Manual de referencia de sbt

Contents

Preface

Guía de inicio de sbt

sbt utiliza unos cuantos conceptos que dan soporte a flexibles y poderosas definiciones de construcción (build definitions). No es que haya muchos conceptos, pero sbt no es exactamente como otros sistemas de construcción y algunos detalles te *confundirán* si no has leído la documentación.

La Guía de inicio cubre los conceptos que necesitas conocer para crear y mantener una definición de construcción de sbt.

¡Es *altamente recomendable* leer la Guía de inicio!

Si tienes prisa, los conceptos esenciales más importantes pueden ser encontrados en Definiciones de construcción, Ámbitos, y Grafos de tareas. Aunque no te aseguramos que saltarte el resto de páginas de la guía sea una buena idea.

Lo mejor es leerlas en orden, ya que las páginas finales de la Guía de inicio se basan en conceptos explicados antes.

Gracias por probar sbt y… ¡*divértete*!

Instalar sbt

Para crear un proyecto sbt necesitarás realizar los siguientes pasos:

- Instalar JDK (recomendamos AdoptOpenJDK JDK 8 u AdoptOpenJDK JDK 11).

- Instalar sbt.
- Configurar un proyecto hola mundo simple
- Continuar con Ejecución para aprender cómo ejecutar sbt.
- Continuar con Definición de construcción para aprender más acerca de definiciones de construcción.

En última instancia, la instalación de sbt se reduce a un lanzador JAR y un script de shell, pero dependiendo de tu plataforma, proporcionamos varias formas de hacer el proceso menos tedioso. Echa un vistazo a los pasos de instalación para macOS, Windows, o Linux.

Consejos y notas

Si has tenido algún problema ejecutando sbt, revisa las Notas sobre configuración sobre codificaciones en el terminal, proxies HTTP y opciones de la JVM.

Instalar sbt on macOS

Instalar JDK

Sigue el link para instalar JDK 8 u 11.

Or use SDKMAN!:

```
$ sdk list java
$ sdk install java 11.0.4.hs-adpt
```

Instalar desde un paquete universal

Descarga el paquete ZIP o TGZ y descomprímelo.

Instalar desde un paquete de terceros

> **Nota:** Puede que algunos paquetes de terceros no proporcionen la última versión. Por favor, asegúrate de reportar cualquier problema con dichos paquetes a sus respectivos mantenedores.

Homebrew

```
$ brew install sbt
```

SDKMAN!

```
$ sdk install sbt
```

Instalar sbt en Windows

Instalar JDK

Sigue el link para instalar JDK 8 u 11.

Instalar desde un paquete universal

Descarga el paquete ZIP o TGZ y descomprímelo.

Instalador Windows

Descarga el instalador msi e instálalo.

Instalar desde un paquete de terceros

> **Nota:** Puede que algunos paquetes de terceros no proporcionen la última versión. Por favor, asegúrate de reportar cualquier problema con dichos paquetes a sus respectivos mantenedores.

Scoop

```
$ scoop install sbt
```

Installing sbt on Linux

Installing from SDKMAN

To install both JDK and sbt, consider using SDKMAN.

```
$ sdk install java $(sdk list java | grep -o "8\.[0-9]*\.[0-9]*\.hs-adpt" | head -1)
$ sdk install sbt
```

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

Instalar JDK

Primero desberás de instalar JDK. Recomendamos AdoptOpenJDK JDK 8 u AdoptOpenJDK JDK 11.

Los detalles sobre el nombre de los paquetes cambian de una distribución a otra. Por ejemplo, Ubuntu xenial (16.04LTS) usa openjdk-8-jdk. La familia Redhat lo llama java-1.8.0-openjdk-devel.

Instalar desde un paquete universal

Descarga el paquete ZIP o TGZ y descomprímelo.

Ubuntu y otras distribuciones basadas en Debian

Los paquetes DEB son oficialmente soportados por sbt.

Ubuntu y otras distribuciones basadas en Debian usan el formato DEB, pero por lo general no necesitas instalar software desde un fichero DEB local. En su lugar lo que se utiliza son los gestores de paquetes, tanto desde la línea de comandos (p.e. `apt-get`, `aptitude`) o con una interfaz gráfica de usuario (p.e. Synaptic). Ejecuta lo siguiente desde el terminal para instalar `sbt` (necesitarás tener privilegios de administrador para hacerlo, de ahí el `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
```

Los gestores de paquetes utilizan los repositorios para buscar los paquetes que se desean instalar. Sólo tienes que añadir el repositorio en aquellos ficheros utilizados por tu gestor de paquetes.

Una vez `sbt` haya sido instalado podrás gestionar el paquete en `aptitude` o Synaptic después de que hayas actualizado la caché de paquetes. También podrás ver el repositorio recién añadido al final de la lista en Preferencias del sistema -> Software y actualizaciones -> Otro software:

Ubuntu Software & Updates Screenshot

Nota: Se han reportado errores de SSL en Ubuntu: `Server access Error: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterExcepti the trustAnchors parameter must be non-empty url=https://repo1.maven.org/maven2/org/scala-sb` los cuales aparentemente impiden a OpenJDK 9 utilizar el formato PKCS12 para `/etc/ssl/certs/java/cacerts` cert-bug. Según https://stackoverflow.com/a/50103533/3827 esto ha sido arreglado en Ubuntu

Cosmic (18.10) aunque Ubuntu Bionic LTS (18.04) aún sigue esperando una release. Mira las respuesta para encontrar soluciones.

Red Hat Enterprise Linux y otras distribuciones basadas en RPM

Los paquetes RPM son oficialmente soportados por sbt.

Red Hat Enterprise Linux y otras distribuciones basadas en RPM utilizan el formato RPM. Ejecuta lo siguiente desde el terminal para instalar `sbt` (necesitarás tener privilegios de administrador para hacerlo, de ahí el `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
```

> **Nota:** Por favor, reporta cualquier problema con estos paquetes al proyecto sbt

Gentoo

El árbol oficial contiene ebuilds para sbt. Para instalar la última versión disponible escribe:

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

sbt mediante ejemplos

Esta página supone que has instalado sbt 1.

Empecemos mostrando algunos ejemplos en lugar de explicar cómo o por qué sbt funciona.

Crear una construcción sbt mínima

```
$ mkdir foo-build
$ cd foo-build
```

```
$ touch build.sbt
```

Iniciar el shell de sbt

```
$ sbt
[info] Updated file /tmp/foo-build/project/build.properties: set sbt.version to 1.1.4
[info] Loading project definition from /tmp/foo-build/project
[info] Loading settings from build.sbt ...
[info] Set current project to foo-build (in build file:/tmp/foo-build/)
[info] sbt server started at local:///Users/eed3si9n/.sbt/1.0/server/abc4fb6c89985a00fd95/so
sbt:foo-build>
```

Salir del shell de sbt

Para salir del shell de sbt, escribe `exit` o pulsa Ctrl+D (Unix) o Ctrl+Z (Windows).

```
sbt:foo-build> exit
```

Compilar un proyecto

Como convención, usaremos el prompt `sbt:...>` o `>` para indicar que estamos en un shell de sbt interactivo.

```
$ sbt
sbt:foo-build> compile
```

Recompilar cuando el código cambie

Si prefijas el comando `compile` (o cualquier otro comando) con `~` harás que dicho comando sea re-ejecutado automáticamente en cuanto uno de los ficheros fuente dentro del proyecto sea modificado. Por ejemplo:

```
sbt:foo-build> ~compile
[success] Total time: 0 s, completed May 6, 2018 3:52:08 PM
1. Waiting for source changes... (press enter to interrupt)
```

Crear un fichero fuente

Deja el comando anterior ejecutándose. Desde un shell diferente (o desde tu gestor de ficheros) crea la siguiente estructura de directorios `src/main/scala/example` en el directorio del proyecto. Después, crea el fichero `Hello.scala` en el directorio `example` utilizando tu editor de texto favorito con este contenido:

```
package example

object Hello extends App {
  println("Hello")
}
```

Este nuevo fichero debería de ser detectado por el comando en ejecución:

```
[info] Compiling 1 Scala source to /tmp/foo-build/target/scala-2.12/classes ...
[info] Done compiling.
[success] Total time: 2 s, completed May 6, 2018 3:53:42 PM
2. Waiting for source changes... (press enter to interrupt)
```

Pulsa `Intro` para salir de `~compile`.

Ejecutar un comando previo

Desde el shell de sbt, pulsa la tecla arriba dos veces para encontrar el comando `compile` que habías ejecutado al principio.

```
sbt:foo-build> compile
```

Obtener ayuda

Usa el comando `help` para obtener ayuda básica sobre los comandos disponibles.

```
sbt:foo-build> help

  about      Displays basic information about sbt and the build.
  tasks      Lists the tasks defined for the current project.
  settings   Lists the settings defined for the current project.
  reload     (Re)loads the current project or changes to plugins project or returns from it.
  new        Creates a new sbt build.
  projects   Lists the names of available projects or temporarily adds/removes extra builds
  project    Displays the current project or changes to the provided `project`.

....
```

Mostrar la descripción de una tarea específica

```
sbt:foo-build> help run
Runs a main class, passing along arguments provided on the command line.
```

Ejecutar tu aplicación

```
sbt:foo-build> run
[info] Packaging /tmp/foo-build/target/scala-2.12/foo-build_2.12-0.1.0-SNAPSHOT.jar ...
[info] Done packaging.
[info] Running example.Hello
Hello
[success] Total time: 1 s, completed May 6, 2018 4:10:44 PM
```

Establecer `ThisBuild / scalaVersion` desde el shell de sbt

```
sbt:foo-build> set ThisBuild / scalaVersion := "2.13.6"
[info] Defining ThisBuild / scalaVersion
```

Comprobar la entrada `scalaVersion`:

```
sbt:foo-build> scalaVersion
[info] 2.13.6
```

Guardar la sesión actual en build.sbt

Podemos guardar la configuración temporal utilizando `session save`.

```
sbt:foo-build> session save
[info] Reapplying settings...
```

El fichero `build.sbt` ahora debería de contener:

```
ThisBuild / scalaVersion := "2.13.6"
```

Dar un nombre a tu proyecto

Utilizando un editor, modifica `build.sbt` con el siguiente contenido:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

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

Recargar la construcción

Usa el comando `reload` para recargar la construcción. El comando hace que el fichero `build.sbt` sea releído y su configuración aplicada.

```
sbt:foo-build> reload
[info] Loading project definition from /tmp/foo-build/project
[info] Loading settings from build.sbt ...
[info] Set current project to Hello (in build file:/tmp/foo-build/)
sbt:Hello>
```

Fíjate en cómo el prompt ha cambiado a `sbt:Hello>`.

Añadir ScalaTest a `libraryDependencies`

Utilizando un editor, modifica `build.sbt` de la siguiente manera:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.7" % Test,
  )
```

Usa el comando `reload` para reflejar los cambios de `build.sbt`.

```
sbt:Hello> reload
```

Lanzar tests

```
sbt:Hello> test
```

Lanzar tests incrementales continuamente

```
sbt:Hello> ~testQuick
```

Escribir un test

Con el comando anterior ejecutándose, crea un fichero llamado `src/test/scala/HelloSpec.scala` utilizando un editor:

```
import org.scalatest.funsuite._

class HelloSpec extends AnyFunSuite {
```

```
  test("Hello should start with H") {
    assert("hello".startsWith("H"))
  }
}
```

`~testQuick` debería de coger el cambio:

```
2. Waiting for source changes... (press enter to interrupt)
[info] Compiling 1 Scala source to /tmp/foo-build/target/scala-2.12/test-classes ...
[info] Done compiling.
[info] HelloSpec:
[info] - Hello should start with H *** FAILED ***
[info]   assert("hello".startsWith("H"))
[info]          |       |          |
[info]          "hello" false      "H" (HelloSpec.scala:5)
[info] Run completed in 135 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed tests:
[error]   HelloSpec
[error] (Test / testQuick) sbt.TestsFailedException: Tests unsuccessful
```

Hacer que el test pase

Utilizando un editor, cambia `src/test/scala/HelloSpec.scala` a:

```
import org.scalatest.funsuite._

class HelloSpec extends AnyFunSuite {
  test("Hello should start with H") {
    // Hello, as opposed to hello
    assert("Hello".startsWith("H"))
  }
}
```

Confirma que el test pasa, luego pulsa `Intro` para salir del test continuo.

Añadir una dependencia de biblioteca

Utilizando un editor, modifica `build.sbt` de esta forma:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

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

```
  .settings(
    name := "Hello",
    libraryDependencies += "com.typesafe.play" %% "play-json" % "2.9.2",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.7" % Test,
  )
```

Usa el comando `reload` para reflejar los cambios de `build.sbt`.

Usar el REPL de Scala

Podemos averiguar qué tiempo hace actualmente en Nueva York.

```
sbt:Hello> console
[info] Starting scala interpreter...
Welcome to Scala 2.12.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_171).
Type in expressions for evaluation. Or try :help.

scala> :paste
// Entering paste mode (ctrl-D to finish)

import scala.concurrent._, duration._
import gigahorse._, support.okhttp.Gigahorse
import play.api.libs.json._

Gigahorse.withHttp(Gigahorse.config) { http =>
  val baseUrl = "https://www.metaweather.com/api/location"
  val rLoc = Gigahorse.url(baseUrl + "/search/").get.
    addQueryString("query" -> "New York")
  val fLoc = http.run(rLoc, Gigahorse.asString)
  val loc = Await.result(fLoc, 10.seconds)
  val woeid = (Json.parse(loc) \ 0 \ "woeid").get
  val rWeather = Gigahorse.url(baseUrl + s"/$woeid/").get
  val fWeather = http.run(rWeather, Gigahorse.asString)
  val weather = Await.result(fWeather, 10.seconds)
  ({Json.parse(_: String)} andThen Json.prettyPrint)(weather)
}

// press Ctrl+D

// Exiting paste mode, now interpreting.

import scala.concurrent._
import duration._
import gigahorse._
import support.okhttp.Gigahorse
```

```
import play.api.libs.json._
res0: String =
{
  "consolidated_weather" : [ {
    "id" : 6446939314847744,
    "weather_state_name" : "Light Rain",
    "weather_state_abbr" : "lr",
    "wind_direction_compass" : "WNW",
    "created" : "2019-02-21T04:39:47.747805Z",
    "applicable_date" : "2019-02-21",
    "min_temp" : 0.48000000000000004,
    "max_temp" : 7.84,
    "the_temp" : 2.1700000000000004,
    "wind_speed" : 5.996333145703094,
    "wind_direction" : 293.12257757287307,
    "air_pressure" : 1033.115,
    "humidity" : 77,
    "visibility" : 14.890539250775472,
    "predictability" : 75
  }, {
    "id" : 5806299509948416,
    "weather_state_name" : "Heavy Cloud",
...

scala> :q // para salir
```

Crear un subproyecto

Cambia `build.sbt` como sigue:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.7" % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
  )
```

Usa el comando `reload` para reflejar los cambios de `build.sbt`.

Listar todos los subproyectos

```
sbt:Hello> projects
[info] In file:/tmp/foo-build/
[info]   * hello
[info]     helloCore
```

Compilar el subproyecto

```
sbt:Hello> helloCore/compile
```

Añadir ScalaTest al subproyecto

Cambia `build.sbt` como sigue:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += scalaTest % Test,
  )
```

Difundir comandos

Añade aggregate para que el comando enviado a `hello` sea difundido también a `helloCore`:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"

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

```
  .aggregate(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += scalaTest % Test,
  )
```

Tras un `reload`, `~testQuick` se ejecuta ahora en ambos subproyectos:

```
sbt:Hello> ~testQuick
```

Pulsa `Intro` para salir del test continuo.

Hacer que hello dependa de helloCore

Usa `.dependsOn(...)` para añadir dependencias sobre otros subproyectos. Además, movamos la dependencia de Gigahorse a `helloCore`.

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0",
    libraryDependencies += scalaTest % Test,
  )
```

Parsear JSON con Play JSON

Vamos a añadir Play JSON a `helloCore`.

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"
val gigahorse = "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0"
val playJson  = "com.typesafe.play" %% "play-json" % "2.9.2"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies ++= Seq(gigahorse, playJson),
    libraryDependencies += scalaTest % Test,
  )
```

Tras un `reload`, añade `core/src/main/scala/example/core/Weather.scala`:

```
package example.core

import gigahorse._, support.okhttp.Gigahorse
import scala.concurrent._, duration._
import play.api.libs.json._

object Weather {
  lazy val http = Gigahorse.http(Gigahorse.config)

  def weather: Future[String] = {
    val baseUrl = "https://www.metaweather.com/api/location"
    val locUrl = baseUrl + "/search/"
    val weatherUrl = baseUrl + "/%s/"
    val rLoc = Gigahorse.url(locUrl).get.
      addQueryString("query" -> "New York")
    import ExecutionContext.Implicits.global
    for {
      loc <- http.run(rLoc, parse)
      woeid = (loc \ 0  \ "woeid").get
```

```
      rWeather = Gigahorse.url(weatherUrl format woeid).get
      weather <- http.run(rWeather, parse)
    } yield (weather \\ "weather_state_name")(0).as[String].toLowerCase
  }

  private def parse = Gigahorse.asString andThen Json.parse
}
```

Ahora, cambia `src/main/scala/example/Hello.scala` como sigue:

```
package example

import scala.concurrent._, duration._
import core.Weather

object Hello extends App {
  val w = Await.result(Weather.weather, 10.seconds)
  println(s"Hello! The weather in New York is $w.")
  Weather.http.close()
}
```

Vamos a ejecutar la aplicación para ver si funciona:

```
sbt:Hello> run
[info] Compiling 1 Scala source to /tmp/foo-build/core/target/scala-2.12/classes ...
[info] Done compiling.
[info] Compiling 1 Scala source to /tmp/foo-build/target/scala-2.12/classes ...
[info] Packaging /tmp/foo-build/core/target/scala-2.12/hello-core_2.12-0.1.0-SNAPSHOT.jar ..
[info] Done packaging.
[info] Done compiling.
[info] Packaging /tmp/foo-build/target/scala-2.12/hello_2.12-0.1.0-SNAPSHOT.jar ...
[info] Done packaging.
[info] Running example.Hello
Hello! The weather in New York is mostly cloudy.
```

Añadir el plugin sbt-native-packager

Utilizando un editor, crea `project/plugins.sbt`:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.4")
```

Después cambia `build.sbt` como sigue para añadir `JavaAppPackaging`:

```
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"
val gigahorse = "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0"
```

```
val playJson  = "com.typesafe.play" %% "play-json" % "2.9.2"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .enablePlugins(JavaAppPackaging)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies ++= Seq(gigahorse, playJson),
    libraryDependencies += scalaTest % Test,
  )
```

Recargar y crear una distribución .zip

```
sbt:Hello> reload
...
sbt:Hello> dist
[info] Wrote /tmp/foo-build/target/scala-2.12/hello_2.12-0.1.0-SNAPSHOT.pom
[info] Wrote /tmp/foo-build/core/target/scala-2.12/hello-core_2.12-0.1.0-SNAPSHOT.pom
[info] Your package is ready in /tmp/foo-build/target/universal/hello-0.1.0-SNAPSHOT.zip
```

Así es cómo puedes ejecutar la app una vez empaquetada:

```
$ /tmp/someother
$ cd /tmp/someother
$ unzip -o -d /tmp/someother /tmp/foo-build/target/universal/hello-0.1.0-SNAPSHOT.zip
$ ./hello-0.1.0-SNAPSHOT/bin/hello
Hello! The weather in New York is mostly cloudy.
```

Dockerizar tu app

```
sbt:Hello> Docker/publishLocal
....
[info] Successfully built b6ce1b6ab2c0
[info] Successfully tagged hello:0.1.0-SNAPSHOT
[info] Built image hello:0.1.0-SNAPSHOT
```

Así es cómo puedes ejecutar la app Dockerizada:

```
$ docker run hello:0.1.0-SNAPSHOT
```

```
Hello! The weather in New York is mostly cloudy
```

Establecer la versión

Cambia `build.sbt` como sigue:

```
ThisBuild / version      := "0.1.0"
ThisBuild / scalaVersion := "2.13.6"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.2.7"
val gigahorse = "com.eed3si9n" %% "gigahorse-okhttp" % "0.5.0"
val playJson  = "com.typesafe.play" %% "play-json" % "2.9.2"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .enablePlugins(JavaAppPackaging)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies ++= Seq(gigahorse, playJson),
    libraryDependencies += scalaTest % Test,
  )
```

Cambiar `scalaVersion` temporalmente

```
sbt:Hello> ++2.12.14!
[info] Forcing Scala version to 2.12.14 on all projects.
[info] Reapplying settings...
[info] Set current project to Hello (in build file:/tmp/foo-build/)
```

Comprueba la entrada `scalaVersion`:

```
sbt:Hello> scalaVersion
[info] helloCore / scalaVersion
[info]  2.12.14
[info] scalaVersion
[info]  2.12.14
```

Esta entrada se esfumará tras un `reload`.

Inspeccionar la tarea `dist`

Para saber más acerca de `dist`, prueba `help` e `inspect`.

```
sbt:Hello> help dist
Creates the distribution packages.
sbt:Hello> inspect dist
```

Para llamar a `inspect` recursivamente en las tareas dependientes utiliza `inspect tree`.

```
sbt:Hello> inspect tree dist
[info] dist = Task[java.io.File]
[info]   +-Universal / dist = Task[java.io.File]
....
```

Modo por lotes

También puedes ejecutar sbt en por lotes, pasando comandos de sbt directamente desde el terminal.

```
$ sbt clean "testOnly HelloSpec"
```

Note: El modo por lotes implica levantar una JVM y JIT cada vez, por lo que **la construcción será mucho más lenta**. Para el desarrollo del día a día recomendamos utilizar el shell de sbt o tests continuos como `~testQuick`.

El comando `new`

Puedes utilizar el comando `new` para generar rápidamente una construcción para un simple "Hola mundo".

```
$ sbt new sbt/scala-seed.g8
....
A minimal Scala project.

name [My Something Project]: hello

Template applied in ./hello
```

Cuando se te pregunte por el nombre del proyecto escribe `hello`.

Esto creará un nuevo proyecto en un directorio llamado `hello`.

Créditos

Esta página está basada en el tutorial Essential sbt escrito por William "Scala William" Narmontas.

Estructura de directorios

Esta página supone que has instalado sbt y leído sbt mediante ejemplos.

Directorio base

En la terminología de sbt, el "directorio base" es el directorio que contiene el proyecto. Así pues, si has creado un proyecto `hello` que contiene `/tmp/foo-build/build.sbt` tal y como se muestra en sbt mediante ejemplos, el directorio base será `/tmp/foo-build`.

Código fuente.

sbt emplea de forma predeterminada la misma estructura de directorios utilizado por Maven para ficheros fuente (todas las rutas son relativas al directorio base):

```
src/
  main/
    resources/
       <ficheros del jar principal aquí>
    scala/
       <fuentes principales de Scala>
    scala-2.12/
       <main Scala 2.12 specific sources>
    java/
       <fuentes principales de Java>
  test/
    resources
       <ficheros del jar de test aquí>
    scala/
       <fuentes de test de Scala>
    scala-2.12/
       <test Scala 2.12 specific sources>
    java/
       <fuentes de test de Java>
```

Cualquier otro directorio en `src/` es ignorado. También son ignorados todos los directorios ocultos.

El código fuente puede estar situado en el directorio base del proyecto como `hello/app.scala`, lo cual puede estar bien para proyectos pequeños, aunque para proyectos normales la gente suele estructurar los proyectos en el directorio `src/main/` para tener las cosas ordenadas.

El hecho de que se permita código fuente del tipo `*.scala` en el directorio base puede parecer algo raro, pero veremos que este hecho es relevante más adelante.

Ficheros de definición de construcción

La definición de construcción es descrita en `build.sbt` (en realidad cualquier fichero llamado `*.sbt`) en el directorio base del proyecto.

```
build.sbt
```

Ficheros auxiliares

Además de `build.sbt`, el directorio `project` puede contener ficheros `.scala` que definen objetos auxiliares y plugins puntuales.

Para más información mira Organizar la construcción.

```
build.sbt
project/
  Dependencies.scala
```

Puede que veas ficheros `.sbt` dentro de `project/` pero no son equivalentes a los ficheros `.sbt` del directorio base del proyecto. La explicación a esto vendrá más adelante, ya que primero necesitarás conocer algunos conceptos.

Construir productos

Los ficheros generados (clases compiladas, jars empaquetados, ficheros gestionados, caches y documentación) son escritos en el directorio `target` de forma predeterminada.

Configurar el control de versiones

Tu `.gitignore` (o el equivalente para otros sistemas de control de versiones) debería de contener:

```
target/
```

Fíjate en que deliberadamente acaba con `/` (para coincidir sólo con directorios) y que deliberadamente no empieza con `/` (para coincidir con `project/target/` además del `target/` en la raíz).

Ejecución

Esta página describe cómo usar sbt una vez has configurado tu proyecto. Se supone que has instalado sbt y leído sbt mediante ejemplos.

El shell de sbt

Lanza sbt en el directorio de tu proyecto sin argumentos:

```
$ sbt
```

Ejecutar sbt sin argumentos hace que se inicie un shell de sbt. El shell de sbt tiene un prompt de comandos (¡con autocompletado e historial!).

Por ejemplo, podrías escribir `compile` en el shell:

```
> compile
```

Para lanzar `compile` de nuevo pulsa la tecla arriba y pulsa `Intro`.

Para ejecutar tu programa, escribe `run`.

Para salir del shell escribe `exit` o usa Ctrl+D (Unix) o Ctrl+Z (Windows).

Modo por lotes

Puedes tambien ejecutar sbt en modo por lotes, proporcionando una lista de comandos sbt separados por espacio como argumentos. Aquellos comandos de sbt que necesiten argumentos pueden ser pasados como un único argumento entrecomillado, por ejemplo:

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

En este ejemplo, `testOnly` tiene como argumentos `TestA` y `TestB`. Los comandos son ejecutados secuencialmente: (`clean`, `compile` y luego `testOnly`).

Note: El modo por lotes implica levantar una JVM y JIT cada vez, por lo que **la construcción será mucho más lenta**. Para el desarrollo del día a día recomendamos utilizar el shell de sbt o construir y testear continuamente tal y como se explica más abajo.

A partir de sbt 0.13.16, se lanza un mensaje informativo cuando sbt se utiliza en el modo por lotes.

```
$ sbt clean compile
[info] Executing in batch mode. For better performance use sbt's shell
...
```

Sólo será mostrado con `sbt compile`. Puede ser desactivado con `suppressSbtShellNotification := true`.

Construir y testear continuamente

Para acelerar el ciclo editar-compilar-testear puedes indicarle a sbt que recompile automáticamente o lance tests cada vez que guardes un fichero fuente.

Haz que un comando se ejecute cuando uno o más ficheros fuente cambien prefijando dicho comando con `~`. Por ejemplo, prueba en el shell de sbt:

```
> ~testQuick
```

Pulsa `Intro` para dejar de observar los cambios.

Puedes utilizar el prefijo `~` tanto en el shell de sbt como en modo por lotes.

Para más información mira Ejecución disparada.

Comandos comunes

A continuación presentamos una lista con algunos de los comandos más comunes de sbt. Para una lista más completa mira Referencia de línea de comandos.

clean

Borra todos los ficheros generados (en el directorio target).

compile

Compila los ficheros fuente principales (en los directorios src/main/scala y src/main/java).

test

Compila y ejecuta todos los tests.

console

Inicia el intérprete de Scala con un classpath que incluye los ficheros fuente compilados y todas sus dependencias. Para volver a sbt, escribe :quit o pulsa Ctrl+D (Unix) o Ctrl+Z (Windows).

run <argument>*

Ejecuta la clase principal del proyecto en la misma máquina virtual que sbt

package

Crea un fichero jar conteniendo los ficheros de src/main/resources y las clases compiladas de src/main/scala y src/main/java.

help <comando>

Muestra ayuda detallada para el comando especificado. Si no se proporciona ningún comando entonces mostrará una breve descripción de cada comando.

reload

Recarga la definición de construcción (los ficheros build.sbt, project/*.scala, project/*.sbt). Necesario si cambias la definición de construcción.

Autocompletado

El shell de sbt tiene autocompletado, incluso con un prompt vacío. Una convención especial de sbt es que si se presiona tab una vez se se mostrarán las opciones más probables mientras que si se pulsa más veces se mostrarán aún más opciones.

Comandos históricos

El shell de sbt recuerda el histórico de comandos, incluso si sales de sbt y lo reinicias. La forma más sencilla de acceder al histórico es con la tecla arriba. Los siguientes comandos están soportados:

!

Muestra la ayuda para comandos históricos.

!!

Ejecuta el comando previo otra vez.

!:

Muestra todos los comandos escritos hasta el momento.

!:n

Muestra los últimos n comandos.

!n

Ejecuta el comando con índice n, como se muestra en el comando !:.

!-n

Ejecuta el n-ésimo comando anterior a este.

!string

Ejecuta el comando más reciente que empiece con 'string.'

!?string

Ejecuta el comando más reciente que contenga 'string.'

Definiciones de construcción

Esta página describe las definiciones de construcción (build definitions), incluyendo algo de "teoría" y la sintaxis de `build.sbt`. Se supone que has instalado una versión reciente de sbt, como sbt 1.5.5, que sabes cómo usar sbt y que has leído las páginas anteriores de la Guía de inicio.

Esta página explica la definición de construcción de `build.sbt`.

Especificar la versión de sbt

Como parte de tu definición de construcción debes de especificar la versión de sbt que tu construcción utiliza.

Esto permitirá a la gente con diferentes versiones del lanzador de sbt construir los mismos proyectos con resultados consistentes. Para hacer esto, crea un fichero llamado `project/build.properties` en el que se especifica la versión de sbt como sigue:

```
sbt.version=1.5.5
```

Si la versión requerida no está disponible localmente, el lanzador `sbt` se la descargará por ti. Si este fichero no está presente, el lanzador `sbt` eligirá una versión arbitraria, lo cual no es aconsejable debido a que hará que tu construcción no sea portable.

¿Qué es una definición de construcción?

Una *definición de construcción* es definida en `build.sbt` y consiste en un conjunto de proyectos (de tipo `Project`). Debido a que el término *project* puede ser ambiguo, con frecuencia utilizaremos el *subproyecto* para referirnos a ellos en esta guía.

Por ejemplo, en `build.sbt` se define el subproyecto ubicado en el directorio actual así:

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.7"
  )
```

Cada subproyecto es configurado como pares clave-valor.

Por ejemplo, una clave es `name` y se mapea a una cadena de texto, el nombre de tu subproyecto. Los pares clave-valor se listan bajo el método `.settings(...)` de tal forma:

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.7"
  )
```

Cómo define build.sbt configuraciones

`build.sbt` define subproyectos, los cuales contienen una secuencia de pares clave-valor llamados *expresiones de configuración* (expression settings) utilizando un *DSL de build.sbt*

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.14"
ThisBuild / version      := "0.1.0-SNAPSHOT"

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

Echemos un vistazo más de cerca al DSL de build.sbt:

setting expression

Cada entrada es llamada una *expresion de configuración*. De entre ellas, algunas son también llamadas expresiones de tareas. Veremos la diferencia entre ellas más tarde en esta página.

Una expresión de configuración consiste en tres partes:

1. La parte izquierda, que es una *clave*.
2. Un *operador*, que en este caso es `:=`
3. La parte derecha es llamada *cuerpo* o *cuerpo de configuración*.

A la izquierda, `name`, `version` y `scalaVersion` son *claves*. Una clave es una instancia de `SettingKey[T]`, `TaskKey[T]` o `InputKey[T]` donde `T` es el tipo esperado. Los tipos de claves son explicados más abajo.

Debido a que la clave `name` tiene tipo `SettingsKey[String]`, el operador `:=` aplicado a `name` también está tipado como `String`. Si usas el tipo incorrecto la definición de construcción no compilará:

```
lazy val root = (project in file("."))
  .settings(
    name := 42  // no compila
  )
```

En `build.sbt` también se pueden entremezclar `val`, `lazy val` y `def`. Los `object` y `class` de primer nivel no están permitidos en `build.sbt`. De hacerlo deberían ir en el directorio `project/` como ficheros fuente de Scala.

Claves

Tipos

Hay tres sabores de claves:

- `SettingKey[T]`: una clave para un valor calculado una única vez (el valor es calculado cuando se carga un subproyecto)
- `TaskKey[T]`: una clave para un valor, llamado una *tarea* que ha de ser recalculado cada vez, potencialmente con efectos colaterales.
- `InputKey[T]`: una clave para una tarea con argumentos de línea de comandos como entrada. Para más información mira Tareas de entrada.

Claves preconfiguradas

Las claves preconfiguradas son simplemente campos de un objeto llamado Keys. Un `build.sbt` tiene de forma implícita un `import sbt.Keys._`, por lo que `sbt.Keys.name` puede ser accedido como `name`.

Claves personalizadas

Las claves personalizadas pueden ser definidas con sus respectivos métodos de creación: `settingKey`, `taskKey` e `inputKey`. Cada método espera el tipo del valor asociado con la clave además de su descripción. El nombre de la clave es tomado del `val` al cual la clave es asignada. Por ejemplo, para definir una clave para una nueva tarea llamada `hello`,

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

Aquí hemos usado el hecho de que un fichero `.sbt` puede contener `val` y `def` además de configuración. Todas estas definiciones son evaluadas antes de la configuración sin importar en qué lugar del fichero han sido definidas.

> **Nota:** Tipicamente, se usan `lazy val` en lugar de `val` para evitar problemas de orden durante la inicialización.

Claves de tarea vs claves de entradas

Se dice de `TaskKey[T]` que define una *tarea*. Las tareas son operaciones tales como `compile` o `package`. A su vez pueden devolver `Unit` (`Unit` es el tipo de Scala para `void`) o pueden devolver un valor relacionado con la tarea, por ejemplo el de `package` es un `TaskKey[File]` y su valor es el fichero jar que crea.

Cada vez que inicias la ejecución de una tarea, por ejemplo escribiendo `compile` en el prompt interactivo de sbt, sbt volverá a ejecutar cualesquiera tareas implicadas exactamente una vez.

Los pares clave-valor de sbt que describen al subproyecto pueden ser almacenados en forma de cadena fija para configuraciones tales como `name`, pero tienen que poder guardar código ejecutable para tareas tales como `compile`. Incluso si dicho código ejecutable acaba devolviendo una cadena, tiene que ser ejecutado cada vez.

Una cierta clave siempre se refiere o bien a una tarea o a una entrada plana. El que tenga que ser ejecutada cada vez o no es una propiedad de la clave, no del valor.

Definir tareas y entradas de configuración

Utilizando `:=` se puede asignar un valor a una entrada y una computación a una tarea. Para una entrada, el valor será computado una única vez durante la carga del proyecto. Para una tarea, la computación será evaluada cada vez que la tarea sea ejecutada.

Por ejemplo, para implementar la tarea `hello` de la sección anterior:

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

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

Ya vimos un ejemplo sobre cómo definir entradas cuando definimos el nombre del proyecto:

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

Tipos para tareas y entradas

Desde una perspectiva del sistema de tipos, el `Setting` creado a partir de una clave tarea es ligeramente diferente de la creada para una clave entrada. `taskKey := 42` resulta en `Setting[Task[T]]` mientras que `settingKey := 42` resulta en `Setting[T]`. En la mayoría de los casos esto no supone ninguna diferencia; la clave tarea aún sigue creando un valor de tipo `T` cuando la tarea es ejecutada.

La diferencia de tipos entre `T` y `Task[T]` tiene la siguiente implicación: una entrada no puede depender de una tarea, debido a que una entrada es evaluada una única vez durante la carga de un proyecto y no es re-evaluada. Más información sobre esto en Grafos de tareas.

Claves en el shell de sbt

En el shell de sbt, puedes escribir el nombre de cualquier tarea para ejecutar dicha tarea. Esta es la razón por la que al escribir `compile` se ejecuta la tarea `compile`. `compile` es una clave tarea.

Si escribes el nombre de una clave entrada en lugar de una clave tarea el valor de la clave entrada será mostrado. Escribir el nombre de una clave tarea ejecuta la tarea pero muestra el valor resultante. Para ver el resultado de una tarea utiliza `show <tarea>` en lugar de simplemente `<tarea>`. La convención para nombres de claves es utilizar `camelCase` por lo que el nombre de linea de comandos y los identificadores de Scala son los mismos.

Para aprender más acerca de cualquier clave escribe `inspect <clave>` en el prompt interactivo de sbt. Alguna de la información que `inspect` muestra no tendrá sentido (de momento), pero al principio de todo mostrará el tipo del valor y una breve descripción de esa entrada.

Importaciones en build.sbt

Puedes incluir sentencias `import` al principio de `build.sbt`. No necesitan estar separadas por líneas en blanco.

Existen algunas importaciones implícitas predeterminadas:

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

(Además, si tienes autoplugins, los nombres marcados bajo `autoImport` serán importados.)

Definiciones de construcción .sbt planas

La configuración puede ser escrita directamente en el fichero `build.sbt` en lugar de ponerla dentro de la llamada a `.settings(...)`. Podemos llamar a esto el "estilo plano".

```
ThisBuild / version := "1.0"
ThisBuild / scalaVersion := "2.12.14"
```

Esta sintaxis es la recomendada para configuraciones con ámbito `ThisBuild` plugins añadidos. Mira secciones posteriores acerca de los ámbitos y los plugins.

Añadir dependencias de biblioteca

Para depender de bibliotecas de terceros hay dos opciones. La primera es copiar jars en `lib/` (dependencias no gestionadas) y la otra es añadir dependencias gestionadas, que en `build.sbt` tienen este aspecto:

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

ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.14"
```

```
ThisBuild / version      := "0.1.0-SNAPSHOT"

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

Así es como se puede añadir una dependencia gestionada de la biblioteca Apache Derby versión 10.4.1.3.

La clave `libraryDependencies` posee dos complejidades: `+=` en lugar de `:=` y el método `%`. `+=` añade al valor antiguo de la clave en lugar de reemplazarlo, como se explica en Grafos de tareas. El método `%` es utilizado para construir un identificador de módulo Ivy a partir de cadenas, como se explica en Dependencias de bibliotecas.

De momento nos saltaremos los detalles de las dependencias de bibliotecas hasta más tarde en la Guía de inicio. Hay una página entera que habla de ellas más tarde.

Construcciones multiproyecto

Esta página introduce múltiples subproyectos en una única construcción.

Por favor, lee primero las páginas anteriores de la Guía de inicio, en particular necesitarás entender el fichero build.sbt antes de leer esta página.

Múltiples subproyectos

Puede ser útil mantener múltiples subproyectos relacionados en una única construcción, especialmente si unos dependen de otros y sueles modificarlos todos a la vez.

Cada subproyecto en una construcción tiene sus propios directorios de código fuente, genera su propio fichero jar cuando ejecutas `package` y en general funcionan como cualquier otro proyecto.

Un proyecto se define declarando un `lazy val` de tipo Project. Por ejemplo:

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

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

El nombre del `val` es usado como ID de subproyecto, el cual es usado para referirse al subproyecto en el shell de sbt.

Opcionalmente, el directorio base puede ser omitido si es el mismo que el nombre del val.

```
lazy val util = project

lazy val core = project
```

Configuración común de construcción

Para extraer configuración común a lo largo de múltiples proyectos puedes definir la configuración en el ámbito de `ThisBuild`. Para ello la parte derecha tiene que ser un valor puro o una configuración en el ámbito de `Global` o `ThisBuild` y no puede haber configuraciones predeterminadas en el ámbito de subproyectos. (Ver Ámbitos)

```
ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.14"

lazy val core = (project in file("core"))
  .settings(
    // otras configuraciones
  )

lazy val util = (project in file("util"))
  .settings(
    // otras configuraciones
  )
```

Ahora podemos incrementar `version` en un único lugar y se verá reflejado a lo largo de los subproyectos cuando recargues la construcción.

Configuración común

Otra forma de extraer configuración común a lo largo de múltiples proyectos es crear una secuencia llamada `commonSettings` y llamar al método `settings` en cada proyecto.

```
lazy val commonSettings = Seq(
  target := { baseDirectory.value / "target2" }
)

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

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

```
    commonSettings,
    // otras configuraciones
  )
```

Dependencias

Los proyectos en la construcción pueden ser completamente independientes uno de otro, pero por lo general estarán relacionados mediante algún tipo de dependencia. Hay dos tipos de dependencias: agregadas y de classpath.

Agregación

Una agregación implica que al ejecutar una tarea en el proyecto que agrega también será ejecutada en los proyectos agregados. Por ejemplo:

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

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

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

En el ejemplo de arriba, el proyecto raíz agrega `util` y `core`. Si ejecutas sbt con dos subproyectos como los del ejemplo e intentas compilar podrás ver cómo los tres proyectos son compilados.

En el proyecto que hace la agregación, el proyecto raíz en este caso, puedes controlar la agregación a nivel de tarea. Por ejemplo, para evitar agregar la tarea `update`:

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

[...]
```

`update / aggregate` es la clave agregada en el ámbito de la tarea `update` (ver Ámbitos).

Nota: La agregación ejecutará las tareas agregadas en paralelo y sin orden predeterminado entre ellas.

Dependencias de classpath

Un proyecto puede depender del código de otro proyecto. Esto se hace añadiendo una llamada al método `dependsOn`. Por ejemplo, si `core` necesita `util` en su classpath deberías de definir `core` así:

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

A partir de ahora el código de `core` puede utilizar las clases de `util`. Esto crea además un orden entre proyectos a la hora de compilarlos. `util` deberá ser actualizado y compilado antes de que `core` pueda ser compilado.

Para depender de múltiples proyectos puedes utilizar múltiples argumentos en `dependsOn`, como `dependsOn(bar, baz)`.

Dependencias de classpath por configuración

`core dependsOn(util)` implica que la configuración de `compile` en `core` dependerá de la configuración de `compile` en `util`. Esto se puede escribir de forma más explícita como `dependsOn(util % "compile->compile")`.

La `->` en `"compile->compile"` significa "depende de", por lo que `"test->compile"` significa que la configuración de `test` en `core` depende de la configuración de `compile` en `util`.

Omitir la parte de `->config` implica `->compile`, por lo que `dependsOn(util % "test")` significa que la configuración de `test` en `core` depende de la configuración de `Compile` en `util`.

Una declaración útil es `"test->test"` que significa que `test` depende de `test`. Esto permite poner código auxiliar para testear en `util/src/test/scala` y luego usar dicho código en `core/src/test/scala`, por ejemplo.

Puedes declarar múltiples configuraciones para una dependencia, separadas por punto y coma. Por ejemplo, `dependsOn(util % "test->test;compile->compile")`.

Dependencias inter-proyecto

En proyectos extremadamente grandes con muchos ficheros y muchos subproyectos, el rendimiento de sbt puede ser menos óptimo al tener que observar qué ficheros han cambiado en una sesión interactiva por tener que realizar muchas operaciones de E/S.

sbt posee las entradas `trackingInternalDependencies` y `exportToInternal`. Éstas pueden ser utilizadas para controlar si la compilación de subproyectos dependientes ha de ser lanzada automáticamente o no cuando se llama a `compile`. Ambas claves pueden tomar uno de estos tres valores: `TrackLevel.NoTracking`, `TrackLevel.TrackIfMissing` y `TrackLevel.TrackAlways`. De forma predeterminada ambas son establecidas a `TrackLevel.TrackAlways`.

Cuando `trackInternalDependencies` es establecido a `TrackLevel.TrackIfMissing`, sbt no volverá a intentar compilar dependencias internas (inter-proyecto) automáticamente, a menos que no haya ficheros `*.class` (o un fichero JAR cuando `exportJars` sea `true`) en el directorio de salida.

Cuando la entrada es establecida a `TrackLevel.NoTracking` la compilación de dependencias internas es omitida. Fíjate en que el classpath aún sigue siendo anexado y que el grafo de dependencias aún sigue mostrándolas como dependencias. La razón es ahorrar el sobrecoste de E/S para observar cambios en una construcción con muchos subproyectos durante el desarrollo. A continuación se muestra cómo establecer todos los subproyectos a `TrackIfMissing`.

```
lazy val root = (project in file(".")).
  aggregate(....).
  settings(
    inThisBuild(Seq(
      trackInternalDependencies := TrackLevel.TrackIfMissing,
      exportJars := true
    ))
  )
```

La entrada `exportToInternal` permite al proyecto del cual se depende optar si puede ser vigilado internamente o no, lo cual puede resultar útil si lo que se quiere es hacer seguimiento de la mayoría de los subproyectos excepto unos cuantos. La intersección de las entradas `trackInternalDependencies` y `exportToInternal` será usada para determinar el nivel de seguimiento real. A continuación se muestra un ejemplo de un proyecto optando de ser seguido o no:

```
lazy val dontTrackMe = (project in file("dontTrackMe")).
  settings(
    exportToInternal := TrackLevel.NoTracking
  )
```

Proyecto raíz predeterminado

Si un proyecto no está definido para el directorio raíz en la construcción, sbt creará uno de forma predeterminada que agrega a los otros proyectos de la construcción.

Debido a que el proyecto `hello-foo` ha sido definido con `base = file("foo")`, él estará contenido en el subdirectorio `foo`. Sus fuentes pueden estar tanto en `foo`, como `foo/Foo.scala` o en `foo/src/main/scala`. La estructura de directorios habitual se aplica a `foo` a excepción de los ficheros de definición de construcción.

Navegando por los proyectos interactivamente

En el prompt interactivo de sbt, escribe `proyectos` para listar tus proyectos y `project <projectname>` para seleccionar el proyecto actual. Al ejecutar una tarea como `compile` ésta se ejecutará sobre el proyecto actual. Por eso no hay por qué compilar el proyecto raíz necesariamente, es posible compilar solamente un subproyecto.

Puedes ejecutar una tarea en otro proyecto especificando explícitamente el ID de proyecto, como en `subproyecto/compile`.

Código común

Las definiciones en los ficheros `.sbt` no son visibles en otros ficheros `.sbt`. Para poder compartir código entre ficheros `.sbt` hay que definir uno o más ficheros de Scala en el directorio `project/` en la construcción raíz.

Para más información ver Organizando la construcción.

Appendix: Subproject build definition files

Cualquier fichero `.sbt` en `foo`, por ejemplo `foo/build.sbt`, será mezclado con la definición de construcción para la construcción principal, pero con ámbito del proyecto `hello-foo`.

Si todo tu proyecto está en `hello`, intenta definir una versión diferente (`version := "0.6"`) en `hello/build.sbt`, `hello/foo/build.sbt`, y `hello/bar/build.sbt`. Ahora `show version` en el prompt interactivo de sbt debería de tener este aspecto (respetando las versiones que hayas definido):

```
> 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` está definido en `hello/foo/build.sbt`, `hello-bar/*:version` está definido en `hello/bar/build.sbt` y `hello/*:version` está definido en `hello/build.sbt`. Recuerda la sintaxis para claves con ámbito. Cada clave `version` está en el ámbito de un proyecto, basado en la ubicación de `build.sbt`. Pero los tres `build.sbt` forman parte de la misma definición de construcción.

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.

No puedes tener un subdirectorio de proyecto o ficheros `project/*.scala` en los subproyectos. `foo/project/Build.scala` sería ignorado.

Grafos de tareas

Continuando con la definición de construcción, esta página explica la definición de `build.sbt` con más detalle.

En lugar de pensar en la configuración como pares clave-valor una analogía más apropiada sería pensar en ella como un *grafo acíclico dirigido* (GAD) de tareas donde los vértices significan **sucede antes de**. Lo llamaremos *grafo de tareas*.

Terminología

Repasemos los términos clave antes de seguir profundizando.

- Expresión de entrada/tarea: entrada dentro de `.settings(...)`.
- Clave: parte izquierda de una expresión. Puede ser de tipo `SettingKey[A]`, `TaskKey[A]` o `InputKey[A]`.
- Entrada: Definida por una expressión de entrada con `SettingKey[A]`. El valor es calculado una única vez durante la carga.
- Tarea: Definida por una expresión de tarea con `TaskKey[A]`. El valor es calculado cada vez que es invocado.

Declarando dependencia de otras tareas

En el DSL de `build.sbt` se utiliza el método `.value` para expresar una dependencia de otra tarea o entrada. El método `value` es especial y sólo puede ser llamado como argumento de `:=` (o `+=` o `++=`, los cuales veremos más adelante).

Como primer ejemplo supongamos que redefinimos `scalacOption` para que dependa de las tareas `update` y `clean`. A continuación se muestran las definiciones de estas claves (tal cual están definidas en Keys).

Nota: Los valores calculados abajo no tienen mucho sentido para `scalacOptions` pero sirven a modo de demostración.

```
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
```

A continuación se muestra cómo podemos redefinir `scalacOptions`:

```
scalacOptions := {
  val ur = update.value  // la tarea update sucede antes de scalacOptions
  val x = clean.value    // la tarea clean sucede antes de scalacOptions
  // ---- scalacOptions empieza aquí ----
  ur.allConfigurations.take(3)
}
```

`update.value` y `clean.value` declaran dependencias de tarea, mientras que `ur.allConfigurations.take(3)` es el cuerpo de la tarea.

`.value` no es un método de Scala normal. El DSL de `build.sbt` utiliza una macro para procesarlo fuera del cuerpo de la tarea. **Ambas tareas, `update` y `clean`, ya han sido completadas en el momento en el que el motor de tareas evalúa el cuerpo de `scalacOptions`, sin importar que esas líneas aparezcan en el cuerpo.**

Mira el siguiente ejemplo:

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.14"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalacOptions := {
      val out = streams.value // la tarea streams sucede antes de scalacOptions
      val log = out.log
      log.info("123")
      val ur = update.value   // la tarea update sucede antes de scalacOptions
      log.info("456")
      ur.allConfigurations.take(3)
    }
  )
```

Después, desde el shell de sbt si se escribe `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
```

Incluso aunque `val ur = ...` aparezca entre `log.info("123")` y `log.info("456")` la evaluación de la tarea `update` se ha realizado antes de tales líneas.

Aquí hay otro ejemplo:

```
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.14"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalacOptions := {
      val ur = update.value  // la tarea update sucede antes de scalacOptions
      if (false) {
        val x = clean.value  // la tarea clean sucede antes de scalacOptions
      }
      ur.allConfigurations.take(3)
    }
  )
```

Después, si desde el shell de sbt se lanza `run` y luego `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
```

Si compruebas `target/scala-2.12/classes/` verás que no existe ya que la tarea `clean` ha sido ejecutado incluso si aparece dentro de `if (false)`.

Otra cosa importante a tener en cuenta es que no hay garantías sobre el orden en el que las tareas `update` y `clean` son ejecutadas. Podría ejecutarse primero `update` y luego `clean`, primero `clean` y luego `update` o ambas ser ejecutadas en paralelo.

Llamadas a .value en línea

Como se ha explicado anteriormente, `.value` es un método especial que es usado para expresar dependencias de otras tareas y entradas. Hasta que te familiarices con `build.sbt` te recomendamos que pongas todas las llamadas `.value` al principio del cuerpo.

Sin embargo, a medida que te vayas sintiendo más cómodo, puedes optar por poner dichas llamadas a `.value` en línea ya que puede hacer que la tarea/entrada sea más concisa, sin tener que utilizar variables.

A continuación se muestran unos cuantos ejemplos de llamadas en línea.

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

Fíjate en que, aunque las llamadas a `.value` estén en línea o en cualquier parte del cuerpo de la tarea, siguen siendo evaluadas antes de entrar al cuerpo de la tarea.

Inspeccionar la tarea

En el ejemplo anterior, `scalacOptions` tiene una *dependencia* de las tareas `update` y `clean`. Si pones ese ejemplo en `build.sbt` y ejecutas la consola interactiva de sbt y luego escribes `inspect scalacOptions` deberías de ver algo similar (en parte):

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

Así es cómo sbt sabe qué tareas dependen de otras.

Por ejemplo, si se lanza `inspect tree compile` verás que depende de otra clave `incCompileSetup` que a su vez depende de otras claves como `dependencyClasspath`. Sigue recorriendo las dependencias y verás cómo ocurre la mágia.

```
> 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]   | | | | |
....
```

Cuando escribes `compile` sbt automáticamente realiza un `update`, por ejemplo. Esto funciona simplemente porque los valores de entrada requeridos por `compile` necesitan que sbt lance un `update` primero.

De esta forma, todas las dependencias de construcción en sbt son *automáticas* en lugar de tener que ser declaradas de forma explícita. Si usas el valor de una clave en otra computación entonces la computación dependerá de dicha clave.

Definir una tarea que depende de otra configuración

`scalacOptions` es una clave tarea. Supongamos que ya ha sido establecida a algún valor, pero que quieres filtrar `"-Xfatal-warnings"` y `"-deprecation"` para las versiones distintas a la 2.12.

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    organization := "com.example",
    scalaVersion := "2.12.14",
    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)
      }
    }
  )
```

A continuación se muestra cómo aparecería en el shell de sbt:

```
> 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
```

Ahora, cojamos estas dos claves (desde 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 for
```

Nota: `scalacOptions` y `checksums` no tienen nada que ver la una con la otra. Simplemente son dos claves con el mismo tipo de valor, donde una es una tarea.

Es posible compilar un `build.sbt` en donde `scalacOptions` hace referencia a `checksums`, pero no en el sentido contrario. Por ejemplo, lo siguiente está permitido:

```
// La tarea scalacOptions puede estar definida en
// terminos de la entrada checksums
scalacOptions := checksums.value
```

No hay forma de ir en la *otra* dirección. Es decir, una clave entrada no puede depender de una clave tarea. Esto se debe a que una clave entrada es computada una única vez cuando se carga el proyecto, por lo que una tarea no sería re-ejecutada cada vez y las tareas esperan justamente lo contrario.

```
// Mal ejemplo: La entrada checksums no puede ser definida en términos de la
// tarea scalacOptions
checksums := scalacOptions.value
```

Definir una entrada que depende de otras entradas

En términos de ejecución, podemos pensar en las entradas como un tipo especial de tarea que se evalúa durante la carga del proyecto.

Consideremos definir la organización del proyecto para que coincida con el nombre del proyecto.

```
// nombramos nuestra organizacón basándonos en el proyecto
// (ambos son SettingKey[String])
organization := name.value
```

A continuación se muestra un ejemplo más realista. Esto cambia el valor de la clave `Compile / scalaSource` a un directorio diferente sólo cuando `scalaBinaryVersion` es `"2.11"`.

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

¿Cuál es el propósito del DSL de build.sbt?

El DSL de `build.sbt` es un lenguage específico del dominio utilizado para construir un GAD de entradas y tareas. Las expresiones de la configuración construyen entradas, tareas y las dependencias entre ellas.

Esta estructura es común a Make (1976), Ant (2000) y Rake (2003).

Introducción a Make

La sintáxis básica de un Makefile tiene este aspecto:

```
objetivo: dependencias
[tab] comando 1
[tab] comando 2
```

En un Makefile el primer objetivo que aparece listado corresponde al objetivo predeterminado (por convenio se suele nombrar `all`).

1. Make comprueba si las dependencias del objetivo han sido construidas y construye cualesquiera dependencias que no hayan sido construídas aún.
2. Make ejecuta los comandos en orden.

Echemos un ojo a un `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 $@
```

Al ejecutar `make` sin parámetros se coge el primer objetivo listado (por convenio `all`). El objetivo lista `hello` como su dependencia, la cual aún no ha sido construida, por lo que Make construirá `hello`.

Después, Make comprueba si las dependencias del objetivo `hello` han sido construidas. `hello` tiene dos objetivos: `main.o` y `hello.o`. Una vez dichos objetivos han sido creados utilizando la última regla de concordancia de patrones el comando de sistema es ejecutado para enlazar `main.o` y `hello.o` en `hello`.

Cuando trabajas con `make` te puedes enfocar en qué objetivos quieres mientras que Make calcula la secuencia y el orden exacto de comandos que necesitan ser lanzados para construir los productos intermedios. Se puede considerar como programación orientada a dependencias o programanción basada en flujo. Make es en realidad considerado como un sistema híbrido porque mientras que su DSL describe las dependencias entre tareas las acciones son delegadas a comandos del sistema.

Rake

Este sistema híbrido es continuado por los sucesores de Make tales como Ant, Rake y sbt. Echemos un vistazo a la sintaxis básica para un Rakefile:

```
task name: [:prereq1, :prereq2] do |t|
  # acciones (puede referenciar prerequisitos tales como t.name, etc...)
end
```

La innovación hecha por Rake fue que usaba un lenguage de programación para describir las acciones en lugar de comandos del sistema.

Ventajas de la programación híbrida basada en flujo

Existen varias razones para organizar la construcción de esta forma.

La primera es evitar la duplicación. Con la programación basada en flujo una tarea es ejecutada una única vez incluso cuando de ella dependen múltiples tareas. Por ejemplo, incluso cuando múltiples tareas a lo largo del grafo de tareas

dependen de `compile in Compile` la compilación será ejecutada exactamente una única vez.

La segunda es la paralelización. Utilizando el grafo de tareas el motor de tareas puede programar tareas mutuamente no dependientes en paralelo.

La tercera es la separación de cometidos y la flexibilidad. El grafo de tareas permite al usuario cablear las tareas juntas de diferentes formas, mientras que sbt y los plugins pueden proporcionar varias características tales como compilación y gestión de dependencias de bibliotecas como funciones que pueden ser reutilizadas.

Resumen

La estructura central de las definiciones de construcción es un GAD de tareas donde los vértices denotan relaciones "sucede antes de". `build.sbt` es un DSL diseñado para expresar programación orientada a dependencias o programación basada en flujo, similar a un `Makefile` o `Rakefile`.

La razón clave para utilizar programación basada en flujo es evitar duplicaciones, procesar en paralelo y la personalización.

Ámbitos

Esta página explica los ámbitos. Se supone que has leído y entendido las páginas anteriores, Definiciones de construcción y Grafos de tareas.

Toda la historia sobre claves

Anteriormente fingimos que una clave tal como `name` correspondía a una única entrada en el mapa de pares clave-valor de sbt. Esto fue una simplificación.

En realidad cada clave puede tener un valor asociado en más de un contexto, llamado *ámbito*.

Algunos ejemplos concretos:

- si tienes múltiples proyectos (también llamados subproyectos) en tu definición de construcción una clave puede tener diferentes valores en cada proyecto.
- la clave `compile` puede tener un valor para tus fuentes principales y otro diferente para tus ficheros de test, si quisieras compilarlos de forma distinta.
- la clave `packageOptions` (que contiene opciones para crear paquetes jar) puede tener diferentes valores cuando se empaquetan ficheros de clases (`packageBin`) o paquetes de código fuente (`packageSrc`).

No existe un único valor para una clave `name` *data*, porque el valor puede diferir según el ámbito.

Sin embargo, existe un único valor para una cierta clave en un ámbito.

Si piensas en sbt cuando procesa una lista de entradas para generar un mapa de clave-valor que describan un proyecto, tal y como se explicó antes, las claves en ese mapa de clave-valor son claves con *ámbito*. Cada entrada definida en la definición de construcción (por ejemplo en `build.sbt`) se aplica también a una clave con ámbito.

Frecuentemente el ámbito está implícito o existe uno predeterminado, pero si el ámbito predeterminado no es el que te interesa deberás mencionar explícitamente el ámbito que deseas en `build.sbt`.

Ejes de ámbito

Un *eje de ámbito* es un constructor de tipo similar a `Option[A]` que es usado para formar un componente en un ámbito.

Existen tres ejes de ámbito:

- El eje de subproyecto
- El eje de configuración
- El eje de tareas

Si el concepto de *eje* no te resulta familiar podemos pensar en un cubo de color RGB como ejemplo:

color cube

En el modelo de color RGB todos los colores son representados por un punto en el cubo cuyos ejes corresponden a las componentes rojo, verde y azul codificadas por un número. De forma similar un ámbito total en sbt está formado por el valor de la **tupla** de un subproyecto, una configuración y un tarea:

```
projA / Compile / console / scalacOptions
```

Que es la sintaxis de barra, introducida en sbt 1.1, equivalente a:

```
scalacOptions in (
  Select(projA: Reference),
  Select(Compile: ConfigKey),
  Select(console.key)
)
```

Ámbito con el eje de subproyecto

Si pones múltiples proyectos en una única construcción, cada proyecto necesitará su propia configuración. Es decir, las claves pueden tener un ámbito u otro de acuerdo al proyecto.

El eje de proyecto puede también ser establecido a `ThisBuild`, que quiere decir la "construcción entera" por lo que una entrada se aplica a toda la construcción en lugar de a un único proyecto.

La configuración a nivel de construcción es frecuentemente utilizada como configuración predeterminada cuando un proyecto no define una entrada específica. Explicaremos configuraciones a nivel de construcción más adelante en esta página.

Ámbito con el eje de configuración

Una *configuración de dependencia* (o simplemente "configuración") define un grafo de dependencias de bibliotecas, potencialmente con su propio classpath, ficheros fuentes, paquetes generados, etc… El concepto de configuración de dependencia proviene de Ivy, el cual es usado por sbt para gestionar dependencias y de los ámbitos de Maven.

Algunas configuraciones que verás en sbt:

- `Compile` que define la construcción de los ficheros principales (`src/main/scala`).
- `Test` que define cómo construir los tests (`src/test/scala`).
- `Runtime` que define el classpath para la tarea `run`.

De forma predeterminada, todas las claves asociadas a la compilación, empaquetado y ejecución tienen como ámbito una configuración y por tanto pueden funcionar de forma diferente en cada configuración. Los ejemplos más ovios son las claves tarea `compile`, `package` y `run`, y todas las claves que afectan a dichas claves (tales como `sourceDirectories`, `scalacOptions` o `fullClasspath`) también tienen una configuración como ámbito.

Otra cosa a tener en cuenta sobre una configuración es que puede extender otras configuraciones. La siguiente figura muestra la relación de extensión entre las configuraciones más comunes.

dependency configurations

`Test` y `IntegrationTest` extienden `Runtime`; `Runtime` extiende `Compile`; `CompileInternal` extiende `Compile`, `Optional`, y `Provided`.

Ámbito con el eje de tarea

La configuración puede afectar a cómo funciona una tarea. Por ejemplo, la tarea `packageSrc` es afectada por la entrada `packageOptions`.

Para soportar esto, una clave tarea (tal como `packageSrc`) puede tener un ámbito para otra clave (tal como `packageOptions`).

Las distintas tareas que construyen un paquete (`packageSrc`, `packageBin`, `packageDoc`) pueden compartir claves relacionadas con el empaquetado, tales como `artifactName` y `packageOptions` Esas claves pueden tener distintos valores para cada tarea de empaquetado.

Componentes con ámbito `Zero`

Cada eje de ámbito puede ser rellenado tanto con una instancia del tipo del eje (análogamente a como ocurre con `Some(_)`), o con el valor especial `Zero`. Podemos pensar en `Zero` como `None`.

`Zero` es un comodín universal para todos los ejes de ámbito pero su uso directo debería de estar reservado para sbt y, en todo caso, para los autores de plugins.

`Global` es un ámbito que establece `Zero` para todos los ejes: `Zero / Zero / Zero`. En otras palabras, `Global / clave` es un atajo para `Zero / Zero / Zero / clave`.

Referenciar ámbitos en la definición de construcción

Si creas una entrada en `build.sbt` con una clave plana entonces tendrá como ámbito (subproyecto actual / configuración `Zero` / tarea `Zero`):

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

Si ejecutas sbt y lanzas `inspect name` podrás comprobar que es proporcionada por `ProjectRef(uri("file:/private/tmp/hello/"), "root") / name`, es decir, el proyecto es `ProjectRef(uri("file:/Users/xxx/hello/"), "root")` y ni el ámbito de la configuración ni el de la tarea son mostrados (lo que significa que son `Zero`).

Una clave plana a la derecha también tiene como ámbito (subproyecto actual / configuración `Zero` / tarea `Zero`):

```
organization := name.value
```

Los tipos de cualesquiera de los ejes de ámbito están extendidos para tener un operador `/`. El argumento de `/` puede ser una clave u otro eje de ámbito. Así que por ejemplo, aunque no hay ninguna razón de peso para hacer lo siguiente, se podría tener una instancia de la clave `name` en el ámbito de la configuración `Compile`:

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

o podrías establecer `name` en el ámbito de la tarea `packageBin` (algo inútil, sólo es un ejemplo).

```
packageBin / name := "hello"
```

o podrías establecer `name` con múltiples ejes de ámbito, por ejemplo con la tarea `packageBin` en la configuración `Compile`:

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

o puedes utilizar `Global`:

```
// same as Zero / Zero / Zero / concurrentRestrictions
Global / concurrentRestrictions := Seq(
  Tags.limitAll(1)
)
```

(`Global / concurrentRestrictions` es convertido implícitamente a `Zero / Zero / Zero / concurrentRestrictions`, estableciendo todos los ejes a ámbito `Zero`. De forma predeterminada las tareas y las configuraciones ya son `Zero` de forma predeterminada por lo que la única utilidad de emplear esto es la de establecer el proyecto a `Zero` en lugar de a `ProjectRef(uri("file:/tmp/hello/"), "root") / Zero / Zero / concurrentRestrictions`)

Referenciar ámbitos desde el shell de sbt

En la línea de comandos y en el shell de sbt, sbt muestra (y analiza) claves con ámbito como esta:

```
ref / Config / intask / key
```

- `ref` identifica el eje de subproyecto. Puede ser `<project-id>`, `ProjectRef(uri("file:..."), "id")`, o `ThisBuild` para denotar el ámbito de la construcción entera.
- `Config` identifica el eje de configuración utilizando el identificador de Scala empezando por mayúscula.
- `intask` identifica el eje de tarea.
- `key` identifica la clave a la que se le aplica el ámbito.

`Zero` puede aparecer en cada eje.

Si se omiten partes del ámbito de la clave, éstas serán inferidas siguiendo las siguientes reglas:

- el proyecto actual será utilizado si el proyecto es omitido
- la configuración dependiente de la clave será autodetectada si se omite la configuración o la tarea.

Para más información ver Interactuar con el sistema de configuración.

Ejemplos de la notación de claves con ámbito

- `fullClasspath` especifica simplemente una clave, por lo que los ámbitos predeterminados son utilizados: proyecto actual, una configuración dependiente de la clave y el ámbito de tarea `Zero`.
- `Test / fullClasspath` emplea una configuración, por lo que es `fullClasspath` en la configuración `Test`, con valores predeterminados para el resto de ejes.
- `root / fullClasspath` especifica el proyecto `root`, donde el proyecto es identificado por el identificador de proyecto.
- `root / Zero / fullClasspath` especifica el proyecto `root` y `Zero` para la configuración, en lugar de la predeterminada.
- `doc / fullClasspath` especifica la clave `fullClasspath` en el ámbito de la tarea `doc`, con los valores predeterminados para el eje del proyecto y el de la configuración.
- `ProjectRef(uri("file:/tmp/hello/"), "root") / Test / fullClasspath` especifica el proyecto `ProjectRef(uri("file:/tmp/hello/"), "root")`. Además especifca la configuración `Test`, dejando el eje de tarea al valor predeterminado.
- `ThisBuild / version` establece el eje de subproyecto a la "construcción entera" donde la construcción es `ThisBuild`, con la configuración predeterminada.
- `Zero / fullClasspath` establece el eje de subproyecto a `Zero`, con la configuración predeterminada.
- `root / Compile / doc / fullClasspath` establece todos los ejes de ámbito.

Inspeccionar ámbitos

En el shell de sbt, puedes utilizar el comando `inspect` para comprender las claves y sus ámbitos. Prueba `inspect test / fullClasspath`:

```
$ sbt
sbt:Hello> inspect Test / fullClasspath
[info] Task: scala.collection.Seq[sbt.internal.util.Attributed[java.io.File]]
[info] Description:
[info]  The exported classpath, consisting of build products and unmanaged and managed, inte
[info] Provided by:
[info]  ProjectRef(uri("file:/tmp/hello/"), "root") / Test / fullClasspath
[info] Defined at:
[info]  (sbt.Classpaths.classpaths) Defaults.scala:1639
[info] Dependencies:
[info]  Test / dependencyClasspath
[info]  Test / exportedProducts
[info]  Test / fullClasspath / streams
```

```
[info] Reverse dependencies:
[info]  Test / testLoader
[info] Delegates:
[info]  Test / fullClasspath
[info]  Runtime / fullClasspath
[info]  Compile / fullClasspath
[info]  fullClasspath
[info]  ThisBuild / Test / fullClasspath
[info]  ThisBuild / Runtime / fullClasspath
[info]  ThisBuild / Compile / fullClasspath
[info]  ThisBuild / fullClasspath
[info]  Zero / Test / fullClasspath
[info]  Zero / Runtime / fullClasspath
[info]  Zero / Compile / fullClasspath
[info]  Global / fullClasspath
[info] Related:
[info]  Compile / fullClasspath
[info]  Runtime / fullClasspath
```

En la primera línea, se puede apreciar que esta es una tarea (y no una entrada, tal y como se explica en Definiciones de construcción). El valor resultante de la tarea es del tipo `scala.collection.Seq[sbt.Attributed[java.io.File]]`.

"Provided by" indica la clave con ámbito que define el valor, en este caso `ProjectRef(uri("file:/tmp/hello/"), "root") / Test / fullClasspath` (que es la clave `fullClasspath` en el ámbito de la configuración `Test` y el proyecto `ProjectRef(uri("file:/tmp/hello/"), "root")` project).

"Dependencies" fue explicado en detalle en la página anterior.

"Delegates" será explicado más adelante.

Si ejecutas `inspect fullClasspath` (en oposición al ejemplo de arriba, inspect `Test / fullClasspath`) podrás apreciar la diferencia. Debido a que la configuración es omitida, es autodetectada como `Compile`. `inspect Compile / fullClasspath` debería por tanto ser lo mismo que `inspect fullClasspath`.

Si ejecutas `inspect ThisBuild / Zero / fullClasspath` podrás obtener otro ejemplo. De forma predeterminada `fullClasspath` no está definido en la ámbito de la configuración `Zero`.

Una vez más, para más información ver Interactuar con el sistema de configuración.

Cuándo especificar un ámbito

Un ámbito necesita ser especificado si la clave en cuestión ya está asociada a otro ámbito. Por ejemplo, la tarea `compile`, de forma predeterminada, tiene

como ámbito las configuraciones `Compile` y `Test` y no existe fuera de dichos ámbitos.

Para cambiar el valor asociado con la clave `compile` necesitas escribir `Compile / compile` o `Test / compile`. Utilizar solamente `compile` definiría una nueva tarea `compile` en el ámbito del proyecto actual, en lugar de sobrescribir la tarea de compilación estándar, la cual tiene como ámbito una configuración.

Si obtienes un error como *"Reference to undefined setting"*, con frecuencia significará que no has especificado un ámbito, o que has especificado el ámbito equivocado. La clave que estás utilizando puede estar definida en otro ámbito. sbt intentará sugerir lo que querías decir como parte del mensaje de error. Busca cosas tipo "Did you mean Compile / compile?"

Una forma de pensar en esto es que un nombre es solo una *parte* de una clave. En realidad, todas las claves consisten tanto en un nombre como en un ámbito (donde el ámbito tiene tres ejes). La expresión completa `Compile / packageBin / packageOptions` es un nombre de clave, dicho de otra forma. `packageOptions` a secas también es un nombre de clave, pero uno diferente (uno donde los ámbitos son implícitamente establecidos: proyecto actual, configuración `Zero` y tarea `Zero`).

Configuración a nivel de construcción

Una técnica avanzada para extraer configuración común a todos los subproyectos es definir valores en el ámbito de `ThisBuild`.

Si una clave que tiene como ámbito un subproyecto en particular no se encuentra, sbt la buscará en el ámbito de `ThisBuild`. Usando este mecanismo, podemos definir valores predeterminados a nivel de construcción para claves usadas con frecuencia tales como `version`, `scalaVersion` y `organization`.

```
ThisBuild / organization := "com.example",
ThisBuild / scalaVersion := "2.12.14",
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    publish / skip := true
  )

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

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

Convenientemente, existe una función `inThisBuild(...)` que definirá tanto la clave como el valor al ámbito de `ThisBuild`. Definir valores aquí sería equivalente a prefijar cada uno con `ThisBuild /` allá donde fuera posible.

Debido a la naturaleza de la delegación de ámbito que explicaremos más adelante, la configuración a nivel de construcción debería ser utilizada sólo para valores puros o para valores en el ámbito de `Global` o `ThisBuild`.

Delegación de ámbito

Una clave con ámbito puede no haber sido definida, si no tiene un valor asociado en su ámbito.

Para cada eje de ámbito, sbt tiene un camino de búsqueda alternativo consistente en otros valores con ámbito. Habitualmente, si una clave no tiene asociado un valor en un ámbito específico, sbt intentará obtener un valor de un ámbito más general, tal como el ámbito `ThisBuild`.

Esta característica te permitirá establecer un valor una única vez en un ámbito general, permitiendo múltiples ámbitos específicos que heredan el valor. Lo discutiremos con mas detalle más tarde en Delegación de ámbito.

Añadir valores

Añadir a valores existentes: `+=` y `++=`

La asignación `:=` es la transformación más simple, pero las claves tienen también otros métodos. Si `T` en `SettingKey[T]` es una secuencia, es decir, si el tipo del valor de una clave es una secuencia, podrás añadir elementos a la secuencia en lugar de reemplazarla.

- `+=` añadirá un único elemento a la secuencia
- `++=` concatenará otra secuencia

Por ejemplo, la clave `Compile / sourceDirectories` tiene un `Seq[File]` por valor. De forma predeterminada el valor de esta clave incluye `src/main/scala`. Si quieres que además se compile el código fuente de un directorio llamado source (en el caso de que por ejemplo no utilices una estructura de directorios estándar) podrás añadir dicho directorio:

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

O convenientemente puedes utilizar la función `file()` del paquete sbt.

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

(`file()` crea un nueva instancia de `File`)

Puedes usar `++=` para añadir más de un directorio a la vez:

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

Donde `Seq(a, b, c, ...)` es la sintaxis estándar de Scala para construir una secuencia.

Por supuesto, para reemplazar totalmente los directorios de código fuente predeterminados puedes seguir utilizando `:=`:

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

Cuándo no está definida una configuración

Cuando una entrada utiliza `:=`, `+=` o `++=` para crear una dependencia sobre sí misma o sobre el valor de otra clave, el valor sobre el que depende debe existir. De lo contrario sbt se quejará. Puede que por ejemplo diga *"Reference to undefined setting"*. Cuando esto suceda asegúrate de que estás utilizando la clave en el ámbito donde está definida.

Es posible crear referencias circulares, lo cual es un error. sbt te lo dirá si lo haces.

Tareas basadas en valores de otras claves

Puedes calcular el valor de algunas tareas o entradas definiendo o añadiendo el valor de otra tarea. Esto se hace utilizando `Def.task` como argumento de `:=`, `+=`, or `++=`.

Como ejemplo, considera añadir un generador de código fuente utilizando el directorio base del proyecto y el classpath de compilación.

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

Añadir con dependencia: `+=` y `++=`

Otras claves pueden ser utilizadas cuando se añade a una entrada o tarea existente, al igual que se hacía para asignar con `:=`.

Por ejemplo, digamos que tienes un informe de cobertura de código cuyo nombre se basa en el nombre del proyecto y que quieres añadirlo a la lista de ficheros que se borran cuando se invoca `clean`:

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

Delegación de ámbito (resolución de .value)

Esta página explica la delegación de ámbito. Se supone que has leído y comprendido las páginas anteriores, Definiciones de construcción y Ámbitos.

Ahora que hemos cubierto todos los detalles de los ámbitos, podemos explicar la resolución de `.value` en detalle. Te puedes saltar esta sección si es la primera vez que lees esta página.

Resumamos lo que hemos aprendido hasta ahora:

- Un ámbito es una tupla de componentes de tres ejes: el eje de subproyecto, el eje de configuración y el eje de tarea.
- Existe el componente de ámbito especial `Zero` utilizado por los tres ejes de ámbito.
- Existe el componente de ámbito especial `ThisBuild` utilizado únicamente por *el eje de subproyecto*.
- `Test` extiende `Runtime` y `Runtime` extiende la configuración `Compile`.
- Una clave definida en build.sbt tiene como ámbito a `${current subproject} / Zero / Zero` de forma predeterminada.
- Una clave puede especificar un ámbito utilizando el operador `/`.

Ahora supongamos que tenemos la siguiente definición de construcción:

```
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
  )
```

Dentro del cuerpo de la entrada de `foo`, la clave con ámbito `Test / bar` ha sido declarada. Sin embargo, a pesar de que `Test / bar` no está definida en `projX`, sbt sigue siendo capaz de resolver `Test / bar` utilizando otra clave con ámbito, lo que lleva a que `foo` sea inicializado a `2`.

sbt tiene un camino alternativo bien definido llamado *delegación de ámbito*. Esto permite establecer un valor una única vez en un ámbito más general, permitiendo que ámbitos más específicos hereden tal valor.

Reglas de delegación de ámbito

Estas son las reglas para la delegación de ámbito:

- Regla 1: los ejes de ámbito tienen la siguiente precedencia: el eje de subproyecto, el eje de configuración y finalmente el eje de tarea.
- Regla 2: Dado un ámbito, la delegación de ámbitos es utilizada sustituyendo el eje de tarea en el siguiente orden: el ámbito de dicha tarea y luego `Zero`, que es la versión tarea-nula con ámbito de este ámbito.
- Regla 3: Dado un ámbito, la delegación de ámbitos es utilizada sustituyendo el eje de configuración en el siguiente orden: la propia configuración, sus ancestros y luego `Zero` (equivalente a un eje de configuración sin ámbito).
- Regla 4: Dado un ámbito, la delegación de ámbitos es utilizada sustituyendo el eje de subproyecto en el siguiente orden: el propio proyecto, `ThisBuild` y luego `Zero`.
- Regla 5: Una clave con delegación de ámbito y sus entradas/tareas dependientes son evaluadas sin acarrear el contexto original.

Estudiaremos cada regla en el resto de esta página.

Regla 1: Precedencia de ejes de ámbito

- Regla 1: los ejes de ámbito tienen la siguiente precedencia: el eje de subproyecto, el eje de configuración y finalmente el eje de tarea.

En otras palabras, dados dos ámbitos candidatos, si uno de ellos tiene un valor más específico en el eje de subproyecto entonces dicho eje siempre ganará sin importar el ámbito de la configuración o la tarea. De forma similar, si los subproyectos son los mismos, ganará el que tenga un valor más específico en el eje de configuración, sin importar lo que tenga en el ámbito de tarea. Veremos más reglas para definir *más específico*.

Regla 2: Delegación del eje de tarea

- Regla 2: Dado un ámbito, la delegación de ámbitos es utilizada **sustituyendo** el eje de tarea en el siguiente orden: el ámbito de dicha tarea y luego `Zero`, que es la versión tarea-nula con ámbito de este ámbito.

Aquí tenemos una regla concreta que muestra cómo sbt empleará la delegación de ámbitos dada una clave. Recuerda, estamos intentando mostrar el camino de búsqueda a partir de un `(xxx / yyy).value` cualquiera.

Ejercicio A: Dada la siguiente definición de construcción:

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

```
    scalaVersion := "2.11.11"
  )
```

¿Cuál es el valor de `projA / name`?

1. `"foo-2.11.11"`
2. `"foo-2.12.14"`
3. ¿otra cosa?

La respuesta es `"foo-2.11.11"`. Dentro de `.settings(...)`, `scalaVersion` tiene automáticamente un ámbito de `projA / Zero / Zero`, por lo que `packageBin / scalaVersion` se convierte en `projA / Zero / packageBin / scalaVersion`. Esa clave con ámbito en particular no está definida. Utilizando la regla 2, sbt sustituirá el eje de tarea a `Zero` como `projA / Zero / Zero` o (`projA / scalaVersion`). Esta clave con ámbito está definida como `"2.11.11"`.

Regla 3: Resolución del eje de configuración

- Regla 3: Dado un ámbito, la delegación de ámbitos es utilizada sustituyendo el eje de configuración en el siguiente orden: la propia configuración, sus ancestros y luego `Zero` (equivalente a un eje de configuración sin ámbito).

El ejemplo es el `projX` que vimos antes:

```
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
  )
```

El ámbito completo es `projX / Test / Zero`. Además, recordemos que `Test` extiende `Runtime` y que `Runtime` extiende `Compile`.

`Test / bar` no está definido pero, debido a la Regla 3, sbt buscará `bar` con ámbito `projX / Test / Zero`, `projX / Runtime / Zero` y finalmente `projX / Compile / Zero`. Este último es encontrado, el cual es `Compile / bar`.

Regla 4: Resolución del eje de subproyecto

- Regla 4: Dado un ámbito, la delegación de ámbitos es utilizada sustituyendo el eje de subproyecto en el siguiente orden: el propio proyecto,

`ThisBuild` y luego `Zero`.

Ejercicio B: Dada la siguiente definición de construcción:

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

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

¿Cuál es el valor de `projB / name`?

1. `"abc-com.example"`
2. `"abc-org.tempuri"`
3. ¿otra cosa?

La respuesta es `abc-org.tempuri`. Aplicando la Regla 4, el primer intento se hace mirando `organization` con ámbito `projB / Zero / Zero`, el cuál está definido en `projB` como `"org.tempuri"`. Éste tiene mayor precedencia que la configuración a nivel de construcción `ThisBuild / organization`.

Precedencia de ejes de ámbito, otra vez

Ejercicio C: Dada la siguiente definición de construcción:

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

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

¿Cuál es el valor de `projC / name`?

1. `"foo-2.12.2"`
2. `"foo-2.11.11"`
3. ¿otra cosa?

La respuesta es `foo-2.11.11`. `scalaVersion` con ámbito `projC / Zero / packageBin` no está definida. La Regla 2 encuentra `projC / Zero / Zero`. La regla 4 encuentra `ThisBuild / Zero / packageBin`. En este caso la Regla 1 dice que el valor más específico del eje de subproyecto gana, el cual es `projC / Zero / Zero`, que está definido como `"2.11.11"`.

Ejercicio D: Dada la siguiente definición de construcción:

```
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
  )
```

¿Qué saldría si ejecutamos `projD / test`?

1. `List()`
2. `List(-Ywarn-unused-import)`
3. ¿otra cosa?

La respuesta es `List(-Ywarn-unused-import)`. La Regla 2 encuentra `projD / Compile / Zero`, la Regla 3 encuentra `projD / Zero / console`, y la Regla 4 encuentra `ThisBuild / Zero / Zero`. La Regla 1 elige `projD / Compile / Zero` porque tiene tiene `projD` en el eje de subproyecto y dicho eje tiene mayor precedencia que el eje de tarea.

Después, `Compile / scalacOptions` hace referencia a `scalacOptions.value`, luego necesitamos encontrar un delegado para `projD / Zero / Zero`. La Regla 4 encuentra `ThisBuild / Zero / Zero` que finalmente resuelve a `(-Ywarn-unused-import)`.

El comando inspect para listar delegaciones

Para saber qué está pasando puedes utilizar el comando `inspect`.

```
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
```

Fíjate en que "Provided by" muestra que `projD / Compile / console / scalacOptions` es proporcionado por `projD / Compile / scalacOptions`. Además, bajo "Delegates", *todos* los posibles candidatos son listados en orden de precedencia.

- Todos los ámbitos con ámbito `projD` en el eje de subproyecto son listados primero, luego `ThisBuild` y luego `Zero`.
- Dentro de un subproyecto, los ámbitos que utilizan `Compile` en el eje de configuración son listados primero y luego los de `Zero`.
- Finalmente, se listan los ejes de tarea con ámbito `console` y luego los que no lo tienen.

Resolución de .value vs enlace dinámico

- Regla 5: Una clave con delegación de ámbito y sus entradas/tareas dependientes son evaluadas sin acarrear el contexto original.

Fíjate en que la delegación de ámbito se parece mucho a la herencia de clases de un lenguaje orientado a objetos, aunque con una diferencia: En un lenguaje orientado a objetos como Scala, cuando existe un método llamado `drawShape` en un trait `Shape` sus subclases pueden sobrescribir el comportamiento incluso cuando `drawShape` es utilizado por otros métodos en el trait, lo es lo que se llama enlace dinámico.

Sin embargo, en sbt la delegación de ámbito puede delegar un ámbito a uno más general, como una configuración a nivel de proyecto hacia una configuración a nivel de construcción, pero dicha configuración a nivel de construcción no puede hacer referencia a la configuración a nivel de proyecto.

Ejercicio E: Dada la siguiente definición de construcción:

```
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"
  )
```

¿Qué devolverá `projE / version`?

1. `"2.12.2_0.1.0"`
2. `"2.11.11_0.1.0"`
3. ¿otra cosa?

La respuesta es `2.12.2_0.1.0`. `projD / version` delega en `ThisBuild / version`, que a su vez depende de `ThisBuild / scalaVersion`. Debido a esto, la configuración a nivel de construcción debería limitarse únicamente a asignaciones simples de valores.

Ejercicio F: Dada la siguiente definición de construcción:

```
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)
    }
  )
```

¿Qué mostrará `projF / test`?

1. `"bippy-D4"`
2. `"bippy-D2-D4"`
3. `"bippy-D0-D3-D4"`
4. ¿otra cosa?

La respuesta es `"bippy-D0-D3-D4"`. Esta es una variación de un ejercicio creado originalmente por Paul Phillips.

Es una gran demostración de todas las reglas porque `someKey += "x"` se expande a

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

Al obtener el valor antiguo se dispara la delegación y debido a la Regla 5 se irá a otra clave con ámbito. Librémonos del += primero y anotemos los delegados para valores antiguos:

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

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

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

Esto se convierte en:

```
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)
    }
  )
```

Dependencias de bibliotecas

Esta página asume que has leído las páginas anteriores de la Guía de inicio, en particular Definiciones de construcción, Ámbitos y Grafos de tareas.

Las dependencias de bibliotecas pueden ser añadidas de dos formas:

- las *dependencias no gestionadas* son jars copiados en el directorio `lib`
- las *dependencias gestionadas* son configuradas en la definición de construcción y descargadas automáticamente desde repositorios

Dependencias no gestionadas

La mayoría de la gente utiliza dependencias gestionadas en lugar de no gestionadas. Pero las segundas son más simples de explicar cuando se empieza con las dependencias.

Las dependencias no gestionadas funcionan así: copia jars en `lib` y serán añadidas en el classpath del proyecto. Y no hay más.

También puedes copiar jars para tests como ScalaCheck, Specs2 y ScalaTest en `lib`.

Las dependencias en `lib` aparecen en todos los classpaths (para `compile`, `test`, `run` y `console`). Si quieres cambiar el classpath para una sola de esas tareas deberías de ajustar `Compile / dependencyClasspath` o `Runtime / dependencyClasspath`, por ejemplo.

No hay que añadir nada en `build.sbt` para empezar a utilizar dependencias no gestionadas, aunque puedes cambiar la clave `unmanagedBase` si quisieras utilizar un directorio diferente a `lib`.

Para utilizar `custom_lib` en lugar de `lib`:

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

`baseDirectory` es el directorio raíz del proyecto, por lo que aquí se está cambiando `unmanagedBase` para que dependa de `baseDirectory` utilizando el método especial `value`, como se explicó en Grafos de tareas.

Existe también una tarea llamada `unmanagedJars` la cual lista los jars del directorio `unmanagedBase`. Si quieres utilizar múltiples directorios o hacer algo más complejo tendrías que reemplazar por completo la tarea `unmanagedJars` con una que hiciese algo diferente, por ejemplo, devolver una lista para la configuración `Compile` en lugar de los ficheros en el directorio `lib`:

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

Dependencias gestionadas

sbt utiliza Apache Ivy para implementar dependencias gestionadas, por lo que no tendrás demasiados problemas si ya estás familiarizado con Ivy o Maven.

La clave `libraryDependencies`

La mayor parte del tiempo, podrás simplemente listar tus dependencias en la entrada `libraryDependencies`. Es incluso posible escribir un fichero POM de Maven o un fichero de configuración de Ivy para configurar externamente tus dependencias y hacer que sbt use esos ficheros de configuración externos. Puedes obtener más información sobre cómo se puede hacer aquí.

Declarar una dependencia es parecido a esto, donde `groupId`, `artifactId` y `revision` son cadenas de caracteres:

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

o a esto, donde `configuration` puede ser una cadena de caracteres o una `Configuration` (such as `Test`):

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

`libraryDependencies` está declarado en Keys de esta forma:

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

El método `%` crea un objeto de tipo `ModuleID` a partir de cadenas de caracteres, que luego puede ser añadido a `libraryDependencies`.

Por supuesto, sbt (via Ivy) tiene que saber de dónde descargar los módulos. Si tu módulo está en uno de los repositorios predeterminados con los que cuenta sbt entonces será suficiente. Por ejemplo, Apache Derby está en el repositorio estándar de Maven2:

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

Si escribes eso en `build.sbt` y luego ejecutas `update`, sbt debería descargar Derby en `~/.ivy2/cache/org.apache.derby/`. (Por cierto, `update` es una dependencia de `compile` por lo que no hay necesidad de estar escribiendo `update` todo el rato.)

Por supuesto, puedes también utilizar `++=` para añadir en un único paso una serie de dependencias:

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
  groupID % otherID % otherRevision
)
```

En contadas ocasiones necesitarás utilizar `:=` con `libraryDependencies`.

Obteniendo la versión de Scala correcta con %%

Si usas `groupID %% artifactID % revision` en lugar de `groupID % artifactID % revision` (la diferencia está en el doble `%%` tras `groupId`), sbt añadirá la versión del binario de Scala de tu proyecto al nombre del artefacto. Esto simplemente es un atajo. Podrías escribir esto sin el `%%`:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11" % "0.3"
```

Si asumimos que `scalaVersion` para tu construcción es `2.11.1` lo siguiente es idéntico a lo anterior (fíjate en el doble `%%` tras `"org.scala-tools"`):

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

La idea es que muchas dependencias son compiladas para múltiples versiones de Scala y con toda seguridad lo que querrás será obtener aquella que ofrece compatibilidad binaria con tu proyecto.

Para más información ver Construcción cruzada.

Revisiones Ivy

`revision` en `groupID % artifactID % revision` no tiene por qué ser una versión fija. Ivy puede seleccionar la última revisión de un módulo de acuerdo a las restricciones que especifiques. En lugar de una revisión fija como `"1.6.1"`, puedes especificar `"latest.integration"`, `"2.9.+"`, or `"[1.0,)"`. Para más información ver la documentación de Revisiones Ivy.

En ocasiones, un "rango de versiones" de Maven es utilizado para especificar una dependencia (transitiva o de otro tipo), tales como `[1.3.0,)`. Si una versión específica de la dependencia es declarada en la construcción y satisface el rango entonces sbt usará la versión especificada. En otro caso, Ivy tendrá que encontrar la última versión. Esto puede ocasionar un comportamiento inesperado donde la versión efectiva va cambiando a lo largo del tiempo, incluso cuando hay una versión específica de la biblioteca que satisface la condición del rango.

Resolvedores

No todos los paquetes viven en el mismo servidor. sbt utiliza el repositorio estándar de Maven2 de forma predeterminada. Si tu dependencia no está en uno de los repositorios predeterminados tendrás que añadir un *resolvedor* para que Ivy lo pueda encontrar.

Para añadir un repositorio adicional utiliza:

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

con el método especial `at` entre dos cadenas de caracteres.

Por ejemplo:

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

La clave `resolvers` está definida Keys de la siguiente forma:

```
val resolvers = settingKey[Seq[Resolver]]("The user-defined additional resolvers for automat
```

El método `at` crea un objecto de tipo `Resolver` a partir de dos cadenas de caracteres.

sbt puede buscar en tu repositorio local de Maven si lo añades como repositorio:

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

o, convenientemente:

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

Para más información sobre cómo definir otros tipos de repositorios ver Resolvedores.

Sobrescribiendo resolvedores predeterminados

`resolvers` no contiene los resolvedores predeterminados, solo los adicionales añadidos por tu definición de construcción.

sbt combina `resolvers` con algunos repositorios predeterminados para formar `externalResolvers`.

Por tanto, para cambiar o eliminar los resolvedores predeterminados tendrías que sobrescribir `externalResolvers` en lugar de `resolvers`.

Dependencias por configuración

A menudo una dependencia es necesaria para tu código de test (en `src/test/scala`, el cual es compilado por la configuración `Test`) pero no así tu código fuente principal.

Si quieres añadir una dependencia que aparezca solo en el classpath de la configuración `Test` pero no en la de `Compile` puedes añadir un `% "test"` de la siguiente forma:

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

También puedes utilizar la versión tipada de la configuración `Test` de esta forma:

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

Ahora, si escribes `show compile:dependencyClasspath` en el prompt interactivo de sbt no deberías ver listado el jar de Derby. Pero si escribes `show test:dependencyClasspath` sí que deberías de verlo.

Habitualmente, las dependencias relacionadas con tests tales como ScalaCheck, Specs2 y ScalaTest debería de ir definidas con `% "test"`.

Existen más detalles, consejos y trucos relacionados con las dependencias de bibliotecas en esta página.

Usar plugins

Por favor, lee primero las páginas anteriores de la Guía de inicio, en particular Definiciones de construcción, Grafos de tareas y Dependencias de bibliotecas antes de leer esta página.

¿Qué es un plugin?

Un plugin extiende la definición de construcción, usualmente añadiendo nueva configuración. La nueva configuración puede incluir nuevas tareas. Por ejemplo, un plugin puede añadir una tarea `codeCoverage` para generar un informe de cobertura de código de test.

Declarar un plugin

Si tu proyecto está en un directorio llamado `hello` y quieres añadir el plugin `sbt-site` a la definición de construcción, crea el fichero `hello/project/site.sbt` y declara la dependencia del plugin pasando el módulo Ivy del plugin a `addSbtPlugin`:

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

Si quieres añadir `sbt-assembly`, crea `hello/project/assembly.sbt` con el siguiente contenido:

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

No todos los plugins están ubicados en uno de los repositorios predeterminados. La documentación del plugin te puede instar a que añadas el repositorio donde se encuentra:

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

Por lo general los plugins proporcionan configuraciones que son añadidas a la del proyecto para habilitar la funcionalidad del plugin. Esto es explicado en la siguiente sección:

Habilitando e inhabilitando autoplugins

Un plugin puede declarar que su configuración debería de ser automáticamente añadida a la definición de construcción, en cuyo caso no tienes que hacer nada para añadirla.

A partir de sbt 0.13.5 se introdujo la nueva característica autoplugins la cual permite asegurar que la configuración de un plugin está disponible en el proyecto de forma automática y de un modo seguro. Muchos autoplugins deberían contar con su configuración predeterminada automáticamente, sin embargo esto requiere la activación explícita.

Si estás utilizando un autoplugin que requiere activación explícita entonces tendrás que añadir lo siguiente en tu `build.sbt`:

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

El método `enablePlugins` permite a los proyectos definir explícitamente qué autoplugins quieren utilizar.

Los proyectos también pueden excluir plugins con el método `disablePlugins`. Por ejemplo, si queremos eliminar la configuración de `IvyPlugin` en `util` tendríamos que modificar nuestro `build.sbt` de la siguiente manera:

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

Los autoplugins deberían documentar si necesitan o no ser explícitamente habilitados. Si sientes curiosidad por saber qué autoplugins están habilitados para un proyecto, puedes ejecutar el comando `plugins` en la consola de sbt.

Por ejemplo:

```
> 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
```

Aquí, la salida de `plugins` nos está mostrando que los plugins predeterminados de sbt están todos habilitados. La configuración predeterminada de sbt es proporcionada via tres plugins:

1. `CorePlugin`: Proporciona los controles de paralelismo de tareas del núcleo.
2. `IvyPlugin`: Proporciona los mecanismos para publicar/resolver módulos.
3. `JvmPlugin`: Proporciona los mecanismos para compilar/testear/ejecutar/empaquetar proyectos de Java/Scala.

Adicionalmente, `JUnitXmlReportPlugin` proporciona soporte experimental para generar junit-xml.

Algunos plugins no automáticos antiguos a menudo necesitan que su configuración sea añadida explícitamente, por lo que las construcciones multiproyecto pueden tener diferentes tipos de proyectos. La documentación del plugin indicará cómo configurarlo, pero típicamente para plugins antiguos esto implica añadir la configuración base del plugin y modificarla según sea necesario.

Por ejemplo, para el plugin `sbt-site`, necesitarías crear un fichero `site.sbt` con el siguiente contenido para habilitarlo para ese proyecto:

```
site.settings
```

Si la construcción define múltiples proyectos, entonces añádelo directamente al proyecto:

```
// inhabilitar el plugin site para el proyecto ´util´
lazy val util = (project in file("util"))

// habilitar el plugin site para el proyecto `core`
lazy val core = (project in file("core"))
  .settings(site.settings)
```

Plugins globales

Los plugins pueden ser instalados para todos tus proyectos a la vez declarándolos en `$HOME/.sbt/1.0/plugins/`. `$HOME/.sbt/1.0/plugins/` es un proyecto sbt cuyo classpath es exportado a todos los proyectos de la definición de construcción. Más o menos, cualquier fichero `.sbt` y `.scala` en `$HOME/.sbt/1.0/plugins/` se comporta como si estuviera en el directorio `project/` de cada uno de los proyectos.

Puedes crear `$HOME/.sbt/1.0/plugins/build.sbt` y poner expresiones `addSbtPlugin()` ahí para añadir plugins a todos tus proyectos a la vez. Debido a que hacer eso incrementaría la dependencia a nivel local, esta característica debería ser utilizada con moderación. Para más información ver Buenas prácticas.

Plugins disponibles

Existe una lista de plugins disponibles.

Algunos plugins especialmente famosos son:

- aquellos para los IDEs (para importar un proyecto sbt en tu IDE)
- aquellos que soportan frameworks web, tales como xsbt-web-plugin

Para más información, incluyendo cómo desarrollar plugins, ver Plugins. Para saber más acerca de las mejores prácticas ver Plugins - Mejores prácticas.

Entradas y tareas personalizadas

Esta página sirve de introducción para crear entradas y tareas personalizadas.

Para entender esta página, asegúrate de que has leído las páginas anteriores de la Guía de inicio, en particular Definiciones de construcción y Grafos de tareas.

Definir una clave

Keys está lleno de ejemplos que ilustran cómo definir claves. La mayoría de las claves están implementadas en Defaults.

Una clave tiene uno de tres posibles tipos: `SettingKey` y `TaskKey` son descritos en Definiciones de construcción. Para saber más acerca de `InputKey` puedes ver la página Tareas con entrada.

Algunos ejemplos de Keys:

```
val scalaVersion = settingKey[String]("The version of Scala used for building.")
val clean = taskKey[Unit]("Deletes files produced by the build, such as generated sources, c
```

Los constructores de claves toman como parámetros dos cadenas de caracteres: el nombre de la clave (`"scalaVersion"`) y una descripción (`"The version of Scala used for building."`).

Como recordarás, en Definición de construcción se explica que el tipo del parámetro `T` en `SettingsKey[T]` indica el tipo del valor que tiene la entrada. `T` en `TaskKey[T]` indica el tipo del resultado de la tarea. También recordarás que una entrada tiene un valor fijo único hasta la siguiente recarga del proyecto, mientras que una tarea es recalculada para cada "ejecución de la tarea" (cada

vez que alguien escribe un comando en el prompt interactivo de sbt o utiliza el modo por lotes).

Las claves pueden estar definidas en un fichero .sbt, un fichero .scala o un autoplugin. Cualesquiera `val` encontrados bajo el objeto `autoImport` de un autoplugin habilitado será importado automáticamente en tus ficheros `.sbt`.

Implementar una tarea

Una vez que hayas definido una clave para tu tarea necesitarás completarla con una definición de tarea. Puedes tanto definir tu propia tarea como redefinir una ya existente. Para cualquiera de los dos casos hay que utilizar `:=` para asociar cierto código con la clave tarea.

```
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 := "2.12.14"

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

Si la tarea tiene dependencias deberías referenciar su valor utilizando `value` tal cual se explicó en Grafos de tareas.

La parte más difícil sobre cómo implementar las tareas normalmente no tiene nada que ver con sbt, ya que las tareas son simplemente código de Scala. La parte difícil sería escribir el "cuerpo" de tu tarea para que haga aquello que estás intentando hacer. Por ejemplo, puede que estés intentando formatear un texto en HTML para lo cual puede que requieras la utilización de una biblioteca de HTML (puede que necesites añadir una dependencia de biblioteca a tu definición de construcción y escribir código basado en dicha biblioteca).

sbt tiene algunas bibliotecas útiles y funciones convenientes, en particular puedes utilizar la APIs de IO para manipular ficheros y directorios.

Semántica de ejecución de las tareas

Cuando una tarea personalizada utiliza `value` para depender de otras tareas, algo importante a tener en cuenta es la semántica de ejecución de las tareas. Por semántica de ejecución nos referimos a *cuándo* exactamente estas tareas son evaluadas.

Si tomamos por ejemplo `sampleIntTask`, cada línea del cuerpo de la tarea debería de ser evaluada estrictamente una tras otra. Eso es semántica secuencial:

```
sampleIntTask := {
  val sum = 1 + 2        // primera
  println("sum: " + sum) // segunda
  sum                    // tercera
}
```

En realidad, la JVM puede ejecutar `sum` en línea y hacer que valga 3, pero el *efecto* observable de la tarea permanecerá intacto como si cada línea fuese ejecutada una tras otra.

Supongamos ahora que definimos dos o más tareas personalizadas `startServer` y `stopServer`, y modificamos `sampleIntTask` como sigue:

```
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 := "2.12.14"

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
    }
  )
```

Al ejecutar `sampleIntTask` desde el prompt interactivo de sbt el resultado será el siguiente:

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

Para revisar qué ha sucedido vamos a mirar la notación gráfica de `sampleIntTask`:

task-dependency

A diferencia de las llamadas a métodos normales de Scala, el hecho de llamar al método `value` en tareas no hará que sea evaluado estrictamente. En su lugar, simplemente servirá para declarar que `sampleIntTask` depende de las tareas `startServer` y `stopServer`. Cuando `sampleIntTask` es invocado, el motor de tareas de sbt hará lo siguiente:

- evaluará las dependencias de las tareas *antes* de evaluar `sampleIntTask` (orden parcial)
- intentará evaluar las dependencias de las tareas en paralelo si son independientes (paralelización)
- cada dependencia de la tarea será evaluada una única vez por cada ejecución del comando (deduplicación)

Deduplicación de dependencias de tareas

Para demostrar el último punto, podemos ejecutar `sampleStringTask` desde el prompt interactivo de sbt.

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

Debido a que `sampleStringTask` depende tanto de `startServer` como de `sampleIntTask`, y `sampleIntTask` depende a su vez de `startServer`, ésta aparece dos veces listada como dependencia. Si fuese una llamada normal a un método de Scala, ésta sería evaluada dos veces, pero debido a que `value` se usa simplemente para indicar la dependencia de otra tarea, ésta es evaluada sólo una vez. A continuación se muestra una notación gráfica de la evaluación de `sampleStringTask`:

task-dependency

Si no hubiésemos deduplicado las dependencias de tareas habríamos acabado compilando el código fuente de los tests muchas veces cuando la tarea `test` hubiese sido invocada, ya que `Test / compile` aparece muchas veces como dependencia de `Test / test`.

Tarea de limpieza

Entonces ¿cómo se podría implementar la tarea `stopServer`? La noción de tarea de limpieza no encaja en el modelo de ejecución de tareas debido a que las tareas tratan de seguir dependencias. La última operación debería ser la tarea que depende de otras tareas intermedias. Por ejemplo `stopServer` debería depender de `sampleStringTask` por lo que `stopServer` debería de ser `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
```

```
    }
  )
```

Para demostrar que esto funciona ejecutemos `sampleStringTask` desde el prompt interactivo:

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

task-dependency

Usar Scala

Otra forma de asegurarse de que algo sucede después de algo es usando Scala. Si se implementa una función simple en `project/ServerUtil.scala` por ejemplo se podrá escribir:

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

Ya que las llamadas a métodos normales siguen la semántica secuencial todo sucede en orden. No hay deduplicación, por lo que ya no hay que preocuparse por eso.

Conversión en plugins

Si te has encontrado con un montón de código personalizado podrías considerar moverlo a un plugin para reutilizarlo a lo largo de múltiples construcciones. Es muy fácil crear un plugin, como se mostró antes y explicó con más detalle aquí.

Esta página ha sido solo un aperitivo. Hay mucho mucho más sobre tareas personalizadas en la página de Tareas.

Organizar la construcción

Esta página explica la organización de la estructura de la construcción.

Por favor, lee primero las páginas anteriores de la Guía de inicio, en particular Definiciones de construcción, Grafos de tareas, Dependencias de bibliotecas y Construcciones multiproyecto antes de leer esta página.

sbt es recursivo

`build.sbt` esconde cómo trabaja en realidad sbt. Las construcciones de sbt son definidas con código de Scala. Ese código, en sí mismo, ha de ser construido. ¿Qué mejor forma de hacerlo que con sbt?

El directorio `project` *es otra construcción dentro de tu construcción*, la cual sabe cómo construir tu construcción. Para distinguir las construcciones a veces utilizamos el término **construcción propia** para referirnos a tu construcción y **metaconstrucción** para referirnos a la construcción que está en `project`. Los proyectos dentro de la metaconstrucción pueden hacer lo que cualquier otro proyecto puede. *Tu definición de construcción es un proyecto sbt.*

Y así hasta el infinito. Si quieres, puedes personalizar la definición de construcción de la definición de construcción del proyecto, creando un directorio `project/project/`.

Aquí se explica con una ilustración.

```
hello/                    # el directorio base del proyecto raíz
                          # de tu construcción

    Hello.scala           # un fichero fuente en el proyecto raíz de tu
                          # construcción (también podría estar en
                          # src/main/scala)

    build.sbt             # build.sbt es parte del código fuente para el
                          # proyecto raíz de la metaconstrucción dentro de
                          # project/; la definición de construcción para
                          # tu construcción

    project/              # directorio base del proyecto raíz de la
                          # metaconstrucción

        Dependencies.scala # un fichero fuente en el proyecto raíz de la
                          # metaconstrucción, es decir, un fichero fuente en
                          # la definición de construcción de la definición
                          # de construcción para tu construcción
```

```
assembly.sbt       # esto es parte del código fuente para el proyecto
                   # raíz de la meta-metaconstrucción en
                   # project/project; la definición de construcción de
                   # la definición de construcción

project/           # directorio base del proyecto raíz de la
                   # meta-metaconstrucción; el proyecto de la definición
                   # de construcción para la definición de construcción

    MetaDeps.scala # fichero fuente en el proyecto raíz de la
                   # meta-metaconstrucción en project/project/
```

¡No te preocupes! La mayor parte del tiempo no necesitarás nada de eso. Pero comprender los principios puede resultar útil.

Por cierto, cada vez que se utilizan ficheros acabados en `.scala` o `.sbt`, el que se llamen `build.sbt` y `Dependencies.scala` son meras convenciones. Esto significa que se permiten múltiples ficheros.

Declarar dependencias en un único lugar

Una forma de aprovechar el hecho de que los ficheros `.scala` bajo `project` acaban convirtiéndose en parte de la definición de construcción es crear `project/Dependencies.scala` para declarar las dependencias en un único lugar.

```
import sbt._

object Dependencies {
  // Versions
  lazy val akkaVersion = "2.3.8"

  // Libraries
  val akkaActor = "com.typesafe.akka" %% "akka-actor" % akkaVersion
  val akkaCluster = "com.typesafe.akka" %% "akka-cluster" % akkaVersion
  val specs2core = "org.specs2" %% "specs2-core" % "2.4.17"

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

El objeto `Dependencies` estará disponible en `build.sbt`. Para simplificar aún más el poder usar los `val` definidos en él, puedes importar `Dependencies._` en tu fichero `build.sbt`.

```
import Dependencies._
```

```
ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.14"

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

Esta técnica es útil cuando tienes construcciones multiproyecto que se empiezan a volver inmanejables y quieres asegurar que las dependencias en los subproyectos son consistentes.

Cuándo usar ficheros `.scala`

En los ficheros `.scala` puedes escribir cualquier código de Scala, incluyendo clases y objetos de primer nivel.

La solución recomendada es definir la mayoría de la configuración en un fichero `build.sbt` de un multiproyecto y utilizar ficheros `project/*.scala` para implementar tareas o compartir valores, tales como claves. El uso de ficheros `.scala` también dependerá de cuán familiarizado estéis tú o tu equipo con Scala.

Definir autoplugins

Para usuarios más avanzados, otra forma de organizar tu construcción es definir autoplugins de usar y tirar en `project/*.scala`. Al definir plugins disparados, los autoplugins pueden ser usados como una forma conveniente de inyectar tareas y comandos personalizados a lo largo de los subproyectos.

Guía de inicio - resumen

Esta página resume la Guía de inicio.

Para usar sbt existe un pequeño número de conceptos que se han de comprender. Cada uno tiene su curva de aprendizaje, pero siendo optimistas, sbt no es mucho más que esos mismos conceptos. sbt usa un pequeño conjunto de poderosos conceptos para hacer todo lo que hace.

Si has leído la Guía de inicio de principio a fin ahora ya sabes lo que necesitas saber.

sbt: Los conceptos esenciales

- Los fundamentos de Scala. Es indudablemente útil estar familiarizado con la sintaxis de Scala. Programming in Scala, escrito por el creador de Scala, es una magnífica introducción.
- Definiciones de construcción
- La definición de construcción es un gran GAD de tareas y sus dependencias.
- Para crear una entrada, emplea uno de los pocos métodos de una clave: `:=`, `+=` o `++=`.
- Cada entrada tiene un valor de un tipo en particular, determinado por la clave.
- Las *tareas* son entradas especiales donde la computación que produce el valor de la clave es re-evaluado cada vez que se lanza una tarea. Las entradas que no son tareas computan su valor una única vez, durante la carga de la definición de construcción.
- Ámbitos
- Cada clave puede tener múltiples valores, con distintos ámbitos.
- Los ámbitos pueden utilizar tres ejes: configuración, proyecto y tarea.
- Los ámbitos permiten tener diferente comportamiento por proyecto, por tarea o por configuración.
- Una configuración es una clase de construcción, como la principal (`Compile`) o la de test (`Test`).
- El eje de proyecto soporta además el ámbito de "construcción entera".
- Un ámbito puede *delegar* en otros ámbitos más generales.
- La mayoría de la configuración debe ir en `build.sbt` y el uso de ficheros `.scala` debería de estar reservado para definir clases e implementaciones de tareas más complejas.
- La definición de construcción es un proyecto sbt como los demás, ubicado en el directorio `project`.
- Los plugins extienden la definición de construcción
- Los plugins se pueden añadir con el método `addSbtPlugin` en `project/plugins.sbt` (y NO en el fichero `build.sbt` del directorio base del proyecto).

Si algunos de estos puntos no te queda claro, por favor, solicita ayuda, vuelve atrás y vuelve a leer, o haz algunos experimentos utilizando el modo interactivo de sbt.

¡Buena suerte!

Notas avanzadas

Ya que sbt es código abierto, ¡no olvides que también puedes echarle un vistazo al código fuente!