Scala Ecosystem

Mikhail Mutcianko, Alexey Shcherbakov

СПБгУ, СП

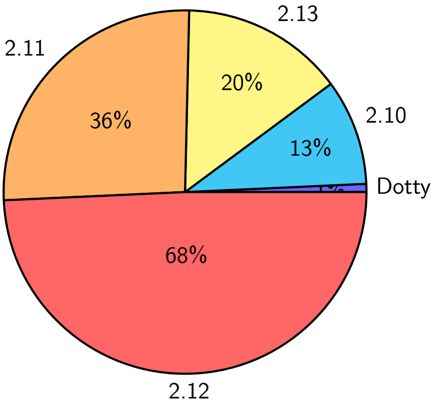
25 марта 2021

Overview

as of Apr 2020

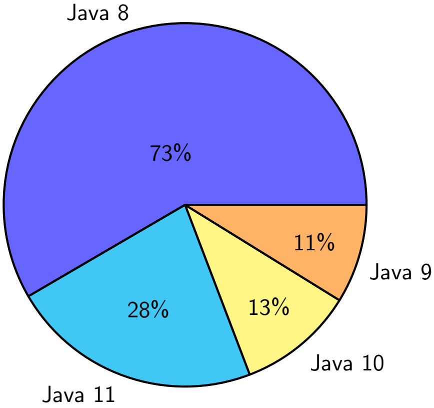
Scala versions

Which versions of Scala do you regularly use?



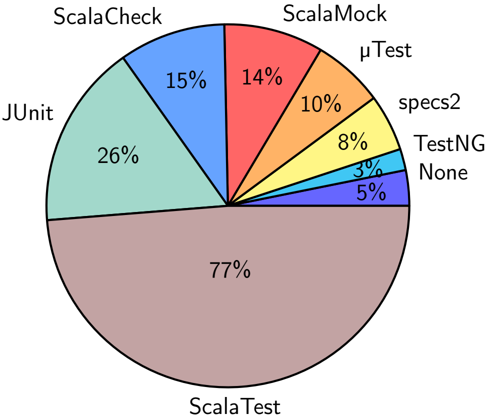
JVM Platform

Which versions of Java do you regularly use?



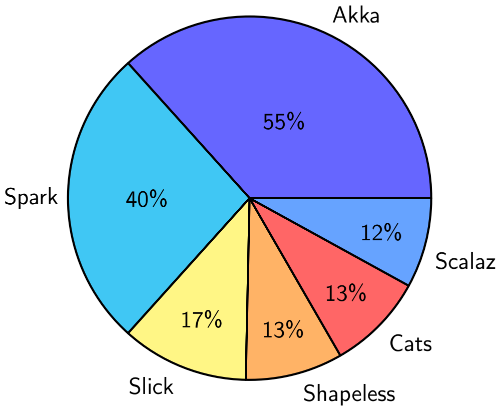
Testing

Which unit-testing frameworks do you regularly use?



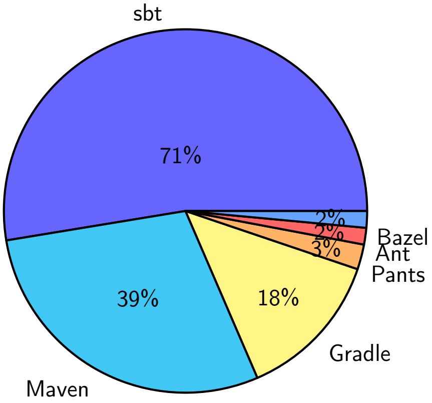
Most Popular Libraries

Which frameworks / libraries do you regularly use?



Build System

Which build systems do you regularly use?



SBT

What is a build system?

Build system is a tool to help automate development processess such as compilation of source coude, running tests and deploying the resulting application.

Real world projects

- $n * 10^5 - n * 10^6 LOC$
- 10-1000 developers on the same codebase
- $n * 10^5$ tests
- 10-100 steps to build the final application

The problem

Given: n source files

Basic tasks:

- compile sources to class files
- find classes that are tests and run? them
- run the application from classes

Build tool interface

- command line: `sbt ;compile;test;`, `mvn install`, `make -j8`
- configuration: Makefile, build.sbt, pom.xml
- remote API: gradle-daemon, sbt-server, BSP

sbt

simple build tool

- build files are defined in Scala
- incremental compilation
- test framork integrations
- build jar artifacts and publish them
- . . .

Project structure

```
build.sbt
src/
  main/
    resources/ <files to include in main jar here>
    scala/     <main Scala sources>
    java/      <main Java sources>
  test/
    scala/     <test Scala sources>
    java/      <test Java sources>
target/        <build results>
```

SBT command line basics

- run sbt in your project directory with no arguments:

```
$ sbt
```

- specify tasks to run:

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

- common commands:
  - reload — Reloads the build definition
  - clean — Deletes all generated files (in the target directory)
  - compile — Compiles the main sources
  - test — Compiles and runs all tests
  - console — run interpreter with a classpath including the compiled sources and all dependencies

Build defiinition

The build definition is described in `build.sbt` (actually any files named *.sbt) in the project's base directory. SBT build definitions consists of:

- a set of subproject definitions

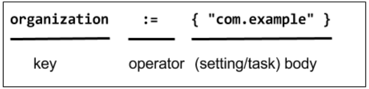
```scala
lazy val root = (project in file(".")).settings(
    name := "Hello",
    scalaVersion := "2.12.7")
```

- setting expressions

```scala
ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.10"
ThisBuild / version      := "0.1.0-SNAPSHOT"
```

SBT Settings

Settings are a sequence of key-value pairs generated by setting operators.



```
organization  :=  { "com.example" }
```

setting/task expression

- 3 kinds of keys:
  - `SettingKey[T]` — a key for a value computed once
  - `TaskKey[T]` — a key for a value, called a task, that has to be recomputed each time, potentially with side effects
  - `InputKey[T]` — a key for a task that has command line arguments as input

Key initialization operators

- := — initialize key discarding prevoius value
- += — append single element to prevoius value
- ++= — append a collection to prevoius value

SBT Task Graph

Rather than thinking of settings as key-value pairs, a better analogy would be to think of it as a directed acyclic graph (DAG) of tasks where the edges denote happens-before. Let's call this the task graph.

- depend on other values by using `.value method`
- setting keys cannot depend on tasks
- show key dependencies with `inspect` task

SBT Task Graph

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

Scopes

Previously we pretended that a key like name corresponded to one entry in sbt's map of key-value pairs. This was a simplification. In truth, each key can have an associated value in more than one context, called a scope

- a key can have a different value in each project
- compile key has a different value for main and test sources
- a global key can be "overridden"in a project

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

Scope Axis

- subproject axis
- configuration axis
- task axis

Subprojects

A project is defined by declaring a lazy val of type Project. For example, :

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

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

Aggregation and Dependencies

Aggregation

Aggregation means that running a task on the aggregate project will also run it on the aggregated projects

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

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

Aggregation and Dependencies

Aggregation

Aggregation means that running a task on the aggregate project will also run it on the aggregated projects

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

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

Classpath dependencies

A project may depend on code in another project. This is done by adding a dependsOn method call

Classpath dependencies

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

lazy val core = (project in file("core"))
  .dependsOn(util)

lazy val app = (project in file("app"))
  .dependsOn(core % "test->test;compile->compile")
```

Configurations

- `Compile` which defines the main build (src/main/scala).
- `Test` which defines how to build tests (src/test/scala).
- `Runtime` which defines the classpath for the run task.

Referring to scopes

Key scoping is done using the "/"operator:

```
organization := name.value
Compile / name := "hello"
packageBin / name := "hello"
Compile / packageBin / name := "hello"
```

Library dependencies

Unmanaged dependencies

Jar files that are assumed to already exist in the filesystem

```
unmanagedBase := baseDirectory.value / "custom_lib_folder"
unmanagedJars += baseDirectory.value / "myLib.jar"
```

Library dependencies

Unmanaged dependencies

Jar files that are assumed to already exist in the filesystem

```
unmanagedBase := baseDirectory.value / "custom_lib_folder"
unmanagedJars += baseDirectory.value / "myLib.jar"
```

Managed dependencies

Dependencies that are managed by some dependendency management programs such as Coursier or Apache Ivy

```
// libraryDependencies += groupID % artifactID % revision % configuration
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

Managed dependencies

- `libraryDependencies` is a pre-defined key in `sbt.Keys`:

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

- % is an extension method of `String` thar creates a `ModuleID`
- %% inserts binary Scala version suffix into the artifact id:

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

- use 3rd % operator to provide scope:

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

Extended build structure

sbt is recursive

The `project` directory is another build inside your build, which knows how to build your build

```
hello/                     # your build's root project's base directory
    Hello.scala         # a source file in your build's root project
    build.sbt              # build.sbt is part of the source code for
                           #   meta-build's root project inside project/;
                           #   the build definition for your build
    project/            # base directory of meta-build's root project
        build.sbt          # this is part of the source code for
                           #   meta-meta-build's root project in project/project;
                           #   build definition's build definition
        project/        # base directory of meta-meta-build's root project;
                        #   the build definition project for the build definition
            MetaDeps.scala # source file in the root project of
                           #   meta-meta-build in project/project/
```

Tracking dependencies in one place

One way of using the fact that .scala files under project is a part of the build is to create `project/Dependencies.scala` to track dependencies in one place.

```scala
object Dependencies {
  lazy val akkaVersion = "2.3.8"
  // Libraries
  val akkaActor = "com.typesafe.akka" %% "akka-actor" % akkaVersion
  val specs2core = "org.specs2" %% "specs2-core" % "2.4.17"
  // Projects
  val backendDeps =
    Seq(akkaActor, specs2core % Test)
}
```

Tracking dependencies in one place

build.sbt

```scala
import Dependencies._

ThisBuild / organization := "com.example"
ThisBuild / version      := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.12.10"

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

SBT plugins

Prts of a build definition can be loaded from classes and jars — thus creating plugins

- plugins are added with `addSbtPlugin(...)`:

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

- since plugin runs inside the build itself, it has to be added by the meta-build

  e.g. into `project/plugins.sbt`

- some plugins are not enabled automatically:

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