

第二章

Spark 快速入门

本章的主要内容包括 Spark 的下载和本地模式的单机运行，主要面向于新接触 Spark 的数据科学家和工程师。

Spark 支持 Python、Java 和 Scala 编程语言。您无需是专家级的编程者即可从本书中受益，我们假设您已熟悉三种编程语言之一。我们尽量将所有的示例都用三种编程语言表达。

Spark 本身用 Scala 语言编写，运行于 Java 虚拟机（JVM）。只要在安装了 Java 6 以上版本的便携式计算机或者集群上都可以运行 spark。如果您想使用 Python API 需要安装 Python 解释器（2.6 或者更高版本），请注意 Spark 暂不支持 Python 3。

下载 Spark

首先下载 Spark 并解压，我们从下载预编译版本的 Spark 开始。在浏览器中访问 <http://spark.apache.org/download.html> 选择 “Pre-built for Hadoop 2.4 and later” 安装包，点击 “Direct Download” 下载名称为 *spark-1.2.0-bin-hadoop2.4.tgz* 的压缩包。



Windows 用户安装时可能会遇到文件夹名称中包含空格的问题，建议 Spark 的安装目录的文件夹中不包含空格，比如 `C:\spark`。

您不需要安装 Hadoop 即可运行 Spark，但是如果您已有 Hadoop 集群或者 HDFS 则需要下载对应的 Spark 版本。您可在 <http://spark.apache.org/downloads.html> 选择不同的安装包，这些安装包的文件名会有所不同。也可以将 Spark 源码重新编译，您可在 Github 下载最新的 Spark 源代码。



大多数 Unix 和 Linux 操作系统，包括 Mac OS X，都包含 tar 命令行解压工具。如果您的操作系统没有安装 tar 的命令行工具，请在互联网搜索免费的解压缩工具。比如在 Windows 系统中您可以使用 7-Zip。

现在我们将已下载的 Spark 解压缩，看看默认的 Spark 分布式。打开终端，切换至下载 Spark 的目录下将其解压缩。执行下面的代码将创建一个与压缩文件同名的新目录。

```
cd ~  
  
tar -xf spark-1.2.0-bin-hadoop2.4.tgz  
  
cd spark-1.2.0-bin-hadoop2.4  
  
ls
```

在包含 tar 的执行命令中，x 表示解压缩，f 表示指定 tar 包名称。ls 命令将列出 Spark 目录下的所有文件。让我们简要介绍下 Spark 目录中的重要文件。

README.md

包含 Spark 入门的简要说明。

bin

包含与 Spark 交互的可执行文件（比如在本章后面介绍的 Spark Shell）

core, streaming, python, ...

包含 Spark 工程主要组件的源码

examples

包含可在 Spark 单机版运行的作业，您可从中了解 Spark API。

您不必对 Spark 工程中包含的如此多的目录和文件所困扰，本书后续章节会涵盖其中的大部分技术内容。现在，让我们深入 Spark 的 Python 和 Scala 交互式 shell。我们将从运行 Spark 官方示例开始，然后编写和运行自己的 Spark 作业。

本章中的 Spark 作业运行于单机模式，即在本地计算机运行的非分布式的模式。Spark 可在不同模式不同环境中运行。除了单机模式，Spark 还可运行于 Mesos 和 YARN，以及 Spark 分布式下的独立调度。我们将在第七章中详细介绍各种部署模式。

Spark 的 Python 和 Scala 交互式 Shell

Spark 的交互式 shell 支持可执行的数据分析。如果您使用其他的 shell 编程，那么您将会对 Spark shell 感觉很亲切。比如 R、Python 和 Scala shell，以及批处理的操作系统编程或者 Windows 命令提示符。

与其他的 Shell 编程只能操作单台计算机的磁盘和内存不同的是，Spark Shell 支持跨多台计算机的分布式磁盘和内存计算，并且 Spark 会自动执行分布式作业处理。

因为 Spark 将数据加载至工作节点内存中，绝大多数分布式计算甚至处理 TB 级的数据也仅需几秒钟。这使得 Spark 适合处理迭代排序、随机和未知分析。Spark 的 Python 和 Scala 的 shell 均支持集群连接。



这本书中绝大部分代码包含 Spark 支持的所有语言，但是交互式 shell 仅支持 Python 和 Scala 语言。因为 shell 是非常有效的学习 API 的方法，我们建议您使用本书中 Python 或者 Scala 语言的示例学习，即使您是一位 Java 开发者。每种语言的 API 差别不大。

让我们用一个简单的数据分析的例子来感受一下 spark shell 的强大，按照 Spark 官方文档的快速入门的步骤：

第一步是打开 Spark 交互式 shell。若要打开 Python 版本的 Spark shell，即 PySpark shell，在 Spark 目录中输入如下指令：

```
bin/pyspark
```

(或者在 Windows 中输入 bin\pyspark)

打开 Scala 版本的 shell，输入：

```
bin/spark-shell
```

shell 提示符应在几秒钟后出现。当 shell 启动时，您会注意到有大量的日志消息提示。您可按下 Enter 键清除日志输出，图 2-1 显示的是打开 PySpark shell 的显示界面。

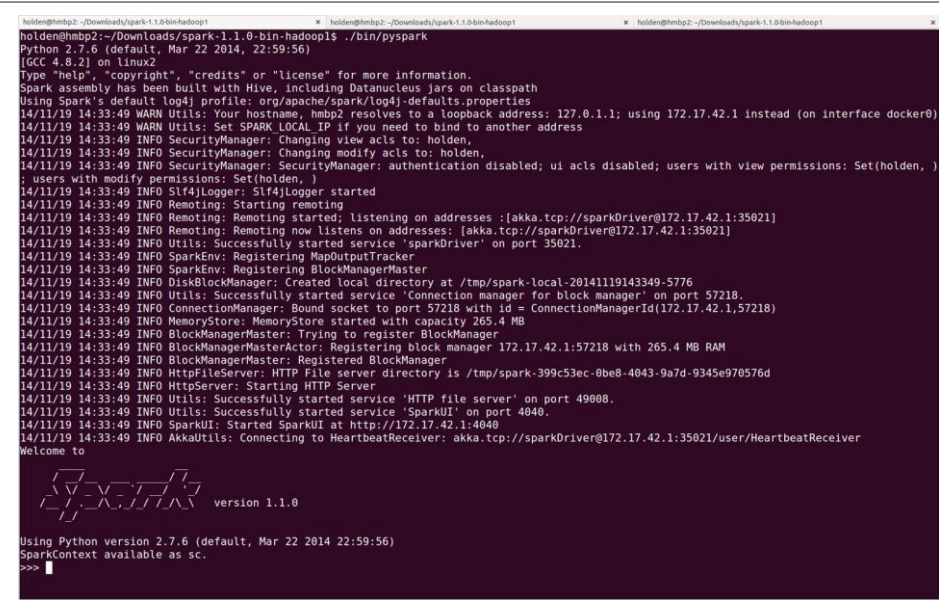
The image shows a terminal window titled 'holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1'. The user has executed 'bin/pyspark'. The terminal displays a series of log messages from the Spark framework, including warnings about local IP resolution and information about security manager settings, remoting, and the successful start of various services like MapOutputTracker, BlockManager, and HTTP server. At the bottom, the PySpark logo is displayed along with the text 'version 1.1.0' and 'Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)'. The prompt 'SparkContext available as sc.' is shown, followed by a 'Welcome to' message and a '++>' prompt.

图 2-1 PySpark shell 的默认日志输出

在 shell 中您可以看到打印的日志信息，您也可以控制日志的详细程度。在 *conf* 目录中创建名称为 *log4j.properties* 的文件，Spark 提供了该文件的模板

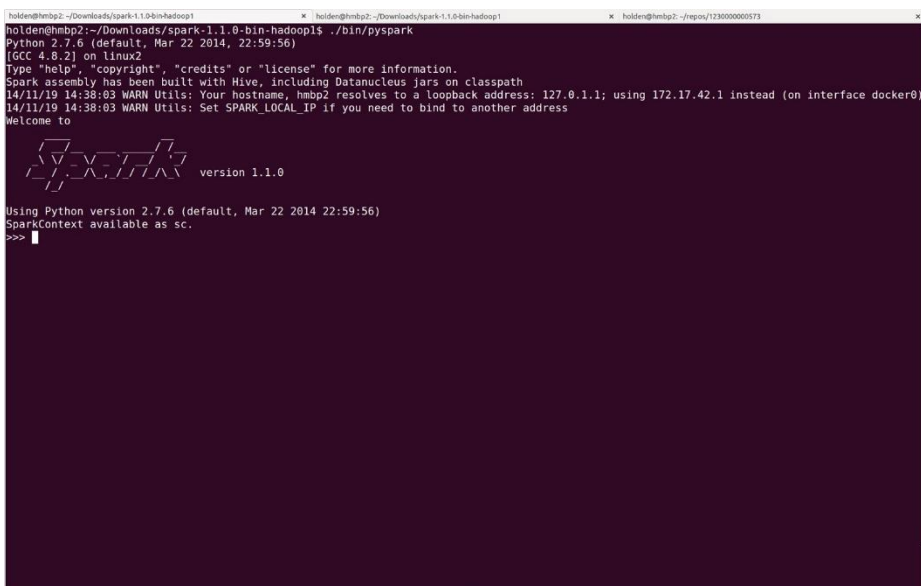
`log4j.properties.template` 。若不需要输出那么冗长的日志，您可以复制该模板并将其改名为 `log4j.properties`，在模板的复制文件中找到下面的代码：

```
log4j.rootCategory=INFO, console
```

降低日志的级别只显示警告信息，将上面的代码修改如下：

```
log4j.rootCategory=WARN, console
```

重新打开 shell，您可以看见输出信息减少了。



```
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
14/11/19 14:38:03 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:38:03 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Welcome to
      ____
     / ___/
    / __/   version 1.1.0
   /___/

Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.
>>>
```

图 2-2. PySpark shell 输出信息减少



使用 IPython

IPython 是颇受 python 使用者追捧的增强版 Python shell，提供诸如 tab 键完成功能。更多信息请查看 <http://ipython.org>。将 IPYTHON 的环境变量设置为 1 即可在 Spark 中使用 IPython。

```
IPYTHON=1 ./bin/pyspark
```

To use the IPython Notebook, which is a web-browser-based version of IPython, use:

若要使用基于浏览器的 IPython Notebook，请使用如下指令：

```
IPYTHON_OPTS="notebook" ./bin/pyspark
```

在 Windows 中设置变量的方法如下：

```
set IPYTHON=1
bin\pyspark
```

在 Spark 中我们通过操作集群的分布式集合进行自动化并行计算，这些集合被称为弹性分布式数据集，或者 RDDs。RDDs 是 Spark 做分布式数据和计算的基础抽象。

在我们说更多的 RDD 之前，让我们创建一个 shell 程序读取本地文本文件并计算简单的特定分析。下面的示例 2-1 是 Python 语音，示例 2-2 是 Scala 语言。

示例 2-1. Python line count

```
>>> lines = sc.textFile("README.md") # Create an RDD called lines
>>> lines.count() # Count the number of items in this RDD
127
>>> lines.first() # First item in this RDD, i.e. first line of
README.md u'# Apache Spark'
```

示例 2-2. Scala line count

```
scala> val lines = sc.textFile("README.md") // Create an RDD
called lines lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // Count the number of items in
this RDD res0: Long = 127

scala> lines.first() // First item in this RDD, i.e. first line of
README.md res1: String = # Apache Spark
```

若要退出 shell，按下 Ctrl-D。



更多的内容在第七章中讨论。您会注意到一条信息：INFO SparkUI: Started SparkUI at [http://\[ipaddress\]:4040](http://[ipaddress]:4040)。您可以通过此 Spark UI 看见更多任务和集群的信息。

在示例 2-1 和 2-2 中，变量 `lines` 为 RDD，它是在本地机器中读取文本文件后被创建的。我们可以对此 RDD 运行各种并行操作，比如在数据集（这里指文件中文本的行数）中统计元素的数量，或者打印元素。在后面的章节中我们将深入讨论 RDD，在这个之前我们先花点时间介绍 Spark 的基本概念。

Spark 核心概念

现在您已经在 shell 中运行了第一个 Spark 代码，是时候开始学习更深入的编程了。

每一个 Spark 应用程序都包含一个在集群上运行各种并行操作的驱动程序，驱动程序包含应用程序的主函数和定义在集群上的分布式数据集。在前面的示例中，驱动程序就是 Spark shell 本身，您只需输入您想要执行的操作即可。

驱动程序通过一个链接到计算集群上的 `SparkContext` 对象访问 Spark 计算集群，在 shell 中，`SparkContext` 被自动创建为名称是 `sc` 的变量，在示例 2-3 中我们输入 `sc`，则 shell 显示其类型。

Example 2-3. Examining the `sc` variable

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

在创建了 `SparkContext` 对象之后，您就可创建 RDD 了。在示例 2-1 和示例 2-2 中，我们调用 `sc.textFile()` 创建一个代表文件中文本行数的 RDD。然后，我们就可以在这些行上进行各种操作，例如 `count()`

若要运行这些操作，驱动程序通常管理者多个拥有 `executor` 的工作节点。比如，我们在集群中执行 `count()` 操作，不同的机器可能计算 `lines` 变量不同的部分。我们只在本地运行 Spark shell，则它被执行在单机中，如果我们将 shell 连接至集群它也可并行的分析数据。示例 2-3 展示了 Spark 如何在集群上执行。

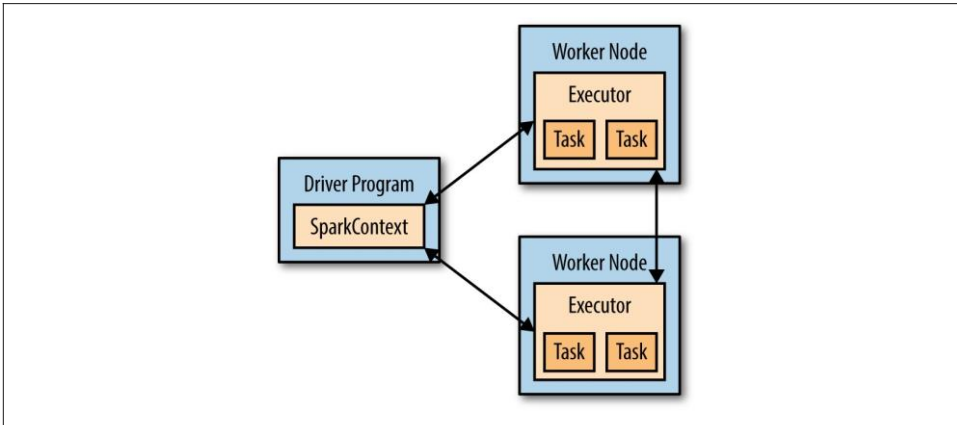


图 2-3. Components for distributed execution in Spark

Spark 的 API 很大程度上依靠在驱动程序里传递函数到集群上运行。比如，我们扩展上面的 README 示例，筛选文本中包含的特定关键词 “Python” 的行，代码如下示例 2-4 (Python)，示例 2-5 (Scala)。

示例 2-4 Python filtering example

```
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first() u'## Interactive Python Shell'
```

Example 2-5. Scala filtering example

```
scala> val lines = sc.textFile("README.md") // Create an RDD
called lines lines: spark.RDD[String] = MappedRDD[...]

scala> val pythonLines = lines.filter(line =>
line.contains("Python")) pythonLines: spark.RDD[String] =
FilteredRDD[...]

scala> pythonLines.first() res0:
String = ## Interactive Python
Shell
```


Spark 传递函数

如果您不熟悉示例 2-4 和 2-5 中的 lambda 表达式 或者 `=>` 语法，那么在此说明其实它是在 Python 和 Scala 中的定义内联函数的简短写法。如果您在 Spark 中使用这些语言，您可定义函数然后将其名称传递给 Spark。比如，在 Python 语言中：

```
def hasPython(line):  
    return "Python" in line  
pythonLines = lines.filter(hasPython)
```

Spark 传递函数也支持 Java 语言，但在此情况下传递函数被定义为类，实现调用函数的接口。比如：

```
JavaRDD<String> pythonLines = lines.filter(  
    new Function<String, Boolean>() {  
        Boolean call(String line) { return  
line.contains("Python"); }  
    }  
);
```

Java 8 中介绍了调用了 lambda 的的简短写法，与 Python 和 Scala 很类似。

```
JavaRDD<String> pythonLines = lines.filter(line ->  
line.contains("Python"));
```

We discuss passing functions further in [“Passing Functions to Spark” on page 30](#). 我们将在 30 页的 “Spark 传递函数” 中深入讨论传递函数。

Spark API 包含许多魅力无穷的基于函数的操作可基于集群并行计算，比如筛选（filter）操作，我们在后面的章节详细介绍。Spark 自动将您的函数传递给执行（executor）节点。因此，您可在单独的驱动程序中编写代码，它会自动的在多个节点中运行。本书第三章涵盖了 RDD API 的详细介绍。

独立（Standalone）应用程序

Spark 快速入门教程中缺少如何在独立（Standalone）应用程序中使用 Spark，其实 Spark 除了可以交互式 shell 运行，还可以在 Java、Scala 和 Python 的独立应用程序中依赖 Spark 运行。唯一与 shell 不同的是，独立应用程序中需要初始化 SparkContext，除此之外所有的 API 都是相同的。

在独立应用程序中依赖 Spark 的方法因语言而异。在 Java 和 Scala 中，您可以在设置 Spark 核心的 Maven 依赖。随着本书版本的书写，最新的 spark 版本为 1.2.0，相应的 Maven 依赖设置为：

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

Maven 是最受欢迎的基于 Java 语言的包管理工具，可以链接至公共的资源库。您可以使用 Maven 创建自己的应用程序，也可以使用其他的工具比如 Scala 的 sbt 或者 Gradle 创建。流行的集成开发环境如 Eclipse 允许直接添加 Maven 依赖至工程中。

在 Python 中，您可编写 Python 脚本的应用程序，然后使用 bin/spark-submit 提交脚本至 Spark 运行。在 spark-submit 脚本中包含供 Python 使用的 Spark 依赖，在此脚本中设置 Spark 的 Python API 的运行环境。

示例 2-6 运行 Python 脚本

Example 2-6. Running a Python script

```
bin/spark-submit my_script.py
```

（请注意在 Windows 中使用反斜杠\替代正斜杠/。）

初始化 SparkContext

如果您将应用程序链接至 Spark，则需在应用程序中引入 Spark 包并创建 SparkContext。首先创建 SparkConf 对象配置应用程序，然后实例化

SparkContext。示例 2-7 至 2-9 以三种语言展示初始化 SparkContext 的方法。

Example 2-7. Initializing Spark in Python

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")

sc = SparkContext(conf = conf)
```

Example 2-8. Initializing Spark in Scala

```
import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

val conf = new SparkConf().setMaster("local").setAppName("My App")

val sc = new SparkContext(conf)
```

Example 2-9. Initializing Spark in Java

```
import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("My
App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

这些示例展示最简单的初始化 SparkContext 的方法，其中传递了两个参数：

- 集群 URL 参数，代表 Spark 连接到集群的方式，本例中设定为 local，表示 Spark 线程仅运行于本地机器而非连接至集群。
- 应用程序名称参数，本例中被定义为 My App，如果您连接至集群，可在集群管理的 UI 界面中通过应用的名称找到您自己的应用程序。

关于应用程序执行或者提交至集群的附加参数配置，将在本书后面的章节中介绍。

在您初始化 `SparkContext` 之后，即可调用我们之前展示给您的所有方法来创建 RDD(比如从文本文件读取)并操纵他们。

最后，您可调用 `stop()` 方法关闭 Spark，或者简单的退出该应用程序（比如 `System.exit(0)` 或者 `sys.exit()`）

以上足以让您在笔记本电脑上运行一个单机(Standalone)的 Spark 应用程序。对于更高级的配置，第七章中将介绍如何将应用程序连接至集群，以及如何将应用程序打包以便代码自动提交至工作节点。目前，我们还是参照 Spark 官方文档的快速入门。

创建独立 (Standalone) 应用程序

如果本章没有字数统计的示例，那么就不是完整大数据图书的导论章节。在单机中运行字数统计的程序很简单，但是在分布式框架中它却是一个常见的示例，因为他需要在众多的工作节点中读取和合并数据。接下来我们分别以 sbt 和 Maven 的方式创建和打包简单的字数统计的示例。我们所有的示例本都可以一起编译，但是为了说明这种最小依赖的精简编译方式，我们将其分解为多个小的程序，代码示例在目录 `learning-sparkexamples/mini-complete-example` 下，您可参阅示例 2-10 (Java) 和 2-11 (Scala)。

Example 2-10. Word count Java application—don't worry about the details yet

```
// Create a Java Spark Context
SparkConf conf = new SparkConf().setAppName("wordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
// Load our input data.
```

```

JavaRDD<String> input = sc.textFile(inputFile);

// Split up into words.
JavaRDD<String> words = input.flatMap(
    new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) {
            return Arrays.asList(x.split(" "));
        }
    });

// Transform into pairs and count.
JavaPairRDD<String, Integer> counts = words.mapToPair(
    new PairFunction<String, String, Integer>(){
        public Tuple2<String, Integer> call(String x){
            return new Tuple2(x, 1);
        }
    }).reduceByKey(new Function2<Integer, Integer, Integer>(){
        public Integer call(Integer x, Integer y){ return x + y;});

// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile);

```

Example 2-11. Word count Scala application—don't worry about the details yet

```

// Create a Scala Spark Context. val conf = new
SparkConf().setAppName("wordCount")

val sc = new SparkContext(conf)

// Load our input data.
val input = sc.textFile(inputFile)

// Split it up into words.
val words = input.flatMap(line => line.split(" "))

// Transform into pairs and count.
val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}

// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile)

```

我们可以使用非常简单的编译文件比如 sbt (示例 2-12) 示例 2-12 和 Maven (示例 2-13) 创建应用程序。我们以 provided 标签标记了 Spark 的核心依赖，以便在稍后的编程中我们可以使用该程序集，而不必导入 spark-coreJAR 包。

Example 2-12. sbt build file

```
name := "learning-spark-mini-example"
version := "0.0.1"
scalaVersion := "2.10.4"
// additional libraries
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.2.0" % "provided"
)
```

Example 2-13. Maven build file

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId> <artifactId>learning-
spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
```

```

    <version>1.2.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<properties>
  <java.version>1.6</java.version>
</properties>
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>${java.version}</source>
          <target>${java.version}</target>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
</project>

```



spark-core 包已经被标记为 provided，在应用程序打包时将自动引入该 JAR 包。更详细的内容在第七章中介绍。

一旦有了自己的编译定义文件，我们可以轻松的将应用程序打包并使用 `bin/spark-submit` 脚本运行。`bin/spark-submit` 脚本包含设置 Spark 运行的环境变量参数。在目录中我们可以编译 Scala (示例 2-14) 和 Java (示例 2-15) 应用。

Example 2-14. Scala build and run

```
sbt clean package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.scala.WordCount \
  ./target/... (as above) \
  ./README.md      ./wordcounts
```

Example 2-15. Maven build and run

```
mvn clean && mvn compile && mvn package
$SPARK_HOME/bin/spark-submit \  --class
com.oreilly.learningsparkexamples.mini.java.WordCount
\  ./target/learning-spark-mini-example-0.0.1.jar \
  ./README.md ./wordcounts
```

更详细的 Spark 应用程序的示例请参阅 Spark 官方文档的快速入门。第七章也将详细介绍 Spark 应用程序的打包方法。

结论

在章中我们介绍了下载 Spark，在笔记本电脑中本地运行，使用交互式方法和以独立应用程序的方式运行 Spark。并简要展示了涉及 Spark 编程的核心概念：在驱动程序中创建 `SparkContext` 和 `RDD`，然后执行并行计算的操作。在下一章节中我们将深入介绍 `RDD` 的操作。

About the Authors

Holden Karau is a software development engineer at Databricks and is active in open source. She is the author of an earlier Spark book. Prior to Databricks, she worked on a variety of search and classification problems at Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a Bachelor of Mathematics in Computer Science. Outside of software she enjoys playing with fire, welding, and hula hooping.

Most recently, **Andy Konwinski** cofounded Databricks. Before that he was a PhD student and then postdoc in the AMPLab at UC Berkeley, focused on large-scale distributed computing and cluster scheduling. He cocreated and is a committer on the Apache Mesos project. He also worked with systems engineers and researchers at Google on the design of Omega, their next-generation cluster scheduling system. More recently, he developed and led the AMP Camp Big Data Bootcamps and Spark Summits, and contributes to the Spark project.

Patrick Wendell is a cofounder of Databricks as well as a Spark Committer and PMC member. In the Spark project, Patrick has acted as release manager for several Spark releases, including Spark 1.0. Patrick also maintains several subsystems of Spark's core engine. Before helping start Databricks, Patrick obtained an MS in Computer Science at UC Berkeley. His research focused on low-latency scheduling for largescale analytics workloads. He holds a BSE in Computer Science from Princeton University.

Matei Zaharia is the creator of Apache Spark and CTO at Databricks. He holds a PhD from UC Berkeley, where he started Spark as a research project. He now serves as its Vice President at Apache. Apart from Spark, he has made research and open source contributions to other projects in the cluster computing area, including Apache Hadoop (where he is a committer) and Apache Mesos (which he also helped start at Berkeley).

Colophon

The animal on the cover of *Learning Spark* is a small-spotted catshark (*Scyliorhinus canicula*), one of the most abundant elasmobranchs in the Northeast Atlantic and Mediterranean Sea. It is a small, slender shark with a blunt head, elongated eyes, and

a rounded snout. The dorsal surface is grayish-brown and patterned with many small dark and sometimes lighter spots. The texture of the skin is rough, similar to the coarseness of sandpaper.

This small shark feeds on marine invertebrates including mollusks, crustaceans, cephalopods, and polychaete worms. It also feeds on small bony fish, and occasionally larger fish. It is an oviparous species that deposits egg-cases in shallow coastal waters, protected by a horny capsule with long tendrils.

The small-spotted catshark is of only moderate commercial fisheries importance, however it is utilized in public aquarium display tanks. Though commercial landings are made and large individuals are retained for human consumption, the species is often discarded and studies show that post-discard survival rates are high.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Wood's Animate Creation. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.