第 1 章
安装 Spark 以及构建 Spark 集群

本章将详细介绍搭建 Spark 的常用方法。Spark 的单机版便于测试，同时本章也会提到通过 SSH 用 Spark 的内置部署脚本搭建 Spark 集群，使用 Mesos、Yarn 或者 Chef 来部署 Spark。对于 Spark 在云环境中的部署，本章将介绍在 EC2（基本环境和 EC2MR）上的部署。如果你的机器或者集群中已经部署了 Spark，可以跳过本章直接开始使用 Spark 编程。

不管如何部署 Spark，首先得从 `http://spark-project.org/download` 获得 Spark 的一个版本，截止到写本书时，Spark 的最新版本为 0.7 版。对于熟悉 github 的程序员，则可以从 `git://github.com/mesos/spark.git` 直接复制 Spark 项目。Spark 提供基本源码压缩包，同时也提供已经编译好的压缩包。为了和 Hadoop 分布式文件系统 (HDFS) 交互，需要在编译源码前设定相应的集群中所使用的 Hadoop 版本。对于 0.7 版本的 Spark，已经编译好的压缩包依赖的是 1.0.4 版本的 Hadoop。如果想更深入地学习 Spark，推荐自己编译基本源码，因为这样可以灵活地选择 HDFS 的版本，如果想对 Spark 源码有所贡献，比如提交补丁，自己编译源码是必须的。你需要安装合适版本的 Scala 和与之对应的 JDK 版本。对于 Spark 的 0.7.1 版本，需要 Scala 2.9.2 或者更高的 Scala 2.9 版本（如 2.9.3 版）。在写本书时，Linux 发行版 Ubuntu 的 LTS 版本已经有 Scala 2.9.1 版，除此之外，最近的稳定版本已经有 2.9.2 版。Fedora 18 已经有 2.9.2 版。软件包的更新信息可以在 `http://packages.ubuntu.com/search?keywords=scala` 查看到。Scala 官网上的最新版在 `http://scala-lang.org/download`。选择 Spark 支持的 Scala 版本十分重要，Spark 对 Scala 的版本很敏感。

下载 Scala 的压缩包，解压后将 `SCALA_HOME` 设置为 Scala

的根目录，然后将 Scala 根目录下的 bin 目录路径加到 PATH 环境变量中。Scala 设置如下：

```
wget http://www.scala-lang.org/files/archive/scala-2.9.3.tgz && tar -xvf
scala-2.9.3.tgz && cd scala-2.9.3 && export PATH=`pwd`/bin:$PATH &&
export SCALA_HOME=`pwd`
```

也可以在 .bashrc 文件中加入：

```
export PATH=`pwd`/bin:\$PATH
export SCALA_HOME=`pwd`
```

Spark 用 sbt（即 simple build tool，现在已经不是个简单工具了）构建，编译源码的时候会花点时间，如果没有安装 sbt，Spark 构建脚本将会为你下载正确的 sbt 版本。

一台双核且安装有 SSD 的笔记本性能不算高，在它之上安装 Spark 的最新版本花了大概 7 分钟。如果从源码开始构建 0.7 版的 Spark，而不是直接下载编译好的压缩包。可以执行：

```
wget http://www.spark-project.org/download-spark-0.7.0-sources-tgz &&
tar -xvf download-spark-0.7.0-sources-tgz && cd spark-0.7.0 && sbt/sbt
package
```

如果底层存储采用 HDFS，而其版本又和 Spark 中的默认 HDFS 版本不一致，则需要修改 Spark 根目录下 project/SparkBuild.scala 文件中的 HADOOP_VERSION，然后重新编译：

```
sbt/sbt clean compile
```

虽然 sbt 工具在依赖性解析方面已经做得非常好了，但是还是强烈推荐开发者去做一次 clean，而不是增量编译。因为增量编译仍然不能保证每次完全正确。

从源码构建 Spark 将花费一些时间，当编译过程停止在 "Resolving [XYZ]...." 很长时间（大概五分钟）之后，停止

然后重新执行 sbt/sbt package 安装。

如果对 HDFS 版本没有特殊要求，只需要下载 Spark 已经编译好的压缩包，解压就能使用：

```
wget http://www.spark-project.org/download-spark-0.7.0-prebuilt-tgz &&
tar -xvf download-spark-0.7.0-prebuilt-tgz && cd spark-0.7.0
```

Spark 最近已经成为 Apache 孵化器项目，作为一个 Spark 应用开发者，所关心的最明显的变化可能就是 org.apache 命名空间下包名的变化。

一些有用的链接和参考如下：

http://spark-project.org/docs/latest
http://spark-project.org/download/
http://www.scala-lang.org

1.1 单机运行 Spark

单机运行是使用 Spark 最简单的方式，同时也是检查 Spark 构建是否有误的明智方法。在 Spark 的根目录下，有个名为 run 的 shell 脚本，能够用来提交一个 Spark 作业。run 脚本的输入是一个代表 Spark 作业的类名和一些参数。./examples/src/main/scala/spark/examples/ 目录下含有大量 Spark 样例作业。

所有的样例程序都有一个输入参数 master，master 参数是分布式集群中 master 节点的 URL，在本地模式下则是 local[N]（N 是线程的个数）。本地模式下用四线程运行 GroupByTest 样例的命令如下：

```
./run spark.examples.GroupByTest local[4]
```

如果出现错误，可能是因为 SCALA_HOME 没有设置。在 bash

中，能通过 export SCALA_HOME=[pathyouextractscalato] 设置。

如果出现如下错误，可能是你在使用 Scala 2.10 版，然而 Spark 0.7 版本还不支持 2.10 版的 Scala：

```
[literal]"Exception in thread "main" java.lang.NoClassDefFoundError:
scala/reflect/ClassManifest"[/literal]
```

Scala 的开发者决定重新组织介于 2.9 版和 2.10 版之间的一些类。要解决上面的错误，你可以降低 Scala 版本，也可以等待新版本的 Spark 构建支持 Scala 2.10 版。

1.2 在 EC2 上运行 Spark

Spark 提供了很多在 EC2 环境下运行的脚本，脚本文件都存储在根目录下的 ec2 目录中。这些脚本可以用来同时运行多个 Spark 集群，甚至是运行 on-the-spot 实例。Spark 也可以运行在 EMR(Elastic MapReduce) 上，EMR 是 Amazon 关于 MapReduce 集群管理的解决方案，它将会给你扩展实例更大的灵活度。

在 EC2 上用脚本运行 Spark

开始之前，你应该确保有 EC2 账号，如果没有，请访问 https://portal.aws.amazon.com/gp/aws/manageYourAccount 注册，对于 Spark 集群，最好访问 https://portal.aws.amazon.com/gp/aws/securityCredentialsR 生成一个单独访问密钥对。同时也需要生成一个 EC2 的密钥对，这样 Spark 的脚本能够 ssh 到其他已经启动的机器上，这通过访问

https://console.aws.amazon.com/ec2/home 并且选择“Network & Security”下的“Key Pairs”选项来实现。注意密钥对是以区域来创建的。所以你需要确保创建的密钥对和将要使用的 Spark 实例在同一个区域。别忘了给密钥对命名（我们将在本章接下来的内容中使用 spark-keypair 来命名示例中的密钥对），因为将要在脚本中使用它。也可以将你 SSH 公钥上传，从而就不需要重新生成密钥对。这些公钥属于安全敏感信息，所以确保它不被泄露，对于 Amazon 的 EC2 的脚本，也需要设置环境变量 AWS_ACCESS_KEY 和 AWS_SECRET_KEY：

```
chmod 400 spark-keypair.pem
export AWS_ACCESS_KEY="..."
export AWS_SECRET_KEY="..."
```

从 http://aws.amazon.com/developertools/Amazon-EC2/351 下载 Amazon 提供的 EC2 脚本非常有用，解压 zip 文件后，将 bin 目录直接加到 PATH 环境变量中，就像对 Spark 中的 bin 目录所做的一样：

```
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
unzip ec2-api-tools.zip
cd ec2-api-tools-*
export EC2_HOME=`pwd`
export PATH=$PATH:`pwd`:/bin
```

Spark 的 EC2 脚本自动地创建一个用来运行 Spark 集群的隔离安全组和防火墙规则。默认情况下 Spark 的外部通用端口为 8080，这种方式并不好，不幸的是 spark_ec2.py 脚本暂时还不提供限制访问你机器的简单方法。如果你有一个静态 IP 地址，强烈建议在 spark_ec2.py 中限制访问，简单地用 [yourip]/32 替换所有的 0.0.0.0/0。这将不会影响集群内的通信，因为在一个安全组内的所有机器都默认能够和其他机器通信。

然后，启动一个 EC2 上的集群：

```
./ec2/spark-ec2 -k spark-keypair -i pk-[....].pem -s 1 launch
myfirstcluster
```

如果遇到错误："The requested Availability Zone is currently constrained and....", 你可以通过传递 --zone 标记指向另一个 zone。

如果不能 SSH 到集群 master 上的话，请确保只有你有权限去读取私钥，否则 SSH 将会拒绝使用私钥。

由于当节点报告它们自己状态的时候，由于竞争条件你还有可能遇到上面的错误，但是 Spark-ec2 脚本还不能 SSH 过去。关于这个问题，在 https://github.com/mesos/spark/pull/555 上有个修复办法。在 Spark 下一个版本出来之前，有一个临时解决问题的简单方法，就是让 setup_cluster 在启动的时候睡眠 120 秒。

如果启动集群的时候遇到一个短暂的错误，可以用下面命令提供的恢复功能完成启动：

```
./ec2/spark-ec2 -i ~/spark-keypair.pem launch myfirstsparkcluster
--resume
```

万事俱备，你将看到屏幕截图的内容，如图 1-1 所示。

这将分配给你最基本的一个 master 实例和一个 worker 实例的集群，两个实例都是默认配置。接下来，确认 master 节点的 8080 端口没有防火墙规则并确认其已经启动。能看见上面屏幕最后输出 master 的名字。

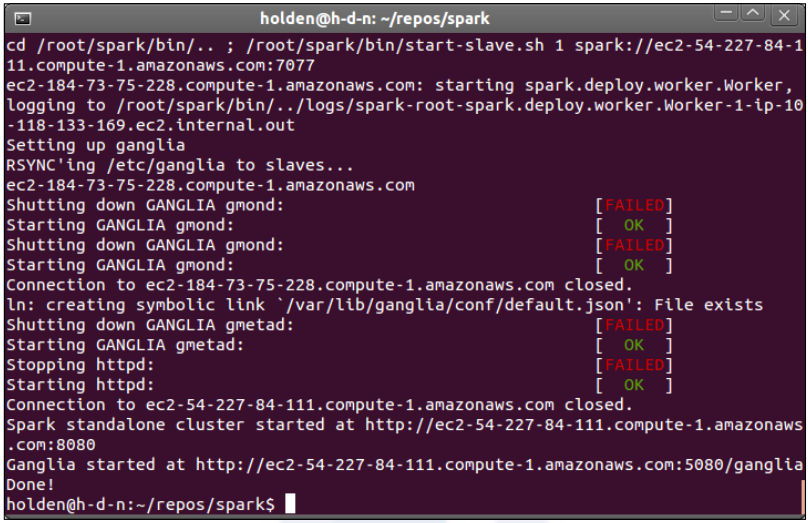

图 1-1

尝试运行一个 Spark 样例作业来确保配置没有问题：

```
sparkuser@h-d-n:~/repos/spark$ ssh -i ~/spark-keypair.pem root@ec2-107-22-48-231.compute-1.amazonaws.com
Last login: Sun Apr  7 03:00:20 2013 from 50-197-136-90-static.hfc.comcastbusiness.net
       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2012.03-release-notes/
There are 32 security update(s) out of 272 total update(s) available
Run "sudo yum update" to apply all updates.
Amazon Linux version 2013.03 is available.
[root@domU-12-31-39-16-B6-08 ~]# ls
ephemeral-hdfs  hive-0.9.0-bin  mesos  mesos-ec2  persistent-hdfs
scala-2.9.2  shark-0.2  spark  spark-ec2
[root@domU-12-31-39-16-B6-08 ~]# cd spark
[root@domU-12-31-39-16-B6-08 spark]# ./run spark.examples.GroupByTest
spark://`hostname`:7077
13/04/07 03:11:38 INFO slf4j.Slf4jEventHandler: Slf4jEventHandler started
13/04/07 03:11:39 INFO storage.BlockManagerMaster: Registered BlockManagerMaster Actor
....
```

```
13/04/07 03:11:50 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 1.100294766 s
2000
```

既然已经能够在 EC2 的集群上运行一个简单的 Spark 作业，现在就可以针对具体 Spark 作业配置一下你的 EC2 集群了。下面是用 `Spark-ec2` 脚本配置集群的多种配置选项。

首先，考虑所需要的实例类型，EC2 提供一个不断丰富的实例集合，能针对 master 和 worker 选择不同的实例类型。实例类型的选择对 Spark 集群性能有较大影响。如果一个 Spark 作业需要很大内存，最好选择一个含有更大内存的实例。可以通过特指 `--instance-type=(name of instance type)` 来指定实例类型。默认情况下，master 和 worker 的实例类型是一样的。当作业是计算密集型的话，将会出现浪费，因为 master 的资源不能充分被利用。你能通过 `--master-instance-type=(name of instance)` 来指定不同的 master 实例类型。

EC2 也有对于 worker 十分有用的 GPU 实例类型。但对 master 来说完全是浪费。注意，由于虚拟机管理程序的高 IO 负载，EC2 的 GPU 性能将比你本地测试的低。

下载实例代码

本书中所有的实例程序都在三个独立的 github repo 中：

- `https://github.com/holdenk/fastdataprocessingwithspark-sharkexamples`
- `https://github.com/holdenk/fastdataprocessingwithsparkexamples`
- `https://github.com/holdenk/chef-cookbook-spark`

Spark 的 EC2 脚本使用 Spark 组提供的 AMI(Amazon Machine Images)。这些 AMI 往往跟不上 Spark 版本的更新速度，如果有对

于 Spark 的自定义补丁（比如要用不同版本的 HDFS），将不会被包含在机器镜像中。目前，AMI 也只能在美国东部地区使用，如果想在其他地区运行它，需要复制 AMI，或者自己在不同的地区制作你的 AMI。

为了使用 Spark 的 EC2 脚本，你所在的地区要有 AMI。为了将默认的 Spark 的 EC2 AMI 复制到一个新的地区，通过查看 spark_ec2.py 脚本中最新 Spark AMI 是什么，这又是通过查看 `LATEST_AMI_URL` 指向的 URL。对于 Spark 0.7，运行下面的命令获得最新 AMI：

```
curl https://s3.amazonaws.com/mesos-images/ids/latest-spark-0.7
```

`ec2-copy-image` 脚本包含你希望获得的复制镜像的功能，但是不能复制不属于自己的镜像。所以必须登录一个前面 AMI 的实例，然后记录它的快照，通过运行以下命令可以获得当前正在运行的镜像的描述信息：

```
ec2-describe-images ami-a60193cf
```

这将显示一个基于 EBS(Elastic Block Store) 的镜像，需要参照 EC2 的命令创建基于 EBS 的实例。通过已获得的启动实例的脚本，可以启动一个 EC2 集群上的实例，然后将其快照记录下来，通过以下命令找到你运行的实例：

```
ec2-describe-instances -H
```

可以复制 `i-[string]` 实例名字，然后保存留有它用

如果想用 spark 的一个普通版本，或者安装工具和依赖，并让它们成为你的 AMI 的一部分，需要在快照复制之前这样做：

```
ssh -i ~/spark-keypair.pem root@[hostname] "yum update"
```

一旦安装了更新，并且安装好你需要的其他软件，你能够继续然后复制快照你的实例：

```
ec2-create-image -n "My customized Spark Instance" i-[instancename]
```

用前面代码中的 AMI 名字，可以通过指定 `[cmd]--ami[/cmd]` 命令行参数启动你自定义的 Spark 版本，也可以将它复制到另一个地区使用：

```
ec2-copy-image -r [source-region] -s [ami] --region [target region]
```

你将获得新的 AMI 名字，可以用它提交你的 EC2 任务，如果想用不同的 AMI 名字，只要指定 `--ami [aminame]` 即可。

当写本书时，有个关于默认 AMI 和 HDFS 的问题，如果 Hadoop 版本和 Spark 编译指定的 Hadoop 版本不一致，需要更新 AMI 上 Hadoop 的版本，参照 `https://spark-project.atlassian.net/browse/SPARK-683` 获得详细信息。

1.3 在 ElasticMapReduce 上部署 Spark

除了 EC2 基本环境之外，Amazon 还提供一个名为 Elastic MapReduce 的 MapReduce 托管解决方案，伴随地也提供一个 Spark 启动脚本，该脚本可以简化在 EMR 上 Spark 的初始使用流程。可以在 Amazon 上通过下面的命令安装 EMR 工具：

```
mkdir emr && cd emr && wget http://elasticmapreduce.s3.amazonaws.com/
elastic-mapreduce-ruby.zip && unzip *.zip
```

为了使 EMR 脚本能够访问你的 AWS(Amazon Web Service) 账

户，你要创建这样一个 credentials.json 文件：

```
{
    "access-id": "<Your AWS access id here>",
    "private-key": "<Your AWS secret access key here>",
    "key-pair": "<The name of your ec2 key-pair here>",
    "key-pair-file": "<path to the .pem file for your ec2 key pair
here>",
    "region": "<The region where you wish to launch your job flows
(e.g us-east-1)>"
}
```

一旦安装了 EMR 工具，就可以通过运行下面的命令启动一个 Spark 集群：

```
elastic-mapreduce --create --alive --name "Spark/Shark Cluster" \
--bootstrap-action s3://elasticmapreduce/samples/spark/install-spark-
shark.sh \
--bootstrap-name "install Mesos/Spark/Shark" \
--ami-version 2.0  \
--instance-type m1.large --instance-count 2
```

5 到 10 分钟之后 EC2MR 实例将会启动。通过执行 `elastic-mapreduce -list` 可以列出集群的状态。一旦其输出 `j-[jobid]`，准备完毕。

有些参考链接如下：

- http://aws.amazon.com/articles/4926593393724923
- http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-install.html

1.4 用 Chef(opscode) 部署 Spark

Chef 是一个渐渐流行的部署大、小集群的自动化管理平台。Chef 可以用来管理一个传统的静态集群，也可以和 EC2 或者其他的云计算提供商一起使用。Chef 用 cookbook 作为最基本的配置单元，可以被泛化或者特指。如果你以前没有用过 Chef，在 `https://learnchef.opscode.com/` 能找到 Chef 的入门教

程。你可以使用一个泛化的 Spark cookbook 作为最基本的启动集群的配置。

为了让 Spark 运行，需要为 master 和 worker 都创建一个角色，同时完成 worker 连接到 master 的配置。首先从 https://github.com/holdenk/chef-cookbook-spark 下载 cookbook。最小化的配置是直接将 master 的 hostname 作为 master（便于 worker 节点连接）和 username，以至于 Chef 可以在正确的位置安装。你也需要接受 Sun Java 的证书，或者切换到可选的 JDK 上。spark-env.sh 中的绝大多数设置都可以通过 cookbook 的配置完成。在 1.7 节“通过 SSH 部署集群”，你能了解这些设置的解释。这些设置可以在每个节点个性化地设定，同时你也可以改变全局的默认值。

为 master 创建一个角色，执行 spark_master_role -e[editor]。你将得到一个可编辑的模板文件，比如对于 master，设置如下：

```
{
  "name": "spark_master_role",

  "description": "",
  "json_class": "Chef::Role",

  "default_attributes": {
    },
  "override_attributes": {
   "username":"spark",
   "group":"spark",
   "home":"/home/spark/sparkhome",
   "master_ip":"10.0.2.15",
  },
  "chef_type": "role",
  "run_list": [
    "recipe[spark::server]",
    "recipe[chef-client]",
  ],
  "env_run_lists": {
    },
}
```

除了将 spark::server 改为 spark::client，用相同的方式创建一个对应于 client 端的角色，然后部署到不同的节点：

```
knife node run_list add master role[spark_master_role]
knife node run_list add worker role[spark_worker_role]
```

最后在节点上运行 chef-client 更新。至此就可以运行一个 Spark 集群了。

1.5 在 Mesos 上部署 Spark

Mesos 是一个能够让多个分布式应用和框架运行在同一集群上的集群管理平台。Mesos 可以在同一个集群上智能地、并发地调度和运行 Spark、Hadoop 以及其他框架（这里 Hadoop 和 Spark 都是一个框架）。Spark 可以将它的任务作为 Mesos 任务提交，或者将 Spark 整体作为一个 Mesos 任务提交。Mesos 支持快速扩展以管理一个更大的集群。Mesos 最初是加州大学伯克利分校的一个研究项目，现在已经成为 Apache 孵化器项目，并已被 Twitter 采用。

在使用 Mesos 之前，先从 http://mesos.apache.org/downloads/ 下载最新版本解压。Mesos 提供各种配置脚本，对于 Ubuntu 下的安装使用 configure.ubuntu-lucid-64 脚本，对于其他的 linux 发行版，请参照 Mesos 的 README 文件指明的对应配置脚本。除了安装 Spark 的要求，检查一下你是否安装了 Python C 头文件（Debian 系统下的 python-dev 包），或者传递 --disable-python 参数给配置脚本。由于 Mesos 需要在每台机器上安装，所以最好不要将 Mesos 安装目录配置到 root 用户的目录下，将 Mesos 安装配置到 Spark 用户的目录下会更简便，如下：

```
./configure --prefix=/home/sparkuser/mesos && make && make check && make
install
```

与 Spark 的 standalone 模式的配置类似，这里也需要检查不同的 Mesos 节点能否找到其他的节点。将 master 的 hostname 添加到 mesossprefix/var/mesos/deploy/masters 中，将 worker 节点的 hostname 添加到 mesossprefix/var/mesos/deploy/slaves 中。然后在 master 的 mesossprefix/var/mesos/conf/mesos.conf 文件下设置 master 与其他节点通信的地址和端口。

一旦 Mesos 构建完毕，就该配置 Spark 和 Mesos 一起协同工作了。将 conf/spark-env.sh.template 复制成 conf/spark-env.sh，然后将 MESOS_NATIVE_LIBRARY 设置为 Mesos 的安装目录。下一小节将提到关于 spark-env.sh 中有关设置的更多信息。

需要在集群中的每台机器上都安装 Mesos 和 Spark，一旦在一台机器上配置了 Mesos 和 Spark，可以用 pscp 将构建复制到所有的节点上：

```
pscp -v -r -h  -l sparkuser ./mesos /home/sparkuser/mesos
```

顺利的话，接下来可以使用 mesosprefix/sbin/mesos-startcluster.sh 脚本开启 Mesos 集群，并且通过 mesos://[host]:5050 作为 master 调度 Mesos 上的 Spark。

1.6 在 Yarn 上部署 Spark

Yarn 是 Apache Hadoop 的第二代 MapReduce。你一旦构建了 Spark 的 assembly jar 包，就可以利用 Spark 提供的简单方式在

Yarn 上调度作业。你建立的 Spark 作业需要使用一个 standalone 模式下 master 的 URL。Spark 应用都是从命令行的参数中读取 master 的 URL，所以指定 `--args` 为 standalone。

运行一下 GroupByTest.scala 样例作业，如下：

```
sbt/sbt assembly #Build the assembly

SPARK_JAR=./core/target/spark-core-assembly-0.7.0.jar ./run spark.deploy.
yarn.Client --jar examples/target/scala-2.9.2/spark-examples_2.9.2-
0.7.0.jar --class spark.examples.GroupByTest --args standalone --num-
workers 2 --worker-memory 1g --worker-cores 1
```

1.7 通过 SSH 部署集群

如果集群未安装任何集群管理软件，你能用一些方便 Spark 部署的脚本通过 SSH 部署 Spark。这种方式在 Spark 文档中叫“ standalone 模式”。如果只有一个 master 和一个 worker，可以在相应的机器上分别用 `./run spark.deploy.master.Master` 和 `./run spark.deploy.worker.Worker spark://MASTERIP:PORT` 启动相应的实例，master 的默认端口是 8080。即使集群由大量机器组成，也不需要到每一台机器上手动地执行命令来启动。bin 目录下提供了大量的启动服务器的脚本。

使用这些脚本的前提是 master 能够无密码登录到所有 worker。推荐创建一个新用户运行 Spark，并让它成为 Spark 的专用用户。本书用 sparkuser 作为 Spark 的用户。在 master 节点上运行 `ssh-keygen` 生成一个 SSH 密钥，在输入密码时提供空密码。一旦生成了密钥，将公钥（如果使用 RSA 密钥，它一般会被默认存储到 `~/.ssh/id_rsa.pub` 中）添加到每个节点的 `./ssh/authorized_keys` 文件中。

Spark 的管理脚本需要你的用户名匹配，如果不匹配的话，你可以在 ~/.ssh/config 配置另一个用户名。

master 能无密码登录到其他机器后，开始配置 Spark。[filepath]conf/spark-env.sh.template[/filepath] 是一个简便的配置文件模板，将它拷贝到 [filepath]conf/spark-env.sh[/filepath]。修改文件中的 SCALA_HOME 为 Scala 的安装目录，同时根据具体环境配置一些（或者所有的）环境变量，如表 1-1 所示。

表 1-1

名　称	意　义	默认值
MESOS_NATIVE_LIBRARY	指向 Mesos 根目录	无
SCALA_HOME	指向 Scala 根目录	无，非空
SPARK_MASTER_IP	master 节点 IP 地址，用于监听和与其他 worker 节点通信	Spark 启动所在的节点 IP
SPARK_MASTER_PORT	Sparkmaster 节点监听端口号	7077
SPARK_MASTER_WEBUI_PORT	master 节点上 Web 界面端口号	8080
SPARK_WORKER_CORES	worker 节点可使用 CPU 核个数	全部使用
SPARK_WORKER_MEMORY	worker 节点可使用内存量	系统最大内存：1GB（或者 512MB）
SPARK_WORKER_PORT	worker 节点运行所在端口号	随机
SPARK_WEBUI_PORT	worker 节点上 Web 界面端口号	8081
SPARK_WORKER_DIR	worker 节点产生文件所在的位置	SPARK_HOME/work_dir

一旦完成了所有的配置，就可以开始启动和运行集群了。用户需要复制相应版本的 Spark 以及构建好的配置到所有的节点

上。安装 PSSH 可以简便部署过程，分布式 SSH 工具 PSSH 包含 PSCP 功能。PSCP 极大地简便了将文件复制到多个目标节点的过程，尽管拷贝过程需要花费一些时间，使用示例如下：

```
pscp -v -r -h conf/slaves -l sparkuser ../spark-0.7.0 ~/
```

如果配置经过改变，需要将改变后的配置传输到所有的 worker 节点上，如下：

```
pscp -v -r -h conf/slaves -l sparkuser conf/spark-env.sh ~/
spark-0.7.0/conf/spark-env.sh
```

如果集群上使用一个共享的 NFS，尽管 Spark 会默认的命名日志文件，但最好区分地指定 worker 的目录，否则它会写到同一个目录中，如果想在共享 NFS 上设置 worker 的日志目录，请考虑添加 hostname 来区别，例如 `SPARK_WORKER_DIR=~/work-'hostname'`。

为了得到更好的性能，也可以考虑将日志文件放到一个独立目录中。

如果 worker 节点上没有安装 Scala，可以通过 pssh 安装：

```
pssh -P -i -h conf/slaves -l sparkuser "wget http://www.scala-lang.org/
downloads/distrib/files/scala-2.9.3.tgz && tar -xvf scala-2.9.3.tgz && cd
scala-2.9.3 && export PATH=$PATH:`pwd`/bin && export SCALA_HOME=`pwd` &&
echo \"export PATH=`pwd`/bin:\\\\$PATH && export SCALA_HOME=`pwd`\" >>
~/.bashrc"
```

接下来启动集群，`start-all.sh` 和 `start-master.sh` 都是在集群的 master 上运行的脚本。启动脚本都是守护进程，所以终端不会输出大量的运行日志。

```
ssh master bin/start-all.sh
```

如果遇到无法找到一个类的错误，类似于 java.lang.NoClassDefFoundError: scala/ScalaObject，请检查是否已经安装并且正确地设置了 SCALA_HOME。

Spark 的脚本假设 Spark 在 master 和 worker 节点上的安装目录是一样的。如果安装目录不同，应该编辑 bin/spark-config.sh，将其设置为合适的目录。

Spark 提供的帮助管理集群的命令如表 1-2 所示。

表 1-2

命　令	用　途
bin/slaves.sh <command>	command 是可以在任意 worker 节点上运行的命令，比如 bin/slaves.sh uptime 能够显示每个 worker 节点运行的总时间
bin/start-all.sh	启动 master 节点和所有 worker 节点，在 master 节点上执行
bin/start-master.sh	启动 master 节点，在 master 节点上执行
bin/start-slaves.sh	启动所有的 worker 节点
bin/start-slave.sh	启动一个 worker 节点
bin/stop-all.sh	停止 master 节点和所有 worker 节点
bin/stop-master.sh	停止 master 节点
bin/stop-slaves.sh	停止所有的 worker 节点

spark 启动脚本运行之后，用户可以通过浏览器获得屏幕截图上显示的界面（图 1-2）。在 master 节点的 8080 端口有个友好的 web 界面；通过 8081 端口可以访问和切换到所有的 worker 节点上。界面中包含正在运行的 worker 的信息，以及现在正在运行和已经完成的作业信息。

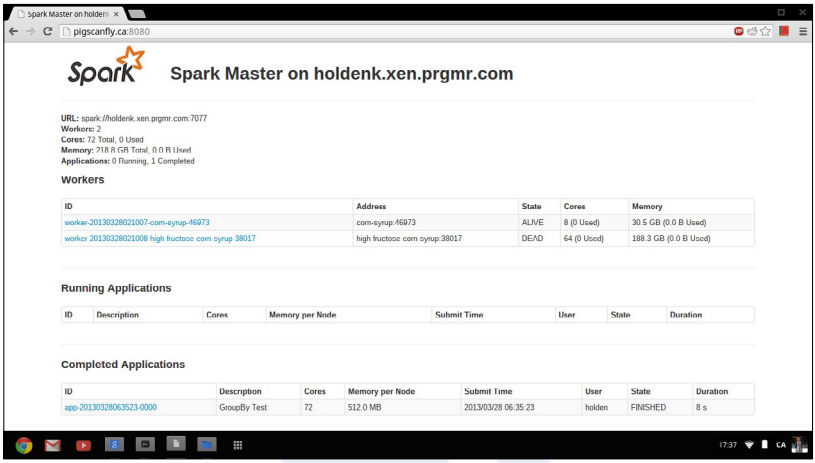

图 1-2

至此 Spark 集群就完成了启动。就像在本地模式下一样，可以使用提供的 run 脚本执行 Spark 作业。所有的样例作业都在 `examples/src/main/scala/spark/examples/` 中，这些作业都需要一个参数指向集群的主节点。假设你在 master 节点上，可以试着运行一个样例作业：

```
./run spark.examples.GroupByTest spark://`hostname`:7077
```

如果遇到了下面这个问题 `java.lang.UnsupportedClassVersionError`，请升级 JDK，如果使用的是 Spark 编译后的版本，需要重新编译源码，Spark 0.7 版本需要 JDK 1.7 版编译，检查一下 JRE 的命令如下：

```
java -verbose -classpath ./core/target/scala-2.9.2/
classes/
spark.SparkFiles | head -n 20
```

注意 49 是 JDK1.5，50 是 JDK1.6，60 是 JDK1.7。

如果不能连接到 localhost，请确认是否已将 master 配置为监

听所有的 IP 地址（或者你没有将 localhost 配置为 master 监听的 ip 地址）。

如果万事俱备，你将看到屏幕上输出一些日志信息。如下：

```
13/03/28 06:35:31 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 2.482816756 s
2000
```

1.8 链接和参考

- http://archive09.linux.com/feature/151340
- http://spark-project.org/docs/latest/spark-standalone.html
- https://github.com/mesos/spark/blob/master/core/src/main/scala/spark/deploy/worker/WorkerArguments.scala
- http://bickson.blogspot.com/2012/10/deploying-graphlabsparkmesos-cluster-on.html
- http://www.ibm.com/developerworks/library/os-spark/
- http://mesos.apache.org/
- http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923
- http://spark-project.org/docs/latest/ec2-scripts.html

1.9 小结

本章介绍了怎样部署 Spark 的本地模式和集群模式，现在可以开始运行 Spark 应用了。在下章中，我们将要学习怎样使用 Spark shell。

第 2 章
Spark shell 的使用

Spark shell 是一个特别适合快速开发 Spark 原型程序的工具，可以帮助我们熟悉 Scala 语言。即使你对 Scala 不熟悉，仍然可以使用这个工具。Spark shell 使得用户可以和 Spark 集群交互，提交查询，这便于调试，也便于初学者使用 Spark。前一章介绍了运行 Spark 实例之前的准备工作，现在你可以开启一个 Spark shell，然后用下面的命令连接你的集群：

```
MASTER=spark://`hostname`:7077 ./spark-shell
```

如果在本地模式下运行 Spark，而且在没有 Spark 实例运行的前提下，直接执行 ./spark-shell 将以单线程启动，多线程需追加 `local[n]`。

2.1 加载一个简单的 text 文件

Spark shell 一旦连接上一个 Spark 集群，终端会输出一些日志信息，这些信息中含有指定 app ID 的内容，类似于 `Connected to Spark cluster with app ID app-20130330015119-0001`。随后 Web 用户界面（默认 8080 端口）上正在运行的 application 中会显示该 app ID。接下来下载一个数据集来做些实验，为《 The Elements of Statistical Learning 》这本书准备的大量数据集，都是以非常便于使用的格式给出的。获得垃圾链接数据集的命令如下：

```
wget http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/spam.data
```

在 Spark shell 中输入以下语句，作用是将 `spam.data` 当作文本文件加载到 Spark 中：

```
scala> val inFile = sc.textFile("./spam.data")
```

上面的语句将 spam.data 文件中的每行作为一个 RDD（Resilient Distributed Datasets）中的单独元素加载到 Spark 中，并返回一个名为 inFile 的 RDD。

注意当你连接到 Spark 的 master 之后，若集群中没有分布式文件系统，Spark 会在集群中每一台机器上加载数据，所以要确保集群中的每个节点上都有完整数据。通常可以选择把数据放到 HDFS、S3 或者类似的分布式文件系统去避免这个问题。在本地模式下，可以将文件从本地直接加载，例如 sc.textFile([filepah])，想让文件在所有机器上都有备份，请使用 SparkContext 类中的 addFile 函数，代码如下：

```
scala> import spark.SparkFiles;
scala> val file = sc.addFile("spam.data")
scala> val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

与大多数的 shell 一样，Spark shell 也有命令的历史记录。按“up 键”可得到前一条执行过的命令。如果你不想完整地输入一条命令或者不确信调用一个对象的什么方法，按“Tab 键”，Spark shell 能够尽可能地自动补全你的代码。

对于逻辑回归的例子，RDD 要是以文件中的每行来组织记录就不是十分有用了，因为逻辑回归需要由空格分割的数值作为输入数据。直接在 RDD 上做映射，能够较快地获得特定格式的数据（注意 _.toDouble 和 x=>x.toDouble 等价）：

```
scala> val nums = inFile.map(x => x.split(' ').map(_.toDouble))
```

然后比较 nums 和 inFile 这两个 RDD，确认一下两种数据的内容是一致的。通过在两个 RDD 上调用 first() 函数查看这两个

RDD 中的第一个元素：

```
scala> inFile.first()
[...]
res2: String = 0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29
1.93 0 0.96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0.778 0 0 3.756 61 278 1

scala> nums.first()
[...]
res3: Array[Double] = Array(0.0, 0.64, 0.64, 0.0, 0.32, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.64, 0.0, 0.0, 0.0, 0.32, 0.0, 1.29, 1.93, 0.0, 0.96,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.778, 0.0, 0.0, 3.756, 61.0, 278.0, 1.0)
```

2.2 用 Spark shell 运行逻辑回归

当执行一条不包含左半部（比如省略 `val x=y` 中的 `val=x`）的语句时，Spark shell 会以 `res[number]` 命名结果值（number 默认是递增的），若显式地写出 `val res[number]=y`，`res[number]` 将用于命名结果值。之前我们获得了更利于使用的格式的数据集，可以用它来做点更有意义的事情，用 Spark 在垃圾链接数据集上运行逻辑回归程序的例子如下：

```
scala> import spark.util.Vector
import spark.util.Vector

scala> case class DataPoint(x: Vector, y: Double)
defined class DataPoint

scala> def parsePoint(x: Array[Double]): DataPoint = {
      DataPoint(new Vector(x.slice(0,x.size-2)) , x(x.size-1))
      }
parsePoint: (x: Array[Double])this.DataPoint

scala> val points = nums.map(parsePoint(_))
points: spark.RDD[this.DataPoint] = MappedRDD[3] at map at
<console>:24

scala> import java.util.Random
import java.util.Random
```

```
scala> val rand = new Random(53)
rand: java.util.Random = java.util.Random@3f4c24
scala> var w = Vector(nums.first.size-2, _ => rand.nextDouble)
13/03/31 00:57:30 INFO spark.SparkContext: Starting job: first at
<console>:20
...
13/03/31 00:57:30 INFO spark.SparkContext: Job finished: first at
<console>:20, took 0.01272858 s
w: spark.util.Vector = (0.7290865701603526, 0.8009687428076777,
0.6136632797111822, 0.9783178194773176, 0.3719683631485643,
0.46409291255379836, 0.5340172959927323, 0.04034252433669905,
0.3074428389716637, 0.8537414030626244, 0.8415816118493813,
0.719935849109521, 0.24316468306717812, 0.17139348575456848,
0.5005137792223062, 0.8915164469396641, 0.7679331873447098,
0.7887571495335223, 0.7263187438977023, 0.40877063468941244,
0.7794519914671199, 0.1651264689613885, 0.18070069370302O1,
0.3227972103818231, 0.2777324549716147, 0.20466985600105037,
0.5823059390134582, 0.4489508737465665, 0.44030858771499415,
0.6419366305419459, 0.5191533842209496, 0.43170678028084863,
0.9237523536173182, 0.5175019655845213, 0.47999523211827544,
0.25862648071479444, 0.020548000101787922, 0.18555332739714137, 0....

scala> val iterations = 100
iterations: Int = 100

scala> import scala.math._

scala> for (i <- 1 to iterations) {
        val gradient = points.map(p =>
          (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
        ).reduce(_ + _)
        w -= gradient
      }
[....]

scala> w
res27: spark.util.Vector = (0.2912515190246098, 1.05257972144256,
1.1620192443948825, 0.764385365541841, 1.3340446477767611,
0.6142105091995632, 0.8561985593740342, 0.7221556020229336,
0.40692442223198366, 0.8025693176035453, 0.7013618380649754,
0.943828424041885, 0.4009868306348856, 0.6287356973527756,
0.3675755379524898, 1.2488466496117185, 0.8557220216380228,
0.7633511642942988, 6.389181646047163, 1.43344096405385,
1.729216408954399, 0.4079709812689015, 0.3706358251228279,
0.8683036382227542, 0.36992902312625897, 0.3918455398419239,
0.2840295056632881, 0.7757126171768894, 0.4564171647415838,
0.6960856181900357, 0.6556402580635656, 0.060307680034745986,
0.31278587054264356, 0.9273189009376189, 0.0538302050535121,
0.545536066902774, 0.9298009485403773, 0.922750704590723,
0.072339496591
```

如果顺利，你已经使用 Spark 运行了逻辑回归程序。至此，我们已经学到了很多，学到了定义类，创建 RDD 以及创建函数。感受到 Spark shell 是如此的方便，因为它很大程度上基于 Scala REPL（Scala 交互式 shell，即 Scala 解释器），并且继承了 Scala REPL（Read-Evaluate-Print-Loop）（读取 – 求值 – 打印 – 循环）的所有功能。Spark shell 虽然强大，但多数时候你还是会运行编译后的代码，而不是使用 REPL 环境。

2.3 交互式地从 S3 加载数据

本节开始 Spark shell 上的第二个实战。Spark 支持从 S3 加载数据，S3 是 Amazon EMR 的一部分，它提供了一些 Wikipedia 的浏览统计数据，这些浏览数据的格式便于 Spark 测试。为了访问数据，首先需要将 AWS 访问证书设置为 shell 的参数。对于 EC2 的注册指令和建立 shell 的参数，请参照第 1 章中的“在 EC2 上用脚本运行 Spark”这一小节（访问 S3，还需要额外的密钥 `fs.s3n.awsAccessKeyId/awsSecretAccessKey` 或者使用这种格式 `s3n://user:pw@`）。以上步骤完成后，开始加载 S3 上的数据，然后查看一下数据的第一行：

```
scala> val file = sc.textFile("s3n://bigdatademo/sample/wiki/")
13/04/21 21:26:14 INFO storage.MemoryStore: ensureFreeSpace(37539)
called with curMem=37531, maxMem=339585269
13/04/21 21:26:14 INFO storage.MemoryStore: Block broadcast_1 stored
as values to memory (estimated size 36.7 KB, free 323.8 MB)
file: spark.RDD[String] = MappedRDD[3] at textFile at <console>:12

scala> file.take(1)
13/04/21 21:26:17 INFO mapred.FileInputFormat: Total input paths to
process : 1
...
13/04/21 21:26:17 INFO spark.SparkContext: Job finished: take at
<console>:15, took 0.533611079 s
res1: Array[String] = Array(aa.b Pecial:Listusers/sysop 1 4695)
```

如果不将 AWS 访问证书设置为 shell 的参数，访问方式就需要这样写：s3n://<AWS ACCESS ID>:<AWS SECRET>@bucket/path。查看一下第一行的数据十分重要，因为除非我们强制使 Spark 去物化数据，否则数据并不会真正被加载。Amazon 提供了一个小的数据集给开发者使用。数据是从稍大的 http://aws.amazon.com/datasets/4182 提供的数据集上拉取的。当以交互式模式开发的时候，通常想得到快速的结果反馈，因此这种实战非常有用。但如果抽样数据太大，执行时间太长，还是通过嵌入在 Spark shell 中的 sample 函数削减 RDD 的大小比较好。

```
scala> val seed  = (100*math.random).toInt
seed: Int = 8
scala> file.sample(false,1/10.,seed)
res10: spark.RDD[String] = SampledRDD[4] at sample at <console>:17

//If you wanted to rerun on the sampled data later, you could write it
back to S3
scala> res10.saveAsTextFile("s3n://mysparkbucket/test")
13/04/21 22:46:18 INFO spark.PairRDDFunctions: Saving as hadoop file
of type (NullWritable, Text)
....
13/04/21 22:47:46 INFO spark.SparkContext: Job finished:
saveAsTextFile at <console>:19, took 87.462236222 s
```

完成数据加载之后，我们开始在样本数据中寻找最受欢迎的文章。首先对每行数据进行解析，将其拆分为 < 文章名，计数 > 键值对。其次，由于同一个文章名可以出现多次，因此需要按照文章名对数据进行一次规约，将同一个文章的多个计数进行累加。最后，规约产生的键值对的键和值将进行一次交换，对交换后的数据执行排序操作后，我们就获得了访问量最多的文章。具体过程如下：

```
scala> val parsed = file.sample(false,1/10.,seed).map(x => x.split("
")).map(x => (x(1), x(2).toInt))
parsed: spark.RDD[(java.lang.String, Int)] = MappedRDD[5] at map at
<console>:16

scala> val reduced = parsed.reduceByKey(_+_)
13/04/21 23:21:49 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable
13/04/21 23:21:49 WARN snappy.LoadSnappy: Snappy native library not
loaded
13/04/21 23:21:50 INFO mapred.FileInputFormat: Total input paths to
process : 1
reduced: spark.RDD[(java.lang.String, Int)] = MapPartitionsRDD[8] at
reduceByKey at <console>:18

scala> val countThenTitle = reduced.map(x => (x._2, x._1))
countThenTitle: spark.RDD[(Int, java.lang.String)] = MappedRDD[9] at
map at <console>:20

scala> countThenTitle.sortByKey(false).take(10)
13/04/21 23:22:08 INFO spark.SparkContext: Starting job: take at
<console>:23
....
13/04/21 23:23:15 INFO spark.SparkContext: Job finished: take at
<console>:23, took 66.815676564 s
res1: Array[(Int, java.lang.String)] = Array((213652,Main_Page),
(14851,Special:Search), (9528,Special:Export/Can_You_Hear_Me),
(6454,Wikipedia:Hauptseite), (4189,Special:Watchlist), (3520,%E7
%89%B9%E5%88%A5:%E3%81%8A%E3%81%BE%E3%81%8B%E3%81%9B%E8%A1%A8%E7
%A4%BA), (2857,Special:AutoLogin), (2416,P%C3%A1gina_principal),
(1990,Survivor_(TV_series)), (1953,Asperger_syndrome))
```

此外也可以利用 Python 和 Spark 交互，通过运行 `./pySpark` 即可。

2.4 小结

本章涵盖了怎样使用 Spark shell 并利用它加载数据，也引导你做了一个简单的机器学习实验。现在你已经了解 Spark 的交互式终端是如何工作的，下章将会讲解怎样在更传统的环境下建立 Spark 作业。

第 3 章
构建并运行 Spark 应用

用 Spark shell 运行 Spark 的交互式模式在代码持久化方面有所局限，而且 Spark shell 不支持 Java 语言。构建一个 Spark 作业比构建一个普通的应用更需要技巧，因为所有要依赖的 jar 包都必须被拷贝到集群中的每台机器上。本章将涵盖用使用 Maven 或者 sbt 构建一个 Java 和 Scala 实现的 Spark 作业，以及用 non-maven-aware 构建系统来构建 Spark 作业。

3.1 用 sbt 构建 Spark 作业

sbt(simple build tool) 是一个流行的 Scala 构建工具，它同时支持 Scala 项目和 Java 项目的构建。用 sbt 构建 Spark 项目是非常好的选择，因为 Spark 本身就是使用 sbt 构建的。使用它，依赖的添加将变得非常容易（这点对 Spark 尤其重要），同时它也支持把所有的东西打成一个可部署的 jar 包。当前，为了简化安装过程，利用 sbt 构建项目时通常都是利用 shell 脚本来引导项目自带的特定版本的 sbt，使构建变得简单。

不妨练下手，拿一个已经可以运行的 Spark 作业，按照以下步骤为它创建一个构建文件。在 Spark 的根目录中，将实例 `GroupByTest` 拷贝到一个新的目录中，如下：

```
mkdir -p example-scala-build/src/main/scala/spark/examples/
cp -af sbt example-scala-build/
cp examples/src/main/scala/spark/examples/GroupByTest.scala example-scala-build/src/main/scala/spark/examples/
```

由于我们的 jar 包要上传至所有节点，所有的依赖包也必须一并上传。你可以挨个儿上传一大把 jar 包，也可以使用一个方便的 sbt 插件 `sbt-assembly` 将所有依赖集合进一个 jar 包中。如果你的项目不涉及错综复杂的依赖关系，那就不必动用 `assembly`

插件了。这种情况下，就必须要在 Spark 的根目录下运行 sbt/sbt assembly，然后把生成的 jar 包 core/target/spark-core-assembly-0.7.0.jar 加到 classpath 环境变量中。sbt-assembly 包是一个非常好的工具，有了它就不用手工管理一大堆 jar 文件了。将以下代码加入构建文件 project/plugins.sbt 中，即可启用 assembly 插件。

```
resolvers += Resolver.url("artifactory",
url("http://scalasbt.artifactoryonline.com/scalasbt/
sbt-plugin-releases"))(Resolver.ivyStylePatterns)

resolvers += "Typesafe Repository" at
"http://repo.typesafe.com/typesafe/releases/"

resolvers += "Spray Repository" at "http://repo.spray.cc/"
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.8.7")
```

sbt 利用 resolver 来定位包的位置，你可以把它们当作额外的 APT(Advanced Package Tool) PPA(Personal Package Archive) 源，只不过其作用范围仅限于待构建的包。在浏览器中打开 resolver 中列出的 URL，大部分情况下你都会看到一堆目录，从中可以看见该 resolver 提供了哪些包。上述 resolver 中的 URL 都指向网络位置，但是也有些指向本地路径，这些 resolver 一般用于简化开发过程。顾名思义，addSbtPlugin 指令的意思是从 com.eed3si9n 引入版本号为 0.8.7 的 sbt-assembly 包，同时也隐式地指定了 Scala 的版本和 sbt 的版本。最后记得执行 sbt reload clean update 安装新插件。

下面是其中一个样例作业 GroupByTest.scala 的构建文件，将以下代码加到 ./build.sbt 中：

```
//Next two lines only needed if you decide to use the assembly plugin
import AssemblyKeys._
assemblySettings

scalaVersion := "2.9.2"
```

```
name := "groupbytest"

libraryDependencies ++= Seq(
  "org.spark-project" % "spark-core_2.9.2" % "0.7.0"
)

resolvers ++= Seq(
  "JBoss Repository" at
  "http://repository.jboss.org/nexus/content/
   repositories/releases/",
  "Spray Repository" at "http://repo.spray.cc/",
  "Cloudera Repository" at
  "https://repository.cloudera.com/artifactory/cloudera-repos/",
  "Akka Repository" at "http://repo.akka.io/releases/",
  "Twitter4J Repository" at "http://twitter4j.org/maven2/"
)
//Only include if using assembly
mergeStrategy in assembly <<= (mergeStrategy in assembly) {
  (old) =>
  {
    case PathList("javax", "servlet", xs @ _*) => MergeStrategy.first
    case PathList("org", "apache", xs @ _*) => MergeStrategy.first
    case "about.html"  => MergeStrategy.rename
    case x => old(x)
  }
}
```

如你所见，build.sbt 构建文件的格式和 plugins.sbt 的格式是一样的。关于这个构建文件的独特之处，值得一提的是，就像在 plugins.sbt 文件中一样，也必须加一些 resolvers 确保 sbt 能够找到所有的依赖。注意必须指定 spark-core 的版本，如 "org.spark-project" % "spark-core_2.9.2" % "0.7.0" 而不是简单的 "org.spark-project" %% "spark-core" % "0.7.0"。可能的话，你应该尽量使用 %% 格式，这样能够自动指定 Scala 版本。上述构建文件中的另一要点是用到了 mergeStrategy。由于多个依赖的 jar 包可能会包含相同的文件，当在将所有文件合并到单个 jar 包中的时候，需要告诉 assembly 插件怎样处理这种情况。除了 mergeStrategy 和手工指定 Spark 对应的 Scala 版本之外，上述构建文件就没什么好解释的了。

如果你在 master 的 JDK 版本和 worker 上的 JDK 不一致，你可以通过在构建文件中加入下面的语句切换到目标 JDK，以便它们保持一致：

```
javacOptions ++= Seq("-target", "1.6")
```

至此构建文件就被定义好了，然后构建 GroupByTest 这个 Spark 作业：

```
sbt/sbt clean compile package
```

这将产生 target/scala-2.9.2/groupbytest_2.9.2-0.1-SNAPSHOT.jar。

在 Spark 根目录中运行 sbt/sbt assembly，检查是否生成 assembly 的 jar 包并将它加入到 classpath 环境变量中。example 程序需要指定 Spark 的 SPARK_HOME 的路径和 SPARK_EXAMPLES_JAR 的路径。也需要用 -cp 指定针对本地 Scala 版本构建得到的 jar 包的 classpath 环境变量。运行下面的例子：

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/scala-2.9.2/
groupbytest-assembly-0.1-SNAPSHOT.jar"  scala -cp/users/
sparkuser/spark-0.7.0/example-scala-build/target/scala-2.9.2/
groupbytest_2.9.2-0.1-SNAPSHOT.jar:/users/sparkuser/spark-0.7.0/
core/target/
spark-core-assembly-0.7.0.jar spark.examples.GroupByTest local[1]
```

如果将所有的依赖用 assembly 插件都打到一个简单的 jar 包中，需要这样调用：

```
sbt/sbt assembly
```

这样能够产生一个 assembly 的 jar 包 target/scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar，然后能够以不指定 spark-core-assembly 的方式运行一遍：

```
SPARK_HOME="../" \ SPARK_EXAMPLES_JAR="./target/scala-2.9.2/groupbytest-
assembly-0.1-SNAPSHOT.jar" \
 scala -cp /users/sparkuser/spark-0.7.0/example-scala-build/target/
scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar spark.examples.
GroupByTest local[1]
```

随着版本的演进，你今后可能还会碰到各种 jar 包合并相关的问题；真到那时候，在网上随便搜索一下，应该就能找到靠谱的解决办法，本书就不操心那些尚未出现的问题了。一般来说，`MergeStrategy.first` 就够用了。

跑通前面的代码并不意味着万事大吉。sbt 会查询本地缓存，依赖解析过程中定位到的依赖包可能是由其他项目写入缓存的，因此在一台机器上能构建成功并不意味着在别的机器上也能成功。保险起见，开始构建之前应该删除本地的 `ivy` 缓存并执行 `sbt clean`。如果发现某些依赖包下载失败，请检查 Spark 的 resolver 列表，并在 `build.sbt` 中自行添加缺失条目。

一些有用的链接如下：

- `http://www.scala-sbt.org/`
- `https://github.com/sbt/sbt-assembly`
- `http://spark-project.org/docs/latest/scala-programming-guide.html`

3.2 用 Maven 构建 Spark 作业

Maven 是一个开源的 Apache 项目，它能构建 Java 版本和 Scala 版本的 Spark 作业。和 sbt 一样，可以通过 Maven 包含 Spark 所有的依赖关系简化构建进程，Maven 能够用插件将 Spark 和所有的依赖捆绑为一个简单的 jar 包，和 `sbt/sbt assembly`

的功能是一样的。

为了描述用 Maven 构建 Spark 作业的过程，本小节以一个 Java 样例作业为例子，因为 Maven 更常用来构建 Java 任务。第一步，用一个可以运行的 Spark 作业，将创建构建文件的步骤过一遍。先把 GroupByTest 例子复制到新的目录中，生成 maven 的模板如下：

```
mkdir example-java-build/; cd example-java-build
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DgroupId=spark.examples \
   -DartifactId=JavaWordCount \
   -Dfilter=org.apache.maven.archetypes:maven-archetype-quickstart
cp ../examples/src/main/java/spark/examples/JavaWordCount.java
JavaWordCount/src/main/java/spark/examples/JavaWordCount.java
```

第二步：更新 Maven 的 pom.xml，让它包含正在使用的 Spark 的版本信息。由于运行的这个例子文件需要 JDK1.5，将需要更新 Maven 所要使用的 Java 版本；在写本书的时候，它默认是 1.3 版本，在 <projects> 标签之间，需要添加下面代码：

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.spark-project</groupId>
    <artifactId>spark-core_2.9.2</artifactId>
    <version>0.7.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
```

```
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

现在用 Maven 的 package 功能构建 jar 包，运行下面的命令：

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/JavaWordCount-1.0-
SNAPSHOT.jar"  java -cp ./target/JavaWordCount-1.0-SNAPSHOT.jar:../../
core/target/spark-core-assembly-0.7.0.jar spark.examples.JavaWordCount
local[1] ../../README
```

就像 sbt 一样，可以用一个插件集合所有的依赖进一个 jar 包中，在 <plugins> 标签中，添加以下代码：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.7</version>
  <configuration>
    <!-- This transform is used so that merging of akka configuration
files works -->
    <transformers>
      <transformer implementation=
  "org.apache.maven.plugins.shade.resource
  .ApacheLicenseResourceTransformer">
      </transformer>
      <transformer implementation=
  "org.apache.maven.plugins.shade
  .resource.AppendingTransformer">
        <resource>reference.conf</resource>
      </transformer>
    </transformers>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

然后执行 mvn assembly，产生 jar 包能够直接拿来执行上节的代码，同时可以省略对 Spark 的 assembly jar 包的指定。

一些有用的链接如下：

- http://maven.apache.org/guides/getting-started/
- http://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html
- http://maven.apache.org/plugins/maven-dependency-plugin/

3.3 用其他工具构建 Spark 作业

如果 sbt 或者 Maven 对你都不是很合适，你可以选择自己习惯的构建系统。还好，Spark 可以将 Spark 依赖的所有 jar 包集合成一个扁平的 jar 包，方便你利用自己的构建系统进行构建。很简单，在 Spark 的根目录中运行一下 `sbt/sbt assembly`，将生成的 jar 包从 `core/target/spark-core-assembly-0.7.0.jar` 拷到你的构建依赖中。

不管你的构建系统是什么，你时不时都会需要使用 Spark 的某个补丁版本。在这种情况下，可以将 Spark 的库文件部署在本地。我建议你给自用的补丁版本单独分配一个版本号，以确保 sbt/maven 不会找错文件。编辑 `project/SparkBuild.scala`，调整“`version:=`”这部分代码即可修改版本号。如果用的是 sbt，请务必运行 `sbt/sbt update`。对于其他的构建系统，只要确保构建时用的是新的 assembly jar 包就可以了。

3.4 小结

现在你已经可以用 Maven、sbt 或者其他的构建系统构建 Spark 作业，之后可以钻进 Spark 系统去干点更有意思的事情了，下章将会介绍怎样创建 SparkContext。