第3章

Spark开发基础

3.1 Spark 概述

3.1.1 Spark 简介

Spark 最初由美国加州大学伯克利分校(UC Berkeley)的 AMP 实验室(AMPLab,现在称为 RISELab)于 2009 年开发,是基于内存计算的大数据并行计算框架,可用于构建大型的、低延迟的数据分析应用程序。研究人员之前曾从事 Hadoop MapReduce 的工作,他们承认 MapReduce 对于交互式或迭代计算作业和复杂的学习框架是低效、难以处理的,因此从一开始,他们就有了使 Spark 更简单、更快、更容易使用的想法。

Spark 在诞生之初属于研究性项目,其诸多核心理念均源自学术研究论文。Spark 项目的核心是借用 Hadoop MapReduce 的思想,但强于后者。Spark 的改进包括:提高容错性和并行性,支持内存存储以在迭代和交互式 Map 和 Reduce 之间减少中间计算,提供支持多种编程语言、且简单易用的 API,并以统一的方式支持其他工作负载。2013 年,Spark 加入 Apache 孵化器项目后,开始迅猛发展,目前已成为使用最多的大数据开源项目之一。

Apache Spark 是为大规模、分布式数据处理而设计的一个分析引擎,为中间计算提供了内存存储,运行速度较 Hadoop MapReduce 快若干倍。Spark 包括用于机器学习的库 MLlib、用于交互式查询的 SQL(Spark SQL)、用于与实时数据交互的流处理(Structured Streaming)以及用于图形处理的 GraphX 库,如图 3-1 所示。

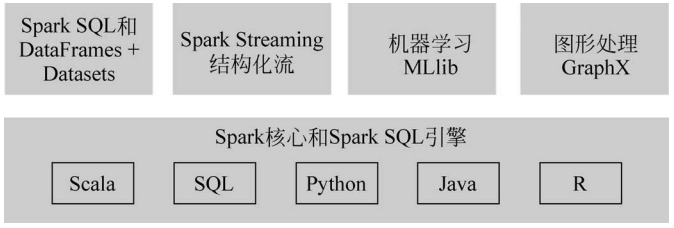

图 3-1 Spark 的组件与 API 接口

Spark 具有如下主要特点。

(1) **运行速度快**：Spark 框架经过优化，可以充分利用多核 CPU 和内存资源，高效利用操作系统的多线程和并行处理机制。其次，Spark 将其查询计算构建为有向无环图(Directed Acyclic Graph，DAG)，执行引擎将其优化为高效的计算图，分解为可以在集群上多个工作线程之间并行执行的任务。此外，所有中间结果都保留在内存中(尽量减少磁盘 I/O)，也极大地提高了性能。

(2) **容易使用**：Spark 支持使用 Scala、Java、Python 和 R 语言进行编程，简洁的 API 设计有助于用户轻松构建并行应用程序，并且可以通过 Spark Shell 进行交互式编程。

(3) **通用性强**：Spark 可以应用于多种类型的工作任务，提供了支持多种编程语言接口的、统一的 API 库，可以将不同的工作负载组合在一个引擎下运行。其核心组件(或模块)，如 Spark SQL、Structured Streaming、MLlib 和 GraphX 等，可以无缝地整合在同一个应用中，足以应对复杂的计算。

(4) **扩展性强**：Spark 专注于其快速的并行计算引擎，而不是存储。使用 Spark 可以读取存储在多种数据源中的数据，如 Apache Hadoop、Cassandra、HBase、Hive、MongoDB，以及各种关系数据库等。Spark 的 DataFrame Reader 和 Writer 可以扩展以支持其他数据源(如 Apache Kafka、Kinesis、Azure Storage 和 Amazon S3 等)。

3.1.2 Spark 架构设计

Spark 是一个分布式数据处理引擎，其组件在集群机器上协同工作。在探讨 Spark 编程之前，需要了解 Spark 分布式架构的所有组件，以及组件之间是如何协同工作和通信的。Spark 的运行架构包括集群管理器、应用程序的任务驱动程序 Driver 和运行作业任务的工作节点(Worker Node)，其中，每个工作节点的程序执行器 Executor 负责具体任务的执行。集群管理器可以是 Spark 自带的资源管理器，也可以是 YARN 等资源管理框架。

与 Hadoop MapReduce 相比，Spark 采用 Executor 的优点体现在两个方面：一是利用多线程执行任务(Hadoop MapReduce 采用的是进程模型)，减少任务的启动开销；二是 Executor 中的 BlockManager 模块会将中间结果存到内存中，当迭代计算需要时，可以直接从内存读数据，而无须读写 HDFS 等文件系统，减少了 I/O 开销。另外，在交互式查询场景下，通过预先将数据表缓存到 BlockManager 中，从而提高 I/O 读写性能。

如图 3-2 所示，Spark 应用程序包含一个 Driver 程序，该程序负责协调 Spark 集群上的并行操作。Driver 程序通过 SparkSession 访问集群中的分布式组件，包括 Spark 程序执行器 Executor 和集群管理器 Cluster Manager。

Spark Driver 是 Spark 应用程序的一部分，负责实例化 SparkSession。Driver 程序与集群管理器通信，请求 Executor(JVM)运行需要的资源(CPU、内存等)；将所有 Spark 操作转换为 DAG(计算、计划)，并将任务分发到 Executor。分配资源后，Driver 将直接与 Executor 通信。

在 Spark 2.0 及其后版本中，**SparkSession** 成为所有数据访问 Spark 操作的统一入口，不仅包含了 Spark 以前的入口点，如 SparkContext、SQLContext、HiveContext、SparkConf 和 StreamingContext 等，还使 Spark 更简单、易用。通过这个入口，可以创建 JVM 运行时参数、定义数据帧和数据集、从数据源读取、访问目录元数据，以及执行 Spark SQL 查询等。

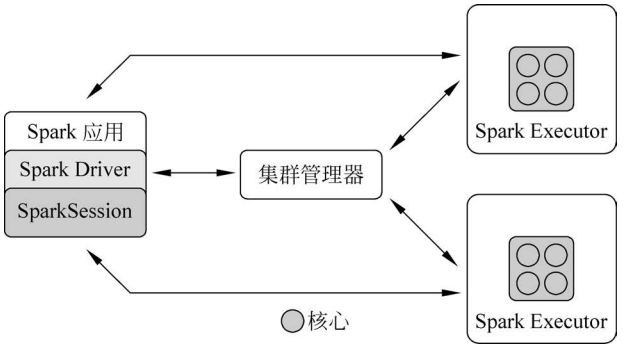

图 3-2 Spark 组件与运行架构

集群管理器负责为运行 Spark 应用程序的集群节点管理和分配资源。目前，Spark 支持 4 类分布式集群管理器，分别是内置的独立集群管理器 Standalone、Apache Hadoop YARN、Apache Mesos 和 Kubernetes。

程序执行器 Executor 运行在集群中的工作节点上。Executor 与 Driver 程序通信，并负责在工作节点上执行任务。在大多数部署模式下，每个节点仅运行一个 Executor。

3.2 Spark 安装及部署

3.2.1 安装 Spark

1. 下载 Spark

Spark 可在官方网站（或镜像网站）下载，选择合适的稳定版本，下载页面如图 3-3 所示。

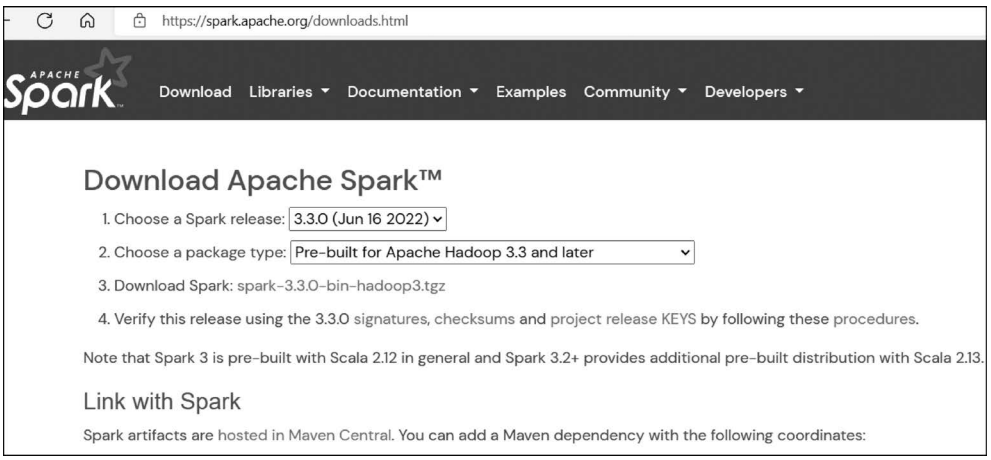

图 3-3 Spark 下载页面

进入下载页面后，即可选择合适的 Spark 版本。本书使用了 Scala 2.13.x 版本，因此，这里选择下载 Pre-built for Apache Hadoop 3.3 and later。也可以从镜像站点下载：

```
$ wget -c https://mirrors.tuna.tsinghua.edu.cn/apache/spark/spark-3.3.0/spark-3.3.0-bin-hadoop3-scala2.13.tgz
```

2. 安装

1）安装 JDK

Spark 运行需要设置 Java 环境，确保系统已安装 Java。具体可参考 2.1.2 节的相关内容。确保已设置 JAVA_HOME 环境变量，如，直接在当前终端窗口设置：

```
$ export JAVA_HOME = ~/jdk - 17
```

注意：请以本地实际安装的路径为准（即替换上述命令的路径~/jdk-17）。

可以根据实际需要，将 JAVA_HOME 设置为用户环境变量或系统环境变量。设置方法与设置 PATH 环境变量类似，具体可参考 2.1.2 节的相关内容，或参考其他资料。

2）安装 Spark

将下载的 Spark 压缩包解压，并根据需要将解包的路径换成容易识记的路径名称，设置必要的用户权限等（注：解包到用户主目录 Home，通常不需要特别设置权限）：

```
$ tar - zxf ./Downloads/spark - 3.3.0 - bin - hadoop3 - scala2.13.tgz
$ mv ./spark - 3.3.0 - bin - hadoop3 - scala2.13 ./spark - 3.3.0
```

3）验证 Spark 安装

如图 3-4 所示，执行以下命令，可以查看 Spark 版本等信息：

```
$ ./spark - 3.3.0/bin/spark - shell -- version
```

```
o@o-VirtualBox:~$ ./spark-3.3.0/bin/spark-shell --version
22/09/28 21:35:28 WARN Utils: Your hostname, o-VirtualBox resolves to a loopback address:
127.0.1.1; using 10.0.2.15 instead (on interface enp0s3)
22/09/28 21:35:28 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.3.0
      /_/

Using Scala version 2.13.8, OpenJDK 64-Bit Server VM, 17.0.4.1
Branch HEAD
Compiled by user ubuntu on 2022-06-09T18:15:33Z
Revision f74867bddfbcdd4d08076db36851e88b15e66556
Url https://github.com/apache/spark
Type --help for more information.
o@o-VirtualBox:~$
```

图 3-4　查看 Spark 版本信息

运行 Spark 自带的例子程序，如计算圆周率 π 近似值，结果如图 3-5 所示。

```
$ ./spark - 3.3.0/bin/run - example SparkPi 4 2 > &1 | grep "Pi "
```

```
o@o-VirtualBox:~$ ./spark-3.3.0/bin/run-example SparkPi 4 2>&1 | grep "Pi "
Pi is roughly 3.1427378568446422
```

图 3-5　运行 Spark 自带的例子程序

说明：执行例子程序时会输出较多的运行信息，因此，这里使用了 grep 命令对输出信息进行过滤（上述命令中的 2 > &1 可以将所有的信息都输出到标准控制台中，否则日志信息还是会输出到屏幕）。具体命令、管道等用法，请参考 Linux 相关文档。

提示：Spark 安装包中附带有一些例子程序，如，基于 RDD API 的 Word Count、圆周率 π 估计程序 SparkPi、基于 DataFrame API 的文本搜索 Text search 以及机器学习等例子程

序。这些例子程序的源代码位于 examples/src/main 目录下。学习例子程序对 Spark 开发有极大的帮助。

3.2.2 Spark 部署方式

Spark 支持多种部署模式,能够运行在不同的配置和环境中。由于集群管理器与它的运行位置无关(只要可以管理 Spark Executor,并满足资源请求),所以 Spark 可以部署在一些最流行的环境中,如 Apache Hadoop YARN 和 Kubernetes,并且可以在不同的模式下运行。

Local 模式是本地模式,常用于本地开发或测试。Standalone 模式使用 Spark 框架自带的资源调度管理服务,可以独立部署到一个集群中。在 Spark on YARN/Kubernetes/Mesos 模式中,Spark 应用程序所需要的各种资源,由相应的资源管理服务器负责调度。表 3-1 给出了可用的部署方式。

表 3-1 Spark 部署方式

模 式	Spark Driver	Spark Executor	Cluster Manager
Local	单个 JVM 中运行,单节点	与 Driver 相同的 JVM	同一台机器
Standalone	集群中的任一节点	每个 Executor 有自己独立的 JVM	集群中的任一主机 Spark Master
YARN (client)	客户机,非集群一部分	Container on YARN's NodeManager	Resource Manager Application Master
YARN (cluster)	Application Master	同上	同上
Kubernetes	运行于 Kubernetes pod	每个节点自己的 pod	Kubernetes Master
Mesos	Framework Scheduler	Mesos Agent	Mesos Master

3.3 配置 Spark 访问 HDFS 数据源

Spark 可以独立使用,也可以与 Hadoop 配合使用。通常情况下,Spark 基于 Hadoop HDFS 的高容错特性来处理大规模数据集。Spark 可以和 Hadoop HDFS 同时安装使用,Spark 可以直接使用 HDFS 存取数据。

Spark 访问 HDFS 数据源,需要部署相应的服务。为保持内容的完整性,本节简要介绍 Hadoop 的安装部署过程,更详细的内容请参考 Hadoop 用户手册或其他相关资料。

3.3.1 Hadoop 部署

1. Hadoop 下载

Hadoop 当前的发行版本是 3.3.4。相比 3.2.x 版本,3.3.x 版本支持 ARM 架构,对 Java 11/17 的支持更全面,且修复了 Guava 包的版本冲突等问题。Hadoop 可以通过官方网站下载,也可以选择镜像下载。如图 3-6 所示,对下载文件进行解压:

```
$ tar -zxf ./Downloads/hadoop-3.3.4.tar.gz
```

图 3-6　Hadoop 下载过程

2. Hadoop 伪分布式部署

1) 依赖环境

(1) Hadoop 集群(单节点)、伪分布部署依赖于 Java、SSH 等。应确保已安装 Java。设置 Java 环境变量,验证 Hadoop 版本信息:

```
$ export JAVA_HOME = ~/jdk - 17
$ ./hadoop - 3.3.4/bin/hadoop version
```

说明: Hadoop 3.3.4 不完全支持 Java 17。根据官方文档,Hadoop 3.3.x 版本完全支持 Java 8/Java 11,Hadoop 3.0.x ～ 3.2.x 仅支持 Java 8。

(2) 安装 SSH:

```
$ sudo apt - get install ssh
$ sudo apt - get install pdsh
```

(3) Hadoop 使用 SSH 登录时,没有提供输入密码的界面,因此需要设置为免密登录(见图 3-7):

```
$ ssh - keygen - t rsa - P '' - f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

图 3-7　设置 SSH 免密登录

2）伪分布配置

（1）修改 Hadoop 配置文件中的环境变量设置，首先运行命令：

```
$ gedit ./hadoop-3.3.4/etc/hadoop/hadoop-env.sh
```

打开文件后，可将其中的 JAVA_HOME 设置为正确的路径，具体如图 3-8 所示。

```
52 # The java implementation to use. By default, this environment
53 # variable is REQUIRED on ALL platforms except OS X!
54 export JAVA_HOME=~/jdk-17
```

图 3-8 配置 JAVA_HOME 环境变量

提示：系统中设置 JAVA_HOME 环境变量后，如果未修改配置文件中的 JAVA_HOME 环境变量设置，那么启动 Hadoop 时，仍会报错“ERROR：**JAVA_HOME is not set** and could not be found”。

思考：Hadoop 3.3.x 读取 hadoop-env.sh 中设置的 JAVA_HOME 环境变量，而不是系统中的 JAVA_HOME 环境变量，这样设计的目的是什么？

（2）修改 Hadoop 配置文件 etc/hadoop/core-site.xml，设置默认的文件系统，示例如下：

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <--! default HDFS server URL, 请根据实际修改 -->
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

（3）修改配置文件 etc/hadoop/**hdfs-site.xml**，设置默认的文件副本数量等。示例如下：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/home/o/hadoop-3.3.4/tmp/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/home/o/hadoop-3.3.4/tmp/dfs/data</value>
    </property>
</configuration>
```

说明：在伪分布模式下，Hadoop 会使用临时目录存储名字服务、数据服务的临时文件，默认的临时文件存放路径是系统/**tmp** 目录。该路径下的文件在 Linux 系统重启时可能会被清理，导致 namenode 启动失败。建议将该临时目录修改为不会被自动清理的路径。

3. 启动

1）文件系统初始化

首次使用 HDFS 服务时，需要格式化文件系统。使用如下命令进行格式化，并等待格式化完成：

```
$ ./hadoop-3.3.4/bin/hdfs namenode -format
```

格式化成功后会有“successfully formatted”提示字样，如图 3-9 所示。

注意：文件系统格式化仅需要在系统初始化时执行一次，以后使用时不再格式化；否则已有数据会丢失。

```
1.1-1664446682328
2022-09-29 18:18:02,394 INFO common.Storage: Storage directory /tmp/hado
 successfully formatted.
2022-09-29 18:18:02,420 INFO namenode.FSImageFormatProtobuf: Saving imag
fs/name/current/fsimage.ckpt_0000000000000000000 using no compression
2022-09-29 18:18:02,483 INFO namenode.FSImageFormatProtobuf: Image file
/current/fsimage.ckpt_0000000000000000000 of size 396 bytes saved in 0 s
2022-09-29 18:18:02,503 INFO namenode.NNStorageRetentionManager: Going t
```

图 3-9　HDFS 格式化成功提示信息

2）启动 HDFS

执行脚本命令，启动 HDFS：

```
$ ./hadoop-3.3.4/sbin/start-dfs.sh
```

注意：脚本命令位于 sbin 目录，而不是 bin 目录下。

正常启动后，可以看到 NameNode、DataNode 等服务进程，也可通过浏览器浏览 NameNode 或 DataNode 的相关信息，如图 3-10 所示。NameNode 的默认 Web 服务地址是：http://localhost:9870/。

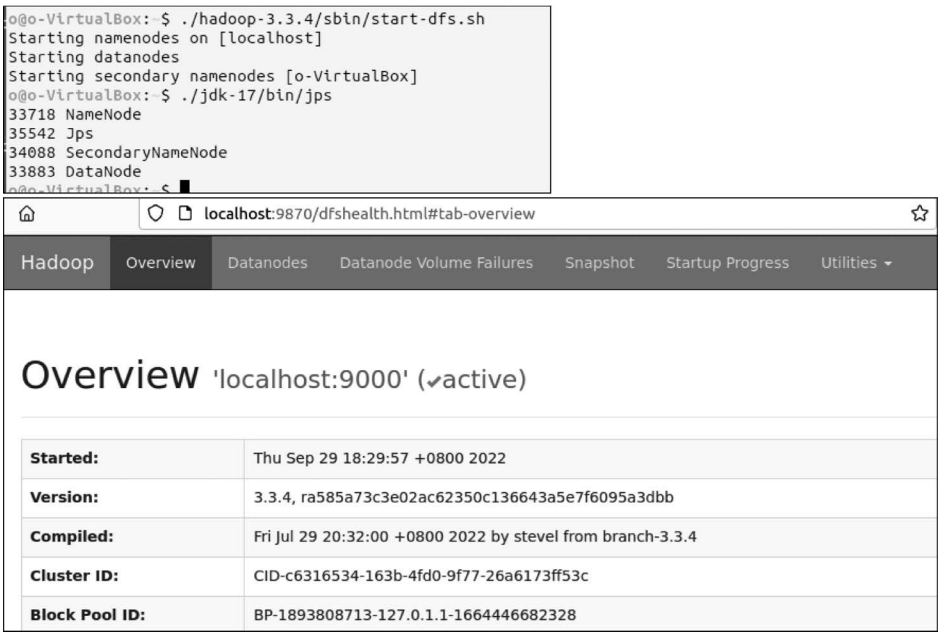

图 3-10　监测 HDFS 服务

4. HDFS 操作

如图 3-11 所示，执行创建目录、上传文件、执行程序、查看结果等命令，实现多种 HDFS 操作。具体可参考 HDFS 用户手册，示例代码如下：

```
$ ./hadoop-3.3.4/bin/hdfs dfs -mkdir -p /user/o/input
$ ./hadoop-3.3.4/bin/hdfs dfs -put ./hadoop-3.3.4/etc/hadoop/*.xml input
$ ./hadoop-3.3.4/bin/hadoop jar ./hadoop-3.3.4/share/hadoop/mapreduce/hadoop-mapreduce
-examples-3.3.4.jar grep input output 'dfs[a-z.]+'
$ ./hadoop-3.3.4/bin/hdfs dfs -cat output/*
```

```
o@o-VirtualBox: $ ./hadoop-3.3.4/bin/hdfs dfs -mkdir -p /user/o/input
o@o-VirtualBox: $ ./hadoop-3.3.4/bin/hdfs dfs -put ./hadoop-3.3.4/etc/hadoop/*.xml input
o@o-VirtualBox: $ ./hadoop-3.3.4/bin/hadoop jar ./hadoop-3.3.4/share/hadoop/mapreduce/hadoop-m
apreduce-examples-3.3.4.jar grep input output 'dfs[a-z.]+'
2022-09-29 21:56:10,697 INFO impl.MetricsConfig: Loaded properties from hadoop-metrics2.proper
ties
2022-09-29 21:56:10,812 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 se
cond(s).
```

图 3-11 HDFS 操作命令

5. 停止 HDFS 服务

如果有必要，则可执行脚本命令，停止 HDFS 服务：

```
$ ./hadoop-3.3.4/sbin/stop-dfs.sh
```

3.3.2 配置 Spark 访问 HDFS

由于下载使用的 Spark 安装包中已经包含 Hadoop 相关的依赖包，所以这里无须进行特殊配置，即可直接访问 HDFS 中的数据。如图 3-12 所示，启动 Spark shell 后，在交互环境中加载 HDFS 中的文件（请确保 HDFS 服务已经启动，且文件路径正确）：

```
val xml2 = sc.textFile("hdfs://localhost:9000/user/o/input/core-site.xml")
xml2.take(5)
```

说明：Spark 对本地文件与 HDFS 文件的访问方法是一致的。HDFS 路径以“hdfs://”修饰，本地文件则以“file://”修饰。如果未指明访问协议，则默认是本地文件。

3.4 使用 Spark shell

Spark shell 是一个交互方式的数据分析工具，也是 Spark API 学习环境，包括 Scala（在 JVM 上运行）版及 Python 版。

3.4.1 启动 Spark shell

输入如下命令，启动 Spark shell：

```
# Scala 环境
$ ./spark-3.3.0/bin/spark-shell
# Python 环境
$ ./spark-3.3.0/bin/pyspark
```

Scala 版本提供 Scala 交互 shell，可以直接执行 Scala 代码（参见第 2 章的有关内容）。

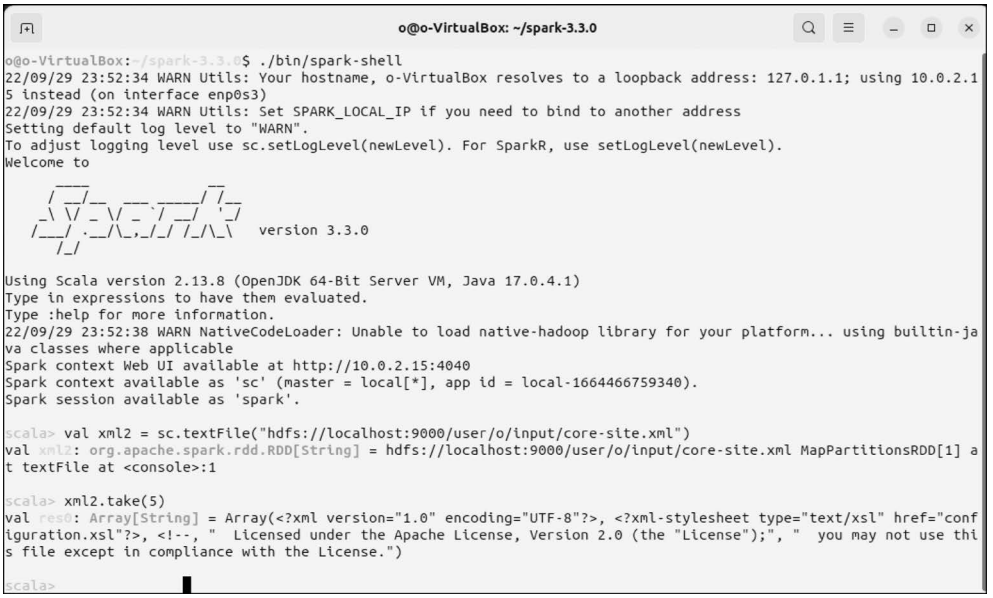

图 3-12　Spark shell 中访问 HDFS 文件

以下以 Scala 环境(版本)的 Spark shell 为例介绍。

3.4.2　使用 Spark shell

Spark 的主要抽象是称为数据集(Dataset)的分布式数据项的集合。数据集可以从本地文件、Hadoop Input Formats(如 HDFS 文件)创建,也可通过转换其他数据集来创建。

Spark shell 启动时,会创建 Spark 环境(context)以及会话(Session)对象,可直接用于后续的数据分析或交互。

在 Spark shell 中加载文本数据,并查看文件首行内容,可使用类似下面的命令(输入命令以**加粗**形式表示。由于是交互式环境,每行命令输入后会获得相应的反馈):

```
scala> val textFile = spark.read.textFile("README.md")
val textFile: org.apache.spark.sql.Dataset[String] = [value: string]
scala> textFile.first()
val res0: String = # Apache Spark
```

注意:"scala>"是 Scala Spark shell 的提示符,不是输入内容。

注意:Spark 采用**惰性机制**。对 RDD 而言,直到执行 RDD 动作(Action)命令时,才真正触发对数据集的加载、转换等一系列操作。因此,即使加载一个不存在的文件,在加载时也不会报错(加载时不报错,查看内容时报错),如图 3-13 所示。

数据集的转换(Transformation)和动作可用于更复杂的计算,执行命令及结果如图 3-14 所示,示例代码如下:

```
// 统计文件中包含"Spark"的行数:
scala> textFile.filter(line => line.contains("Spark")).count()
// 最长行所包含的单词数
```

```
scala> val textFile2 = sc.textFile("README.md2")
val textFile2: org.apache.spark.rdd.RDD[String] = README.md2 MapPartitionsRDD[4] at textFi
le at <console>:1

scala> textFile2.take(5)
org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: file:/home/o/sp
ark-3.3.0/README.md2
  at org.apache.hadoop.mapred.FileInputFormat.singleThreadedListStatus(FileInputFormat.jav
```

图 3-13 Spark 中动作操作触发实际计算

```
scala > textFile. map(line = > line. split(" "). size)
            . reduce((a, b) = > if (a > b) a else b)
// 大数据经典例子 Word Count
scala > val wordCounts = textFile. flatMap(
        line = > line. split("[ ,. ] + ")). groupByKey(identity). count(). collect()
```

```
scala> textFile.filter(line => line.contains("Spark")).count()
val res1: Long = 20

scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
val res2: Int = 16

scala> val wordCounts = textFile.flatMap(line => line.split("[ ,.]+")).groupByKey(identity
).count().collect()
val wordCounts: Array[(String, Long)] = Array(([![PySpark,1), (online,1), (graphs,1), (dis
tributions,1), (API,1), (com/apache/spark/actions/workflows/build_and_test,2), (["Building
,1), (documentation,4), (thread,1), (count(),2), (abbreviated,1), (```bash,6), (overview,1
), (rich,1), (set,2), (-DskipTests,1), (name,1), (workloads,1), (["Specifying,1), (org/con
tributing,1), (stream,1), (run:,1), (not,1), (programs,3), (tests,2), (instructions,1), (w
ill,1), ([run,1), (particular,2), (must,1), (using,3), (you,4), (MLlib,1), (variable,1), (
Note,1), (core,1), (protocols,1), (shields,1), (guidance,2), (shell:,2), (can,6), (html#sp
ecifying-the-hadoop-version-and-enabling-yarn),1), (*,4), ([building,1), (configure,1), (f
or,13), (README,1), (Interactive,2), (```python,1),...
```

图 3-14 执行数据转换和动作

3.4.3 退出 Scala Spark shell

在 Spark shell 中执行“:quit”命令(简写为“:q”),或按组合键 Ctrl+D 退出:

```
scala > :quit
```

3.4.4 Spark shell 常用选项

启动 Spark shell 时,使用“--help”(或“-h”)选项,可列出 shell 命令选项(参数),如图 3-15 所示。常用选项如下:

(1) “--master”用于指定连接的服务器地址(默认为本地模式,“local[*]”)。

(2) “--deploy-mode”用于指定客户应用启动模式(默认为 client)。

(3) “--conf”用于设置相关配置项(如内存负载、Shuffle 分区数等)。

(4) “--jars”用于导入依赖的 jar 包等。

各选项的具体用法,请参考 Spark 用户手册。

```
o@o-VirtualBox:~/spark-3.3.0$ ./bin/spark-shell --help
Usage: ./bin/spark-shell [options]

Scala REPL options:
  -I <file>                   preload <file>, enforcing line-by-line interpretation

Options:
  --master MASTER_URL         spark://host:port, mesos://host:port, yarn,
                              k8s://https://host:port, or local (Default: local[*]).
  --deploy-mode DEPLOY_MODE   Whether to launch the driver program locally ("client") or
                              on one of the worker machines inside the cluster ("cluster")
                              (Default: client).
  --class CLASS_NAME          Your application's main class (for Java / Scala apps).
  --name NAME                 A name of your application.
  --jars JARS                 Comma-separated list of jars to include on the driver
                              and executor classpaths.
  --packages                  Comma-separated list of maven coordinates of jars to include
                              on the driver and executor classpaths. Will search the local
                              maven repo, then maven central and any additional remote
                              repositories given by --repositories. The format for the
                              coordinates should be groupId:artifactId:version.
  --exclude-packages          Comma-separated list of groupId:artifactId, to exclude while
                              resolving the dependencies provided in --packages to avoid
                              dependency conflicts.
  --repositories              Comma-separated list of additional remote repositories to
                              search for the maven coordinates given with --packages.
  --py-files PY_FILES         Comma-separated list of .zip, .egg, or .py files to place
                              on the PYTHONPATH for Python apps.
  --files FILES               Comma-separated list of files to be placed in the working
                              directory of each executor. File paths of these files
                              in executors can be accessed via SparkFiles.get(fileName).
  --archives ARCHIVES         Comma-separated list of archives to be extracted into the
                              working directory of each executor.

  --conf, -c PROP=VALUE       Arbitrary Spark configuration property.
  --properties-file FILE      Path to a file from which to load extra properties. If not
                              specified, this will look for conf/spark-defaults.conf.
```

图 3-15　Spark shell 命令选项

3.5　Spark 开发环境

本书基于 Scala 编程语言进行 Spark 应用开发。因此，所介绍的开发环境主要考虑 Spark 的 Scala 开发环境，包括命令行工具、IDE 环境等。

3.5.1　SBT

SBT 全称是 Simple Build Tool，是 Scala、Java 构建工具，与 Maven 或 Gradle 类似。SBT 依赖于 Java 8(1.8)或以上版本。

官方推荐的安装方式是使用 cs setup(参考 2.1.2 节的有关内容)。本节仅介绍 Linux 环境下的 SBT 的手动安装、部署过程。

1. 安装 SBT

(1) 可到官方网站(或镜像网站)选择合适的稳定版本的 SBT 进行下载，当前版本是 SBT 1.7.2，镜像下载命令如图 3-16 所示。

说明：考虑网络等原因，建议使用断点续传(wget)**"-c"**选项；为便于管理可下载到指定路径(使用**"-P"**选项)。

Ubuntu 用户也可以下载安装包安装，或直接在线安装。具体请参考相关资料。

(2) SBT 依赖于 Java 环境，所以需要确保系统已安装 Java。支持 SBT 1.7.2 的 Java

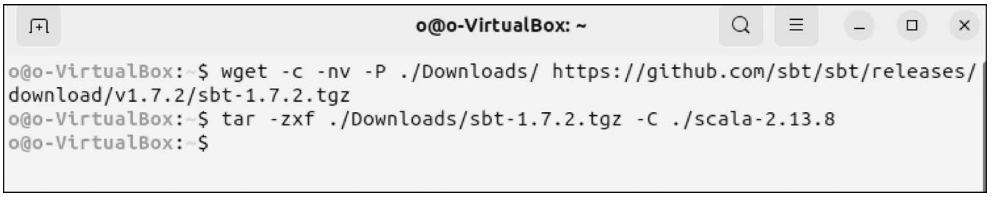

图 3-16 SBT 的镜像下载

版本包括 8、11、17 等。下载完成后，对压缩包进行解压：

```
$ tar -zxf ./Downloads/sbt-1.7.2.tgz -C ./scala-2.13.8
```

可以根据实际需要，将 SBT 添加到搜索路径(设置为 PATH 环境变量)，具体可参考 2.1.2 节的相关内容。

(3) 如图 3-17 所示，运行 sbt shell 命令(请确保 Java 安装正确)，查看 SBT 版本信息：

```
$ ./scala-2.13.8/sbt/bin/sbt --version
```

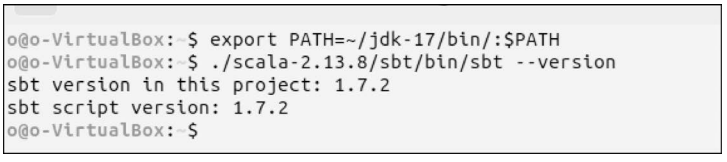

图 3-17 查看 SBT 版本信息

注意：在 SBT 交互环境中，可以使用 help 命令查看 SBT shell 的相关命令，使用 exit 命令退出 SBT shell。

2. SBT 使用示例

1) 最小 SBT 构建

最小 SBT 构建使用默认设置，仅需要一个空的 SBT 文件，创建的 build.sbt 文件如下：

```
$ mkdir ./scala-examples
$ cd ./scala-examples
$ touch build.sbt                    # 创建空的 SBT 文件
```

2) 启动 SBT shell

如图 3-18 所示，运行命令启动 SBT shell。

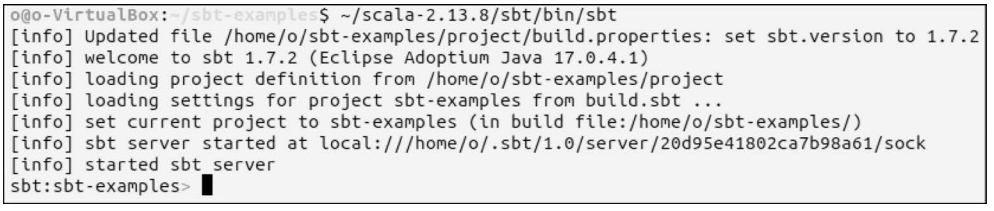

图 3-18 启动 SBT shell

3) 编译

启动 SBT 编译命令，运行结果如图 3-19 所示，可以看到编译成功的提示信息。

SBT shell 支持即时编译模式，即在修改代码的同时进行编译，运行结果如图 3-20 所示。

```
sbt:sbt-examples> compile
[success] Total time: 0 s, completed Oct 7, 2022, 5:07:24 PM
```

图 3-19　SBT 编译

```
sbt:sbt-examples> ~compile
[success] Total time: 0 s, completed Oct 7, 2022, 5:07:35 PM
[info] 1. Monitoring source files for sbt-examples/compile...
[info]    Press <enter> to interrupt or '?' for more options.
```

图 3-20　即时编译模式

3. 使用 SBT shell 进行构建

1）编写项目 Scala 代码

创建 SBT 构建所需的目录结构，并编写源代码（Hello.scala）：

```
$ mkdir -p ./src/main/scala/example
$ gedit ./src/main/scala/example/Hello.scala
```

在源文件（Hello.scala）中输入如下示例代码：

```
package example
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello")
    println("Welcome to Wenzhou University!")
    println("Welcome to AI School,Wenzhou University!")
  }
}
```

2）编译项目

运行 SBT shell 的 compile 命令，对项目进行编译（或即时编译），如图 3-21 所示。

图 3-21　编译项目

如果源代码有错误，则根据提示信息进行修改后再编译。

3）运行代码

运行 SBT shell 的 run 命令，执行编译后的项目，如图 3-22 所示。

```
sbt:sbt-examples> run
[info] running example.Hello
Hello
Welcome to Wenzhou University!
Welcome to AI School, Wenzhou University!
[success] Total time: 0 s, completed Oct 7, 2022, 5:43:53 PM
```

图 3-22　代码运行

4）项目属性设置

（1）运行 SBT shell 的 set 命令，设置构建版本信息，运行结果如图 3-23 所示，代码如下：

```
sbt:sbt - examples > set ThisBuild / scalaVersion := "2.13.8"
sbt:sbt - examples > scalaVersion                    # 验证设置
```

```
sbt:sbt-examples> set ThisBuild / scalaVersion := "2.13.8"
[info] Defining ThisBuild / scalaVersion
[info] The new value will be used by Compile / bspBuildTarget, Compile / dependencyTreeCros
sProjectId and 50 others.
[info]  Run `last` for details.
[info] Reapplying settings...
[info] set current project to sbt-examples (in build file:/home/o/sbt-examples/)
sbt:sbt-examples> scalaVersion
[info] 2.13.8
```

图 3-23 构建版本信息

（2）运行 SBT shell 的 session save 命令，将交互环境中的会话设置信息保存到 build.sbt 文件中：

```
sbt:sbt - examples > session save
```

（3）编辑 build.sbt 文件，对项目进行命名，设置版本，添加依赖或设置其他属性，如：

```
$ gedit ./build.sbt
```

文件内容参考如下：

```
// project version
ThisBuild / version := "0.1.0"
ThisBuild / scalaVersion := "2.13.8"
ThisBuild / organization := "cn.edu.wzu.example"
lazy val hello = (project in file("."))
  .settings(
    // project name
    name := "Hello",
    // project dependency
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.7" % Test,
    )
```

（4）运行 SBT shell 的 reload 命令，重新加载属性更新后的项目，如图 3-24 所示。

```
sbt:sbt-examples> reload
[info] welcome to sbt 1.7.2 (Eclipse Adoptium Java 17.0.4.1)
[info] loading project definition from /home/o/sbt-examples/project
[info] loading settings for project hello from build.sbt ...
[info] set current project to Hello (in build file:/home/o/sbt-examples/)
sbt:Hello>
```

图 3-24 重新加载项目

注意：重新加载项目后，SBT shell 的提示符发生了变化（更新后的项目名）。

（5）在 SBT shell 中执行 exit 命令，或按 Ctrl+D 组合键退出：

```
sbt:sbt - examples > exit
```

4. 使用 SBT 构建 Spark 应用程序

用 Spark API 编写的独立应用程序，可以使用 SBT 构建。

首先，按照 SBT 构建目录结构要求创建相应的目录，并编写源代码文件。例如，创建一

个 Simple Spark App 项目，其目录结构如下：

```
$ find .
    .
    ./build.sbt
    ./src
    ./src/main
    ./src/main/scala
    ./src/main/scala/SimpleSparkApp.scala
```

代码 SimpleSparkApp.scala 的内容示例如下：

```
/**
 * SimpleSparkApp.scala
 *
 * @author Rujun Cao
 * @date 2022/10/00
 */
package cn.edu.wzu.SparkExample

import org.apache.spark.sql.SparkSession
// This example illustrates Spark session / Dataset
// Counting the number of lines with 's' or 't'
object SimpleSparkApp {
  def main(args: Array[String]): Unit = {
    // A file from command-line, or default = "README.md"
    val logFile = if (args.length > 0) args(0) else "README.md"
    val spark = SparkSession.builder
            .appName("Simple Spark Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numSs = logData.filter(line => line.contains("s")).count()
    val numTs = logData.filter(line => line.contains("t")).count()
    println(s"Lines with s: $numSs, Lines with t: $numTs")
    spark.stop()
  }
}
```

构建文件 build.sbt 的内容示例如下：

```
name := "Simple Spark Project"
version := "0.1.0"
scalaVersion := "2.13.8"
libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.3.0"
```

创建好项目结构及内容后，使用 SBT 程序进行打包：

```
$ ~/scala-2.13.8/sbt/bin/sbt package
```

打包过程中 SBT 会自动下载相关依赖包。初次打包时需要下载的内容较多，下载时间稍长，后续打包速度相对较快。打包完成后，会有 success 的提示字样。如果源代码有错误，则根据提示信息进行修改后，再次编译、打包。

说明：SBT 程序打包的输出路径为./**target**/**scala-2.xx**。

打包完成后即可将应用程序提交给 Spark 执行，如图 3-25 所示。

```
$ ../bin/spark-submit ./target/scala-2.13/simple-spark-project_2.13-0.1.0.jar ../
README.md
```

```
o@o-VirtualBox:~/spark-3.3.0/SimpleApp$ ../bin/spark-submit ./target/scala-2.13/
simple-spark-project_2.13-0.1.0.jar ../README.md 2>&1 | grep "Lines "
Lines with s: 63, Lines with t: 60
o@o-VirtualBox:~/spark-3.3.0/SimpleApp$
```

图 3-25 执行打包程序

程序执行后,可以查看运行结果(例子程序的结果类似于图 3-25 的"Lines with ...")。

说明:对简单的应用程序,也可以直接使用 scalac 进行编译、打包。如对于上述例子可以使用下面的命令:

```
$ ~/scala-2.13.8/bin/scalac -classpath ../jars/spark-core_2.13-3.3.0.jar:../jars/
scala-library-2.13.8.jar:../jars/spark-sql_2.13-3.3.0.jar ./src/main/scala/
SimpleSparkApp.scala -d SimpleSparkProj.jar
```

5. 使用 SBT 镜像服务器

使用 SBT 会自动下载依赖包,默认会使用官方服务器地址。由于网络等原因,默认的官方服务器下载速度较慢。这时,可以使用国内镜像站点或镜像仓库。

使用镜像仓库,需要编辑 SBT 的仓库配置文件 repositories,该文件位于~/.sbt 目录(隐藏目录,如果目录不存在则需要创建)

```
$ gedit ~/.sbt/repositories
```

其内容参考如下:

```
[repositories]
local
# 设置华为云镜像仓库
huaweicloud-maven: https://repo.huaweicloud.com/repository/maven/
```

具体镜像仓库地址等可以查找相关资料。

3.5.2 IntelliJ IDEA

IntelliJ IDEA 是由 JetBrains 创建的集成开发环境(Integrated Development Environment,IDE),其社区版(Community)是基于 Apache v2 许可协议的开源版本。IntelliJ 集成了许多构建工具(包括 SBT),可以用来导入项目。

1. 安装 IntelliJ IDEA

可在 JetBrains 官方网站下载 IntelliJ IDEA 社区版。下载后进行解包、安装。Ubuntu 用户也可以通过软件中心(Software Center)直接搜索、安装,安装界面如图 3-26 所示。

说明:IntelliJ IDEA 教育版(Educational)是针对学生或教师提供的免费旗舰版(Ultimate),学生或教师用户可根据需要选用。

2. 安装 Scala 插件

启动 IDEA 后,选择插件页面(Plugins Tab),在可用插件列表中选择 Scala 后(如果未列出 Scala,那么可以用搜索框搜索),单击 Install 按钮,如图 3-27 所示。

插件安装完成后,根据提示重启 IDE。

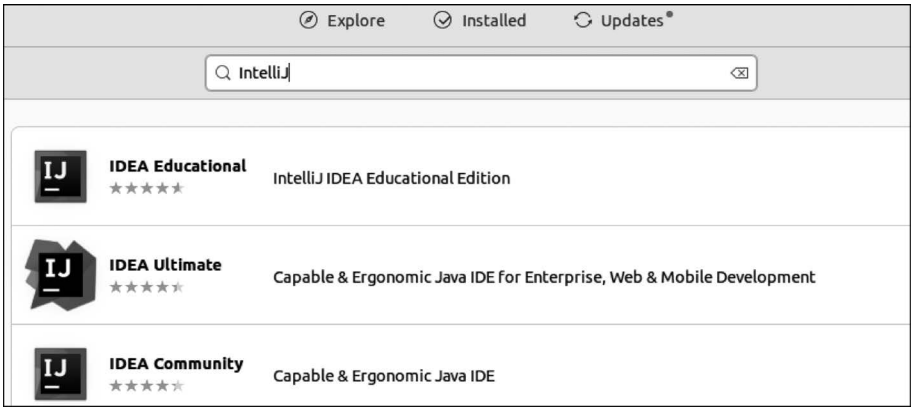

图 3-26　安装 IDEA

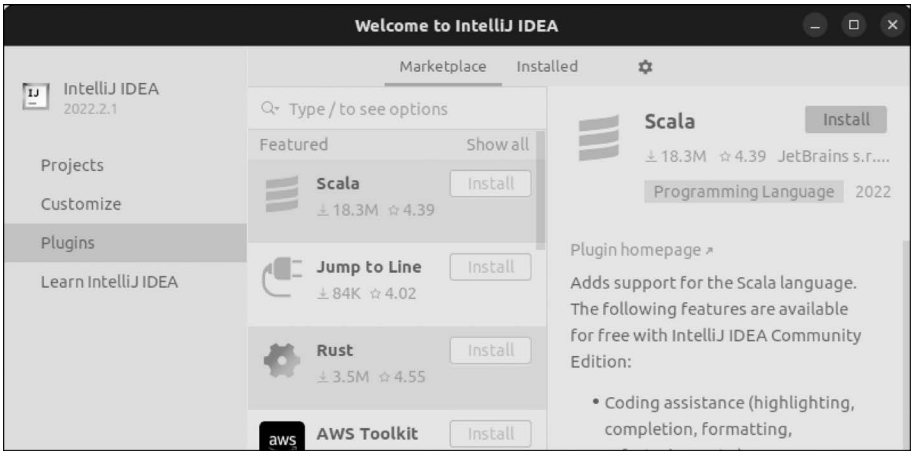

图 3-27　在 IDEA 中安装 Scala 插件

3. 导入 Spark 项目

启动 IDEA 后，依次选择 Projects→Open，在弹出的窗口中选择已创建的 Spark/Scala 项目（目录中包含 build.sbt）后，单击 OK 按钮，如图 3-28 所示。

提示：也可以通过选择“从已有代码创建项目”的方式导入项目，具体操作步骤是：依次选择 File→New→Project from Existing Sources 菜单项，再选择项目目录中的 build.sbt 文件。

说明：IntelliJ Scala 插件使用自己的轻量级编译引擎来检测错误，速度快但可能有错误。可以将 IntelliJ 配置为使用 Scala 编译器，以突出错误显示。

4. 代码交互调试

使用 IDEA 便于交互式调试程序。在 IDEA 中打开需要调试的代码，在适当位置设置断点（Break point）后，即可启动程序调试模式，进入交互调试过程，如图 3-29 所示。

当测试达到断点时，可以检查变量的值等信息。

提示：IDEA 中设置/取消断点的组合键是 Ctrl+F8，也可以在编辑窗口左侧的代码行提示位置处，用鼠标单击设置/取消断点。进入调试模式的组合键是 Shift+F9。

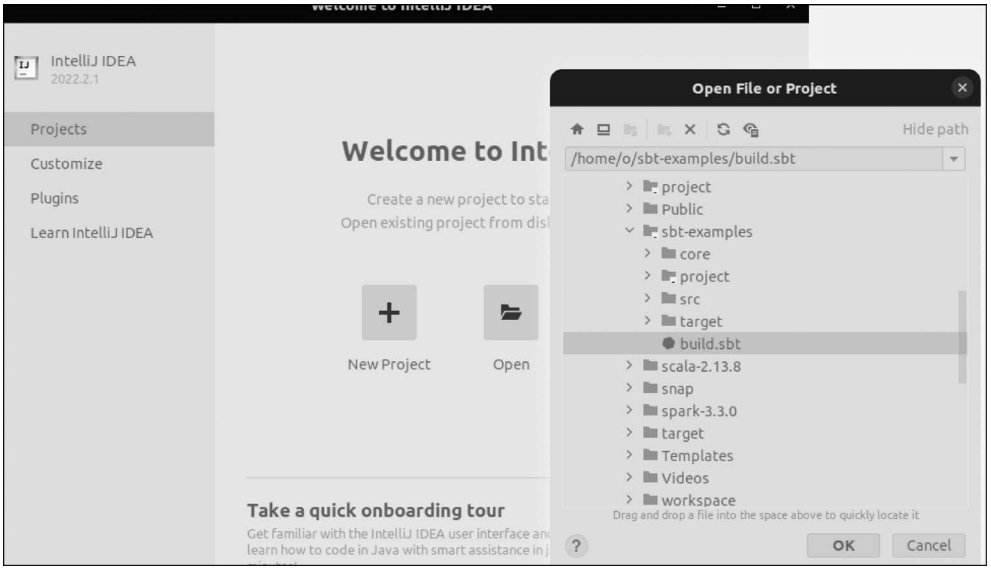

图 3-28 导入 Spark 项目

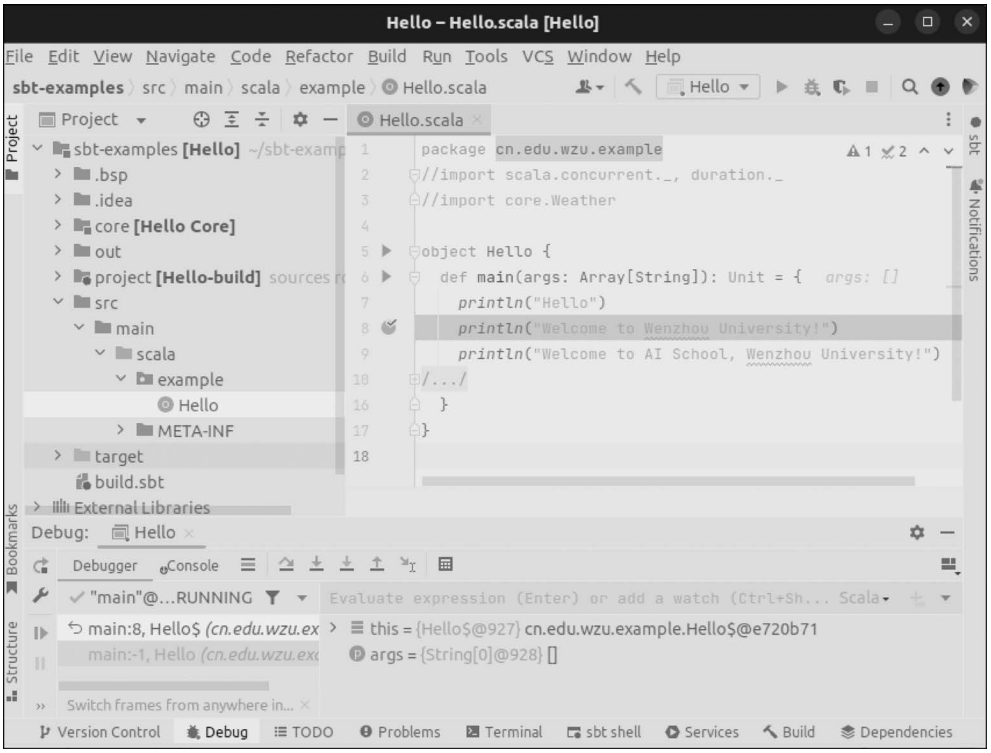

图 3-29 在 IDEA 中调试代码