递消息的实现细节。

**3.3.3　LiveListenerBus 详解**

LiveListenerBus 继承了 SparkListenerBus，并实现了将事件异步投递给监听器，达到实时刷新 UI 界面数据的效果。LiveListenerBus 主要由以下部分组成。

* eventQueue：是 SparkListenerEvent 事件的阻塞队列，队列大小可以通过 Spark 属性 spark.scheduler.listenerbus.eventqueue.size 进行配置，默认为 10000（Spark 早期版本中属于静态属性，固定为 10000）。
* started：标记 LiveListenerBus 的启动状态的 AtomicBoolean 类型的变量。
* stopped：标记 LiveListenerBus 的停止状态的 AtomicBoolean 类型的变量。
* droppedEventsCounter：使用 AtomicLong 类型对删除的事件进行计数，每当日志打印了 droppedEventsCounter 后，会将 droppedEventsCounter 重置为 0。
* lastReportTimestamp：用于记录最后一次日志打印 droppedEventsCounter 的时间戳。
* processingEvent：用来标记当前正有事件被 listenerThread 线程处理。
* logDroppedEvent：AtomicBoolean 类型的变量，用于标记是否由于 eventQueue 已满，导致新的事件被删除。
* eventLock：用于当有新的事件到来时释放信号量，当对事件进行处理时获取信号量。
* listeners：继承自 LiveListenerBus 的监听器数组。
* listenerThread：处理事件的线程。

**1.异步事件处理线程**

listenerThread 用于异步处理 eventQueue 中的事件，为了便于说明，这里将展示 listener-Thread 及 LiveListenerBus 中的主要代码片段，如代码清单 3-49 所示。

代码清单 3-49　LiveListenerBus 主要逻辑的代码片段

private lazy val EVENT\_QUEUE\_CAPACITY = validateAndGetQueueSize()  
private lazy val eventQueue = new LinkedBlockingQueue[SparkListenerEvent](EVENT\_QUEUE\_CAPACITY)  
private def validateAndGetQueueSize(): Int = {  
 val queueSize = sparkContext.conf.get(LISTENER\_BUS\_EVENT\_QUEUE\_SIZE)  
 if (queueSize <= 0) {  
 throw new SparkException("spark.scheduler.listenerbus.eventqueue.size must be > 0!")  
 }  
 queueSize  
}  
  
private val started = new AtomicBoolean(false)  
private val stopped = new AtomicBoolean(false)  
private val droppedEventsCounter = new AtomicLong(0L)  
@volatile private var lastReportTimestamp = 0L  
private var processingEvent = false  
private val logDroppedEvent = new AtomicBoolean(false)  
private val eventLock = new Semaphore(0)  
  
private val listenerThread = new Thread(name) {  
 setDaemon(true)  
 override def run(): Unit = Utils.tryOrStopSparkContext(sparkContext) {  
 LiveListenerBus.withinListenerThread.withValue(true) {  
 while (true) {  
 eventLock.acquire() // 获取信号量  
 self.synchronized {  
 processingEvent = true  
 }  
 try {  
 val event = eventQueue.poll //从eventQueue中获取事件  
 if (event == null) {  
 // Get out of the while loop and shutdown the daemon thread  
 if (!stopped.get) {  
 throw new IllegalStateException("Polling 'null' from eventQueue means" +  
 " the listener bus has been stopped. So 'stopped' must be true")  
 }  
 return  
 }  
 postToAll(event) // 事件处理  
 } finally {  
 self.synchronized {  
 processingEvent = false  
 }  
 }  
 }  
 }  
 }  
}

通过分析代码清单 3-49，listenerThread 的工作步骤如下。

1. 不断获取信号量（当可以获取信号量时，说明还有事件未处理）。
2. 通过同步控制，将 processingEvent 设置为 true。
3. 从 eventQueue 中获取事件。
4. 调用超类 ListenerBus 的 postToAll 方法（postToAll 方法对监听器进行遍历，并调用 SparkListenerBus 的 doPostEvent 方法对事件进行匹配后执行监听器的相应方法）。
5. 每次循环结束依然需要通过同步控制，将 processingEvent 设置为 false。

值得一提的是，listenerThread 的 run 方法中调用了 Utils 的 tryOrStopSparkContext，tryOrStopSparkContext 方法可以保证当 listenerThread 的内部循环抛出异常后启动一个新的线程停止 SparkContext（SparkContext 的内容将在第 4 章详细介绍，tryOrStopSparkContext 方法的具体实现请阅读附录 A）。

**2.LiveListenerBus 的消息投递**

在解释了异步线程 listenerThread 的工作内容后，还有一个要点没有解释：eventQueue 中的事件是如何放进去的呢？由于 eventQueue 定义在 LiveListenerBus 中，因此 ListenerBus 和 SparkListenerBus 中并没有操纵 eventQueue 的方法，要将事件放入 eventQueue，只能依靠 LiveListenerBus 自己了，其 post 方法就是为此目的而生的，如代码清单 3-50 所示。

代码清单 3-50　向 LiveListenerBus 投递 SparkListenerEvent 事件

def post(event: SparkListenerEvent): Unit = {  
 if (stopped.get) {  
 logError(s"$name has already stopped! Dropping event $event")  
 return  
 }  
 val eventAdded = eventQueue.offer(event) // 向eventQueue中添加事件  
 if (eventAdded) {  
 eventLock.release()  
 } else {  
 onDropEvent(event)  
 droppedEventsCounter.incrementAndGet()  
 }  
 // 打印删除事件数的日志  
 val droppedEvents = droppedEventsCounter.get  
 if (droppedEvents > 0) {  
 if (System.currentTimeMillis() - lastReportTimestamp >= 60 \* 1000) {  
 if (droppedEventsCounter.compareAndSet(droppedEvents, 0)) {  
 val prevLastReportTimestamp = lastReportTimestamp  
 lastReportTimestamp = System.currentTimeMillis()  
 logWarning(s"Dropped $droppedEvents SparkListenerEvents since " +  
 new java.util.Date(prevLastReportTimestamp))  
 }  
 }  
 }  
}

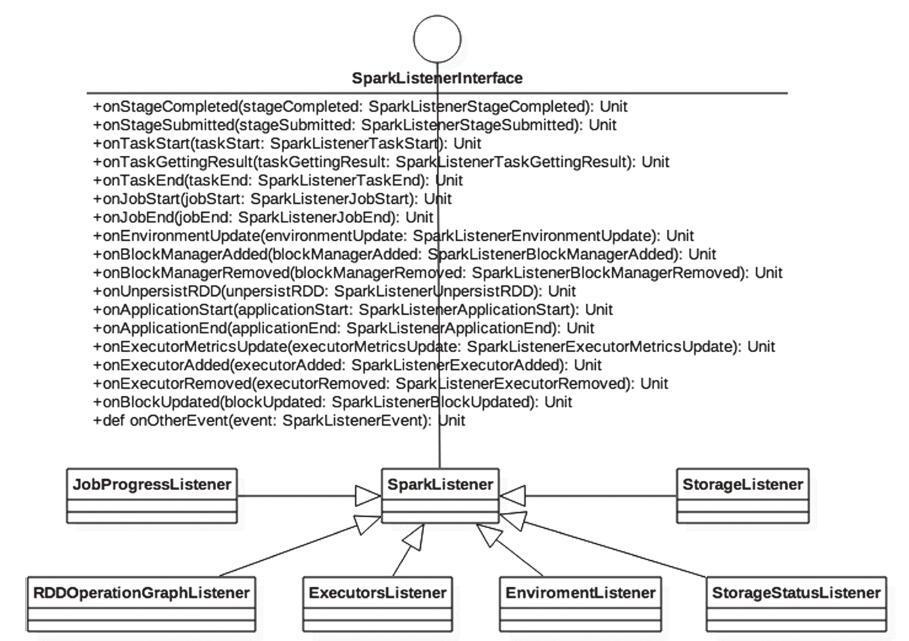
从代码清单 3-50 看到 post 方法的处理步骤如下。

1. 判断 LiveListenerBus 是否已经处于停止状态。
2. 向 eventQueue 中添加事件。如果添加成功，则释放信号量，进而催化 listener-Thread 有效工作。如果 eventQueue 已满造成添加失败，则移除事件，并对删除事件计数器 droppedEventsCounter 进行自增。
3. 如果有事件被删除，并且当前系统时间距离上一次打印 droppedEventsCounter 超过了 60 秒，则将 droppedEventsCounter 打印到日志。

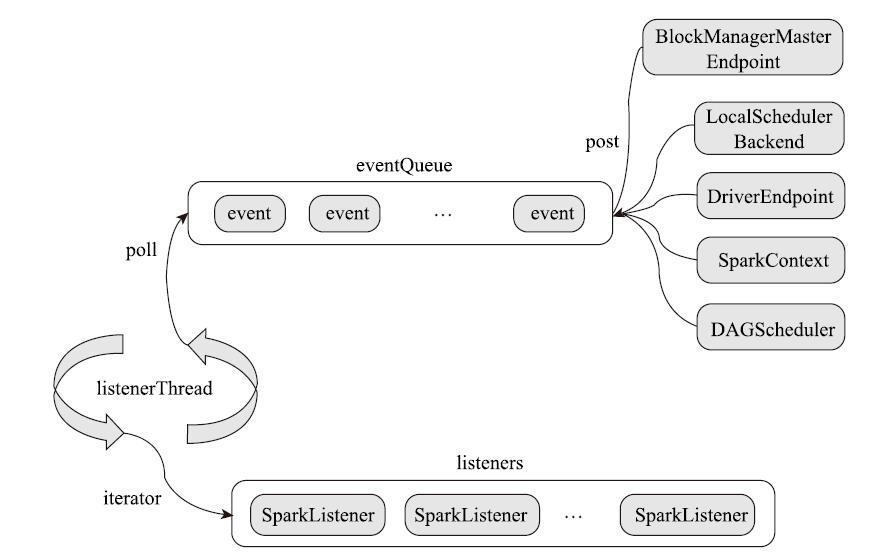
**3.LiveListenerBus 与监听器**

与 LiveListenerBus 配合使用的监听器，并非是父类 SparkListenerBus 的类型参数 Spark ListenerInterface，而是继承自 SparkListenerInterface 的 SparkListener 及其子类。图 3-10 列出了 Spark 中监听器 SparkListener 及它的 6 种最常用的实现㊟。

SparkListener 虽然实现了 SparkListenerInterface 中的每个方法，但是其实都是空实现，具体的实现需要交给子类去完成。

图 3-10　SparkListener 的类继承体系

本节首先对事件总线的接口定义进行了一些介绍，之后选择 ListenerBus 的子类 SparkListenerBus 与 LiveListenerBus 作为具体的实现例子进行分析，这里将通过图 3-11 更加直观地展示 ListenerBus、SparkListenerBus 及 LiveListenerBus 的工作原理。

图 3-11　LiveListenerBus 的工作流程图

最后对图 3-11 作一些补充说明：图中的 DAGScheduler、SparkContext、BlockManager-MasterEndpoint、DriverEndpoint 及 LocalSchedulerBackend 都是 LiveListenerBus 的事件来源，它们都是通过调用 LiveListenerBus 的 post 方法将消息交给异步线程 listenerThread 进行处理的。

## 3.4　度量系统

对于一个系统而言，首先考虑要满足一些业务场景，并实现功能。随着系统功能越来越多，代码量级越来越高，系统的可维护性、可测试性、性能都会成为新的挑战，这使监控功能就变得越来越重要了。在国内，绝大多数 IT 公司的项目都以业务为导向，以完成功能为目标，这些项目在立项、设计、开发、上线的各个阶段，很少有人会考虑到监控的问题。开发人员能够认真地在代码段落中打印日志，就已经属于比较优秀的程序员了。然而，在国外很多项目则不会这样，看看久负盛名的 Hadoop 的监控系统就可见一斑，尤其是在 Facebook 中，更是把功能、日志及监控列为同等重要，这是作为一个合格工程师的「三驾马车」。

Spark 作为优秀的开源系统，在监控方面也有自己的一整套体系。一个系统有了监控功能后将收获诸多益处，如可测试性、性能优化、运维评估、数据统计等。Spark 的度量系统使用 codahale 提供的第三方度量仓库 Metrics，本节将着重介绍 Spark 基于 Metrics 构建度量系统的原理与实现。对于 Metrics 感兴趣的读者，可以参考阅读附录 D 中的内容。

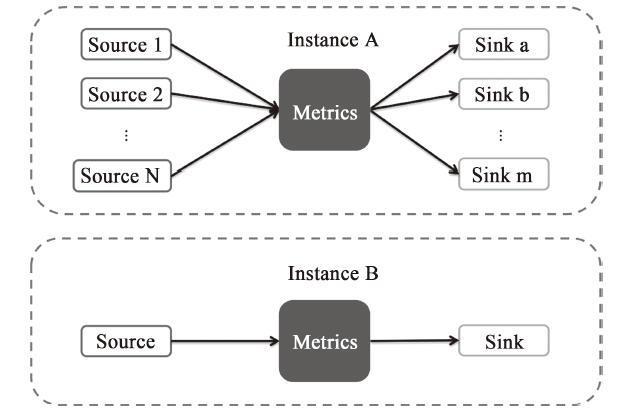
Spark 的度量系统中有三个概念，分别如下。

* Instance：指定了度量系统的实例名。Spark 按照 Instance 的不同，区分为 Master、Worker、Application、Driver 和 Executor。
* Source：指定了从哪里收集度量数据，即度量数据的来源。Spark 提供了应用的度量来源（ApplicationSource）、Worker 的度量来源（WorkerSource）、DAGScheduler 的度量来源（DAGSchedulerSource）、BlockManager 的度量来源（BlockManagerSource）等诸多实现，对各个服务或组件进行监控。
* Sink：指定了往哪里输出度量数据，即度量数据的输出。Spark 中使用 Metrics-Servlet 作为默认的 Sink，此外还提供了 ConsoleSink、CsvSink、JmxSink、Metrics Servlet、GraphiteSink 等实现。

为了更加直观地表现上述概念，我们以图 3-12 来表示 Spark 中度量系统的工作流程。

**3.4.1　Source 继承体系**

任何监控都离不开度量数据的采集，离线的数据采集很容易做到和被采集模块之间的解耦，但是对于实时的度量数据，尤其是那些内存中数据的采集就很难解耦。这就类似于网页监控数据的埋点一样，你要在网页中加入一段额外的 js 代码（如 Google 分析，即便你只是引入一个 JS 文件，也很难让前端工程师感到开心）。还有一类监控，比如在 Java Web 中增加一个负责监控的 Servlet 或者一个基于 Spring 3.0 的拦截器，这种方式虽然将耦合度从代码级别降低到配置级别，但却无法有效地对内存中的数据结构进行监控。Spark 的度量系统对系统功能来说是在代码层面耦合的，这种牺牲对于能够换取对实时的、处于内存中的数据进行更有效的监控是值得的。

图 3-12　度量系统的工作流程

Spark 将度量来源抽象为 Source，其定义如代码清单 3-51 所示。

代码清单 3-51　度量源的定义

private[spark] trait Source {  
 def sourceName: String  
 def metricRegistry: MetricRegistry  
}

从代码清单 3-51 可以看到，Source 是一个特质，其中定义了两个方法。

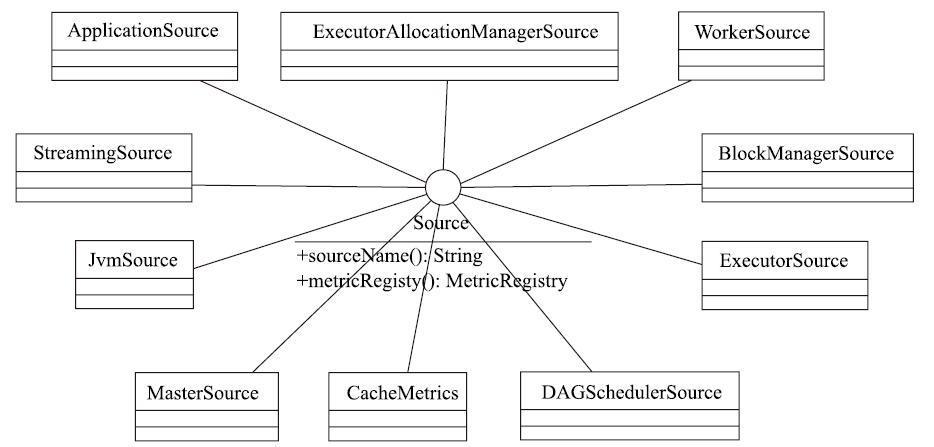
* sourceName：度量源的名称。
* metricRegistry：当前度量源的注册表。MetricRegistry 是 Metrics 库提供的 API，在附录 D 中有更详细的介绍。

Spark 中有很多 Source 的具体实现，可以通过图 3-13 来了解。

为了说明 Source 该如何实现，我们选择 ApplicationSource（也是因为其实现简单明了，足以说明问题），其实现如代码清单 3-52 所示。

代码清单 3-52　ApplicationSource 的实现

private[master] class ApplicationSource(val application: ApplicationInfo) extends Source {  
override val metricRegistry = new MetricRegistry()  
override val sourceName = "%s.%s.%s".format("application", application.desc.name,  
 System.currentTimeMillis())  
metricRegistry.register(MetricRegistry.name("status"), new Gauge[String] {  
 override def getValue: String = application.state.toString  
})  
 metricRegistry.register(MetricRegistry.name("runtime\_ms"), new Gauge[Long] {  
 override def getValue: Long = application.duration  
 })  
 metricRegistry.register(MetricRegistry.name("cores"), new Gauge[Int] {  
 override def getValue: Int = application.coresGranted  
 })  
}

图 3-13　Source 的继承体系

望文生义，ApplicationSource 用于采集 Spark 应用程序相关的度量。代码清单 3-52 中 ApplicationSource 重载了 metricRegistry 和 sourceName，并且向自身的注册表注册了 status（即应用状态，包括 WAITING，RUNNING，FINISHED，FAILED，KILLED，UNKNOWN）、runtime\_ms（运行持续时长）、cores（授权的内核数）等度量。这三个度量的取值分别来自于 ApplicationInfo 的 state、duration 和 coresGranted 三个属性。这三个度量都由 Gauge 的匿名内部类实现，Gauge 是 Metrics 提供的用于估计度量值的特质。有关 Gauge、MetricRegistry、MetricRegistry 注册度量的方法 register 及命名方法 name 的更详细介绍，请阅读附录 D。

**3.4.2　Sink 继承体系**

Source 准备好度量数据后，我们就需要考虑如何输出和使用的问题。这里介绍一些常见的度量输出方式：阿里数据部门采用的一种度量使用方式是输出到日志；在命令行运行过 Hadoop 任务（如 mapreduce）的使用者会发现控制台打印的内容中也包含度量信息；用户可能希望将有些度量信息保存到文件（如 CSV），以便将来能够查看；如果觉得使用 CSV 或者控制台等方式不够直观，还可以将采集到的度量数据输出到专用的监控系统界面。这些最终对度量数据的使用，或者说是输出方式，Spark 将它们统一抽象为 Sink。Sink 的定义如代码清单 3-53 所示。

代码清单 3-53　度量输出的定义

private[spark] trait Sink {  
 def start(): Unit  
 def stop(): Unit  
 def report(): Unit  
}

从代码清单 3-53 可以看到，Sink 是一个特质，包含三个接口方法。

* start：启动 Sink。
* stop：停止 Sink。
* report：输出到目的地。

从这三个方法的解释来看，很难让读者获得更多的信息。我们先把这些困惑放在一边，来看看 Spark 中 Sink 的类继承体系，如图 3-14 所示。

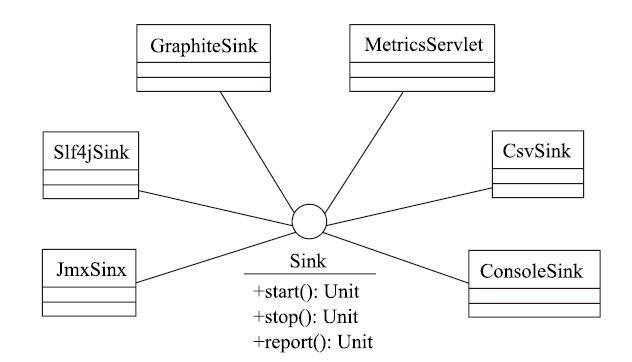
图 3-14　Sink 的类继承体系

图 3-14 中展示了 6 种 Sink 的具体实现。

* ConsoleSink：借助 Metrics 提供的 ConsoleReporter 的 API，将度量输出到 System.out，因此可以输出到控制台。
* CsvSink：借助 Metrics 提供的 CsvReporter 的 API，将度量输出到 CSV 文件。
* MetricsServlet：在 Spark UI 的 jetty 服务中创建 ServletContextHandler，将度量数据通过 Spark UI 展示在浏览器中。
* JmxSink：借助 Metrics 提供的 JmxReporter 的 API，将度量输出到 MBean 中，这样就可以打开 Java VisualVM，然后打开 Tomcat 进程监控，给 VisualVM 安装 MBeans 插件后，选择 MBeans 标签页可以对 JmxSink 所有注册到 JMX 中的对象进行管理。
* Slf4jSink：借助 Metrics 提供的 Slf4jReporter 的 API，将度量输出到实现了 Slf4j 规范的日志输出。
* GraphiteSink：借助 Metrics 提供的 GraphiteReporter 的 API，将度量输出到 Graphite（一个由 Python 实现的 Web 应用，采用 django 框架，用来收集服务器状态的监控系统）。

了解了 Sink 的类继承体系，我们挑选 Slf4jSink 作为 Spark 中 Sink 实现类的例子，来了解 Sink 具体该如何实现。Slf4jSink 的实现如代码清单 3-54 所示。

代码清单 3-54　Slf4jSink 的实现

private[spark] class Slf4jSink(  
 val property: Properties,  
 val registry: MetricRegistry,  
 securityMgr: SecurityManager)  
extends Sink {  
val SLF4J\_DEFAULT\_PERIOD = 10  
val SLF4J\_DEFAULT\_UNIT = "SECONDS"  
  
val SLF4J\_KEY\_PERIOD = "period"  
val SLF4J\_KEY\_UNIT = "unit"  
  
val pollPeriod = Option(property.getProperty(SLF4J\_KEY\_PERIOD)) match {  
 case Some(s) => s.toInt  
 case None => SLF4J\_DEFAULT\_PERIOD  
}  
  
val pollUnit: TimeUnit = Option(property.getProperty(SLF4J\_KEY\_UNIT)) match {  
 case Some(s) => TimeUnit.valueOf(s.toUpperCase())  
 case None => TimeUnit.valueOf(SLF4J\_DEFAULT\_UNIT)  
}  
  
MetricsSystem.checkMinimalPollingPeriod(pollUnit, pollPeriod)  
  
val reporter: Slf4jReporter = Slf4jReporter.forRegistry(registry)  
 .convertDurationsTo(TimeUnit.MILLISECONDS)  
 .convertRatesTo(TimeUnit.SECONDS)  
 .build()  
  
override def start() {  
 reporter.start(pollPeriod, pollUnit)  
}  
  
override def stop() {  
 reporter.stop()  
}  
  
override def report() {  
 reporter.report()  
 }  
}

从 Slf4jSink 的实现可以看到，Slf4jSink 的 start、stop 及 report 实际都是代理了 Metrics 库中的 Slf4jReporter 的 start、stop 及 report 方法。Slf4jReporter 的 start 方法实际是其父类 ScheduledReporter 的 start 实现。而传递的两个参数 pollPeriod 和 pollUnit，正是被 Scheduled Reporter 使用作为定时器获取数据的周期和时间单位。有关 ScheduledReporter 中 start、stop 及 Slf4jReporter 的 report 方法的实现可以参阅附录 D。

## 3.5　小结

本章首先对 Spark 配置类 SparkConf 进行了介绍，SparkConf 是 Spark 中使用频度较高的类之一，Spark 处处都能看到它的身影。Spark 系统中很多不同的功能模块都需要使用 RPC 的功能，通过对 Netty 的 API 进行封装，提供了 Spark 内部统一的 RPC 框架。Spark 中有多个模块都采用了监听器模式实现事件与监听器，通过事件总线的抽象，非常有利于 Spark 的并发性能和功能扩展，其高度抽象对开发人员来讲，非常值得借鉴。度量系统采用了 Metrics 库，Spark 又抽象出了 Source、Sink 和 Instance，Source 和 Sink 的具体实现都依赖于 Metrics 的 API。

**第 4 章　SparkContext 的初始化**

「合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。」

——《道德经》

**本章导读**

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

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

本章主要讲解的内容如下。

* SparkContext 概述。
* 创建 Spark 环境。
* Spark UI。
* 其他组件的创建和启动。
* 环境更新。

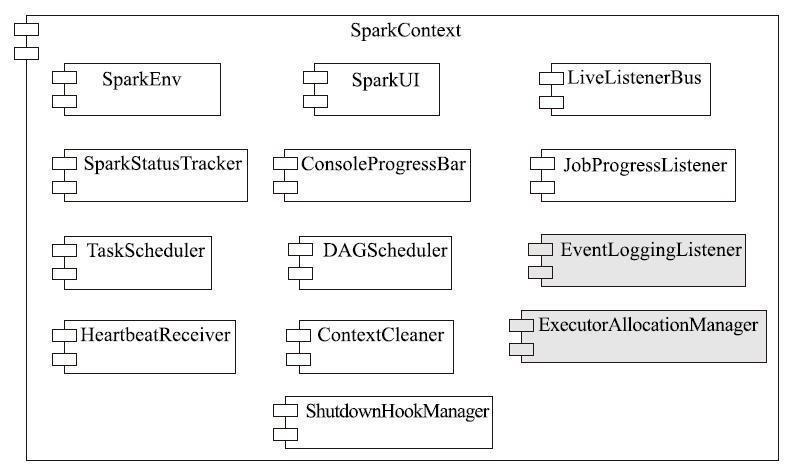
## 4.1　SparkContext 概述

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

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

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

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

图 4-1　SparkContext 的组成 ㊟

从图 4-1 可以知道，SparkContext 主要由以下部分组成。

* SparkEnv：Spark 运行时环境。Executor 是处理任务的执行器，它依赖于 SparkEnv 提供的运行时环境。此外，在 Driver 中也包含了 SparkEnv，这是为了保证 local 模式下任务的执行。SparkEnv 内部包含了很多组件，例如，serializerManager、RpcEnv、BlockManager、mapOutputTracker 等，在这里只需要认识它们即可，第 5 章将会对这些组件进行具体介绍。
* LiveListenerBus：SparkContext 中的事件总线，可以接收各个使用方的事件，并且通过异步方式对事件进行匹配后调用 SparkListener 的不同方法。LiveListenerBus 的具体内容已经在 3.3 节详细介绍，此处不再赘述。
* SparkUI：Spark 的用户界面。SparkUI 间接依赖于计算引擎、调度系统、存储体系，作业（Job）、阶段（Stage）、存储、执行器（Executor）等组件的监控数据都会以 SparkListenerEvent 的形式投递到 LiveListenerBus 中，SparkUI 将从各个 SparkListener 中读取数据并显示到 Web 界面。
* SparkStatusTracker：提供对作业、Stage（阶段）等的监控信息。SparkStatusTracker 是一个低级的 API，这意味着只能提供非常脆弱的一致性机制。
* ConsoleProgressBar：利用 SparkStatusTracker 的 API，在控制台展示 Stage 的进度。由于 SparkStatusTracker 存在的一致性问题，所以 ConsoleProgressBar 在控制台的显示往往有一定的时延。
* DAGScheduler：DAG 调度器，是调度系统中的重要组件之一，负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage、提交 Stage 等。SparkUI 中有关 Job 和 Stage 的监控数据都来自 DAGScheduler。
* TaskScheduler：任务调度器，是调度系统中的重要组件之一。TaskScheduler 按照调度算法对集群管理器已经分配给应用程序的资源进行二次调度后分配给任务。TaskScheduler 调度的 Task 是由 DAGScheduler 创建的，所以 DAGScheduler 是 TaskScheduler 的前置调度。
* HeartbeatReceiver：心跳接收器。所有 Executor 都会向 HeartbeatReceiver 发送心跳信息，HeartbeatReceiver 接收到 Executor 的心跳信息后，首先更新 Executor 的最后可见时间，然后将此信息交给 TaskScheduler 作进一步处理。
* ContextCleaner：上下文清理器。ContextCleaner 实际用异步方式清理那些超出应用作用域范围的 RDD、ShuffleDependency 和 Broadcast 等信息。
* JobProgressListener：作业进度监听器。JobProgressListener 在 3.3.3 节介绍 Spark-Listener 的继承体系时提到过，根据之前的介绍我们知道，JobProgressListener 将注册到 LiveListenerBus 中作为事件监听器之一使用。
* EventLoggingListener：将事件持久化到存储的监听器，是 SparkContext 中的可选组件。当 spark.eventLog.enabled 属性为 true 时启用。
* ExecutorAllocationManager：Executor 动态分配管理器。顾名思义，可以根据工作负载动态调整 Executor 的数量。在配置 spark.dynamicAllocation.enabled 属性为 true 的前提下，在非 local 模式下或者当 spark.dynamicAllocation.testing 属性为 true 时启用。
* ShutdownHookManager：用于设置关闭钩子的管理器。可以给应用设置关闭钩子，这样就可以在 JVM 进程退出时，执行一些清理工作。
* 除了以上介绍的这些 SparkContext 内部组件，SparkContext 内部还包括以下属性。
* creationSite㊟：类型为 CallSite，其中保存着线程栈中最靠近栈顶的用户定义的类及最靠近栈底的 Scala 或者 Spark 核心类信息，CallSite 的 shortForm 属性保存着以上信息的简短描述，CallSite 的 longForm 属性则保存着以上信息的完整描述。Spark 自带的 examples 项目中有对单词进行计数的应用例子 JavaWordCount，运行 JavaWordCount 得到的 CallSite 对象的属性值分别如下。
* shortForm：getOrCreate at JavaWordCount.java:48。
* longForm：org.apache.spark.sql.SparkSession$Builder.getOrCreate(SparkSession.scala:860)org.apache.spark.examples.JavaWordCount.main(JavaWordCount.java:48)。
* allowMultipleContexts：是否允许多个 SparkContext 实例。默认为 false，可以通过属性 spark.driver.allowMultipleContexts 来控制。
* startTime：SparkContext 启动的时间戳。
* stopped：标记 SparkContext 是否已经停止的状态，采用原子类型 AtomicBoolean。
* addedFiles：用于每个本地文件的 URL 与添加此文件到 addedFiles 时的时间戳之间的映射缓存。
* addedJars：用于每个本地 Jar 文件的 URL 与添加此文件到 addedJars 时的时间戳之间的映射缓存。
* persistentRdds：用于对所有持久化的 RDD 保持跟踪。
* executorEnvs：用于存储环境变量。executorEnvs 中环境变量都将传递给执行任务的 Executor 使用。
* sparkUser：当前系统的登录用户，也可以通过系统环境变量 SPARK\_USER 进行设置。这里使用的 Utils 的 getCurrentUserName 方法的更多介绍，请阅读附录 A。
* checkpointDir：RDD 计算过程中保存检查点时所需要的目录。
* localProperties：由 InheritableThreadLocal 保护的线程本地变量，其中的属性值可以沿着线程栈传递下去，供用户使用。
* \_conf：SparkContext 的配置，通过调用 SparkConf 的 clone 方法的克隆体。在 Spark-Context 初始化的过程中会对 conf 中的配置信息做校验，例如，用户必须给自己的应用程序设置 spark.master（采用的部署模式）和 spark.app.name（用户应用的名称）；用户设置的 spark.master 属性为 yarn 时，spark.submit.deployMode 属性必须为 cluster，且必须设置 spark.yarn.app.id 属性。
* \_jars：用户设置的 Jar 文件。当用户选择的部署模式是 YARN 时，\_jars 是由 spark.jars 属性指定的 Jar 文件和 spark.yarn.dist.jars 属性指定的 Jar 文件的并集。其他模式下只采用由 spark.jars 属性指定的 Jar 文件。这里使用了 Utils 的 getUserJars 方法，其具体介绍请阅读附录 A。
* \_files：用户设置的文件。可以使用 spark.files 属性进行指定。
* \_eventLogDir：事件日志的路径。当 spark.eventLog.enabled 属性为 true 时启用。默认为/tmp/spark-events，也可以通过 spark.eventLog.dir 属性指定。
* \_eventLogCodec：事件日志的压缩算法。当 spark.eventLog.enabled 属性与 spark.eventLog.compress 属性皆为 true 时启用。压缩算法默认为 lz4，也可以通过 spark.io.compression.codec 属性指定。Spark 目前支持的压缩算法包括 lzf、snappy 和 lz4 这 3 种。
* \_hadoopConfiguration：Hadoop 的配置信息，具体根据 Hadoop（Hadoop 2.0 之前的版本）和 Hadoop YARN（Hadoop2.0+ 的版本）的环境有所区别。如果系统属性 SPARK\_YARN\_MODE 为 true 或者环境变量 SPARK\_YARN\_MODE 为 true，那么将会是 YARN 的配置，否则为 Hadoop 配置。
* \_executorMemory：Executor 的内存大小。默认值为 1024MB。可以通过设置环境变量（SPARK\_MEM 或者 SPARK\_EXECUTOR\_MEMORY）或者 spark.executor.memory 属性指定。其中，spark.executor.memory 的优先级最高，SPARK\_EXECUTOR\_MEMORY 次之，SPARK\_MEM 是老版本 Spark 遗留下来的配置方式，未来将会废弃。
* \_applicationId：当前应用的标识。TaskScheduler 启动后会创建应用标识，SparkContext 中的\_applicationId 就是通过调用 TaskScheduler 的 applicationId 方法获得的。
* \_applicationAttemptId：当前应用尝试执行的标识。Spark Driver 在执行时会多次尝试执行，每次尝试都将生成一个标识来代表应用尝试执行的身份。
* \_listenerBusStarted：LiveListenerBus 是否已经启动的标记。
* nextShuffleId：类型为 AtomicInteger，用于生成下一个 Shuffle 的身份标识

## 4.2　创建 Spark 环境

在 Spark 中，凡是需要执行任务的地方就需要 SparkEnv。在生产环境中，SparkEnv 往往运行于不同节点的 Executor 中。但是由于 local 模式在本地执行的需要，因此在 Driver 本地的 Executor 也需要 SparkEnv。SparkContext 中创建 SparkEnv 的实现如代码清单 4-1 所示。

代码清单 4-1　创建 SparkEnv

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

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

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

createSparkEnv 方法创建 SparkEnv 的代码如下。

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

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

private[spark] def createDriverEnv(  
 conf: SparkConf,  
 isLocal: Boolean,  
 listenerBus: LiveListenerBus,  
 numCores: Int,  
 mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv = {  
 assert(conf.contains(DRIVER\_HOST\_ADDRESS),  
 s"${DRIVER\_HOST\_ADDRESS.key} is not set on the driver!")  
 assert(conf.contains("spark.driver.port"), "spark.driver.port is not set on the driver!")  
 val bindAddress = conf.get(DRIVER\_BIND\_ADDRESS)  
 val advertiseAddress = conf.get(DRIVER\_HOST\_ADDRESS)  
 val port = conf.get("spark.driver.port").toInt  
 val ioEncryptionKey = if (conf.get(IO\_ENCRYPTION\_ENABLED)) {  
 Some(CryptoStreamUtils.createKey(conf))  
 } else {  
 None  
 }  
 create(  
 conf,  
 SparkContext.DRIVER\_IDENTIFIER,  
 bindAddress,  
 advertiseAddress,  
 port,  
 isLocal,  
 numCores,  
 ioEncryptionKey,  
 listenerBus = listenerBus,  
 mockOutputCommitCoordinator = mockOutputCommitCoordinator  
 )  
}

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

* bindAddress：Driver 实例的 host。此属性通过从 SparkConf 中获取 DRIVER\_BIND\_ADDRESS 指定的属性值。DRIVER\_BIND\_ADDRESS 的定义如下。

private[spark] val DRIVER\_HOST\_ADDRESS = ConfigBuilder("spark.driver.host")  
 .doc("Address of driver endpoints.")  
 .stringConf  
 .createWithDefault(Utils.localHostName())  
  
private[spark] val DRIVER\_BIND\_ADDRESS = ConfigBuilder("spark.driver.bind Address")  
 .doc("Address where to bind network listen sockets on the driver.")  
 .fallbackConf(DRIVER\_HOST\_ADDRESS)

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

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

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

**第 10 章　Spark API**

「我心素已闲，清川澹如此。请留盘石上，垂钓将已矣。」

——《清溪》

**本章导读**

王维的《清溪》是我喜欢的唐诗之一。每当我读到这首诗，王维那种淡泊名利、悠然自在的心境，也让我对田园、山水有了无限的向往！这首诗表面上写王维在黄花川游玩的快乐趣途，实际上也是对自己身心的修炼。只有心性崇高、返璞归真，才会使自己坦然、淡然、自然！

静下心来读读山水田园诗，徜徉在浪漫与理想中，固然是很多人的追求，可是如何将自己从繁忙的工作中解放出来，更为关键！Spark 早期版本只提供了基于 SparkContext 的 API。现在 Spark 又增加了 SparkSession 和 DataFrame 相关的 API㊟，熟练使用它们，可以使大数据的开发工作变得更加简单。

word count 的例子是比较易于理解和讲解的大数据应用场景，所以 Hadoop 的入门例子选择了它。类似于任何介绍编程语言的 hello world 例子，word count 能用最短的时间，带给读者最直观的感受，从而降低学习曲线。作为大数据的入门实例，笔者为便于描述，自然选择经典的 word count 例子。

之所以讲解 Spark 的 word count 例子，也是为了能够把之前各章的内容串联起来。Spark 的任务运行在存储体系、调度系统及计算引擎之上。Spark Application、Application 的配置信息、Application 依赖的 Jar 包、map 任务的输出缓存、reduce 任务的缓存等都基于存储体系；程序转换为 RDD 构成的 DAG 后，RDD 的阶段划分、提交 Task、资源申请等都离不开调度系统；Task 在执行时对数据的缓存、聚合、Shuffle 管理等都由计算引擎支撑。

本章主要讲解的内容如下。

* DataType、Metadata、StructType 与 StructField 等基本概念。
* 数据源 DataSource。
* 检查点。
* RDD 的 API。
* 数据集合 Dataset 与 DataFrame。
* DataFrameReader。
* SparkSession。
* word count。

## 10.1　基本概念

本章会涉及一些基本概念，包括数据类型（DataType）、结构类型（StructType）、结构字段（StructField）及元数据（Metadata）。通过本节对这些基本概念的介绍，有利于本章后续内容的展开。

**1.DataType 简介**

DataType 是 Spark SQL 的所有数据类型的基本类型，Spark SQL 的所有数据类型都继承自 DataType，DataType 的继承体系如图 10-1 所示。

从图 10-1 中可以看到，Spark SQL 中定义的数据类型与 Java 的基本数据类型大部分都是一致的。由于 Spark SQL 不是本书要讲解的内容，所以读者在这里只需要了解这些内容即可。

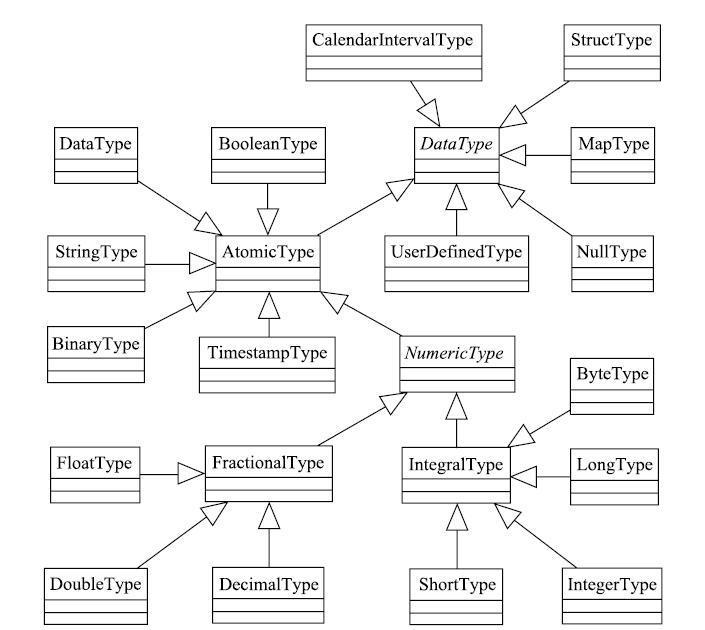
**2.Metadata 简介**

Metadata 用来保存 StructField 的元数据信息，其本质是底层的 Map[String，Any]。Meta-data 可以对 Boolean、Long、Double、String、Metadata、Array[Boolean]、Array[Long]、Array[Double]、Array[String]和 Array[Metadata]等类型的元数据进行存储或读取。Metadata 属于 Spark SQL 中的内容，这里不多介绍。

**3.StructType 与 StructField**

样例类 StructType 与样例类 StructField 共同构建起数据源的数据结构。

StructField 中共定义了 4 个属性：字段名称（name）、数据类型（dataType）、是否允许为 null（nullable）、元数据（metadata）。StructField 的定义如下。

图 10-1　DataType 的继承体系

case class StructField(  
 name: String,  
 dataType: DataType,  
 nullable: Boolean = true,  
 metadata: Metadata = Metadata.empty)

StructType 的属性中最为重要的是类型为 Array[StructField]的 fields，由此可以看出一个 StructType 中可以包括零到多个 StructField。为了便于理解，下面定义了一个简单的 StructType。

val struct =  
 StructType(  
 StructField("a", IntegerType, true) ::  
 StructField("b", LongType, false) ::  
 StructField("c", BooleanType, false) :: Nil)

对于构建的 struct，可以用以下语句获取名称为 b 的 StructField。

val singleField = struct("b")

如果要获取 struct 中不存在的 StructField，就像下面这样。

val nonExisting = struct("d")

那么 nonExisting 等于 null。

如果想要获取 struct 中的多个 StructField，需要像下面这样使用。

val twoFields = struct(Set("b", "c"))

此时的 twoFields 实际为 StructType（List（StructField（b，LongType，false），StructField（c，BooleanType，false）））。

如果像下面这样获取多个 StructField。

val ignoreNonExisting = struct(Set("b", "c", "d"))

那么其中的 d 会被忽略，此时的 ignoreNonExisting 实际为 StructType（List（StructField（b，LongType，false），StructField（c，BooleanType，false）））。

如果我们以以下代码定义一个新的数据结构 struct。

val innerStruct =  
 StructType(  
 StructField("f1", IntegerType, true) ::  
 StructField("f2", LongType, false) ::  
 StructField("f3", BooleanType, false) :: Nil)  
  
 val struct = StructType(  
 StructField("a", innerStruct, true) :: Nil)

那么我们可以使用这个 schema 创建 Row，就像下面这样。

val row = Row(Row(1, 2, true))

## 10.2　数据源 DataSource

从 Spark 1.3.0 开始，Spark 推出了 DataFrame 的 API，与此同时 DataSource 也被引入到 Spark 中。Spark 将文本文件、CSV 文件、JSON 文件等一系列格式的输入数据都作为数据源。特质 DataSourceRegister 是对所有数据源的抽象，DataSourceRegister 的所有具体实现都被注册到了 DataSource 中。DataSource 将负责对不同类型数据源的查找、创建不同类型数据源的信息、解析不同数据源的关系等。

**10.2.1　DataSourceRegister 详解**

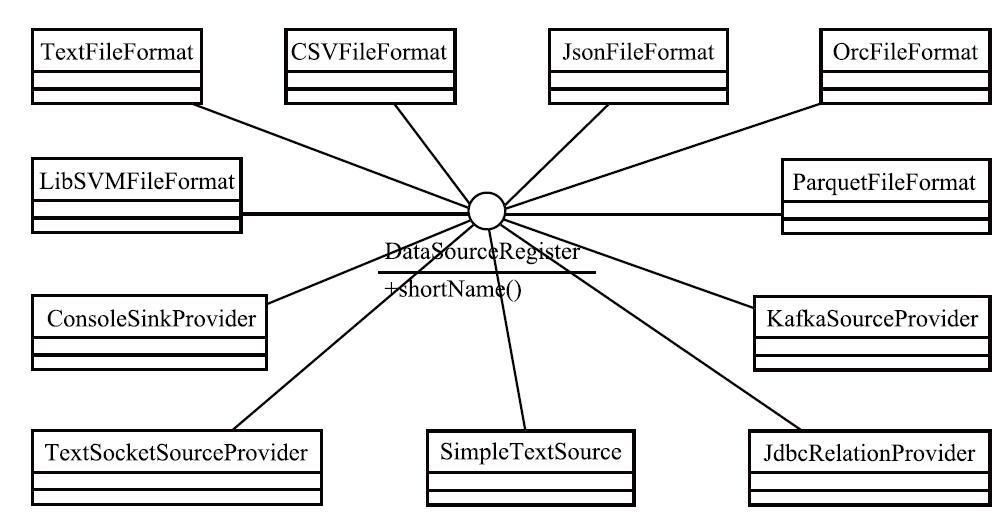
DataSourceRegister 是数据源的抽象，所有数据源都应该实现它。DataSourceRegister 的定义非常简单，代码如下。

trait DataSourceRegister {  
 def shortName(): String  
}

shortName 方法意在获取数据源提供者使用的格式或格式的别名。

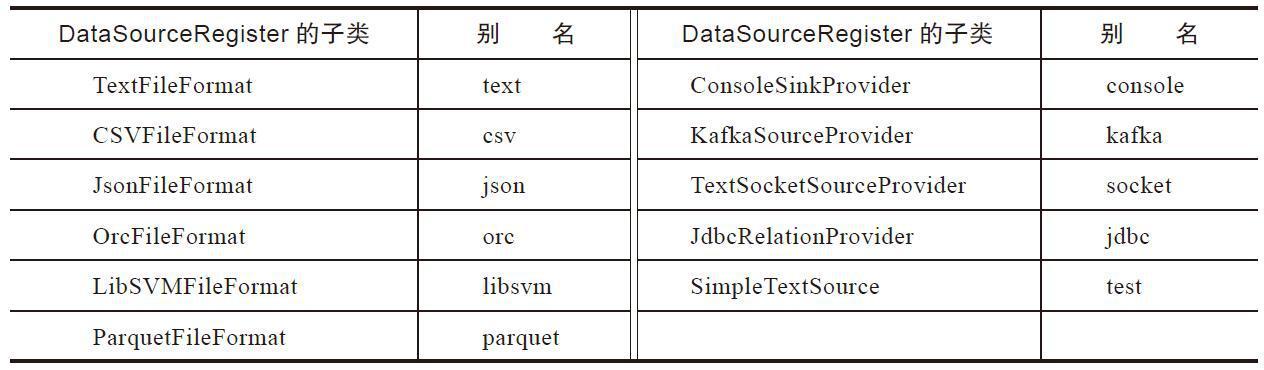
Spark 中实现了大量的数据源提供者，如图 10-2 所示。

这里以 TextFileFormat 实现的 shortName 方法为例。

图 10-2　DataSourceRegister 的继承体系

override def shortName（）：String="text"

可以看到 TextFileFormat 的别名为 text。此外，其他 DataSourceRegister 的实现类也都有属于自己的别名，表 10-1 对此进行了展示。

表 10-1　DataSourceRegister 的子类及其别名

**10.2.2　DataSource 详解**

DataSource 用于表示在 Spark SQL 中可插拔的数据源。DataSource 中包括以下属性。

* sparkSession：即 SparkSession。
* className：此名称用于决定要使用的类。
* paths：数据源的多个路径。
* userSpecifiedSchema：用户指定的 StructType。
* partitionColumns：分区字段的序列。
* bucketSpec：类型为 BucketSpec。BucketSpec 是一种用于将数据集分解为更多可管理部分的技术。由于桶的数量是固定的，因此数据不会随数据而波动。
* options：类型为 Map[String，String]，用于保存选项。
* catalogTable：类型为 CatalogTable。CatalogTable 是用于定义表的字典。
* providingClass：数据源的类。providingClass 实际是以 className 为参数，调用 Data-Source 伴生对象的 lookupDataSource 方法（见代码清单 10-1）获得的。
* sourceInfo：类型为 SourceInfo。样例类 SourceInfo 用于表示数据源的名称和元数据信息，其定义如下。

case class SourceInfo(name: String, schema: StructType, partitionColumns: Seq[String])

sourceInfo 实际是通过调用 DataSource 的 sourceSchema 方法（见代码清单 10-2）得到的。

* caseInsensitiveOptions：忽略大小写的选项配置。caseInsensitiveOptions 实际是通过 CaseInsensitiveMap 对 options 的包装，以忽略配置的大小写。

DataSource 伴生对象的 backwardCompatibilityMap 属性缓存了类名与对应的 DataSource-Register 实现类之间的映射关系，代码如下。

private val backwardCompatibilityMap: Map[String, String] = {  
 val jdbc = classOf[JdbcRelationProvider].getCanonicalName  
 val json = classOf[JsonFileFormat].getCanonicalName  
 val parquet = classOf[ParquetFileFormat].getCanonicalName  
 val csv = classOf[CSVFileFormat].getCanonicalName  
 val libsvm = "org.apache.spark.ml.source.libsvm.LibSVMFileFormat"  
 val orc = "org.apache.spark.sql.hive.orc.OrcFileFormat"  
  
 Map(  
 "org.apache.spark.sql.jdbc" -> jdbc,  
 "org.apache.spark.sql.jdbc.DefaultSource" -> jdbc,  
 "org.apache.spark.sql.execution.datasources.jdbc.DefaultSource" -> jdbc,  
 "org.apache.spark.sql.execution.datasources.jdbc" -> jdbc,  
 "org.apache.spark.sql.json" -> json,  
 "org.apache.spark.sql.json.DefaultSource" -> json,  
 "org.apache.spark.sql.execution.datasources.json" -> json,  
 "org.apache.spark.sql.execution.datasources.json.DefaultSource" -> json,  
 "org.apache.spark.sql.parquet" -> parquet,  
 "org.apache.spark.sql.parquet.DefaultSource" -> parquet,  
 "org.apache.spark.sql.execution.datasources.parquet" -> parquet,  
 "org.apache.spark.sql.execution.datasources.parquet.DefaultSource" -> parquet,  
 "org.apache.spark.sql.hive.orc.DefaultSource" -> orc,  
 "org.apache.spark.sql.hive.orc" -> orc,  
 "org.apache.spark.ml.source.libsvm.DefaultSource" -> libsvm,  
 "org.apache.spark.ml.source.libsvm" -> libsvm,  
 "com.databricks.spark.csv" -> csv  
 )  
}

**1.lookupDataSource**

DataSource 的伴生对象中提供了根据指定的名字找到对应的数据源类的方法 lookup-DataSource，其实现如代码清单 10-1 所示。

代码清单 10-1　根据指定的名字查找数据源

def lookupDataSource(provider: String): Class[\_] = {  
 val provider1 = backwardCompatibilityMap.getOrElse(provider, provider)  
 val provider2 = s"$provider1.DefaultSource"  
 val loader = Utils.getContextOrSparkClassLoader  
 val serviceLoader = ServiceLoader.load(classOf[DataSourceRegister], loader)  
  
 try {  
 serviceLoader.asScala.filter(\_.shortName().equalsIgnoreCase(provider1)).toList match {  
 case Nil =>  
 try {  
 Try(loader.loadClass(provider1)).orElse(Try(loader.loadClass(provider2))) match {  
 case Success(dataSource) =>  
 dataSource  
 case Failure(error) =>  
 // 忽略抛出异常的代码  
 }  
 } catch {  
 // 忽略捕获异常的处理代码  
 }  
 case head :: Nil =>  
 head.getClass  
 case sources =>  
 sys.error(s"Multiple sources found for $provider1 " +  
 s"(${sources.map(\_.getClass.getName).mkString(", ")}), " +  
 "please specify the fully qualified class name.")  
 }  
 } catch {  
 // 忽略捕获异常的处理代码  
 }  
}

根据代码清单 10-1，lookupDataSource 方法的执行步骤如下。

1. 从 backwardCompatibilityMap 中查找指定名称对应的数据源类 provider1，还在 provider1 后拼接 DefaultSource 构成 provider2。
2. 由于所有的数据源类都必须实现 DataSourceRegister，且 DataSourceRegister 的 shortName 方法定义了获取数据源类的别名的规范，所以这里使用类加载器找到与 provider1 匹配的数据源类，如果未能匹配，则尝试加载由 provider1 或 provider2 指定的类。

DataSource 中提供了很多方法，限于篇幅不能一一介绍。笔者挑选出 DataSource 中与本章内容相关的一些方法进行介绍。

**2.sourceSchema**

sourceSchema 方法用于根据 providingClass 得到对应的 SourceInfo，其实现如代码清单 10-2 所示。

代码清单 10-2　根据 providingClass 构造 SourceInfo

private def sourceSchema(): SourceInfo = {  
 providingClass.newInstance() match {  
 case s: StreamSourceProvider =>  
 val (name, schema) = s.sourceSchema(  
 sparkSession.sqlContext, userSpecifiedSchema, className, caseInsensitive-Options)  
 SourceInfo(name, schema, Nil)  
  
 case format: FileFormat =>  
 val path = caseInsensitiveOptions.getOrElse("path", {  
 throw new IllegalArgumentException("'path' is not specified")  
 })  
  
 val hdfsPath = new Path(path)  
 if (!SparkHadoopUtil.get.isGlobPath(hdfsPath)) {  
 val fs = hdfsPath.getFileSystem(sparkSession.sessionState.newHadoop-Conf())  
 if (!fs.exists(hdfsPath)) {  
 throw new AnalysisException(s"Path does not exist: $path")  
 }  
 }  
  
 val isSchemaInferenceEnabled = sparkSession.sessionState.conf.streaming-SchemaInference  
 val isTextSource = providingClass == classOf[text.TextFileFormat]  
 // 省略抛出IllegalArgumentException的代码  
 val (dataSchema, partitionSchema) = getOrInferFileFormatSchema(format)  
 SourceInfo(  
 s"FileSource[$path]",  
 StructType(dataSchema ++ partitionSchema),  
 partitionSchema.fieldNames)  
  
 case \_ =>  
 throw new UnsupportedOperationException(  
 s"Data source $className does not support streamed reading")  
 }  
}

根据代码清单 10-2，sourceSchema 方法通过反射得到 providingClass 对应的实例，然后根据实例的不同类型构造 SourceInfo。

**3.resolveRelation**

resolveRelation 方法用于根据数据源的提供类（providingClass）和 userSpecifiedSchema 创建读写数据源所需的 BaseRelation。由于 resolveRelation 方法的内容很多，这里只展示 resolveRelation 方法的一部分内容，如代码清单 10-3 所示。

代码清单 10-3　resolveRelation 方法的部分实现

def resolveRelation(checkFilesExist: Boolean = true): BaseRelation = {  
 val relation = (providingClass.newInstance(), userSpecifiedSchema) match {  
 case (dataSource: SchemaRelationProvider, Some(schema)) =>  
 dataSource.createRelation(sparkSession.sqlContext, caseInsensitiveOptions, schema)  
 case (dataSource: RelationProvider, None) =>  
 dataSource.createRelation(sparkSession.sqlContext, caseInsensitiveOptions)  
 case (\_: SchemaRelationProvider, None) =>  
 // 忽略创建BaseRelation的代码  
 case (format: FileFormat, \_) =>  
 // 忽略创建BaseRelation的代码  
 case \_ =>  
 throw new AnalysisException(  
 s"$className is not a valid Spark SQL Data Source.")  
 }  
 relation  
}

[继续阅读 10.3 检查点的实现](https://www.zhihu.com/pub/reader/119559409/chapter/931554613075775488)

## 10.3　检查点的实现

检查点是很多分布式系统为了容灾容错引入的机制，其实质是将系统运行期的内存数据结构和状态持久化到磁盘上，在需要时通过对这些持久化数据的读取，重新构造出之前的运行期状态。Spark 使用检查点主要是为了将 RDD 的执行状态保留下来，在重新执行时就不用重新计算，而直接从检查点读取。CheckpointRDD 是对检查点数据进行操作的 RDD，例如，读写检查点数据。RDDCheckpointData 表示 RDD 检查点的数据。Spark 的检查点离不开 CheckpointRDD 和 RDDCheckpointData 的支持，本节将对它们的代码实现进行分析。

**10.3.1　CheckpointRDD 的实现**

CheckpointRDD 是特殊的 RDD，用来从存储体系中恢复检查点数据。CheckpointRDD 的定义如下。

private[spark] abstract class CheckpointRDD[T: ClassTag](sc: SparkContext)  
extends RDD[T](sc, Nil) {  
  
override def doCheckpoint(): Unit = { }  
override def checkpoint(): Unit = { }  
override def localCheckpoint(): this.type = this  
  
protected override def getPartitions: Array[Partition] = ???  
override def compute(p: Partition, tc: TaskContext): Iterator[T] = ???  
}

可以看到 CheckpointRDD 重写了 RDD 的 5 个方法，分别如下。

* doCheckpoint：保存检查点。
* checkpoint：读取检查点数据。
* localCheckpoint：本地的检查点。
* getPartitions：获取检查点的分区数组。
* compute：名为计算，实际是从检查点恢复数据。

CheckpointRDD 有 LocalCheckpointRDD 和 ReliableCheckpointRDD 两个子类，它们都没有完全实现 CheckpointRDD 中的方法。下面以 ReliableCheckpointRDD 为例，来介绍 Check-pointRDD 的具体实现。

ReliableCheckpointRDD 中定义了以下属性。

* sc：即 SparkContext。
* checkpointPath：检查点目录的字符串表示。
* \_partitioner：调用方指定的分区计算器，默认为 None。
* hadoopConf：SparkContext 的\_hadoopConfiguration 属性，即 Hadoop 的配置信息。
* cpath：类型为 org.apache.hadoop.fs.Path，表示 checkpointPath 对应的 Hadoop 文件系统中的路径。
* fs：使用 hadoopConf 得到的 org.apache.hadoop.fs.FileSystem。
* broadcastedConf：调用 SparkContext 的 broadcast 方法对 hadoopConf 进行广播后返回的 Broadcast 对象。
* partitioner：ReliableCheckpointRDD 的分区计算器，优先采用\_partitioner 指定的，否则调用 ReliableCheckpointRDD 的伴生对象的 readCheckpointedPartitionerFile 方法从 checkpointPath 指定的检查点目录下读取分区计算器。

ReliableCheckpointRDD 的伴生对象中提供了很多工具方法，下面逐个介绍。

**1.writePartitionToCheckpointFile**

writePartitionToCheckpointFile 方法用于将 RDD 分区的数据写入到检查点目录下的文件中，其实现如代码清单 10-4 所示。

代码清单 10-4　writePartitionToCheckpointFile 方法

def writePartitionToCheckpointFile[T: ClassTag](  
 path: String,  
 broadcastedConf: Broadcast[SerializableConfiguration],  
 blockSize: Int = -1)(ctx: TaskContext, iterator: Iterator[T]) {  
 val env = SparkEnv.get  
 val outputDir = new Path(path)  
 val fs = outputDir.getFileSystem(broadcastedConf.value.value)  
  
 val finalOutputName = ReliableCheckpointRDD.checkpointFileName(ctx.partition-Id())  
 val finalOutputPath = new Path(outputDir, finalOutputName)  
 val tempOutputPath =  
 new Path(outputDir, s".$finalOutputName-attempt-${ctx.attemptNumber()}")  
  
 val bufferSize = env.conf.getInt("spark.buffer.size", 65536)  
  
 val fileOutputStream = if (blockSize < 0) {  
 fs.create(tempOutputPath, false, bufferSize)  
 } else {  
 // 省略与测试相关的代码  
 }  
 val serializer = env.serializer.newInstance()  
 val serializeStream = serializer.serializeStream(fileOutputStream)  
 Utils.tryWithSafeFinally {  
 serializeStream.writeAll(iterator)  
 } {  
 serializeStream.close()  
 }  
  
 if (!fs.rename(tempOutputPath, finalOutputPath)) {  
 if (!fs.exists(finalOutputPath)) {  
 logInfo(s"Deleting tempOutputPath $tempOutputPath")  
 fs.delete(tempOutputPath, false)  
 throw new IOException("Checkpoint failed: failed to save output of task: " +  
 s"${ctx.attemptNumber()} and final output path does not exist: $finalOut-putPath")  
 } else {  
 // Some other copy of this task must've finished before us and renamed it  
 logInfo(s"Final output path $finalOutputPath already exists; not overwrit-ing it")  
 if (!fs.delete(tempOutputPath, false)) {  
 logWarning(s"Error deleting ${tempOutputPath}")  
 }  
 }  
 }  
}

**2.writePartitionerToCheckpointDir**

writePartitionerToCheckpointDir 方法用于将分区计算器的数据写入到检查点的目录下，其实现如代码清单 10-5 所示。

代码清单 10-5　writePartitionerToCheckpointDir 方法

private def writePartitionerToCheckpointDir(  
 sc: SparkContext, partitioner: Partitioner, checkpointDirPath: Path): Unit = {  
 try {  
 val partitionerFilePath = new Path(checkpointDirPath, checkpointPartitioner-FileName)  
 val bufferSize = sc.conf.getInt("spark.buffer.size", 65536)  
 val fs = partitionerFilePath.getFileSystem(sc.hadoopConfiguration)  
 val fileOutputStream = fs.create(partitionerFilePath, false, bufferSize)  
 val serializer = SparkEnv.get.serializer.newInstance()  
 val serializeStream = serializer.serializeStream(fileOutputStream)  
 Utils.tryWithSafeFinally {  
 serializeStream.writeObject(partitioner)  
 } {  
 serializeStream.close()  
 }  
 logDebug(s"Written partitioner to $partitionerFilePath")  
 } catch {  
 case NonFatal(e) =>  
 logWarning(s"Error writing partitioner $partitioner to $checkpointDirPath")  
 }  
}

**3.writeRDDToCheckpointDirectory**

writeRDDToCheckpointDirectory 方法用于将 RDD 的数据写入检查点目录，其实现如代码清单 10-6 所示。

代码清单 10-6　writeRDDToCheckpointDirectory 方法

def writeRDDToCheckpointDirectory[T: ClassTag](  
 originalRDD: RDD[T],  
 checkpointDir: String,  
 blockSize: Int = -1): ReliableCheckpointRDD[T] = {  
  
 val sc = originalRDD.sparkContext  
  
 val checkpointDirPath = new Path(checkpointDir)  
 val fs = checkpointDirPath.getFileSystem(sc.hadoopConfiguration)  
 if (!fs.mkdirs(checkpointDirPath)) {  
 throw new SparkException(s"Failed to create checkpoint path $checkpoint-DirPath")  
 }  
  
 val broadcastedConf = sc.broadcast(  
 new SerializableConfiguration(sc.hadoopConfiguration))  
 sc.runJob(originalRDD,  
 writePartitionToCheckpointFile[T](checkpointDirPath.toString, broadcastedConf) \_)  
  
 if (originalRDD.partitioner.nonEmpty) {  
 writePartitionerToCheckpointDir(sc, originalRDD.partitioner.get, checkpointDirPath)  
 }  
  
 val newRDD = new ReliableCheckpointRDD[T](  
 sc, checkpointDirPath.toString, originalRDD.partitioner)  
 if (newRDD.partitions.length != originalRDD.partitions.length) {  
 throw new SparkException(  
 s"Checkpoint RDD $newRDD(${newRDD.partitions.length}) has different " +  
 s"number of partitions from original RDD $originalRDD(${originalRDD.partitions.length})")  
 }  
 newRDD  
}

根据代码清单 10-6，writeRDDToCheckpointDirectory 方法的执行步骤如下。

1. 调用 SparkContext 的 runJob 方法将 RDD 的数据写入到检查点目录。将数据写入磁盘的函数是 ReliableCheckpointRDD 的伴生对象的 writePartitionToCheckpointFile 方法，其实现如代码清单 10-4 所示。
2. 如果 RDD 有分区计算器，那么调用 ReliableCheckpointRDD 的伴生对象的 write-PartitionerToCheckpointDir 方法（见代码清单 10-5）将分区计算器的信息也写入到检查点目录。
3. 创建并返回 ReliableCheckpointRDD。

**4.readCheckpointFile**

readCheckpointFile 方法用于从检查点目录下的文件中读取 RDD 的数据，其实现如代码清单 10-7 所示。

代码清单 10-7　readCheckpointFile 方法

def readCheckpointFile[T](  
 path: Path,  
 broadcastedConf: Broadcast[SerializableConfiguration],  
 context: TaskContext): Iterator[T] = {  
 val env = SparkEnv.get  
 val fs = path.getFileSystem(broadcastedConf.value.value)  
 val bufferSize = env.conf.getInt("spark.buffer.size", 65536)  
 val fileInputStream = fs.open(path, bufferSize)  
 val serializer = env.serializer.newInstance()  
 val deserializeStream = serializer.deserializeStream(fileInputStream)  
  
 context.addTaskCompletionListener(context => deserializeStream.close())  
  
 deserializeStream.asIterator.asInstanceOf[Iterator[T]]  
}

上面介绍了 ReliableCheckpointRDD 的伴生对象提供的方法，下面介绍 ReliableCheck-pointRDD 对父类 RDD 重写的部分方法。

**5.getPartitions**

ReliableCheckpointRDD 实现的 getPartitions 方法（见代码清单 10-8）用于从检查点文件中获取分区数组。

代码清单 10-8　getPartitions 方法

protected override def getPartitions: Array[Partition] = {  
 val inputFiles = fs.listStatus(cpath)  
 .map(\_.getPath)  
 .filter(\_.getName.startsWith("part-"))  
 .sortBy(\_.getName.stripPrefix("part-").toInt)  
 inputFiles.zipWithIndex.foreach { case (path, i) =>  
 if (path.getName != ReliableCheckpointRDD.checkpointFileName(i)) {  
 throw new SparkException(s"Invalid checkpoint file: $path")  
 }  
 }  
 Array.tabulate(inputFiles.length)(i => new CheckpointRDDPartition(i))  
}

**6.getPreferredLocations**

ReliableCheckpointRDD 实现的 getPreferredLocations 方法（见代码清单 10-9）用于从检查点文件中获取偏好位置。

代码清单 10-9　getPreferredLocations 方法

protected override def getPreferredLocations(split: Partition): Seq[String] = {  
 val status = fs.getFileStatus(  
 new Path(checkpointPath, ReliableCheckpointRDD.checkpointFileName(split.index)))  
 val locations = fs.getFileBlockLocations(status, 0, status.getLen)  
 locations.headOption.toList.flatMap(\_.getHosts).filter(\_ != "localhost")  
}

**7.compute**

ReliableCheckpointRDD 实现的 compute 方法（见代码清单 10-10）用于从检查点文件中获取检查点数据。

代码清单 10-10　compute 方法

override def compute(split: Partition, context: TaskContext): Iterator[T] = {  
 val file = new Path(checkpointPath, ReliableCheckpointRDD.checkpointFile-Name(split.index))  
 ReliableCheckpointRDD.readCheckpointFile(file, broadcastedConf, context)  
}

根据对 CheckpointRDD 的各个方法的分析，检查点似乎不再那么神秘，其实现不过是对文件的读与写。

**10.3.2　RDDCheckpointData 的实现**

RDDCheckpointData 用于保存与检查点相关的信息。每个 RDDCheckpointData 实例都与一个 RDD 实例相关联。RDDCheckpointData 中一共有三个属性。

* rdd：RDDCheckpointData 关联的 RDD。
* cpState：检查点的状态。默认为 Initialized。cpState 的值来自枚举类型 Checkpoint-State，CheckpointState 中定义了检查点的状态，包括初始化完成（Initialized）、正在保存检查点（CheckpointingInProgress）和保存检查点完毕（Checkpointed）。
* cpRDD：保存检查点数据的 RDD，即 CheckpointRDD 的实现类。

RDDCheckpointData 中定义了一些方法，分别如下。

**1.isCheckpointed**

isCheckpointed 方法用于判断是否已经为 RDDCheckpointData 关联的 RDD 保存了检查点数据，其实现如下。

def isCheckpointed: Boolean = RDDCheckpointData.synchronized {  
 cpState == Checkpointed  
}

**2.checkpoint**

checkpoint 方法是用于将 RDDCheckpointData 关联的 RDD 的数据保存到检查点的模板方法，其实现如代码清单 10-11 所示。

代码清单 10-11　保存检查点的模板方法 checkpoint

final def checkpoint(): Unit = {  
 RDDCheckpointData.synchronized {  
 if (cpState == Initialized) {  
 cpState = CheckpointingInProgress  
 } else {  
 return  
 }  
 }  
 val newRDD = doCheckpoint()  
  
 // Update our state and truncate the RDD lineage  
 RDDCheckpointData.synchronized {  
 cpRDD = Some(newRDD)  
 cpState = Checkpointed  
 rdd.markCheckpointed()  
 }  
}

根据代码清单 10-11，checkpoint 方法的执行步骤如下。

1. 如果检查点的状态是 Initialized，那么将 cpState 设置为 CheckpointingInProgress，否则返回。
2. 调用 doCheckpoint 方法保存检查点并生成 CheckpointRDD。doCheckpoint 方法需要 RDDCheckpointData 的子类实现。RDDCheckpointData 的子类有 LocalRDDCheckpointData 和 ReliableRDDCheckpointData 两种，10.3.3 节以 ReliableRDDCheckpointData 为例，介绍了其实现的 doCheckpoint 方法。
3. 由 cpRDD 持有刚生成的 CheckpointRDD，然后将 cpState 设置为 Checkpointed，最后调用 RDD 的 markCheckpointed 方法（见代码清单 10-12）清空依赖。之所以清空依赖，是因为现在已经有了 CheckpointRDD，之前的依赖关系不再需要了。

代码清单 10-12　标记 RDD 已经保存了检查点

private[spark] def markCheckpointed(): Unit = {  
 clearDependencies()  
 partitions\_ = null  
 deps = null // Forget the constructor argument for dependencies too  
}  
  
protected def clearDependencies() {  
 dependencies\_ = null  
}

**3.checkpointRDD**

checkpointRDD 方法用于获取 cpRDD 持有的 CheckpointRDD，其实现如下。

def checkpointRDD: Option[CheckpointRDD[T]] = RDDCheckpointData.synchronized { cpRDD }

**4.getPartitions**

getPartitions 方法用于获取 CheckpointRDD 的分区数组，其实现如下。

def getPartitions: Array[Partition] = RDDCheckpointData.synchronized {  
 cpRDD.map(\_.partitions).getOrElse { Array.empty }  
}

有了对 RDDCheckpointData 的理解，下一小节以 RDDCheckpointData 的子类 Reliable-RDDCheckpointData 为例，来看看 RDDCheckpointData 该如何实现。

**10.3.3　ReliableRDDCheckpointData 的实现**

本节以 RDDCheckpointData 的子类 ReliableRDDCheckpointData 为例，来看看 RDD-CheckpointData 的具体实现。

ReliableRDDCheckpointData 除继承了 RDDCheckpointData 的属性外，还有自身的一个属性 cpDir。cpDir 是保存 ReliableRDDCheckpointData 所关联的 RDD 数据的检查点目录，是通过调用 ReliableRDDCheckpointData 的伴生对象的 checkpointPath 方法（见代码清单 10-13）生成的。

ReliableRDDCheckpointData 及其伴生对象提供了以下方法。

**1.checkpointPath**

checkpointPath 方法用于在 SparkContext 的 checkpointDir 属性指定的 RDD 计算过程中保存检查点的目录下创建子目录，作为保存 ReliableRDDCheckpointData 所关联的 RDD 数据的检查点目录。ReliableRDDCheckpointData 的伴生对象提供的 checkpointPath 方法如代码清单 10-13 所示。

代码清单 10-13　创建保存 RDD 数据的检查点目录

def checkpointPath(sc: SparkContext, rddId: Int): Option[Path] = {  
 sc.checkpointDir.map { dir => new Path(dir, s"rdd-$rddId") }  
}

**2.doCheckpoint**

ReliableRDDCheckpointData 实现了父类 RDDCheckpointData 定义的 doCheckpoint 方法如代码清单 10-14 所示。

代码清单 10-14　保存检查点

protected override def doCheckpoint(): CheckpointRDD[T] = {  
 val newRDD = ReliableCheckpointRDD.writeRDDToCheckpointDirectory(rdd, cpDir)  
  
 if (rdd.conf.getBoolean("spark.cleaner.referenceTracking.cleanCheckpoints", false)) {  
 rdd.context.cleaner.foreach { cleaner =>  
 cleaner.registerRDDCheckpointDataForCleanup(newRDD, rdd.id)  
 }  
 }  
  
 logInfo(s"Done checkpointing RDD ${rdd.id} to $cpDir, new parent is RDD ${newRDD.id}")  
 newRDD  
}

根据代码清单 10-14，ReliableRDDCheckpointData 实现的 doCheckpoint 方法的执行步骤如下。

1. 调用 ReliableCheckpointRDD 的伴生对象的 writeRDDToCheckpointDirectory 方法（见代码清单 10-6）将 RDD 的数据保存到检查点目录。
2. 如果 spark.cleaner.referenceTracking.cleanCheckpoints 属性指定为 true，那么将生成的 ReliableCheckpointRDD 注册到 SparkContext 的子组件 ContextCleaner 的 referenceBuffer 中，以便于 ContextCleaner 对 ReliableCheckpointRDD 进行清理。
3. 返回生成的 ReliableCheckpointRDD。

## 10.4　RDD 的再次分析

笔者早在 7.2.2 节就对 RDD 的实现进行了分析，但当时只介绍了与调度系统相关的 API。RDD 还提供了很多其他类型的 API，包括对 RDD 进行转换的 API、对 RDD 进行计算（动作）的 API 及 RDD 检查点相关的 API。转换 API 里的计算是延迟的，也就是说调用转换 API 不会向 Spark 集群提交 Job，更不会执行转换计算。只有调用了动作 API，才会提交 Job 并触发对转换计算的执行。由于 RDD 提供的 API 非常多，本书不可能一一展示。由于在 10.8 节将要介绍 word count 的例子，因此本节主要挑选 word count 例子中使用到的 API 进行分析。

**10.4.1　转换 API**

转换（transform）是指对现有 RDD 执行某个函数后转换为新的 RDD 的过程。转换前的 RDD 与转换后的 RDD 之间具有依赖和血缘关系。RDD 的多次转换将创建出多个 RDD，这些 RDD 构成了一张单向依赖的图，也就是 DAG。下面挑选 10.8 节的 word count 例子所涉及的转换 API 进行介绍。

**1.mapPartitions**

mapPartitions 方法用于将 RDD 转换为 MapPartitionsRDD，其实现如代码清单 10-15 所示。

代码清单 10-15　RDD 的 mapPartitions 方法

def mapPartitions[U: ClassTag](  
 f: Iterator[T] => Iterator[U],  
 preservesPartitioning: Boolean = false): RDD[U] = withScope {  
 val cleanedF = sc.clean(f)  
 new MapPartitionsRDD(  
 this,  
 (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(iter),  
 preservesPartitioning)  
}

为便于理解，这里假设函数 f 的作用是过滤出大于 0 的数字，那么 mapPartitions 方法的执行可以用图 10-3 表示。

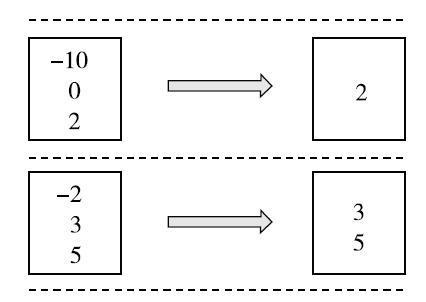
**2.mapPartitionsWithIndex**

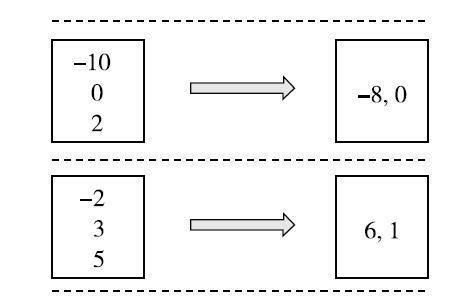
mapPartitionsWithIndex 方法（见代码清单 10-16）用于创建一个将与分区索引相关的函数应用到 RDD 的每个分区的 MapPartitionsRDD。

代码清单 10-16　RDD 的 mapPartitionsWithIndex 方法

def mapPartitionsWithIndex[U: ClassTag](  
 f: (Int, Iterator[T]) => Iterator[U],  
 preservesPartitioning: Boolean = false): RDD[U] = withScope {  
 val cleanedF = sc.clean(f)  
 new MapPartitionsRDD(  
 this,  
 (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),  
 preservesPartitioning)  
}

mapPartitionsWithIndex 与 mapPartitions 相似，区别在于多接收分区索引的参数。我们假设函数 f 的作用是将每个分区的数字累加并且与分区索引以逗号分隔输出，那么 mapPartitionsWithIndex 方法的执行可以用图 10-4 表示。

图 10-3　mapPartitions 示例

图 10-4　mapPartitionsWithIndex 示例

**3.mapPartitionsWithIndexInternal**

mapPartitionsWithIndexInternal 方法（见代码清单 10-17）用于创建一个将函数应用到 RDD 的每个分区的 MapPartitionsRDD。由于此方法是私有的，所以只在 Spark SQL 内部使用。

代码清单 10-17　RDD 的 mapPartitionsWithIndexInternal 方法

private[spark] def mapPartitionsWithIndexInternal[U: ClassTag](  
 f: (Int, Iterator[T]) => Iterator[U],  
 preservesPartitioning: Boolean = false): RDD[U] = withScope {  
 new MapPartitionsRDD(  
 this,  
 (context: TaskContext, index: Int, iter: Iterator[T]) => f(index, iter),  
 preservesPartitioning)  
}

**4.flatMap**

flatMap 方法（见代码清单 10-18）用于向 RDD 中的所有元素应用函数，并对结果扁平化处理。

代码清单 10-18　RDD 的 flatMap 方法

def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {  
 val cleanF = sc.clean(f)  
 new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.flatMap(cleanF))  
}

根据代码清单 10-18，flatMap 方法也将返回 MapPartitionsRDD。我们假设函数 f 的作用是将给每个数字加上 5，那么 flatMap 方法的执行可以用图 10-5 表示。

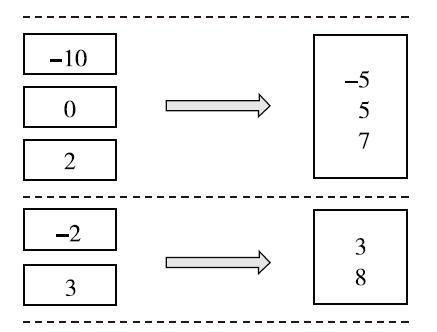
**5.map**

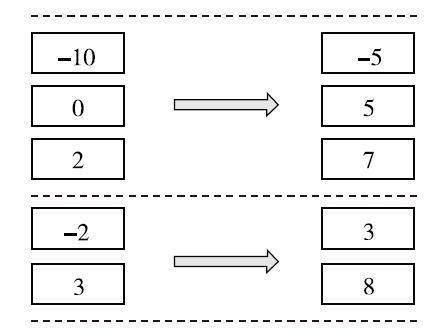
map 方法（见代码清单 10-19）用于向 RDD 中的所有元素应用函数。

代码清单 10-19　RDD 的 map 方法

def map[U: ClassTag](f: T => U): RDD[U] = withScope {  
 val cleanF = sc.clean(f)  
 new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))  
}

我们假设函数 f 的作用是将给每个数字加上 5，那么 map 方法的执行可以用图 10-6 表示。

图 10-5　flatMap 示例

图 10-6　map 示例

**6.toJavaRDD**

toJavaRDD 方法用于将 RDD 自己转换为 JavaRDD，其实现如代码清单 10-20 所示。

代码清单 10-20　RDD 的 toJavaRDD 方法

def toJavaRDD() : JavaRDD[T] = {  
 new JavaRDD(this)(elementClassTag)  
}

有了以上几个转换 API 的分析，读者对于转换的含义应该有更清楚的认识了。

**10.4.2　动作 API**

由于转换 API 都是预先编织好，但是不会执行的，所以 Spark 需要一些 API 来触发对转换的执行。动作 API 触发对数据的转换后，将接收到一些结果数据，动作 API 因此还具备对这些数据进行收集、遍历、叠加的功能。下面挑选 10.8 节的 word count 例子使用的动作 API——collect 进行介绍。此外再介绍 foreach 和 reduce 两个动作 API。

**1.collect**

collect 方法（见代码清单 10-21）将调用 SparkContext 的 runJob 方法（见代码清单 4-29）提交基于 RDD 的所有分区上的作业，并返回数组形式的结果。

代码清单 10-21　RDD 的 collect 方法

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

collect 可以用图 10-7 来表示。

**2.foreach**

foreach 方法（见代码清单 10-22）将调用 SparkContext 的 runJob 方法（见代码清单 4-29）提交将函数应用到 RDD 中所有元素的作业。

代码清单 10-22　RDD 的 foreach 方法

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

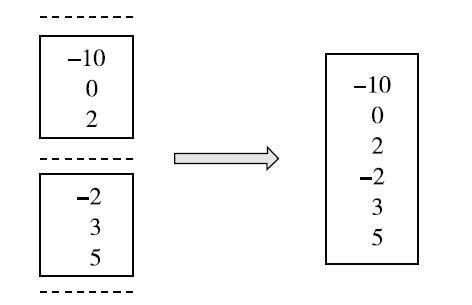
**3.reduce**

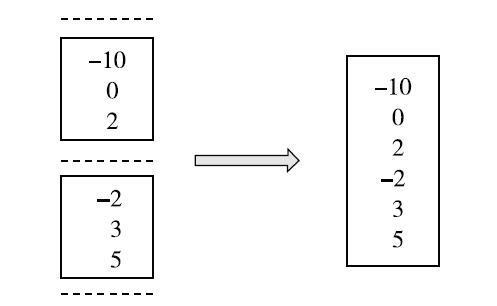
reduce 方法按照指定的函数对 RDD 中的元素进行叠加操作，其实现如代码清单 10-23 所示。

代码清单 10-23　RDD 的 reduce 方法

def reduce(f: (T, T) => T): T = withScope {  
 val cleanF = sc.clean(f)  
 val reducePartition: Iterator[T] => Option[T] = iter => {  
 if (iter.hasNext) {  
 Some(iter.reduceLeft(cleanF))  
 } else {  
 None  
 }  
 }  
 var jobResult: Option[T] = None  
 val mergeResult = (index: Int, taskResult: Option[T]) => {  
 if (taskResult.isDefined) {  
 jobResult = jobResult match {  
 case Some(value) => Some(f(value, taskResult.get))  
 case None => taskResult  
 }  
 }  
 }  
 sc.runJob(this, reducePartition, mergeResult)  
 jobResult.getOrElse(throw new UnsupportedOperationException("empty collection"))  
}

为了便于说明 reduce 的作用，这里假设函数 f 的定义是：（L，R）=>（L+R），那么可以用图 10-8 来表示 reduce 的效果。

图 10-7　collect 的示例

图 10-8　reduce 示例

**10.4.3　检查点 API 的实现分析**

RDD 中提供了很多与检查点相关的 API，通过对这些 API 的使用，Spark 应用程序才能够启用、保存及使用检查点，提高应用程序的容灾和容错能力。下面进行介绍。

**1.检查点的启用**

用户提交的 Spark 作业必须主动调用 RDD 的 checkpoint 方法（见代码清单 10-24），才会启动检查点功能。

代码清单 10-24　RDD 的 checkpoint 方法

def checkpoint(): Unit = RDDCheckpointData.synchronized {  
 if (context.checkpointDir.isEmpty) {  
 throw new SparkException("Checkpoint directory has not been set in the SparkContext")  
 } else if (checkpointData.isEmpty) {  
 checkpointData = Some(new ReliableRDDCheckpointData(this))  
 }  
}

根据代码清单 10-24，给 SparkContext 指定 checkpointDir 是启用检查点机制的前提。可以使用 SparkContext 的 setCheckpointDir 方法（见代码清单 4-30）设置 checkpointDir。如果没有指定 RDDCheckpointData，那么创建 ReliableRDDCheckpointData。

**2.检查点的保存**

RDD 的 doCheckpoint 方法用于将 RDD 的数据保存到检查点，其实现如代码清单 10-25 所示。由于此方法是私有的，只能在 RDD 内部使用。

代码清单 10-25　RDD 的 doCheckpoint 方法

private[spark] def doCheckpoint(): Unit = {  
 RDDOperationScope.withScope(sc, "checkpoint", allowNesting = false, ignore-Parent = true) {  
 if (!doCheckpointCalled) {  
 doCheckpointCalled = true  
 if (checkpointData.isDefined) {  
 if (checkpointAllMarkedAncestors) {  
 dependencies.foreach(\_.rdd.doCheckpoint())  
 }  
 checkpointData.get.checkpoint()  
 } else {  
 dependencies.foreach(\_.rdd.doCheckpoint())  
 }  
 }  
 }  
}

根据代码清单 10-25，doCheckpoint 方法的执行步骤如下。

1. 如果 checkpointData 中保存了 RDDCheckpointData，调用 RDDCheckpointData 的 checkpoint 方法（见代码清单 10-11）保存检查点。如果需要对祖先 RDD 保存检查点，那么还会调用每个依赖的 RDD 的 doCheckpoint 方法。由于在启用检查点时，保存到 check-pointData 中的是 RDDCheckpointData 的子类 ReliableRDDCheckpointData，因此 RDDCheck-pointData 的 checkpoint 方法中将调用 ReliableRDDCheckpointData 的 doCheckpoint 方法（见代码清单 10-14）。
2. 如果 checkpointData 中没有保存 RDDCheckpointData，那么调用每个依赖的 RDD 的 doCheckpoint 方法。

**3.检查点的使用**

在 7.2.2 节曾介绍过获取 RDD 的分区数组的 partitions 方法（见代码清单 7-1）、获取指定分区的偏好位置的 preferredLocations 方法（见代码清单 7-2）、获取当前 RDD 的所有依赖的 dependencies 方法（见代码清单 7-3）。虽然这几个方法的作用不同，但是实现方式却是类似的，即首先从 RDD 关联的 CheckpointRDD 中查找对应信息。

根据对检查点的启用和保存的分析，负责为 RDD 提供检查点服务的实际是 Reli-ableCheckpointRDD。因此当调用 RDD 的 partitions 方法时，会首先调用 ReliableCheck-pointRDD 的 partitions 方法，进而调用 ReliableCheckpointRDD 的 getPartitions 方法（见代码清单 10-8），最后才调用 RDD 自己的 getPartitions 方法。当调用 RDD 的 preferred-Locations 方法时，首先会调用 ReliableCheckpointRDD 的 getPreferredLocations 方法（见代码清单 10-9），当调用 RDD 的 dependencies 方法时，首先会尝试将 ReliableCheck-pointRDD 封装为 OneToOneDependency。

除了以上场景外，对 RDD 的迭代计算也涉及对检查点的使用，其中将调用 Reliable-CheckpointRDD 的 compute 方法（见代码清单 10-10）。迭代计算的内容将在下一小节介绍。

**10.4.4　迭代计算**

在 8.5.3 节和 8.5.4 节分析 ShuffleMapTask 和 ResultTask 的 runTask 方法时已经看到，Task 的执行离不开对 RDD 的 iterator 方法的调用。RDD 的 iterator 方法是迭代计算的入口，其实现如代码清单 10-26 所示。

代码清单 10-26　迭代计算的入口

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

根据代码清单 10-26，iterator 方法的执行步骤如下。

1. 如果 RDD 的存储级别（StorageLevel）不是 NONE，那么根据 6.1.2 节对 StorageLevel 的分析，StorageLevel 的构造器是私有的，而 Spark 内置的存储级别都定义在代码清单 6-7 中。这些内置的存储级别除 NONE 外，至少会使用磁盘、堆内内存、堆外内存三者之一，因此可以调用 getOrCompute 方法（见代码清单 10-27）从这些存储中尝试获取计算结果。
2. 如果 RDD 的存储级别（StorageLevel）是 NONE，那么说明分区任务可能是初次执行且存储中还没有任务的执行结果，所以会调用 computeOrReadCheckpoint 方法（见代码清单 10-28）计算或者从检查点恢复。

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

代码清单 10-27　获取或计算 RDD 的分区

private[spark] def getOrCompute(partition: Partition, context: TaskContext): Iterator[T] = {  
 val blockId = RDDBlockId(id, partition.index)  
 var readCachedBlock = true  
 SparkEnv.get.blockManager.getOrElseUpdate(blockId, storageLevel, elementClass-Tag, () => {  
 readCachedBlock = false  
 computeOrReadCheckpoint(partition, context)  
 }) match {  
 case Left(blockResult) =>  
 if (readCachedBlock) {  
 val existingMetrics = context.taskMetrics().inputMetrics  
 existingMetrics.incBytesRead(blockResult.bytes)  
 new InterruptibleIterator[T](context, blockResult.data.asInstanceOf[Iterator-[T]]) {  
 override def next(): T = {  
 existingMetrics.incRecordsRead(1)  
 delegate.next()  
 }  
 }  
 } else {  
 new InterruptibleIterator(context, blockResult.data.asInstanceOf[Iterator[T]])  
 }  
 case Right(iter) =>  
 new InterruptibleIterator(context, iter.asInstanceOf[Iterator[T]])  
 }  
}

根据代码清单 10-27，getOrCompute 方法的执行步骤如下。

1. 调用 BlockManager 的 getOrElseUpdate 方法（见代码清单 6-81）先尝试从存储体系中获取 RDD 分区的 Block，否则调用 computeOrReadCheckpoint 方法（见代码清单 10-28）从检查点读取或计算。
2. 对 getOrElseUpdate 方法返回的结果进行匹配，将返回的 BlockResult 的 data 属性或返回的 Iterator 封装为 InterruptibleIterator。

computeOrReadCheckpoint 方法在存在检查点时直接从检查点读取数据，否则需要调用 compute 继续计算。computeOrReadCheckpoint 方法的实现如代码清单 10-28 所示。

代码清单 10-28　计算或从检查点读取数据

private[spark] def computeOrReadCheckpoint(split: Partition, context: Task-Context): Iterator[T] =  
{  
 if (isCheckpointedAndMaterialized) {  
 firstParent[T].iterator(split, context)  
 } else {  
 compute(split, context)  
 }  
}  
  
private[spark] def isCheckpointedAndMaterialized: Boolean =  
checkpointData.exists(\_.isCheckpointed)

根据代码清单 10-28，computeOrReadCheckpoint 方法的执行步骤如下。

1. 如果 checkpointData 中保存了 RDDCheckpointData 且其检查点的状态（cpState）是 Checkpointed，那么调用 firstParent 方法（见代码清单 10-29）找到其父 RDD，然后调用父 RDD 的 iterator 方法。由于 firstParent 中调用了 dependencies，且当前 RDD 的父 RDD 实际是 ReliableCheckpointRDD，那么对 ReliableCheckpointRDD 的 iterator 方法的调用最终将转变为对 ReliableCheckpointRDD 的 compute 方法（见代码清单 10-10）的调用，从而从检查点文件读取之前保存的计算结果。
2. 如果 checkpointData 中没有保存 RDDCheckpointData 或其检查点的状态（cpState）不是 Checkpointed，那么调用 compute 方法进行计算。

代码清单 10-29　找到父亲 RDD

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

每个 RDD 实现的 compute 方法都不相同。在代码清单 10-10 中曾经介绍了 Reliable-CheckpointRDD 的 compute 方法。此处再以 MapPartitionsRDD 和 ShuffledRDD 为例，来看看它们各自实现的 compute 方法。

MapPartitionsRDD 实现的 compute 方法如代码清单 10-30 所示。

代码清单 10-30　MapPartitionsRDD 的 compute 方法

override def compute(split: Partition, context: TaskContext): Iterator[U] =  
 f(context, split.index, firstParent[T].iterator(split, context))

ShuffledRDD 实现的 compute 方法如代码清单 10-31 所示。

代码清单 10-31　ShuffledRDD 的 compute 方法

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

可以看到，ShuffledRDD 的 compute 方法首先调用 SortShuffleManager 的 getReader 方法（见代码清单 8-96）获取 BlockStoreShuffleReader，然后调用 BlockStoreShuffleReader 的 read 方法（见代码清单 8-90）获取 map 任务输出的 Block 并在 reduce 端进行聚合或排序。