sbt Reference Manual

Contents

Preface

sbt 入门

sbt 使用少数的几个概念来支撑它灵活并且强大的构建定义。其实没有太多的概念，但是 sbt 并不完全像其他的构建体系，而且如果你没有看过文档的话，你偶尔 将会遇到一些细节问题。

这篇入门指南覆盖了一些你在创建和维护一个 sbt 构建定义时需要知道的概念。

强烈建议看完该入门指南!

如果你非常急切，你可以直接进入以下几小节了解最重要的概念背景 .sbt 构建定义，scopes，更多关于设置。但是我们不保证跳过该指南中的其他小节是一个好的想法。

最好是按顺序阅读，因为该指南中后面的小节建立在前面介绍的概念的基础上。

感谢尝试 sbt 并且 体验其中的乐趣!

安装 sbt

创建一个 sbt 工程，你需要经过以下几步：

- 安装 JDK (建议使用 Eclipse Adoptium Temurin JDK 8, 11, 或 17)。
- 安装 sbt 并且创建脚本来运行它。
- 建立一个简单的 hello world 工程
  - 创建一个工程目录并且将源文件放在其中。
  - 创建你的构建定义。
- 继续前往 运行 sbt 学习怎么运行 sbt。
- 然后前往 .sbt 构建定义 学习更多关于构建的定义。

最后，安装步骤就简化为一个 Jar 文件和一个 Shell 脚本，但是取决于你的平台，我们提供了好几种方式来使得步骤不是那么单调。macOS，Windows，或Linux 提供了相应的安装步骤。

提示和技巧

如果你在运行 sbt 时遇到任何问题，查看 安装建议 中的终端编码 (terminal encoding)，HTTP 代理，JVM 参数。

在 macOS 上安装 sbt

Install sbt with cs setup

Follow Install page, and install Scala using Coursier. This should install the latest stable version of `sbt`.

Install JDK

Follow the link to install [JDK 8 or 11][AdoptOpenJDK], or use SDKMAN!

通过 SDKMAN! 安装

```
$ sdk install java $(sdk list java | grep -o "\b8\.[0-9]*\.[0-9]*\-tem" | head -1)
$ sdk install sbt
```

通过通用的包安装

下载 ZIP 或者 TGZ 包并解压。

通过第三方的包安装

> **注意：**第三方的包可能没有提供最新的版本，请记得将任何问题反馈给这些包相关的维护者。

通过 Homebrew 安装

```
$ brew install sbt
```

在 Windows 上安装 sbt

Install sbt with cs setup

Follow Install page, and install Scala using Coursier. This should install the latest stable version of `sbt`.

通过通用的安装包安装

下载 ZIP 或者 TGZ 包并解压。

通过 Windows 安装包安装

下载 msi 安装包 并安装。

通过第三方的包安装

> **注意：**第三方的包可能没有提供最新的版本，请记得将任何问题反馈给这些包相关的维护者。

通过 Scoop 安装

```
$ scoop install sbt
```

在 Linux 上安装 sbt

Install sbt with cs setup

Follow Install page, and install Scala using Coursier. This should install the latest stable version of `sbt`.

Installing from SDKMAN

To install both JDK and sbt, consider using SDKMAN.

```
$ sdk install java $(sdk list java | grep -o "\b8\.[0-9]*\.[0-9]*\-tem" | head -1)
$ sdk install sbt
```

Using Coursier or SDKMAN has two advantages.

1. They will install the official packaging by Eclipse Adoptium, as opposed to the "mystery meat OpenJDK builds".
2. They will install `tgz` packaging of sbt that contains all JAR files. (DEB and RPM packages do not to save bandwidth)

Install JDK

You must first install a JDK. We recommend **Eclipse Adoptium Temurin JDK 8**, **JDK 11**, or **JDK 17**.

The details around the package names differ from one distribution to another. For example, Ubuntu xenial (16.04LTS) has openjdk-8-jdk. Redhat family calls it java-1.8.0-openjdk-devel.

通过通用的安装包安装

下载 ZIP 或者 TGZ 包并解压。

Ubuntu 和其他基于 Debian 的发行版

DEB 安装包由 sbt 官方支持。

Ubuntu 和其他基于 Debian 的发行版使用 DEB 格式，但通常你不从本地的 DEB 文件安装软件。相反，他们由程序包管理器安装，通过命令行（如 `apt-get`，`aptitude`）或图形用户界面（如 Synaptic)。从终端运行下面的命令安装 `sbt`（你需要超级用户权限，因此需要 `sudo`)。

```
echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/sources.l
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.list.d/s
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499
sudo apt-get update
sudo apt-get install sbt
```

软件包管理器将检查若干个提供安装软件包的配置存储库。sbt 二进制文件发布到 Bintray，而 Bintray 方便地提供了 APT 资源库。你只需要将存储库添加到你的软件包管理器将检查的地方。一旦安装了 `sbt`，你会能够在 `aptitude` 或 Synaptic 的包缓存更新后管理了。你也应该能够看到添加的存储库，在底部的 `System Settings -> Software & Updates -> Other Software`:

Ubuntu Software & Updates Screenshot

红帽企业版 Linux 和其他基于 RPM 的发行版

RPM 安装包由 sbt 官方支持。

红帽企业版 Linux 和其他基于 RPM 的发行版使用 RPM 格式。从终端运行下面的命令安装 sbt（你需要超级用户权限，因此需要 sudo）。

```
# remove old Bintray repo file
sudo rm -f /etc/yum.repos.d/bintray-rpm.repo
curl -L https://www.scala-sbt.org/sbt-rpm.repo > sbt-rpm.repo
sudo mv sbt-rpm.repo /etc/yum.repos.d/
sudo yum install sbt
```

On Fedora (31 and above), use `sbt-rpm.repo`:

```
# remove old Bintray repo file
sudo rm -f /etc/yum.repos.d/bintray-rpm.repo
curl -L https://www.scala-sbt.org/sbt-rpm.repo > sbt-rpm.repo
sudo mv sbt-rpm.repo /etc/yum.repos.d/
sudo dnf install sbt
```

> **注意：**请将任何和这两个包相关的问题反馈到 sbt-launcher-package 项目。

Gentoo

在 sbt 官方的树中没有提供 ebuild。但是有从二进制合并 sbt 的 ebuilds。可以通过以下方式从这些 ebuilds 中合并 sbt：

```
emerge dev-java/sbt
```

Hello, World

这一小节假设你已经 安装 sbt 了。

创建一个有源码的项目目录

一个合法的 sbt 项目可以是一个包含单个源码文件的目录。尝试创建一个 `hello` 目录，包含内容如下的源码文件 `hw.scala`：

```
object Hi {
  def main(args: Array[String]) = println("Hi!")
}
```

现在在 `hello` 目录下启动 sbt，然后执行 `run` 命令进入到 sbt 的交互式命令行。在 Linux 或者 OS X 上的命令可能是这样：

```
$ mkdir hello
$ cd hello
$ echo 'object Hi { def main(args: Array[String]) = println("Hi!") }' > hw.scala
$ sbt
...
> run
...
Hi!
```

在这个例子中，sbt 完全按照约定工作。sbt 将会自动找到以下内容：

- 项目根目录下的源文件
- src/main/scala 或 src/main/java 中的源文件
- src/test/scala 或 src/test/java 中的测试文件
- src/main/resources 或 src/test/resources 中的数据文件
- lib 中的 jar 文件

默认情况下，sbt 会用和启动自身相同版本的 Scala 来构建项目。你可以通过执行 sbt run 来运行项目或者通过 sbt console 进入 Scala REPL。sbt console 已经帮你设置好项目的 classpath，所以你可以根据项目的代码尝试实际的 Scala 示例。

构建定义

大多数项目需要一些手动设置。基本的构建设置都放在项目根目录的 build.sbt 文件里。例如，如果你的项目放在 hello 下，在 hello/build.sbt 中可以这样写：

```
lazy val root = (project in file("."))
  .settings(
    name := "hello",
    version := "1.0",
    scalaVersion := "2.12.16"
  )
```

在 .sbt 构建定义 这节中你将会学到更多关于如何编写 build.sbt 脚本的东西。

如果你准备将你的项目打包成一个 jar 包，在 build.sbt 中至少要写上 name 和 version。

设置 sbt 版本

你可以通过创建 hello/project/build.properties 文件强制指定一个版本的 sbt。在这个文件里，编写如下内容来强制使用 1.8.2：

```
sbt.version=1.8.2
```

sbt 在不同的 release 版本中是 99% 兼容的。但是在 project/build.properties 文件中设置 sbt 的版本仍然能避免一些潜在的混淆。

目录结构

这一小节假设你已经 安装 sbt 并且已经阅读过 Hello, World 了。

基础目录

在 sbt 的术语里，“基础目录” 是包含项目的目录。所以，如果你创建了一个和 Hello, World 一样的项目 `hello` ，包含 `hello/build.sbt` 和 `hello/hw.scala`，`hello` 就是基础目录。

源代码

源代码可以像 `hello/hw.scala` 一样的放在项目的基础目录中。然而，大多数人不会在真实的项目中这样做，因为太杂乱了。sbt 和 Maven 的默认的源文件的目录结构是一样的（所有的路径都是相对于基础目录的）：

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    scala-2.12/
       <main Scala 2.12 specific sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    scala-2.12/
       <test Scala 2.12 specific sources>
    java/
       <test Java sources>
```

`src/` 中其他的目录将被忽略。而且，所有的隐藏目录也会被忽略。

sbt 构建定义文件

你已经在项目的基础目录中看到了 `build.sbt`。其他的 sbt 文件在 `project` 子目录中。`project` 目录可以包含 `.scala` 文件，这些文件最后会和 `.sbt` 文件合并共同构成完整的构建定义。想知道更多请参见 组织构建。

```
build.sbt
project/
```

```
Build.scala
```

你可能在 `project/` 中也看到了 `.sbt` 文件，但是它不等同于项目基础目录中的 `.sbt` 文件。这将在 稍后 解释，因为首先你需要一些背景知识。

构建产品

构建出来的文件（编译的 classes，打包的 jars，托管文件，caches 和文档）默认写在 `target` 目录中。

配置版本管理

你的 `.gitignore` 文件（或者其他版本控制系统等同的文件）应该包含：

```
target/
```

注意：这里后面需要跟一个 `/` （只匹配目录）且前面不能有 `/` （除了匹配普通的 `target/` 还匹配 `project/target/` ）。

运行

这一小节将讲述在你建立好项目之后如何去使用 sbt。假设你已经 安装 sbt 并且已经创建过 Hello, World 项目或其他项目。

交互模式

在你的项目目录下运行 sbt 不跟任何参数：

```
$ sbt
```

执行 sbt 不跟任何命令行参数将会进入交互模式。交互模式有一个命令行（含有 tab 自动补全功能和历史记录）。

例如，在 sbt 命令行里输入 `compile`：

```
> compile
```

再次 `compile`，只需要按向上的方向键，然后回车。输入 `run` 来启动程序。输入 `exit` 或者 Ctrl+D （Unix）或者 Ctrl+Z （Windows）可以退出交互模式。

批处理模式

你也可以用批处理模式来运行 sbt，可以以空格为分隔符指定参数。对于接受参数的 sbt 命令，将命令和参数用引号引起来一起传给 sbt。例如：

```
$ sbt clean compile "testOnly TestA TestB"
```

在这个例子中，`testOnly` 有两个参数 `TestA` 和 `TestB`。这个命令会按顺序执行（`clean`，`compile`，然后 `testOnly`）。

持续构建和测试

为了加快编辑-编译-测试循环，你可以让 sbt 在你保存源文件时自动重新编译或者跑测试。在命令前面加上前缀 `~` 后，每当有一个或多个源文件发生变化时就会自动运行该命令。例如，在交互模式下尝试：

```
> ~ compile
```

按回车键停止监视变化。你可以在交互模式或者批处理模式下使用 `~` 前缀。参见 触发执行 获取详细信息。

常用命令

下面是一些非常常用的的 sbt 命令。更加详细的列表请参见 命令行参考。

clean

删除所有生成的文件（在 target 目录下）。

compile

编译源文件（在 src/main/scala 和 src/main/java 目录下）。

test

编译和运行所有测试。

console

进入到一个包含所有编译的文件和所有依赖的 classpath 的 Scala 解析器。输入:quit，Ctrl+D (Unix)，或者 Ctrl+Z （Windows）返回到 sbt。

run < 参数 >*

在和 sbt 所处的同一个虚拟机上执行项目的 main class。

package

将 src/main/resources 下的文件和 src/main/scala 以及 src/main/java 中编译出来的 class 文件打包成一个 jar 文件。

help < 命令 >

显示指定的命令的详细帮助信息。如果没有指定命令，会显示所有命令的简介。

reload

重新加载构建定义（build.sbt，project/*.scala*，*project/*.sbt 这些文件中定义的内容）。在修改了构建定义文件之后需要重新加载。

Tab 自动补全

交互模式下包括空的命令行都有 tab 自动补全。sbt 的一个特别的约定是，当按 tab 键一次的时候可能只会显示所有命令中最有可能的自动补全的子集，当按多次时才会显示详细的选项。

命令历史记录

交互模式有历史记录，即使你退出 sbt 然后重新进入。最简单的访问历史记录的方法时用上方向键。还支持以下一些命令：

!

显示历史记录命令帮助。

!!

重新执行前一条命令。

!:

显示所有之前的命令。

!:n

显示之前的最后 n 条命令。

!n

执行!: 命令显示的结果中下标为 n 的命令。

!-n

执行从该命令往前数第 n 条命令。

!string

执行最近执行过的以 string 打头的命令。

!?string

执行最近执行过的包含 string 的命令。

.sbt 构建定义

这一小节描述 sbt 构建定义，包含一些“理论”和 `build.sbt` 的语法。假设你已经知道如何 使用 sbt 并且阅读过入门指南前面的几小节。

构建定义的二种风格

构建定义有二种风格。

1. 多工程 `.sbt` 构建定义

2. bare .sbt 构建定义

这一小节将讨论最新的多工程.sbt 构建定义，它结合了两种老风格的优点，并且适用于所有情况。当你处理新的构建的时候可能会遇见另外两个老的风格。参见 [bare .sbt 构建定义][Bare-Def] 和.scala 构建定义（在入门指南的后面部分）了解更多其它风格的内容。

此外，构建定义可以包含以.scala 结尾的文件，位于基目录的 project/文件夹下，来定义常用的函数和值。

什么是构建定义?

sbt 在检查项目和处理构建定义文件之后，形成一个 Project 定义。

在 build.sbt 中你可以创建一个本目录的Project工程定义，像这样：

```
lazy val root = (project in file("."))
```

每一个工程对应一个不可变的映射表（immutable map）（一些键值对的集合）来描述工程。

例如，一个叫做 name 的 key，映射到一个字符串的值，即项目的名称。

构建定义文件不会直接影响 *sbt* 的 *map*。

取而代之的是，构建定义会创建一个类型为 Setting[T] 的庞大的对象列表，T 是映射表中值（value）的类型。一个 Setting 描述的是一次 对映射表（*map*）的转换，像增加一个新的键值对或者追加到一个已经存在的 value 上。（在函数式编程关于使用不可变数据结构和值的思想中，一次转换返回一个新的 map —— 它不会就地更新旧的 map。）

你可以为本目录下的项目名称关联一个 Setting[String]，像这样：

```
lazy val root = (project in file("."))
  .settings(
    name := "hello"
  )
```

这个 Setting[String] 会通过增加（或者替换）name 键的值为 "hello" 来对 map 做一次转换。转换后的 map 成为 sbt 新的 map。

为了创建这个 map，sbt 会先对所有设置的列表进行排序，这样对同一个 key 的改变可以放在一起操作，而且如果 value 依赖于其他的 key，会先处理其他被依赖的 key。然后，sbt 会对 Settings 排好序的列表进行遍历，按顺序把每一项都应用到 map 中。

总结：一个构建定义是一个 Project，拥有一个类型为 Setting[T] 的列表，Setting[T] 是会影响到 sbt 保存键值对的 map 的一种转换，T 是每一个 value 的类型。

如何在 build.sbt 中定义设置

build.sbt 定义了一个 Project，它持有一个名为 settings 的 scala 表达式列表。

下面是一个例子：

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.16"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "hello"
  )
```

每一项 Setting 都定义为一个 Scala 表达式。在 settings 中的表达式是相互独立的，而且它们仅仅是表达式，不是完整的 Scala 语句。

这些表达式可以用 val，lazy val，def 声明。build.sbt 不允许使用顶层的 object 和 class。它们必须写到 project/ 目录下作为完整的 Scala 源文件。

在左边，name，version 和 scalaVersion 都是 键（*keys*）。一个键（key）就是一个 SettingKey[T]，TaskKey[T] 或者 InputKey[T] 的实例，T 是期望的 value 的类型。key 的类别将在下面讲解。

键（Keys）有一个返回 Setting[T] 的 := 方法。你可以像使用 Java 的语法一样调用该方法：

```
lazy val root = (project in file("."))
  .settings(
    name.:=("hello")
  )
```

但是，Scala 允许 name := "hello" 这样调用（在 Scala 中，一个只有单个参数的方法可以使用任何一种语法调用）。

键（key）name 上的 := 方法会返回一个 Setting，在这里特指 Setting[String]。String 也出现在 name 自身的类型 SettingKey[String] 中。在这个例子中，返回的 Setting[String] 是一个在 sbt 的 map 中增加或者替换键为 name 的转换，赋值为 "hello"。

如果你使用了错误类型的 value，构建定义会编译不通过：

```
lazy val root = (project in file("."))
  .settings(
    name := 42  // 编译不通过
  )
```

键（Keys）

类型（Types）

有三种类型的 key：

- SettingKey[T]：一个 key 对应一个只计算一次的 value（这个值在加载项目的时候计算，然后一直保存着）。

- `TaskKey[T]`：一个 key 对应一个称之为 *task* 的 value，每次都会重新计算，可能存在潜在的副作用。
- `InputKey[T]`：一个 key 对应一个可以接收命令行参数的 task。详细内容参见 Input Tasks。

内置的 Keys

内置的 keys 实际上是对象 Keys 的字段。`build.sbt` 会隐式包含 `import sbt.Keys._`，所以可以通过 `name` 取到 `sbt.Keys.name`。

自定义 Keys

可以通过它们各自的创建方法：`settingKey`，`taskKey` 和 `inputKey` 创建自定义 keys。每个方法都期待 key 和 value 的类型以及一段描述。key 的名称取自于赋给 `val` 变量的值。例如，给一个新的 task `hello` 定义一个 key，

```
lazy val hello = taskKey[Unit](" 一个 task 示例")
```

这里我们用事实说明了 `.sbt` 文件除了可以包含设置（settings）外，还可以包含 `val`s 和 `def`s。所有这些定义都会在设置（settings）之前被计算而跟它们在文件里定义的位置无关。`val`s 和 `def`s 必须以空行和设置（settings）分隔。

> **注意：**通常，使用 lazy val 而不是 val 可以避免初始化顺序的问题。

Task vs Setting keys

`TaskKey[T]` 是用来定义 *task* 的。Tasks 就是像 `compile` 或者 `package` 这样的操作。它们可能返回 `Unit`（`Unit` 在 Scala 中表示 `void`），或者可能返回 task 相关的返回值，例如 `package` 就是一个类型为 `TaskKey[File]` 的 task，它的返回值是其生成的 jar 文件。

每当你执行一个 task，例如在 sbt 命令行中输入 `compile`，sbt 将会对涉及到的每个 task 恰好执行一次。

sbt 描述项目的 map 会将设置（setting）保存为固定的字符串，比如像 name；但是它不得不保存 task 的可执行代码，比如 `compile` – 即使这段可执行的代码最终返回一个字符串，它也需要每次都重新执行。

一个给定的 *key* 总是指向一个 *task* 或者一个普通的设置（*setting*）。也就是说，“taskiness”（是否每次都重新执行）是 key 的一个属性（property），而不是一个值（value）。

定义 tasks 和 settings

你可以使用 `:=` 给一个 setting 赋一个值或者给一个 task 赋一种计算。对于 setting，这个值（value）只会在项目加载的时候执行一次。对于 task，这个计算会在 task 每次执行的时候重新计算。

例如，实现前面一部分中的 `hello` task：

```
lazy val hello = taskKey[Unit]("An example task")

lazy val root = (project in file("."))
  .settings(
    hello := { println("Hello!") }
  )
```

我们已经在定义项目名称时见过定义 settings 的例子，

```
lazy val root = (project in file("."))
  .settings(
    name := "hello"
  )
```

Tasks 和 Settings 的类型

从类型系统的角度来讲，通过 task key 创建的 `Setting` 和通过 setting key 创建的 `Setting` 有稍微不同。`taskKey := 42` 的类型是 `Setting[Task[T]]` 而 `settingKey := 42` 的类型是 `Setting[T]`。这对于绝大多数情况并无影响；task key 在执行的时候仍然创建一个类型为 `T` 的值（value）。

`T` 类型和 `Task[T]` 类型的不同的含义是：一个 setting 不能依赖一个 task，因为一个 setting 只会在项目加载的时候计算一次，不会重新计算。更多关于设置 的内容很快就会讲到。

sbt 交互模式中的 Keys

在 sbt 的交互模式下，你可以输入任何 task 的 name 来执行该 task。这就是为什么输入 `compile` 就是执行 `compile` task。`compile` 就是该 task 的 key。

如果你输入的是一个 setting key 的 name 而不是一个 task key 的 name，setting key 的值 (value) 会显示出来。输入一个 task key 的 name 会执行该 task 但是不会显示执行结果的值 (value)；输入 `show <task name>` 而不是简单的 `<task name>` 可以看到该 task 的执行结果。对于 key name 的一个约定就是使用 `camelCase`，这样命令行里的 name 和 Scala 的标识符就一样了。

了解更多关于任何 key 内容，可以在 sbt 交互模式的命令行里输入 `inspect <keyname>`。虽然 `inspect` 显示的一些信息没有意义，但是在顶部会显示 setting 的 value 的类型和 setting 的简介。

build.sbt 中的引入

你可以将 `import` 语句放在 `build.sbt` 的顶部；它们可以不用空行分隔。

下面是一些默认的引入：

```
import sbt._
import Keys._
```

(另外，如果你有 .scala 文件，这些文件中任何 Build 对象或者 Plugin 对象里的内容都会被引入。更多关于这些的内容放在 .scala 构建定义。)

bare .sbt 构建定义

bare .sbt 构建定义由一个 Setting[_] 表达式的列表组成，而不是定义 Project。

```
name := "hello"
version := "1.0"
scalaVersion := "2.12.16"
```

添加依赖库

有两种方式添加第三方的依赖。一种是将 jar 文件放入 lib/（非托管的依赖）中，另一种是在 build.sbt 中添加托管的依赖，像这样：

```
val derby = "org.apache.derby" % "derby" % "10.4.1.3"

ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.16"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "hello",
    libraryDependencies += derby
  )
```

就是像这样添加版本为 10.4.1.3 的 Apache Derby 库作为依赖。

key libraryDependencies 包含两个方面的复杂性：+= 方法而不是 :=，第二个就是 % 方法。+= 方法是将新的值追加该 key 的旧值后面而不是替换它，这将在 更多设置 中介绍。% 方法是用来从字符串构造 Ivy 模块 ID 的，将在 库依赖 中介绍。

目前，一直到入门指南的后面部分，我们跳过了库依赖的一些细节。后面有一整节 库依赖 来介绍这些内容。

任务图

> This page was translated mostly with Google Translate. Please send a pull request to improve it.

继.sbt 构建定义，此页面更详细地解释了 build.sbt 定义。

与其将 settings 视为键值对，不如将其更好地比喻为将任务表示为边 happens-before 的任务的有向无环图 (DAG)。我们将此称为**任务图** (task graph)。

术语

在深入探讨之前，让我们先回顾一下关键术语。

- setting/task 式: `.settings(...)` 条目。
- key: setting 式的左侧。它可以是 `SettingKey[A]`,`TaskKey[A]` 或 `InputKey[A]`。
- setting: 由带有 `SettingKey[A]` 的 setting 式定义。该值在加载期间仅计算一次。
- task: 由带有 `TaskKey[A]` 的 task 式定义。每次调用时都会计算该值。

声明对其他任务的依赖

在 `build.sbt` DSL 中，我们使用 `.value` method 来表示对另一个任务或 setting 的依赖性。value method 是特殊的，只能在 `:=` 的参数中调用（或 `+=` 或 `++=` 我们将在后面介绍）。

作为第一个示例，请考虑定义依赖于 `update` 和 `clean` 任务的 `scalacOption`。这些是这些 key 的定义（来自 Keys）。

注意：下面计算的值对于 `scalaOptions` 是毫无 `scalaOptions`，仅用于演示目的：

```
val scalacOptions = taskKey[Seq[String]]("Options for the Scala compiler.")
val update = taskKey[UpdateReport]("Resolves and optionally retrieves dependencies, producir
val clean = taskKey[Unit]("Deletes files produced by the build, such as generated sources, c
```

这是我们如何重新连接 `scalacOptions`:

```
scalacOptions := {
  val ur = update.value  // update task happens-before scalacOptions
  val x = clean.value    // clean task happens-before scalacOptions
  // ---- scalacOptions begins here ----
  ur.allConfigurations.take(3)
}
```

`update.value` 和 `clean.value` 声明了任务依赖性,而 `ur.allConfigurations.take(3)` 是任务的主体。

`.value` 不是正常的 Scala method 调用。`build.sbt` DSL 使用宏将它们提升到任务主体之外。**在任务引擎评估 `scalacOptions` 的打开 `{`，无论它出现在主体中的哪一行，`update` 和 `clean` 任务都已完成**。

请参见以下示例:

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.16"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalacOptions := {
      val out = streams.value // streams task happens-before scalacOptions
```

```
      val log = out.log
      log.info("123")
      val ur = update.value   // update task happens-before scalacOptions
      log.info("456")
      ur.allConfigurations.take(3)
    }
  )
```

接下来，在 sbt shell 中键入 scalacOptions:

```
> scalacOptions
[info] Updating {file:/xxx/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[info] 123
[info] 456
[success] Total time: 0 s, completed Jan 2, 2017 10:38:24 PM
```

即使 val ur = ... 出现在 log.info("123") 和 log.info("456")，update 任务的评估还是要先于它们进行。

这是另一个例子:

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.16"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalacOptions := {
      val ur = update.value  // update task happens-before scalacOptions
      if (false) {
        val x = clean.value  // clean task happens-before scalacOptions
      }
      ur.allConfigurations.take(3)
    }
  )
```

接下来，在 sbt shell 中键入 run，然后键入 scalacOptions。

```
> run
[info] Updating {file:/xxx/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[info] Compiling 1 Scala source to /Users/eugene/work/quick-test/task-graph/target/scala-2.1
[info] Running example.Hello
hello
[success] Total time: 0 s, completed Jan 2, 2017 10:45:19 PM
```

```
> scalacOptions
[info] Updating {file:/xxx/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[success] Total time: 0 s, completed Jan 2, 2017 10:45:23 PM
```

现在，如果您检查 target/scala-2.12/classes/，它将不存在，因为即使它在 if (false) 内，clean 任务也已运行。

需要注意的另一件事是，不能保证 update 和 clean 任务的顺序。他们可能同时运行 update 然后 clean，clean 然后 update 或同时运行。

内联.value 调用

如上所述，.value 是一种特殊的 method，用于表达对其他任务和 setting 的依赖性。在您熟悉 build.sbt 之前，我们建议您将所有 .value 调用放在任务正文的顶部。

但是，当您变得更加舒适时，您可能希望内联 .value 调用，因为它可以使 task/setting 更简洁，并且不必提供变量名。

我们内联了一些示例：

```
scalacOptions := {
  val x = clean.value
  update.value.allConfigurations.take(3)
}
```

请注意，.value 调用是内联的还是放在任务正文中的任何位置，在进入任务正文之前仍会对它们进行评估。

检查任务

在上面的示例中，scalacOptions 对 update 和 clean 任务具有**依赖性**。如果将以上内容放置在 build.sbt 并运行 sbt shell，则键入 inspect scalacOptions，您应该看到（部分）：

```
> inspect scalacOptions
[info] Task: scala.collection.Seq[java.lang.String]
[info] Description:
[info]  Options for the Scala compiler.
....
[info] Dependencies:
[info]  *:clean
[info]  *:update
....
```

这就是 sbt 如何知道哪些任务取决于哪些其他任务的方式。

例如，如果您 `inspect tree compile` 您将看到它依赖于另一个 key `incCompileSetup`，而后者又依赖于其他 key，如 `dependencyClasspath`。继续遵循依赖性链，魔术就会发生。

```
> inspect tree compile
[info] compile:compile = Task[sbt.inc.Analysis]
[info]   +-compile:incCompileSetup = Task[sbt.Compiler$IncSetup]
[info]   | +-*/*:skip = Task[Boolean]
[info]   | +-compile:compileAnalysisFilename = Task[java.lang.String]
[info]   | | +-*/*:crossPaths = true
[info]   | | +-{.}/*:scalaBinaryVersion = 2.12
[info]   | |
[info]   | +-*/*:compilerCache = Task[xsbti.compile.GlobalsCache]
[info]   | +-*/*:definesClass = Task[scala.Function1[java.io.File, scala.Function1[java.lang
[info]   | +-compile:dependencyClasspath = Task[scala.collection.Seq[sbt.Attributed[java.io.
[info]   | | +-compile:dependencyClasspath::streams = Task[sbt.std.TaskStreams[sbt.Init$Scop
[info]   | | | +-*/*:streamsManager = Task[sbt.std.Streams[sbt.Init$ScopedKey[_ <: Any]]]
[info]   | | |
[info]   | | +-compile:externalDependencyClasspath = Task[scala.collection.Seq[sbt.Attribute
[info]   | | | +-compile:externalDependencyClasspath::streams = Task[sbt.std.TaskStreams[sbt
[info]   | | | | +-*/*:streamsManager = Task[sbt.std.Streams[sbt.Init$ScopedKey[_ <: Any]]]
[info]   | | | |
[info]   | | | +-compile:managedClasspath = Task[scala.collection.Seq[sbt.Attributed[java.io
[info]   | | | | +-compile:classpathConfiguration = Task[sbt.Configuration]
[info]   | | | | | +-compile:configuration = compile
[info]   | | | | | +-*/*:internalConfigurationMap = <function1>
[info]   | | | | | +-*:update = Task[sbt.UpdateReport]
[info]   | | | | |
....
```

例如，当您键入 `compile` sbt 时，它会自动执行 `update`。它之所以行之有效，是因为作为 `compile` 计算的输入所需的值需要 sbt 首先进行 `update` 计算。

这样，sbt 中的所有构建依赖项都是**自动的**，而不是显式声明的。如果在另一个计算中使用 key 的值，则该计算取决于该 key。

定义依赖于其他 setting 的任务

`scalacOptions` 是 task key。假设已经将其设置为某些值，但是您想为非 2.12 过滤掉 `"-Xfatal-warnings"` 和 `"-deprecation"`。

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    organization := "com.example",
    scalaVersion := "2.12.16",
    version := "0.1.0-SNAPSHOT",
    scalacOptions := List("-encoding", "utf8", "-Xfatal-warnings", "-deprecation", "-uncheck
```

```
    scalacOptions := {
      val old = scalacOptions.value
      scalaBinaryVersion.value match {
        case "2.12" => old
        case _      => old filterNot (Set("-Xfatal-warnings", "-deprecation").apply)
      }
    }
  )
```

这是它在 sbt shell 上的外观：

```
> show scalacOptions
[info] * -encoding
[info] * utf8
[info] * -Xfatal-warnings
[info] * -deprecation
[info] * -unchecked
[success] Total time: 0 s, completed Jan 2, 2017 11:44:44 PM
> ++2.11.8!
[info] Forcing Scala version to 2.11.8 on all projects.
[info] Reapplying settings...
[info] Set current project to Hello (in build file:/xxx/)
> show scalacOptions
[info] * -encoding
[info] * utf8
[info] * -unchecked
[success] Total time: 0 s, completed Jan 2, 2017 11:44:51 PM
```

接下来，使用这两个 key (来自 Keys)：

```
val scalacOptions = taskKey[Seq[String]]("Options for the Scala compiler.")
val checksums = settingKey[Seq[String]]("The list of checksums to generate and to verify fo
```

注意：scalacOptions 和 checksums 彼此无关。它们只是两个具有相同值类型的键，其中一个是一项任务。

可以编译一个将 build.sbt 别名为 checksums scalacOptions，但不能以其他方式编译。例如，这是允许的：

```
// The scalacOptions task may be defined in terms of the checksums setting
scalacOptions := checksums.value
```

没有**其他**方向可以走。也就是说，setting key 不能依赖于 task key。这是因为 setting key 仅在 subproject 加载时计算一次，因此该任务不会每次都重新运行，并且任务希望每次都重新运行。

```
// Bad example: The checksums setting cannot be defined in terms of the scalacOptions task!
checksums := scalacOptions.value
```

定义取决于其他 setting 的 setting

在执行时间方面，我们可以将 setting 视为在加载期间评估的特殊任务。

考虑将 subproject 组织定义为与项目名称相同。

```
// name our organization after our project (both are SettingKey[String])
organization := name.value
```

Here's a realistic example. This rewires `Compile / scalaSource` key to a different directory only when `scalaBinaryVersion` is `"2.11"`.

```
Compile / scalaSource := {
  val old = (Compile / scalaSource).value
  scalaBinaryVersion.value match {
    case "2.11" => baseDirectory.value / "src-2.11" / "main" / "scala"
    case _      => old
  }
}
```

build.sbt DSL 的意义是什么?

build.sbt DSL 是一种领域特定语言，用于构建设置和任务的 DAG。setting 式对 setting，任务及其之间的依赖关系进行编码。

这种结构在 Make (1976)，Ant (2000)，和 Rake (2003) 中很常见。

Make 简介

基本的 Makefile 语法如下所示：

```
target: dependencies
[tab] system command1
[tab] system command2
```

给定一个目标（默认目标名为 `all`），

1. Make 检查目标的依赖项是否已构建，并构建尚未构建的任何依赖项。
2. Make 按顺序运行系统命令。

让我们看一下 `Makefile`：

```
CC=g++
CFLAGS=-Wall

all: hello

hello: main.o hello.o
    $(CC) main.o hello.o -o hello
```

```
%.o: %.cpp
    $(CC) $(CFLAGS) -c $< -o $@
```

运行 make，默认情况下它将选择名为 all 的目标。目标将 hello 作为其依赖项列出，但尚未建立，因此 Make 将建立 hello 。

接下来，Make 检查是否已经建立了 hello 目标的依赖关系。hello 列出了两个目标：main.o 和 hello.o。一旦使用最后一个模式匹配规则创建了这些目标，只有执行系统命令，才能将 main.o 和 hello.o 链接到 hello。

如果您只是运行 make，则可以专注于作为目标的目标，并且 Make 会确定构建中间产品所需的确切时间和命令。我们可以将其视为面向依赖的编程或基于 flow-based 编程。Make 实际上被认为是混合系统，因为虽然 DSL 描述了任务相关性，但操作被委派给系统命令。

Rake

对于 Make 后继者（例如 Ant，Rake 和 sbt），这种混合状态仍在继续。看一下 Rakefile 的基本语法：

```
task name: [:prereq1, :prereq2] do |t|
  # actions (may reference prereq as t.name etc)
end
```

Rake 的突破之处在于它使用一种编程语言来描述操作而不是系统命令。

基于混合 flow-based 编程的好处

以这种方式组织构建有多种动机。

首先是重复数据删除。使用基于 flow-based 编程，即使一个任务由多个任务依赖，它也只能执行一次。例如，即使沿着任务图的多个任务依赖 Compile / compile 也将只执行一次。

第二是并行处理。使用任务图，任务引擎可以并行调度互不相关的任务。

第三是关注点和灵活性的分离。任务图使构建用户可以以不同的方式将任务连接在一起，而 sbt 和插件可以提供各种功能（例如，编译和库依赖管理）作为可重复使用的功能。

摘要

构建定义的核心数据结构是任务的 DAG，其中边缘表示 happens-before 关系。build.sbt 是一种 DSL，旨在表达面向依赖的程序或基于 flow-based 程序，类似于 Makefile 和 Rakefile。

基于 flow-based 编程的主要动机是重复数据删除，并行处理和可定制性。

Scope

这一小节介绍 scope。假设你已经阅读并且理解了前一小节 .sbt 构建定义。

关于 Key 的所有故事

前一小节 我们认为像 `name` 的一个 key 相当于 sbt 保存键值对的 map 中的一条记录，这只是一种简化。

事实上，每一个 key 可以在多个上下文中关联一个值，每个上下文称之为 "scope"。

一些具体的例子：

- 如果在你的构建定义中有多个项目，在每个项目中同一个 key 可以有不同的值。
- 如果你想根据不同的情形编译它们，key `compile` 对于 main 源文件和 test 源文件可以有不同的值。
- Key `packageOptions`（包含创建 jar 包的一些选项）可以有不同的值，在对 class 文件打包时是 `packageBin`，对源代码打包时是 `packageSrc`。

给定的 *key* ***name*** 没有单一的值，因为在不同的 scope 下它的值不同。

然而，给定的 `scoped` key 的值是单一的。

如果你想起 前面 我们讨论过的，sbt 生成一个 map 来处理描述项目的 settings 列表，这个 map 中的 key 就是 *scope* 下的 key。构建定义中定义的每一个 setting（例如在 `build.sbt` 中）都有一个 scope 下的 key。

一般 scope 是隐式的或者是默认的，但是一旦默认的是错误的，你就需要在 `build.sbt` 中指定你期待的 scope。

Scope 轴

Scope 轴是一种类型，该类型的每个实例都能定义自己的 scope（也就是说，每个实例的 key 可以有自己唯一的值）。

有三种类型的 scope 轴：

- Projects
- Configurations
- Tasks

通过 Project 轴划分 Scope

如果你将 多个项目 放在同一个构建中，每个项目需要有属于自己的 settings。也就是说，keys 可以根据项目被局限在不同的上下文中。

Project 轴可以设置成构建全局的，因此一个 setting 可以应用到全局构建而不是单个项目。当一个项目没有定义项目层级的 setting 的时候，构建层级的 setting 通常作为默认的设置。

通过 Configuration 轴划分 Scope

一个 *configuration* 定义一种特定的构建，可能包含它自己的 classpath，源文件和生成的包等。Configuration 的概念来自于它用来管理 库依赖 的 Ivy 和 MavenScopes。

在 sbt 中你可以看到这些 configurations：

- `Compile` 定义主构建（`src/main/scala`）
- `Test` 定义如何构建测试（`src/test/scala`）
- `Runtime` 为 task `run` 定义 classpath

默认情况下，和编译、打包、运行相关的所有 key 都局限于一个 configuration，因此在不同的 configuration 中可能产生不同的效果。最明显的例子就是 task key：`compile`，`package` 和 `run`；然而能够 影响到这些 key 的所有其他 key（例如 `sourceDirectories`，`scalacOptions` 和 `fullClasspath`）也都局限于该 configuration。

通过 Task 轴划分 Scope

Settings 可以影响一个 task 如何工作。例如，task `packageSrc` 就会被 setting `packageOptions` 影响。

为了支持这种特性，一个 task key（例如 `packageSrc`）可以作为另一个 key（例如 `packageOptions`）的 scope。

一些和打包相关的 task（`packageSrc`，`packageBin`，`packageDoc`）可以共享和打包相关的 key，例如 `artifactName` 和 `packageOptions`。这些 key 对于每个打包的 task 可以有唯一的值。

全局 Scope

每一种 scope 轴都可以用和该轴类型一致的实例代替（例如 task 轴可以用一个 task 代替），或者该轴可以被特定的值 `Global` 代替。

`Global` 的意义就是你所期待的：将 setting 的值应用到该轴所有的实例上。例如如果一个 task 轴的值是 `Global`，那么该 setting 的值将被应用到所有的 task 上。

代理

如果在一个 scope 中某一个 key 没有关联的值，那么该 key 就是未定义的。

对于每一个 scope，sbt 有由其他 scope 构成的替代选项的搜索路径。通常，如果一个 key 在特定的 scope 下没有关联的值，sbt 会尝试从更加一般的 scope（例如 `Global` scope 或者构建全局 scope）中获取值。

这个特性允许你一旦在更加一般的 scope 中设置了某一项设置的值之后，使得多个特定的 scope 能够继承该值。

你可以像下面这样用 `inspect` 命令查看一个 key 的替代选项的查找路径或者“代理”。往下看。

运行 sbt 时涉及 scope 下的 key

在命令行的交互模式下，sbt 像这样显示（和解析）scope 下的 keys：

{<build-uri>}<project-id>/config:intask::key

- {<build-uri>}/<project-id> 标识 project 轴。如果 project 轴有构建全局 scope，将没有 <project-id> 部分。
- config 标识 configuration 轴。
- intask 标识 task 轴。
- key 标识 scope 下的 key。

“*” 号可以出现在任意轴，参考 Global scope。

如果你省略部分 scoped key，它会像下面这样推断：

- 如果省略 project，当前的 project 会被使用。
- 如果省略 configuration 或者 task，会自动检测 key 所有依赖的 configuration。

更多详细内容，请参见 与 Configuration 系统交互。

使用 scoped key 标识的例子

- fullClasspath 仅仅指定了一个 key，所以会使用默认的 scope：当前的 project，key 所依赖的 configuration 和全局 task 的 scope。
- Test/fullClasspath 指定为 configuration，所以这个 fullClasspath 就在 test configuration scope 下，其他两个 scope 轴均为默认值。
- *:fullClasspath 将 configuration 指定为 Global，而不是默认的 configuration。
- doc::fullClasspath 将 key fullClasspath 局限在 doc task 下，project 轴和 configuration 轴还是默认的。
- {file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath 指定了一个 project,在 {file:/home/hp/checkout/hello/}default-aea33a 中，{file:/home/hp/checkout/hello/} 标识 project，而且 project id 在 default-aea33a 构建中。也指定了 configuration 为 test，但是将 task 轴留为默认的。
- {file:/home/hp/checkout/hello/}/test:fullClasspath 将构建为 {file:/home/hp/checkout/hello/} 的 project 轴设置为全局构建。
- {.}/test:fullClasspath 将构建为 {.} 的 project 轴设置为全局构建。{.} 可以在 Scala 代码中写成 ThisBuild。
- {file:/home/hp/checkout/hello/}/compile:doc::fullClasspath 设置了全部的三个 scope 轴。

审查 scope

在 sbt 的交互模式下，你可以使用 inspect 命令来理解 key 和它对应的 scope。尝试 inspect Test/fullClasspath，

```
$ sbt
> inspect Test/fullClasspath
[info] Task: scala.collection.Seq[sbt.Attributed[java.io.File]]
```

```
[info] Description:
[info]  The exported classpath, consisting of build products and unmanaged and managed, inte
[info] Provided by:
[info]  {file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath
[info] Dependencies:
[info]  test:exportedProducts
[info]  test:dependencyClasspath
[info] Reverse dependencies:
[info]  test:runMain
[info]  test:run
[info]  test:testLoader
[info]  test:console
[info] Delegates:
[info]  test:fullClasspath
[info]  runtime:fullClasspath
[info]  compile:fullClasspath
[info]  *:fullClasspath
[info]  {.}/test:fullClasspath
[info]  {.}/runtime:fullClasspath
[info]  {.}/compile:fullClasspath
[info]  {.}/*:fullClasspath
[info]  */test:fullClasspath
[info]  */runtime:fullClasspath
[info]  */compile:fullClasspath
[info]  */*:fullClasspath
[info] Related:
[info]  compile:fullClasspath
[info]  compile:fullClasspath(for doc)
[info]  test:fullClasspath(for doc)
[info]  runtime:fullClasspath
```

在第一行，你可以看到这是一个 task（和 .sbt 构建定义 中介绍的 setting 相对)。该 task 得到的值的类型为 `scala.collection.Seq[sbt.Attributed[java.io.File]]`。

“Provided by” 表明定义该值的 scoped key,在这个例子中是 `{file:/home/hp/checkout/hello/}default-aea33a/tes` (`fullClasspath` 在 `test` configuration 中且在 `{file:/home/hp/checkout/hello/}default-aea33a` project 下)。

“Dependencies” 可能没有意义；请继续阅读 下一节。

你也可以看到一些代理；如果没有定义，sbt 会通过以下途径查找：

- 其他两个 configuration(`Runtime/fullClasspath` 和 `Compile/fullClasspath`)。在这些 scoped key 中，project 没有指定的话就意味着是“当前 project” 而且 task 没有指定的话就意味着是 `Global`。
- 当 project 没有指定“当前 project” 并且 task 没有指定为 `Global` 时，configuration 会被设置成 `Global` (`*:fullClasspath`)。
- 当全局构建中没有指定特定的 project 时，project 会被设置成 `{.}` 或者 `ThisBuild`。

- 将 project 轴设置成 Global（*/test:fullClasspath）（记住，不指定 project 表示用当前的 current，所以这里查找 Global 是一个新的方式；例如：* 和“显示没有 project”对于 project 轴是不一样的；例如：*/test:fullClasspath 和 test:fullClasspath 不是一回事）。
- project 轴和 configuration 轴都会被设置成 Global（*/*:fullClasspath）（还记得我们已经说过不指定 task 表示用 Global，所以 */*:fullClasspath 表示三个轴都用 Global）。

尝试用 inspect fullClasspath（和上面例子中的 inspect test:fullClasspath 相对）来查看它们的不同。因为 configuration 被省略了，sbt 自动检测并设置为 compile。因此 inspect Compile/fullClasspath 得到的结果看起来应该和 inspect fullClasspath 得到的结果一样。

尝试用 inspect *:fullClasspath 作为对比。默认情况下，fullClasspath 没有定义在 Global configuration 中。

更多详细的内容请参见 与 Configuration 系统交互。

在构建定义中涉及 scope

如果你创建的 build.sbt 中有一个 bare key，它的作用于将是当前的 project 下，configuration 和 task 均为 Global：

```
lazy val root = (project in file("."))
  .settings(
    name := "hello"
  )
```

启动 sbt 并且执行 inspect name 可以看到它由 {file:/home/hp/checkout/hello/}default-aea33a/*:name 提供，也就是说，project 是 {file:/home/hp/checkout/hello/}default-aea33a，configuration 是 *（表示全局），task 没有显示出来（实际上也是全局）。

Keys 会调用一个重载的 in 方法设置 scope。传给 in 方法的参数可以是任何 scope 轴的实例。比如说，你可以将 name 局限在 Compile configuration 中，尽管没有真实的理由要这样做：

```
Compile / name := "hello"
```

或者你可以把 name 局限在 packageBin task 中（没有什么意义！仅仅是个例子）：

```
name in packageBin := "hello"
```

或者你可以把 name 局限在多个 scope 轴中，例如在 Compile configuration 的 packageBin task 中：

```
name in (Compile, packageBin) := "hello"
```

或者你可以用 Global 表示所有的轴：

```
name in Global := "hello"
```

(`name in Global` 隐式的把 scope 轴的值 `Global` 转换为 scope 所有轴的值均为 `Global`; task 和 configuration 默认是 `Global`,因此这里的效果是将 project 设置成 `Global`,也就是说,定义了 `*/*:name` 而不是 `{file:/home/hp/checkout/hello/}default-aea33a/*:name`)

如果你之前不熟悉 Scala，提醒一下：in 和 `:=` 仅仅是方法，不是魔法，理解这点很重要。Scala 让你用一种更好的方式编写它们，但是你也可以用 Java 的风格：

```
name.in(Compile).:=("hello")
```

毫无理由使用这种丑陋的语法，但是它阐明这实际上是方法。

指定 scope 的时机

如果一个 key 通常的作用域有问题，你需要指定 scope。例如，`compile` task 默认是在 `Compile` 和 `Test` configuration 的 scope 中，而且在这些 scope 之外它并不存在。

为了改变 key `compile` 的值，你需要写成 `Compile / compile` 或者 `Test / compile`。用普通的 `compile` 会在当前 project 的 scope 中定义一个新的 task，而不是覆盖 configuration 的 scope 标准的 `compile` task。

如果你遇到像“引用未定义的设置”这样的错误，通常是你指定 scope 失败了，或者你指定了一个错误的 scope。你使用的 key 可能定义在其他的 scope 中。sbt 会尝试在错误消息里面提示你的想法是什么；如“你是指 Compile/compile? ”

一种方式是你可以这样认为,name 只是 key 的一部分。实际上,所有的 key 都有 name 和 scope 组成 (scope 有三个轴)。换句话说，`packageOptions in (Compile, packageBin)` 是表示 key name 的完整的表达式。其简写 `packageOptions` 也是一个 key name，但是是不同的（对于没有 in 方法的 key，会隐式的假设一个 scope：当前的 project，global config，global task)。

追加值

追加值：`+=` 和 `++=`

通过 `:=` 方法赋值是最简单的转换，但是 key 也有很多其他的方法。如果 `SettingKey[T]` 中的 `T` 是一个列表，例如，一个 key 的值的类型是 sequence，你就可以往列表中追加而不是替换。

- `+=` 会追加单个元素到列表中。
- `++=` 会连接两个列表。

例如，一个 key `Compile / sourceDirectories` 的值是 `Seq[File]`。默认情况下该 key 的值会包含 `src/main/scala`。如果你也想编译叫做 source 的目录下的源代码（因为你不得不成为非标准的)，你可以添加该目录：

```
Compile / sourceDirectories += new File("source")
```

或者，遵循约定使用 sbt 包中的 `file()` 函数：

```
Compile / sourceDirectories += file("source")
```

(file() 只是创建了一个新的 File。)

你可以用 ++= 一次添加多个目录:

```
Compile / sourceDirectories ++= Seq(file("sources1"), file("sources2"))
```

Seq(a, b, c, ...) 是 Scala 用来构建列表的标准语法。

要完全替换默认的 source 目录，当然可以使用 := 方法:

```
Compile / sourceDirectories := Seq(file("sources1"), file("sources2"))
```

当设置未定义时

无论何时一个设置用 :=，+= 或者 ++= 时依赖于自己或者另一个 key 的值，它依赖的值必须存在。如果不存在，sbt 就会抱怨。例如，它可能会说“引用了未定义的设置”。当这发生时，确认一下你使用的 key 在 scope 中并且已经定义了。

在 sbt 中创建循环引用是可能的，这是错误的；如果你循环引用了，sbt 会告诉你。

依赖于其他 key 的值的 task

你可以计算一些 task 或者 setting 的值来定义另一个 task 或者为另一个 task 追加值。通过使用 Def.task 作为:=，+= 或者 ++= 的参数可以做到。

作为第一个例子，考虑追加一个使用项目基目录和编译 classpath 的 source generator。

```
Compile / sourceGenerators += Def.task {
  myGenerator(baseDirectory.value, (Compile / managedClasspath).value)
}
```

追加依赖：+= 和 ++=

当追加到一个已经存在的 setting 或者 task 时可以使用另一些 key，就像它们可以通过 := 赋值一样。例如，比方说你有一个以项目名称命名的覆盖率报告，而且你想在每次清除文件的时候都清除它:

```
cleanFiles += file("coverage-report-" + name.value + ".txt")
```

Scope 委托 (.value 查找)

> This page was translated mostly with Google Translate. Please send a pull request to improve it.

此页面描述 scope 委托。假定您已经阅读并理解了先前的页面 .sbt 构建定义 和 scopes。

既然我们已经涵盖了 scope 界定的所有细节，我们就可以详细解释 .value 查找。如果您是第一次阅读此页面，则可以跳过本节。

总结到目前为止我们已经学到的东西:

- scope 是三个轴上的组件的元组: subproject 轴、configuration 轴、task 轴。
- 任何 scope 轴都有一个特殊的 scope 组件 `Zero`。
- 在 subproject 轴上有一个特殊的 scope 组件 `ThisBuild`。
- `Test` 扩展了 `Runtime`，而 `Runtime` 扩展了 `Compile` configuration。
- 默认情况下，放置在 build.sbt 中的 key 的 scope 为 `${current subproject} / Zero / Zero`。
- 可以使用 `/` 运算符确定 key 的 scope。

现在，假设我们具有以下构建定义:

```
lazy val foo = settingKey[Int]("")
lazy val bar = settingKey[Int]("")

lazy val projX = (project in file("x"))
  .settings(
    foo := {
      (Test / bar).value + 1
    },
    Compile / bar := 1
  )
```

在 `foo` 的 setting 主体内部，声明了对 scoped key `Test / bar` 的依赖。但是，尽管在 `projX` 中未定义 `Test / bar`，sbt 仍然能够将 `Test / bar` 解析为另一个 scoped key，导致 `foo` 初始化为 2。

sbt 具有定义明确的后备搜索路径，称为 **scope 委托**。此功能使您可以在更广泛的 scope 内设置一次值，从而允许多个更特定的 scope 继承该值。

scope 委托规则

以下是 scope 委托的规则:

- 规则 1: scope 轴具有以下优先级: subproject 轴，configuration 轴，然后是 task 轴。
- 规则 2: 在给定 scope 的情况下，可以通过按以下顺序替换 task 轴来搜索委托 scope: 给定的 task scope，然后是 `Zero` （这是 scope 的非 task scope 版本）。
- 规则 3: 在给定 scope 的情况下，可以通过按以下顺序替换 configuration 轴来搜索委托 scope: 给定 configuration，其父项，其父项等等，然后 `Zero`（与无作用域的 configuration 轴相同）。
- 规则 4: 给定一个 scope ，通过按以下顺序替换 subproject 轴来搜索委托 scope: 给定的 subproject，`ThisBuild`，然后为 `Zero`。
- 规则 5: 在不携带原始上下文的情况下，评估委托 scoped key 及其相关的 settings/tasks。

我们将在本页面的其余部分中查看每个规则。

规则 1: scope 轴优先级

- 规则 1: scope 轴具有以下优先级: subproject 轴，configuration 轴，然后是 task 轴。

换句话说，给定两个作用域候选者，如果一个在 subproject 轴上具有更特定的值，则无论 configuration 或 task scope 如何，它将始终获胜。同样，如果 subproject 相同，则无论 task scope 如何，具有更具体 configuration 值的子项目将始终获胜。我们将看到更多定义**更具体的**规则。

规则 2: task 轴委托

- 规则 2: 在给定 scope 的情况下，可以通过按以下顺序**替换** task 轴来搜索委托 scope: 给定的 task scope，然后是 `Zero` (这是 scope 的非 task scope 版本)。

对于给定 key，sbt 将如何生成委托 scope，这里有一个具体规则。记住，我们试图显示给定任意 `(xxx / yyy).value` 的搜索路径。

练习题 A: 给出以下构建定义:

```
lazy val projA = (project in file("a"))
  .settings(
    name := {
      "foo-" + (packageBin / scalaVersion).value
    },
    scalaVersion := "2.11.11"
  )
```

`projA / name` 的值是什么?

1. `"foo-2.11.11"`
2. `"foo-2.12.16"`
3. 还有什么吗

答案是 `"foo-2.11.11"`。在 `.settings(...)` 内部，`scalaVersion` 的 scope 将自动设置为 `projA / Zero / Zero`，因此 `packageBin / scalaVersion` 变为 `projA / Zero / packageBin / scalaVersion`。该特定 scoped key 是未定义的。通过使用规则 2，sbt 将把 task 轴替换 `Zero` 作为 `projA / Zero / Zero` (或 `projA / scalaVersion`)。该 scoped key 定义为 `"2.11.11"`。

规则 3: configuration 轴搜索路径

- 规则 3: 在给定 scope 的情况下，可以通过按以下顺序替换 configuration 轴来搜索委托 scope: 给定 configuration，其父项，其父项等等，然后 `Zero` (与无作用域的 configuration 轴相同)。

我们前面看到的例子是 `projX`:

```
lazy val foo = settingKey[Int]("")
lazy val bar = settingKey[Int]("")

lazy val projX = (project in file("x"))
  .settings(
    foo := {
      (Test / bar).value + 1
    },
    Compile / bar := 1
  )
```

如果我们再次写出完整 scope，projX / Test / Zero 。还记得 Test 扩展了 Runtime ，Runtime 扩展了 Compile 。

Test / bar 是未定义的，但是由于规则 3，sbt 将查找 scope 为 projX / Test / Zero，projX / Runtime / Zero，然后 projX / Compile / Zero。找到最后一个，即 Compile / bar。

规则 4: subproject 轴搜索路径

- 规则 4: 给定一个 scope ，通过按以下顺序替换 subproject 轴来搜索委托 scope: 给定的 subproject，ThisBuild，然后为 Zero。

练习题 B: 给出以下构建定义:

```
ThisBuild / organization := "com.example"

lazy val projB = (project in file("b"))
  .settings(
    name := "abc-" + organization.value,
    organization := "org.tempuri"
  )
```

projB / name 的值是什么?

1. "abc-com.example"
2. "abc-org.tempuri"
3. 还有什么吗

答案是 abc-org.tempuri。因此，根据规则 4，第一个搜索路径是具有 projB / Zero / Zero scope 的 organization，在 projB 中定义为 "org.tempuri"。它的优先级高于构建级别 setting ThisBuild / organization。

scope 轴优先级再次

练习题 C: 给出以下构建定义:

```
ThisBuild / packageBin / scalaVersion := "2.12.2"
```

```
lazy val projC = (project in file("c"))
  .settings(
    name := {
      "foo-" + (packageBin / scalaVersion).value
    },
    scalaVersion := "2.11.11"
  )
```

projC / name 值是什么?

1. "foo-2.12.2"
2. "foo-2.11.11"
3. 还有什么吗

答案是 foo-2.11.11。scope 为 projC / Zero / packageBin 的 scalaVersion 未定义。

scalaVersion scoped to projC / Zero / packageBin is undefined. 规则 2 找到 projC / Zero / Zero。规则 4 找到 ThisBuild / Zero / packageBin。在这种情况下，规则 1 决定在 subproject 轴上赢得更具体的价值，这是定义为 "2.11.11" 的 projC / Zero / Zero。

练习题 D: 给出以下构建定义:

```
ThisBuild / scalacOptions += "-Ywarn-unused-import"

lazy val projD = (project in file("d"))
  .settings(
    test := {
      println((Compile / console / scalacOptions).value)
    },
    console / scalacOptions -= "-Ywarn-unused-import",
    Compile / scalacOptions := scalacOptions.value // added by sbt
  )
```

如果您进行了 projD/test 您会看到什么?

1. List()
2. List(-Ywarn-unused-import)
3. 还有什么吗

答案是 List(-Ywarn-unused-import)。规则 2 找到 projD / Compile / Zero,规则 3 找到 projD / Zero / console，规则 4 找到 ThisBuild / Zero / Zero。规则 1 选择 projD / Compile / Zero 因为它具有 subproject 轴 projD，并且 configuration 轴的优先级高于 task 轴。

接下来，Compile / scalacOptions 引用 scalacOptions.value，我们接下来需要找到 projD / Zero / Zero 的委托。规则 4 找到 ThisBuild / Zero / Zero，然后解析为 List(-Ywarn-unused-import)。

inspect 命令列出委托

您可能需要快速查找正在发生的事情。这是可以使用 inspect 地方。

```
sbt:projd> inspect projD / Compile / console / scalacOptions
[info] Task: scala.collection.Seq[java.lang.String]
[info] Description:
[info]  Options for the Scala compiler.
[info] Provided by:
[info]  ProjectRef(uri("file:/tmp/projd/"), "projD") / Compile / scalacOptions
[info] Defined at:
[info]  /tmp/projd/build.sbt:9
[info] Reverse dependencies:
[info]  projD / test
[info]  projD / Compile / console
[info] Delegates:
[info]  projD / Compile / console / scalacOptions
[info]  projD / Compile / scalacOptions
[info]  projD / console / scalacOptions
[info]  projD / scalacOptions
[info]  ThisBuild / Compile / console / scalacOptions
[info]  ThisBuild / Compile / scalacOptions
[info]  ThisBuild / console / scalacOptions
[info]  ThisBuild / scalacOptions
[info]  Zero / Compile / console / scalacOptions
[info]  Zero / Compile / scalacOptions
[info]  Zero / console / scalacOptions
[info]  Global / scalacOptions
```

请注意，“Provided by” 如何显示 projD / Compile / console / scalacOptions 提供了 projD / Compile / scalacOptions。同样在 “Delegates” (委托)，按优先顺序列出了**所有**可能的委托候选人！

- 首先列出在 subproject 轴上具有 projD scope 的所有 scope，然后列出 ThisBuild 和 Zero。
- 在 subproject 中，首先列出在 configuration 轴上具有 Compile scope 的 scope，然后退回到 Zero。
- 首先列出在 task 轴上具有 task scope console / 的所有 scope，然后列出没有 task scope console / 的所有 scope。

.value 查找与动态调度

- 规则 5：在不携带原始上下文的情况下，评估委托 scoped key 及其相关的 settings/tasks。

请注意，scope 委托感觉类似于面向对象语言中的类继承，但是有区别。在像 Scala 这样的 OO 语言中，如果在 trait Shape 上有一个名为 drawShape 的 method，则即使 Shape trait

中的其他 method 使用了 drawShape，其子类也可以覆盖行为，这称为动态调度。

但是，在 sbt 中，scope 委托可以将 scope 委托给更通用的 scope，例如将 project-level 的 setting 委托给 build-level setting，但是该 build-level setting 不能引用 project-level setting。

练习题 E: 给出以下构建定义：

```
lazy val root = (project in file("."))
  .settings(
    inThisBuild(List(
      organization := "com.example",
      scalaVersion := "2.12.2",
      version      := scalaVersion.value + "_0.1.0"
    )),
    name := "Hello"
  )

lazy val projE = (project in file("e"))
  .settings(
    scalaVersion := "2.11.11"
  )
```

projE / version 返回什么?

1. "2.12.2_0.1.0"
2. "2.11.11_0.1.0"
3. 还有什么吗

答案是 2.12.2_0.1.0。projE / version 委托 ThisBuild / version，它取决于 ThisBuild / scalaVersion。因此，build-level setting 应主要限于简单的值分配。

练习题 F: 给出以下构建定义：

```
ThisBuild / scalacOptions += "-D0"
scalacOptions += "-D1"

lazy val projF = (project in file("f"))
  .settings(
    compile / scalacOptions += "-D2",
    Compile / scalacOptions += "-D3",
    Compile / compile / scalacOptions += "-D4",
    test := {
      println("bippy" + (Compile / compile / scalacOptions).value.mkString)
    }
  )
```

projF / test 显示什么?

1. "bippy-D4"
2. "bippy-D2-D4"

3. "bippy-D0-D3-D4"
4. 还有什么吗

答案是 "bippy-D0-D3-D4"。这是 Paul Phillips 最初创建的练习的变体。这是所有规则的很好展示，因为 someKey += "x" 扩展为

```
someKey := {
  val old = someKey.value
  old :+ "x"
}
```

检索旧值将导致委托，并且由于规则 5，它将转到另一个 scoped key。让我们先摆脱 +=，然后为旧值注释委托：

```
ThisBuild / scalacOptions := {
  // Global / scalacOptions <- Rule 4
  val old = (ThisBuild / scalacOptions).value
  old :+ "-D0"
}

scalacOptions := {
  // ThisBuild / scalacOptions <- Rule 4
  val old = scalacOptions.value
  old :+ "-D1"
}

lazy val projF = (project in file("f"))
  .settings(
    compile / scalacOptions := {
      // ThisBuild / scalacOptions <- Rules 2 and 4
      val old = (compile / scalacOptions).value
      old :+ "-D2"
    },
    Compile / scalacOptions := {
      // ThisBuild / scalacOptions <- Rules 3 and 4
      val old = (Compile / scalacOptions).value
      old :+ "-D3"
    },
    Compile / compile / scalacOptions := {
      // projF / Compile / scalacOptions <- Rules 1 and 2
      val old = (Compile / compile / scalacOptions).value
      old :+ "-D4"
    },
    test := {
      println("bippy" + (Compile / compile / scalacOptions).value.mkString)
    }
  )
```

变成:

```
ThisBuild / scalacOptions := {
  Nil :+ "-D0"
}

scalacOptions := {
  List("-D0") :+ "-D1"
}

lazy val projF = (project in file("f"))
  .settings(
    compile / scalacOptions := List("-D0") :+ "-D2",
    Compile / scalacOptions := List("-D0") :+ "-D3",
    Compile / compile / scalacOptions := List("-D0", "-D3") :+ "-D4",
    test := {
      println("bippy" + (Compile / compile / scalacOptions).value.mkString)
    }
  )
```

库依赖

阅读这一小节时，假设你已经阅读过新手入门前面的内容，特别是 .sbt 构建定义，Scopes 和 更多关于设置。

可以通过下面这两种方式添加库依赖:

- 非托管依赖为放在 lib 目录下的 jar 文件
- 托管依赖配置在构建定义中，并且会自动从仓库（repository）中下载

非托管依赖

大多数人会用托管依赖而不是非托管依赖。但是非托管依赖在起步阶段会简单很多。

非托管依赖像这样工作: 将 jar 文件放在 lib 文件夹下，然后它们将会被添加到项目的 classpath 中。没有更多的事情了!

你也可以将测试依赖的 jar 文件放在 lib 目录下，比如 ScalaCheck，Specs2，ScalaTest。

lib 目录下的所有依赖都会在 classpaths（对 compile，test，run 和 console 都成立）。如果你想对其中的一个改变 classpath，你需要做适当调整，例如 Compile / dependencyClasspath 或者 Runtime / dependencyClasspath。

如果用非托管依赖的话，不用往 build.sbt 文件中添加任何内容，不过你可以改变 unmanagedBase key，如果你想用一个不同的目录而非 lib。

用 custom_lib 替代 lib:

```
unmanagedBase := baseDirectory.value / "custom_lib"
```

baseDirectory 是项目的根目录，所以在这里你依据 baseDirectory 的值改变了 unmanagedBase，通过在 更多关于设置 中介绍的一个特殊的 value 方法。

同时也有一个列举 unmanagedBase 目录下所有 jar 文件的 task 叫 unmanagedJars。如果你想用多个目录或者做一些更加复杂的事情，你可能需要用一个可以做其他事情的 task 替换整个 unmanagedJars task，例如清空 Compile configuration 的列表，不考虑 lib 目录下的文件：

```
Compile / unmanagedJars := Seq.empty[sbt.Attributed[java.io.File]]
```

托管依赖

sbt 使用 Apache Ivy 来实现托管依赖，所以如果你对 Ivy 或者 Maven 比较熟悉的话，你不会有太多的麻烦。

libraryDependencies Key

大多数时候，你可以很简单的在 libraryDependencies 设置项中列出你的依赖。也可以通过 Maven POM 文件或者 Ivy 配置文件来配置依赖，而且可以通过 sbt 来调用这些外部的配置文件。你可以从这里获取更详细的内容。

可以像这样定义一个依赖，其中 groupId，artifactId 和 revision 都是字符串：

```
libraryDependencies += groupID % artifactID % revision
```

或者像这样，用字符串或者 Configuration val (Test) 当做 configuration：

```
libraryDependencies += groupID % artifactID % revision % configuration
```

libraryDependencies 在 Keys 中像这样声明：

```
val libraryDependencies = settingKey[Seq[ModuleID]]("Declares managed dependencies.")
```

方法 % 从字符串创建 ModuleID 对象，然后将 ModuleID 添加到 libraryDependencies 中。

当然，要让 sbt（通过 Ivy）知道从哪里下载模块。如果你的模块和 sbt 来自相同的某个默认的仓库，这样就会工作。例如，Apache Derby 在标准的 Maven2 仓库中：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

如果你在 build.sbt 中输入上面这些内容，然后执行 update，sbt 会将 Derby 下载到 $COURSIER_CACHE/https/repo1.maven.org/maven2/org/apache/derby。(顺便提一下，compile 依赖于 update，所以大多数时候不需要手动的执行 update。)

当然，你也可以通过 ++= 一次将所有依赖作为一个列表添加：

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
```

```
  groupID % otherID % otherRevision
)
```

在很少情况下，你也会需要在 libraryDependencies 上用 := 方法。

通过 %% 方法获取正确的 Scala 版本

如果你用是 groupID %% artifactID % revision 而不是 groupID % artifactID % revision（区别在于 groupID 后面是 %%），sbt 会在工件名称中加上项目的 Scala 版本号。这只是一种快捷方法。你可以这样写不用 %%：

```
libraryDependencies += "org.scala-stm" % "scala-stm_2.13" % "0.9.1"
```

假设这个构建的 scalaVersion 是 2.13.8，下面这种方式是等效的（注意 "org.scala-stm" 后面是 %%）：

```
libraryDependencies += "org.scala-stm" %% "scala-stm" % "0.9.1"
```

这个想法是很多依赖都会被编译给多个 Scala 版本，而你想确保和项目匹配的 jar 是二进制兼容的。

参见 交叉构建 获取更多信息。

Ivy 修正

groupID % artifactID % revision 中的 revision 不需要是一个固定的版本号。Ivy 能够根据你指定的约束选择一个模块的最新版本。你指定 "latest.integration"，"2.9.+" 或者 "[1.0,)"，而不是一个固定的版本号，像 "1.6.1"。参看 Ivy 修订 文档获取详细内容。

解析器

不是所有的依赖包都放在同一台服务器上，sbt 默认使用标准的 Maven2 仓库。如果你的依赖不在默认的仓库中，你需要添加 *resolver* 来帮助 Ivy 找到它。

通过以下形式添加额外的仓库：

```
resolvers += name at location
```

在两个字符串中间有一个特殊的 at。

例如：

```
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snap
```

resolvers key 在 Keys 中像这样定义：

```
val resolvers = settingKey[Seq[Resolver]](" 用户为托管依赖定义的额外的解析器。")
```

at 方法通过两个字符串创建了一个 Resolver 对象。

sbt 会搜索你的本地 Maven 仓库如果你将它添加为一个仓库：

```
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/reposit
```

或者，为了方便起见：

```
resolvers += Resolver.mavenLocal
```

参见 解析器 获取更多关于定义其他类型的仓库的内容。

覆写默认的解析器

resolvers 不包含默认的解析器，仅仅通过构建定义添加额外的解析器。

sbt 将 resolvers 和一些默认的仓库组合起来构成 externalResolvers。

然而，为了改变或者移除默认的解析器，你需要覆写 externalResolvers 而不是 resolvers。

Per-configuration dependencies

通常会有依赖只被测试代码使用（在 src/test/scala 中，通过 Test configuration 编译）而没有在主应用中使用。

如果你想要一个依赖只在 Test configuration 的 classpath 中出现而不是 Compile configuration，像这样添加 % "test"：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % "test"
```

也可能也会像这样使用类型安全的 Test configuration：

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3" % Test
```

现在，如果你在 sbt 的命令提示行里输入 show Compile/dependencyClasspath，你不应该看到 derby jar。但是如果你输入 show Test/dependencyClasspath，你应该在列表中看到 derby jar。

通常，测试相关的依赖，如 ScalaCheck，Specs2 和 ScalaTest 将会被定义为 % "test"。

库依赖更详细的内容和技巧在这里。

多项目构建

这一小节介绍在一个构建中定义多个项目。

请先阅读入门指南前面的内容，尤其需要在阅读本小节之前理解 .sbt 构建定义。

多项目

将多个相关的项目定义在一个构建中是很有用的，尤其是如果它们依赖另一个，而且你倾向于一起修改它们。

每个子项目在构建中都有它们自己的源文件夹，当打包时生成各自的 jar 文件，而且通常和其他的项目一样运转。

通过声明一个类型为 Project 的 lazy val 定义一个项目，例如：

```
lazy val util = project

lazy val core = project
```

val 的名称被用作项目的 ID 和基目录名。该 ID 用于在命令行中引用该项目。基目录可能通过 in 方法改变。例如，下面是一个更加显示的方式来实现前一个例子：

```
lazy val util = project.in(file("util"))

lazy val core = project in file("core")
```

公共设定

To factor out common settings across multiple projects, create a sequence named commonSettings and call settings method on each project. 要跨多个项目提取公共设置，请创建一个名为 commonSettings 的序列，并在每个项目上调用 settings 方法。

```
lazy val commonSettings = Seq(
  organization := "com.example",
  version := "0.1.0",
  scalaVersion := "2.12.16"
)

lazy val core = (project in file("core"))
  .settings(
    commonSettings,
    // other settings
  )

lazy val util = (project in file("util"))
  .settings(
    commonSettings,
    // other settings
  )
```

现在我们可以在一处修改 version，当重新加载构建时，将在各个子项目相应更新。

依赖

构建中的项目完全可以彼此独立，但是通常情况下它们会有依赖上的一些相关性。有两种类型的依赖：aggregate 和 classpath。

Aggregation

Aggregation 意味着在 aggregate 项目上执行一个 task 也会在 aggregated 的项目执行。例如，

```
lazy val root = (project in file(".")).aggregate(util, core)

lazy val util = project

lazy val core = project
```

在上面的例子中，root 项目聚合了 util 和 core。像例子中一样，随着有两个子项目的情况下启动 sbt，然后尝试编译。你应该会看到全部的三个项目都被编译了。

在进行聚合的项目中，像这个例子中的 root 项目一样，你可以按 task 来控制聚合。例如，为了避免聚合 update task:

```
lazy val root = (project in file("."))
  .aggregate(util, core)
  .settings(
    aggregate in update := false
  )

[...]
```

aggregate in update 是 update task 的 scope 下的聚合的 key。(参见 scopes。)

注意：聚合会并行的执行聚合的 task，task 之间的顺序是不确定的。

Classpath 依赖

一个项目可能依赖另一个项目的代码。这是通过添加 dependsOn 方法来实现的。例如，如果 core 在 classpath 中需要 util，你将这样定义 core:

```
lazy val core = project.dependsOn(util)
```

现在 core 中的代码可以使用 util 的类。在编译时也会在两个项目之间创建顺序；在编译 core 之前，util 必须被更新和编译过。

为了依赖多个项目，像这样 dependsOn(bar, baz) 给 dependsOn 多个参数。

configuration 粒度的 classpath 依赖

foo dependsOn(bar) 表示 foo 中的 compile configuration 依赖于 bar 中的 compile configuration。你可以显示的写成这样：dependsOn(bar % "compile->compile")。

"compile->compile" 中的 -> 表示 “depends on”，所以 "test->compile" 表示 foo 中的 test configuration 将依赖于 bar 中的 compile configuration。

省略 ->config 部分暗示 ->compile，所以 dependsOn(bar % "test") 表示 foo 中的 test configuration 依赖于 bar 中的 Compile configuration。

一个实用的声明 "test->test" 表示 test 依赖于 test。例如，这样允许你将测试工具代码放在 bar/src/test/scala 中，然后在 foo/src/test/scala 中使用这些代码，

对于一个依赖你可以有多个 configuration，以分号分隔，例如: dependsOn(bar % "test->test;compile->compile")。

默认的 root 项目

如果在构建中根目录没有定义项目，sbt 会在构建中创建一个默认的项目并将其他项目也聚合起来。

因为 hello-foo 项目定义了 base = file("foo")，它将会被包含在 foo 子目录中。它的源文件可以直接放在 foo 下，像 foo/Foo.scala，或者在 foo/src/main/scala 中。通常 sbt 的 目录结构 应用在 foo 目录下除了构建定义文件。

交互式引导项目

在 sbt 的命令行中，输入 projects 列出你的项目，执行 project <projectname> 可以选择当前项目。当你执行 task 像 compile，它会在当前项目上执行。所以你没有必要去编译 root 项目，你可以只编译子项目。

你可以通过显示的指定项目 ID 在另一个项目上执行一个 task,例如 subProjectID/compile。

通用代码

在一个 .sbt 文件中的定义对于其他的 .sbt 文件不可见。为了在不同的 .sbt 文件中共享代码，在构建根目录下的 project/ 目录下定义一个或多个 Scala 文件。

参见 组织构建 获取详细内容。

Appendix: Subproject build definition files

foo 中的任何 .sbt 文件,比如说 foo/build.sbt,将会和整个构建合并,但是在 hello-foo 项目的 scope 中。

如果你的的整个项目都在 hello 中，尝试在 hello/build.sbt，hello/bar/build.sbt 和 hello/foo/build.sbt 中定义一个不同的版本 (version := "0.6")。现在在 sbt 的命令行中执行 show version。你应该得到这样的信息 (随着你定义的任何版本):

```
> show version
[info] hello-foo/*:version
[info]  0.7
[info] hello-bar/*:version
[info]  0.9
[info] hello/*:version
[info]  0.5
```

hello-foo/*:version 定义在 hello/foo/build.sbt 中，hello-bar/*:version 定义在 hello/bar/build.sbt 中，hello/*:version 定义在 hello/build.sbt 中。记住 scoped keys 的语法。每个 version key 在对应的项目的 scope 中，基于 build.sbt 文件的位置。但是所有的三个 build.sbt 文件都只是整个构建定义的一部分。

Style choices:

- Each subproject's settings can go into *.sbt files in the base directory of that project, while the root build.sbt declares only minimum project declarations in the form of lazy val foo = (project in file("foo")) without the settings.
- We recommend putting all project declarations and settings in the root build.sbt file in order to keep all build definition under a single file. However, it up to you.

在子项目中,你不能有项目的子目录或者 project/*.scala 文件。foo/project/Build.scala 将会被忽略。

使用插件

请先阅读入门指南前面的内容，尤其需要在阅读本小节之前理解 build.sbt 和 库依赖。

什么是插件

插件继承了构建定义，大多数通常是通过添加设置。新的设置可以是新的 task。例如，一个插件可以添加一个 codeCoverage task 来生成一个测试覆盖率报告。

声明一个插件

如果你的项目在 hello 目录下，而且你正在往构建定义中添加一个 sbt-site 插件，创建 hello/project/site.sbt 并且通过传递插件的 Ivy 模块 ID 声明插件依赖给 addSbtPlugin:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-site" % "0.7.0")
```

如果你添加 sbt-assembly，像下面这样创建 hello/project/assembly.sbt :

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

不是所有的插件都在同一个默认的仓库中，而且一个插件的文档会指导你添加能够找到它的仓库:

```
resolvers ++= Resolver.sonatypeOssRepos("public")
```

插件通常提供设置将它添加到项目并且开启插件功能。这些将在下一小节描述。

启用和禁用自动插件

一个插件能够声明它自己的设置被自动添加到构建定义中去，在这种情况下你不需要为添加它做任何事情。

作为 0.13.5 版本的 sbt，有一个新的特性叫自动插件，它能够在自动的、安全的、确保所有依赖都在项目里的前提下开启插件。很多自动插件应该能够自动开启，然而有些却需要显式开启。

如果你正在使用一个需要显示开启的自动插件，那么你需要添加这样的代码到你的 build.sbt 文件：

```
lazy val util = (project in file("util"))
  .enablePlugins(FooPlugin, BarPlugin)
  .settings(
    name := "hello-util"
  )
```

enablePlugins 方法允许项目显式定义它们需要使用的自动插件。

项目也可以使用 disablePlugins 方法排除掉一些插件。例如，如果我们希望能够从 util 中移除 IvyPlugin 插件的设置，我们将 build.sbt 修改如下：

```
lazy val util = (project in file("util"))
  .enablePlugins(FooPlugin, BarPlugin)
  .disablePlugins(plugins.IvyPlugin)
  .settings(
    name := "hello-util"
  )
```

自动插件会在文档中说明是否需要显示的开启。如果你对一个项目中开启了哪些插件好奇，只需要在 sbt 命令行中执行 plugins 命令。

例如：

```
> plugins
In file:/home/jsuereth/projects/sbt/test-ivy-issues/
        sbt.plugins.IvyPlugin: enabled in scala-sbt-org
        sbt.plugins.JvmPlugin: enabled in scala-sbt-org
        sbt.plugins.CorePlugin: enabled in scala-sbt-org
        sbt.plugins.JUnitXmlReportPlugin: enabled in scala-sbt-org
```

这里，plugins 的输出显示 sbt 默认的插件都被开启了。sbt 默认的设置通过 3 个插件提供：

1. CorePlugin: 提供对 task 的核心并行控制。
2. IvyPlugin: 提供发布、解析模块的机制。
3. JvmPlugin: 提供编译、测试、执行、打包 Java/Scala 项目的机制。

另外，JUnitXmlReportPlugin 提供对生成 junit-xml 的试验性支持。

老的非自动的插件通常需要显示的添加设置，以致于多项目构建可以有不同的项目类型。插件的文档会指出如何配置它，但是特别是对于老的插件，这包含添加对插件必要的基本设置和自定义。

例如，对于 sbt-site 插件，为了在项目中开启它，需要创建包含如下内容的 site.sbt 文件来。

```
site.settings
```

如果构建定义了多个项目，往项目中直接添加如下内容替而代之：

```
// 在 `util` 项目中不使用 site 插件
lazy val util = (project in file("util"))

// 在`core` 项目中开启 site 插件
lazy val core = (project in file("core"))
  .settings(site.settings)
```

全局插件

可以一次给所有项目安装插件，只要在 $HOME/.sbt/1.0/plugins/ 中声明它们。$HOME/.sbt/1.0/plugins/ 是一个将自己的 classpath 导出给所有项目的 sbt 构建定义。概略地讲，在 $HOME/.sbt/1.0/plugins/ 中的任何 .sbt 或者 .scala 文件就和所有项目的 project/ 目录下的一样。

为了一次给所有的项目添加插件，你可以创建 $HOME/.sbt/1.0/plugins//build.sbt 并且添加 addSbtPlugin() 表达式。因为这样做会增加机器上的依赖，所以这个特性应该少用。参见最佳实践。

可用的插件

这里有一个可用的插件列表。

一些特别流行的插件如下：

- 对 IDE 的支持（为了将 sbt 项目导入到 IDE）
- 对 web 框架的支持，例如xsbt-web-plugin。

更多详细信息，包含开发插件的方法，参见插件。关于最佳实践，参见插件最佳实践。

自定义设置和任务

这一小节讲解如何创建自定义设置和任务。

在理解本节之前，请先阅读 sbt 入门前面的章节，尤其是 .sbt 构建定义和更多关于设置。

定义一个键

这里介绍了如何定义键。大多数的默认键定义在这里。

键有三种类型。SettingKey 和 TaskKey 在 .sbt 构建定义讲解。关于 InputKey 的内容在输入任务页面。

列举一些来自 Keys 的例子：

```
val scalaVersion = settingKey[String]("scala 的版本")
val clean = taskKey[Unit](" 删除构建产生的文件，包括生成的 source 文件，编译的类和任务缓存。")
```

键的构造函数有两个字符串参数：键的名称（“scalaVersion”）和文档字符串（“用于构建工程的 scala 的版本。”）。

还记得.sbt 构建定义中，类型 T 在 SettingKey[T] 中表示的设置的值的类型。类型 T 在 TaskKey [T] 中指示任务的结果的类型。在.sbt 构建定义中，一个设置有一个固定的值，直到项目重新加载。任务会在每一个“任务执行”（用户在交互输入中或在 batch 模式下输入一个命令）被重新计算。

键可以在定义在.sbt 构建定义，.scala 文件或一个自动插件中。任何在启用的自动插件的 autoImport 对象的 val 将被自动导入到你的 .sbt 文件。

执行任务

一旦你定义了一个任务的键，你需要用它完成任务定义。你可以定义自己的任务，或者重新定义现有的任务。无论哪种方式看起来是一样的；用 := 使任务的键和部分代码相关联：

```
val sampleStringTask = taskKey[String]("A sample string task.")
val sampleIntTask = taskKey[Int]("A sample int task.")

ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.16"

lazy val library = (project in file("library"))
  .settings(
    sampleStringTask := System.getProperty("user.home"),
    sampleIntTask := {
      val sum = 1 + 2
      println("sum: " + sum)
      sum
    }
  )
```

在更多关于设置里有描述，如果任务有依赖关系，你使用 value 来引用值。

有关任务实现最困难的部分往往不是 sbt 专用；任务只是 Scala 代码。困难的部分可能是写你的任务体，即做什么，或者说你正在试图做的。例如，你要格式化 HTML，在这种情况下，你可能需要使用一个 HTML 库（也许你将为构建定义添加一个库的依赖来编写基于 HTML 库代码）。

sbt 具有一些实用工具库和方便的函数，特别是可以经常使用 API 中的 IO 来操作文件和目录。

任务的执行语义

当从依赖于其他任务的自定义任务中使用 value 时，一个要注意的重要细节是是任务的执行语义。对执行语义，我们的意思是到底何时这些任务被取值。

以 sampeIntTask 为例，任务体中的每一行应严格地一个接一个被取值。这就是顺序语义：

```
sampleIntTask := {
  val sum = 1 + 2         // first
  println("sum: " + sum) // second
  sum                     // third
}
```

在现实中，JVM 可能内联 sum 为 3，但任务可观察到的行为仍将与严格地一个接一个被执行完全相同。

现在假设我们定义了另外两个的自定义任务 startServer 和 stopServer，并修改 sampeIntTask，如下所示：

```
val startServer = taskKey[Unit]("start server")
val stopServer = taskKey[Unit]("stop server")
val sampleIntTask = taskKey[Int]("A sample int task.")
val sampleStringTask = taskKey[String]("A sample string task.")

ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.16"

lazy val library = (project in file("library"))
  .settings(
    startServer := {
      println("starting...")
      Thread.sleep(500)
    },
    stopServer := {
      println("stopping...")
      Thread.sleep(500)
    },
    sampleIntTask := {
      startServer.value
      val sum = 1 + 2
      println("sum: " + sum)
      stopServer.value // THIS WON'T WORK
      sum
    },
    sampleStringTask := {
      startServer.value
      val s = sampleIntTask.value.toString
```

```
      println("s: " + s)
      s
    }
  )
```

从 sbt 交互式提示符中运行 sampleIntTask 将得到如下结果：

```
> sampleIntTask
stopping...
starting...
sum: 3
[success] Total time: 1 s, completed Dec 22, 2014 5:00:00 PM
```

若要查看发生了什么事，让我们看一下 sampleIntTask 图形表示：

不同于普通的 Scala 方法调用，调用任务的 value 方法将不被严格取值。相反，他们只是充当占位符来表示 sampleIntTask 依赖于 startServer 和 stopServer 任务。当你调用 sampleIntTask 时，sbt 的任务引擎将：

- 在对 sampleIntTask 取值前对依赖任务取值（偏序）
- 如果依赖任物是相互独立的，尝试并行取值（并行）
- 每次命令执行，每个任务依赖项将被评估且仅被评估一次（去重）

任务依赖项去重

为证明这最后一点，我们可以从 sbt 交互式提示符运行 sampleStringTask。

```
> sampleStringTask
stopping...
starting...
sum: 3
s: 3
[success] Total time: 1 s, completed Dec 22, 2014 5:30:00 PM
```

因为 sampleStringTask 依赖于 startServer 和 sampleIntTask 两个任务，而 sampleIntTask 也依赖于 startServer 任务，它作为任务依赖出现了两次。如果这是一个普通的 Scala 方法调用，它会被计算两次，但由于任务的依赖项被标记为 value 类型，它将只被计算一次。以下是 sampeStringTask 如何取值的图形表示：

如果我们不做重复任务相关项的去重，则当我们执行 test 时最终会编译测试源代码很多次，因为 Test / compile 作为 Test / test 的依赖项出现了很多次。

清理任务

应该如何实现 stopServer 任务？清理任务的概念并不适合任务的执行模型，因为任务关心的是依赖项跟踪。最后一次操作应成为依赖其他中间任务的任务。例如 stopServer 应依赖于 sampleStringTask，在其中 stopServer 应该是 sampleStringTask。

```
lazy val library = (project in file("library"))
  .settings(
    startServer := {
      println("starting...")
      Thread.sleep(500)
    },
    sampleIntTask := {
      startServer.value
      val sum = 1 + 2
      println("sum: " + sum)
      sum
    },
    sampleStringTask := {
      startServer.value
      val s = sampleIntTask.value.toString
      println("s: " + s)
      s
    },
    sampleStringTask := {
      val old = sampleStringTask.value
      println("stopping...")
      Thread.sleep(500)
      old
    }
  )
```

为了证明它可以工作，在交互式提示符中运行 sampleStringTask：

```
> sampleStringTask
starting...
sum: 3
s: 3
stopping...
[success] Total time: 1 s, completed Dec 22, 2014 6:00:00 PM
```

task-dependency

直接使用 Scala

确保一些事发生在其它一些事物之后的另一种方式是使用 Scala。例如，在 project/ServerUtil.scala 中实现一个简单的函数，你可以编写：

```
sampleIntTask := {
  ServerUtil.startServer
```

```
  try {
    val sum = 1 + 2
    println("sum: " + sum)
  } finally {
    ServerUtil.stopServer
  }
  sum
}
```

因为普通的方法调用遵循顺序语义，所有事情按顺序发生。这里没有去重，所以你必须要小心。

将它们转为插件

如果你发现自己有很多自定义代码，可以考虑将其移动到插件，从而可以在多个构建中重复利用。

创建一个插件很容易，在使用插件和插件中有详细讨论。

本小节是个快速的向导；更多关于自定义任务可以在任务中找到。

组织构建

本页面将讨论构建结构的组织。

本小节假设你已经阅读了之前的章节，尤其是 build.sbt，库依赖和多工程构建。

sbt 是递归的

build.sbt 很简单，隐藏了 sbt 是如何工作的。sbt 构建是用 Scala 代码定义的。代码本身也必须是能被构建的。有比 sbt 更好的建立方式么?

project 目录 是你的工程内另一个工程的项目，它知道如何构建你的工程。为了区分这两种构建，我们有时使用**正常构建**表示你的构建，使用**元构建**指代在 project 中的构建。在元构建中的项目能做任何其他项目可以做的事情。你的构建定义是一个 *sbt* 项目。

递归可以继续下去。如果你喜欢, 你可以通过创建 project/project/ 目录稍稍调整项目的构建定义。

下面是一个例子:

```
hello/                  # 项目的基目录

    Hello.scala         # 一个项目源文件（也可以在src/main/scala）

    build.sbt           # build.sbt 是project/ 中元构建根项目的源代码。是构建定义项目的一部

    project/            # 元构建根项目的基目录
```

```
Build.scala     # 元构建根项目的一个源文件，是你的构建定义的构建定义源文件
build.sbt       # 元元构建的根项目——project/project的源代码；构建定义的构建定义
project/        # 元元构建的根项目的基目录；构建定义的构建定义工程
    Build.scala # project/project/ 元元构建的根项目中的源文件
```

不用担心！大部分时候不需要 project/project/ 目录。但是理解它是有帮助的。

另外，任何以 .scala 或者 .sbt 结尾的文件都会被使用，命名为 build.sbt 和 Build.scala 只是惯例。多个文件也是允许的。

在同一个地方跟踪依赖项

用 project 下的.scala 文件组成构建定义的一个实际用例是创建 project/Dependencies.scala 来在同一个地方跟踪依赖项。

```
import sbt._

object Dependencies {
  // Versions
  lazy val akkaVersion = "2.6.19"

  // Libraries
  val akkaActor = "com.typesafe.akka" %% "akka-actor" % akkaVersion
  val akkaCluster = "com.typesafe.akka" %% "akka-cluster" % akkaVersion
  val specs2core = "org.specs2" %% "specs2-core" % "4.16.0"

  // Projects
  val backendDeps =
    Seq(akkaActor, specs2core % Test)
}
```

Dependencies 对象将在 build.sbt 中可用。如果要让使用 val 的代码更加简单，可以引入 Dependencies._。

```
import Dependencies._

ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.16"

lazy val backend = (project in file("backend"))
  .settings(
    name := "backend",
```

```
    libraryDependencies ++= backendDeps
  )
```

当你的多工程构建变得很大，并且想要确保子项目有一致的依赖关系时，这种技术很有用。

何时用 .scala 文件

在 .scala 文件，你可以写任意的 Scala 代码，包括顶层的类和对象。

推荐的方法是定义大部分设置在多工程的 build.sbt 文件中，并且使用 project/*.scala 文件来做任务实现或在多个文件中共享键值。对.scala 文件的使用也取决于你的团队对 scala 的熟练程度。

定义自动插件

对于更高级的用户，另一种方式组织你的构建是在 project/*.scala 中定义一次性自动插件。通过定义触发的插件，自动插件可以用作一种简便方法来注入跨所有子项目的自定义任务和命令。

总结

这一节将入门指南总结一下。

为了使用 sbt，有一些概念你必须理解。这有一些学习曲线，但是乐观的讲，除了这些概念对于 sbt 并不多。sbt 用一小部分核心概念来使得它工作。

如果你已经阅读过所有的入门指南，现在你知道了你需要知道什么。

sbt：核心概念

- Scala 基础。不可否认，熟悉 Scala 语法非常有帮助。Programming in Scala，Scala 的作者写的非常好的介绍。
- .sbt 构建定义
- 你的构建定义是一个大的 Setting 对象列表，sbt 使用 Setting 转换之后的键值对执行 task。
- 为了创建 Setting，在一个 key 上调用其中的一个方法：:=，+= 或者 ++=。
- 没有可变的状态，至于转换；例如，一个 Setting 将 sbt 的键值对集合转换成一个新的集合。不会就地改变任何代码。
- 每一个设置都有一个特定类型的值，由 key 决定。
- *tasks* 是特殊的设置，通过 key 产生 value 的计算在每次出发 task 的时候都会重新执行。Non-task 计算只会在构建定义的第一次加载时执行。
- Scopes
- 每一个 key 都可能有多个 value，按照 scope 划分。
- scope 会用三个轴：configuration，project，task。
- scope 允许你按项目、按 task、按 configuration 有不同的行为。
- 一个 configuration 是一种类型的构建，例如 Compile 或者 Test。

- project 轴也支持“构建全局” scope。
- scopes 回滚或 代理到更通用的 scope。
- 将大部分配置放在 `build.sbt` 中，但是用 `.scala` 构建定义文件定义类和更大的 task 实现。
- 构建定义是一个 sbt 项目，来自于项目目录。
- 插件是对构建定义的扩展
- 通过在 `addSbtPlugin` 方法在 `project/plugins.sbt` 中添加插件。(不是在项目基目录下的 `build.sbt` 中)。

如果你怀疑这些细枝末节中的任何一个，请寻求帮助，返回重新阅读或者在 sbt 的交互式命令行中做实验。

祝你好运!

附录

因为 sbt 是一个开源项目，别忘记签出项目源代码!