

sbt Reference Manual

目 次

Preface	4
始める sbt	4
sbt のインストール	5
豆知識	5
Mac への sbt のインストール	5
サードパーティパッケージを使ってのインストール	5
ユニバーサルパッケージからのインストール	6
Lightbend Activator	6
手動インストール	6
Windows への sbt のインストール	6
Windows インストーラ	6
ユニバーサルパッケージからのインストール	6
Lightbend Activator	6
手動インストール	6
Linux への sbt のインストール	6
ユニバーサルパッケージからのインストール	6
Ubuntu 及びその他の Debian ベースの Linux ディストリ ビューション	7
Red Hat Enterprise Linux 及びその他の RPM ベースのディ ストリビューション	7
Gentoo	8
Lightbend Activator	9
手動インストール	9
手動インストール	9
Unix	9
Windows	9
Lightbend Activator (sbt を含む) のインストール	11
Hello, World	12

ソースコードの入ったプロジェクトディレクトリを作る	12
ビルド定義	13
sbt バージョンの設定	13
ディレクトリ構造	13
ベースディレクトリ	13
ソースコード	14
sbt ビルド定義ファイル	14
ビルド成果物	15
バージョン管理の設定	15
実行	15
インタラクティブモード	15
バッチモード	16
継続的ビルドとテスト	16
よく使われるコマンド	16
タブ補完	17
履歴コマンド	18
.sbt ビルド定義	18
3 種類のビルド定義	18
ビルド定義とは何か	19
build.sbt はどのように settings を定義するか	20
キー	21
タスクとセッティングの定義	22
sbt インタラクティブモードにおけるキー	23
build.sbt 内の import 文	24
ライブラリへの依存性を加える	24
スコープ	25
キーに関する本当の話	25
スコープ軸	26
グローバルスコープ	27
委譲	27
sbt 実行中のスコープ付きキーの参照方法	28
スコープ付きキーの表記例	28
スコープの検査	29
ビルド定義からスコープを参照する	31
いつスコープを指定すべきか	32
他の種類のセッティング	33
復習: セッティング	33
既存の値に追加する: += と ++=	33
他のキーの値を基に値を計算	34

依存性を用いた追加: += と ++=	36
ライブラリ依存性	36
アンマネージ依存性 (Unmanaged Dependencies)	37
マネージ依存性 (Managed Dependencies)	37
マルチプロジェクト・ビルド	41
複数のプロジェクト	41
依存関係	42
デフォルトルートプロジェクト	44
プロジェクトの切り替え	45
共通のコード	46
プラグインの使用	46
プラグインとは何か	46
プラグインの宣言	46
auto plugin の有効化と無効化	47
グローバル・プラグイン	48
利用可能なプラグイン	49
カスタムセッティングとタスク	49
キーを定義する	49
タスクを実装する	50
タスクの実行意味論	51
プラグイン化しよう	56
ビルドの整理	56
sbt は再帰的だ	56
ライブラリ依存性を一箇所にまとめる	57
いつ .scala ファイルを使うべきか	58
auto plugin を定義する	59
まとめ	59
sbt: コア・コンセプト	59
上級者への注意	60
付録: bare .sbt ビルド定義	60
bare .sbt ビルド定義とは何か	60
(0.13.7 以前) 設定は空白行で区切る	61
付録: .scala ビルド定義	61
build.sbt と Build.scala の関係	61
インタラクティブモードにおけるビルド定義	64
注意: 全てが immutable だ	64
各論	65
プラグインとベストプラクティス	65
sbt プラグインをテストする	65

scripted test framework	65
ステップ 1: snapshot	65
ステップ 2: scripted-plugin	66
ステップ 3: src/sbt-test	66
ステップ 4: スクリプトを書く	67
ステップ 5: スクリプトを実行する	68
ステップ 6: カスタムアサーション	68
ステップ 7: テストをテストする	69
ステップ 8: インスパイアされる	70
How to	70
逐次実行	70
Def.sequential を用いて逐次タスクを定義する	71
Def.taskDyn を用いて動的タスクを定義する	72
インプットタスクの後で何かする	73
Def.inputTaskDyn を用いた動的インプットタスクの定義	75
コマンドを用いた逐次実行	76

Preface

始める sbt

sbt には、柔軟かつ強力なビルド定義 (Build Definition) を支えるための独自の概念がいくつか存在している。その概念は決して多くはないが、sbt は他のビルドシステムとは一味違うので、ドキュメントを読まずに使おうとすると、きっと細かい点でつまづいてしまうだろう。

この「始める sbt」では、sbt ビルド定義を作成してメンテナンスしていく上で知っておくべき概念を説明していく。

このガイドを一通り読んでおくことを強く推奨したい。

もしどうしても時間がないというなら、最も重要な概念は sbt ビルド定義、スコープ、と他の種類のセッティング に書かれている。ただし、それ以外のページを読み飛ばしても大丈夫かは保証できない。

このガイドの読み方だが、後ろの方のページはその前のページで紹介された概念の理解を前提に書かれているので、最初から順番に読み進めていくのがベストだ。

sbt を試してくれることに感謝する。ぜひ楽しいんでほしい！

誤訳の報告はこちらへ。sbt0.13 での変更点や新機能に興味があるなら、sbt 0.13.0 の変更点 を読むとよいだろう。

sbt のインストール

sbt プロジェクトを作るためには、以下の手順をたどる必要がある:

- sbt をインストールして起動スクリプトを作る。
- 簡単な hello world プロジェクトをセットアップする。
- ソースファイルの入ったプロジェクトディレクトリを作る。
- ビルド定義を作る。
- 実行するを読んで、sbt の実行方法を知る。
- .sbt ビルド定義を読んで、ビルド定義についてもっと詳しく知る。

究極的には sbt のインストールはランチャー JAR とシェルスクリプトの 2 つを用意するだけだが、利用するプラットフォームによってはもう少し簡単なインストール方法もいくつか提供されている。Mac、Windows、Linux、Lightbend Activator、もしくは手動インストールの手順を参照してほしい。

豆知識

sbt の実行が上手くいかない場合は、Setup Notes のターミナルの文字エンコーディング、HTTP プロキシ、JVM のオプションに関する説明を参照してほしい。

Mac への sbt のインストール

サードパーティパッケージを使っのインストール

注意: サードパーティが提供するパッケージは最新版を使っているとは限らない。何か問題があれば、パッケージメンテナに報告してほしい。

Macports

```
$ port install sbt
```

Homebrew

```
$ brew install sbt
```

ユニバーサルパッケージからのインストール

ZIP か TGZ をダウンロードしてきて解凍する。

Lightbend Activator

Lightbend Activator の手順を参照。

手動インストール

手動インストールの手順を参照。

Windows への sbt のインストール

Windows インストーラ

msi インストーラをダウンロードしてインストールする。

ユニバーサルパッケージからのインストール

ZIP か TGZ をダウンロードしてきて解凍する。

Lightbend Activator

Lightbend Activator の手順を参照。

手動インストール

手動インストールの手順を参照。

Linux への sbt のインストール

ユニバーサルパッケージからのインストール

ZIP か TGZ をダウンロードしてきて解凍する。

Ubuntu 及びその他の Debian ベースの Linux ディストリビューション

DEB は sbt による公式パッケージだ。

Ubuntu 及びその他の Debian ベースのディストリビューションは DEB フォーマットを用いるが、ローカルの DEB ファイルからソフトウェアをインストールすることは稀だ。これらのディストロは通常コマンドラインや GUI 上から使えるパッケージ・マネージャがあつて (例: `apt-get`、`aptitude`、`Synaptic` など)、インストールはそれらから行う。ターミナル上から以下を実行すると sbt をインストールできる (superuser 権限を必要とするため、`sudo` を使っている)。

```
echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF7
sudo apt-get update
sudo apt-get install sbt
```

パッケージ・マネージャは設定されたりポジトリに指定されたパッケージがあるか確認しに行く。sbt のバイナリは Bintray にて公開されており、都合の良いことに Bintray は APT リポジトリを提供している。そのため、このリポジトリをパッケージ・マネージャに追加しさえすればよい。

注意 `sbt/website#127` で報告されている通り、`https` を使用するとセグメンテーション違反が発生する場合がある。

sbt を最初にインストールした後は、このパッケージは `aptitude` や `Synaptic` 上から管理することができる (パッケージ・キャッシュの更新を忘れずに)。追加された APT リポジトリは「システム設定 -> ソフトウェアとアップデート -> 他のソフトウェア」の一番下に表示されているはずだ:

Red Hat Enterprise Linux 及びその他の RPM ベースのディストリビューション

RPM は sbt による公式パッケージだ。

Red Hat Enterprise Linux 及びその他の RPM ベースのディストリビューションは RPM フォーマットを用いる。ターミナル上から以下を実行すると sbt をインストールできる (superuser 権限を必要とするため、`sudo` を使っている)。

```
curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
sudo yum install sbt
```



図 1: Ubuntu Software & Updates Screenshot

sbt のバイナリは Bintray にて公開されており、Bintray は RPM リポジトリを提供する。そのため、このリポジトリをパッケージ・マネージャに追加する必要がある。

注意: これらのパッケージに問題があれば、sbt-launcher-package プロジェクトに報告してほしい。

Gentoo

公式には sbt の ebuild は提供されていないが、バイナリから sbt をマージする ebuild が公開されているようだ。この ebuild を使って sbt をマージするには:

```
mkdir -p /usr/local/portage && cd /usr/local/portage
git clone git://github.com/whiter4bbit/overlays.git
echo "PORTDIR_OVERLAY=$PORTDIR_OVERLAY /usr/local/portage/overlays" >> /etc/make.conf
emerge sbt-bin
```

注意: この ebuild に関する問題はこちらへ報告してほしい。

Lightbend Activator

Lightbend Activator の手順を参照。

手動インストール

手動インストールの手順を参照。

手動インストール

sbt-launch.jar をダウンロードして起動スクリプトを書くことで手動でインストールできる。

Unix

sbt-launch.jar を ~/bin 配下に置く。

以下のようなスクリプトを ~/bin/sbt として作成し JAR を起動する:

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

スクリプトを実行可能にする:

```
$ chmod u+x ~/bin/sbt
```

Windows

ターミナルの種類と Cygwin を使っているかによって Windows 環境での手動インストールは変わってくる。いずれにせよ、バッチファイルもしくはスクリプトにパスを通すことでコマンドプロンプトから sbt と打ち込めば sbt が起動できるようにする。あとは、必要に応じて JVM の起動設定も調整する。

非 Cygwin

標準の Windows ターミナルを使っている非 Cygwin ユーザは、以下のバッチファイル `sbt.bat` を作る:

```
set SCRIPT_DIR=%~dp0
java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M -jar "%~dp0\sbt-launch.jar" %*

そしてダウンロードしてきた sbt-launch.jar はバッチファイルと同じディレクトリに置く。
```

標準 Windows ターミナルを使った Cygwin

標準 Windows ターミナルとともに Cygwin を使っている場合は、`~/bin/sbt` という名前で `bash` スクリプトを作る:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
java $SBT_OPTS -jar sbt-launch.jar "$@"

sbt-launch.jar の所はダウンロードしてきた sbt-launch.jar へのパスで置き換える。必要ならば cygpath を使う。スクリプトを実行可能にする:

$ chmod u+x ~/bin/sbt
```

Ansi ターミナルを使った Cygwin

Ansi ターミナル (Ansi エスケープをサポートして、`stty` によって設定できる) を使って Cygwin を実行している場合は、`~/bin/sbt` という名前で `bash` スクリプトを作る:

```
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=256M"
stty -icanon min 1 -echo > /dev/null 2>&1
java -Djline.terminal=jline.UnixTerminal -Dsbt.cygwin=true $SBT_OPTS -jar sbt-launch.jar %*
stty icanon echo > /dev/null 2>&1

sbt-launch.jar の所はダウンロードしてきた sbt-launch.jar へのパスで置き換える。必要ならば cygpath を使う。そして、スクリプトを実行可能にする:

$ chmod u+x ~/bin/sbt
```

scala コンソールでバックスペースが正しく動作するためには、バックスペースが `stty` で設定された消去文字を送信している必要がある。デフォルトの Cygwin のターミナル (`mintty`) を使っていて、消去文字が Cygwin のデフォルトである `^H` を使っている場合は Options -> Keys “Backspace sends `^H`” の設定をチェックする必要がある。

注意: 他の設定は現在サポートしていない。何か良い方法があれば pull request を送ってほしい。

訳注:

- 32bitOS の場合 `-Xmx1536M` だと JVM のメモリの制限によりうまくいかないで、`-Xmx1024M` などに減らす必要がある。
- sbt0.13.0 以降、windows の場合は、`-Dinput.encoding=Cp1252` を指定しないと矢印キーでの履歴参照などが文字化けするようなので、設定してください。詳しい議論などはここを参照

Lightbend Activator (sbt を含む) のインストール

Lightbend Activator は `activator ui` と `activator new` という 2 つのコマンドを追加するカスタム版の sbt だ。つまり、`activator` は sbt のスーパーセットであると言える。

Activator は typesafe.com で入手できる。

このガイドで `sbt ~test` というようなコマンドラインがあれば、`activator ~test` と打ち込めばそのまま動作するはずだ。Activator の「中の人」は sbt なので、全ての Activator プロジェクトは sbt で開くことができ、そのまた逆も成り立つ。

Activator をダウンロードすると `activator` スクリプトと `activator-launch.jar` が含まれている。これは手動インストールで解説されている sbt スクリプトと sbt launcher JAR に相当する。以下が sbt の手動インストールとの違いだ:

- 引数なしで `activator` と入力すると `activator shell` モードか `activator ui` モードに入るかを推論する。コマンドラインプロンプトを強制したい場合は `activator shell` と入力する。
- `activator new` を使うことで豊富な テンプレートのカタログを元にプロジェクトを新規作成することができる。例えば、`play-scala` テンプレートを使うと Scala の Play Framework アプリを作れる。
- `activator ui` は、クイックスタート UI を起動する。これを使ってテンプレート付属のチュートリアルを読みながら作業できる (カタログ内の多くのテンプレートにはチュートリアルが付属している)。

Activator には、起動スクリプトと起動 JAR のみのミニマル版ダウンロードと、Scala、Akka、そして Play Framework の JAR がすぐに使えるよう Ivy リポジトリを含む完全版ダウンロードの 2 種類がある。

Hello, World

このページは、既に sbt をインストールしたことを前提とする。

ソースコードの入ったプロジェクトディレクトリを作る

一つのソースファイルを含むディレクトリでも、一応有効な sbt プロジェクトとなりうる。試しに、hello ディレクトリを作って、以下の内容の hw.scala というファイルを作成する:

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

次に hello ディレクトリ内から sbt を起動して sbt のインタラクティブコンソールに run と入力する。Linux、Mac OS X の場合、コマンドは以下のようになる:

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

この例では、sbt はただデフォルトの規約によって動作している。sbt は以下のものを自動的に検知する:

- ベースディレクトリにあるソースファイル
- src/main/scala か src/main/java 内のソースファイル
- src/test/scala か src/test/java 内のテストソースファイル
- src/main/resources か src/test/resources 内のデータファイル
- lib 内の jar ファイル

デフォルトでは、sbt は sbt 自身が使っている Scala のバージョンを使ってプロジェクトをビルドする。

sbt run でプロジェクトを実行したり、sbt console で Scala REPL に入ることができる。sbt console は君のプロジェクトにクラスパスを通すので、プロジェクトのコードを使った Scala コード例をその場で試すことができる。

ビルド定義

ほとんどのプロジェクトでは何らかの手動設定が必要になるだろう。基本的なビルド設定方法はプロジェクトのベースディレクトリに `build.sbt` というファイルとして配置されるものだ。例えば、君のプロジェクトが `hello` ディレクトリにある場合、`hello/build.sbt` はこんな感じになる：

```
lazy val root = (project in file(".")).
  settings(
    name := "hello",
    version := "1.0",
    scalaVersion := "2.11.8"
  )
```

.sbt ビルド定義で、`build.sbt` の書き方をもっと詳しく説明する。もし君のプロジェクトを jar ファイルにパッケージ化するつもりなら、最低でも `build.sbt` に `name` と `version` は書いておこう。

sbt バージョンの設定

`hello/project/build.properties` というファイルを作ること、特定のバージョンの sbt を強制することができる。このファイルに、以下のように書く：

```
sbt.version=0.13.12
```

sbt はリリース間で 99% ソースコード互換性を維持しているが、sbt バージョンを `project/build.properties` に設定しておくことで、不要な混乱を避けることができるだろう。

ディレクトリ構造

このページは、sbt をインストールして、Hello, World を読んだことを前提とする。

ベースディレクトリ

sbt 用語では「ベースディレクトリ (base directory)」はプロジェクトが入ったディレクトリを指す。Hello, World での例のように、`hello/build.sbt` と `hello/hw.scala` が入った `hello` プロジェクトを作った場合、ベースディレクトリは `hello` となる。

ソースコード

ソースコードは `hello/hw.scala` のようにプロジェクトのベースディレクトリに置くこともできる。しかし、現実のプロジェクトでは、ほとんどの場合、そのようにはしないだろう。きっとプロジェクトがゴチャゴチャになってしまうからだ。

sbt はデフォルトで Maven と同じディレクトリ構造を使う（全てのパスはベースディレクトリからの相対パスとする）：

```
src/  
  main/  
    resources/  
      <メインの jar に含むデータファイル>  
    scala/  
      <メインの Scala ソースファイル>  
    java/  
      <メインの Java ソースファイル>  
  test/  
    resources/  
      <テストの jar に含むデータファイル>  
    scala/  
      <テストの Scala ソースファイル>  
    java/  
      <テストの Java ソースファイル>
```

`src/` 内の他のディレクトリは無視される。また、隠しディレクトリも無視される。

sbt ビルド定義ファイル

プロジェクトのベースディレクトリに `build.sbt` を置くことはご理解いただけたかと思う。それ以外の sbt 関連の設定ファイルは `project` 配下のサブディレクトリに置かれる。

`project` には `.scala` ファイルを含むことができ、それは `.sbt` ファイルと組み合わせあって一つの完全なビルド定義を構成する。詳しくは、ビルドの整理を参照。

```
build.sbt  
project/  
  Build.scala
```

project 内に .sbt があるのを見ることがあるかもしれないが、それはプロジェクトのベースディレクトリ下の .sbt とはまた別物だ。これに関しては他に前提となる知識が必要なので後ほど説明する。

ビルド成果物

生成されたファイル（コンパイルされたクラスファイル、パッケージ化された jar ファイル、managed 配下のファイル、キャッシュとドキュメンテーション）は、デフォルトでは target ディレクトリに出力される。

バージョン管理の設定

.gitignore（もしくは、他のバージョン管理システムの同様のファイル）には以下を追加しておくといだろう：

```
target/
```

ここでは（ディレクトリだけにマッチさせるために）語尾の / は意図的につけていて、一方で（普通の target/ に加えて project/target/ にもマッチさせるために）先頭の / は意図的に つけていないことに注意。

実行

このページではプロジェクトをセットアップした後の sbt の使い方を説明する。君が sbt をインストールして、Hello, World か他のプロジェクトを作ったことを前提とする。

インタラクティブモード

プロジェクトのベースディレクトリで、sbt を引数なしで実行する：

```
$ sbt
```

sbt をコマンドライン引数なしで実行するとインタラクティブモードで起動する。インタラクティブモードにはコマンドプロンプト（とタブ補完と履歴も！）がある。

例えば、compile と sbt プロンプトに入力する：

```
> compile
```

もう一度 `compile` するには、上矢印を押して、エンターキーを押す。

君のプログラムを実行するには、`run` と入力する。

インタラクティブモードを終了するには、`exit` と入力するか、`Ctrl+D` (Unix) か `Ctrl+Z` (Windows) を押す。

バッチモード

`sbt` のコマンドを空白で区切られたリストとして引数に指定すると `sbt` をバッチモードで実行することができる。引数を取る `sbt` コマンドの場合は、コマンドと引数の両方を引用符で囲むことで一つの引数として `sbt` に渡す。例えば、

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

この例では、`testOnly` は `TestA` と `TestB` の二つの引数を取る。コマンドは順に実行される（この場合 `clean`、`compile`、そして `testOnly`）。

継続的ビルドとテスト

編集 || コンパイル || テストのサイクルを速めるために、ソースファイルを保存する度 `sbt` に自動的に再コンパイルを実行させることができる。

ソースファイルが変更されたことを検知してコマンドを実行するには、コマンドの先頭に `~` をつける。例えば、インタラクティブモードで、これを試してみよう：

```
> ~ compile
```

このファイル変更監視状態を止めるにはエンターキーを押す。

先頭の `~` はインタラクティブモードでもバッチモードでも使うことができる。

詳しくは、[Triggered Execution](#) 参照。

よく使われるコマンド

最もよく使われる `sbt` コマンドを紹介する。全ての一覧は [Command Line Reference](#) を参照。

<td><tt>clean</tt></td>

<td> (<tt>target</tt> ディレクトリにある) 全ての生成されたファイルを削除する。</td>

<td><tt>compile</tt></td>

<td>

(<tt>src/main/scala</tt> と <tt>src/main/java</tt> ディレクトリにある)

メインのソースをコンパイルする。</td>

<td><tt>test</tt></td>

<td>全てのテストをコンパイルし実行する。</td>

<td><tt>console</tt></td>

<td>コンパイル済のソースと依存ライブラリにクラスパスを通して、Scala インタプリタを開始する。

sbt に戻るには、:quit と入力するか、Ctrl+D (Unix) か Ctrl+Z (Windows) を押す。

<td><nobr><tt>run <argument>*</tt></nobr></td>

<td><tt>sbt</tt> と同じ仮想マシン上で、プロジェクトのメインクラスを実行する。</td>

<td><tt>package</tt></td>

<td><tt>src/main/resources</tt> 内のファイルと <tt>src/main/scala</tt> と

<tt>src/main/java</tt>

からコンパイルされたクラスファイルを含む jar を作る。</td>

<td><tt>help <command></tt></td>

<td>指定されたコマンドの詳細なヘルプを表示する。コマンドが指定されていない場合は、

全てのコマンドの簡単な説明を表示する。

<td><tt>reload</tt></td>

<td>ビルド定義 (<tt>build.sbt</tt>、<tt>project/*.scala</tt>、<tt>project/*.sbt</tt> ファイル) を再読み込みする。

ビルド定義を変更した場合に必要。

タブ補完

インタラクティブモードには、空のプロンプトの状態を含め、タブ補完がある。sbt の特殊な慣例として、タブを一度押すとよく使われる候補だけが表示され、複数回押すと、より多くの冗長な候補一覧が表示される。

履歴コマンド

インタラクティブモードは、sbt を終了して再起動した後でも履歴を覚えて
いる。履歴にアクセスする最も簡単な方法は矢印キーを使うことだ。以下の
コマンドも使うことができる:

<td><tt>!
</td>

<td>履歴コマンドのヘルプを表示する。</td>

<td><tt>!!</td>

<td>直前のコマンドを再実行する。</td>

<td><tt>!:</td>

<td>全てのコマンド履歴を表示する。</td>

<td><tt>!:n</td>

<td>最後の <tt>n</td>

<td><tt>!n</td>

<td><tt>!:</td>

<td><tt>!-n</td>

<td><tt>n</td>

<td><tt>!string</td>

<td>'string' から始まる最近のコマンドを実行する。</td>

<td><tt>!?string</td>

<td>'string' を含む最近のコマンドを実行する。</td>

.sbt ビルド定義

このページでは、多少の「理論」も含めた sbt のビルド定義 (build definition)
と build.sbt の構文を説明する。君が sbt の使い方を分かっている、「始め
る sbt」の前のページも読んだことを前提とする。

3 種類のビルド定義

ビルド定義には以下の 3 種類がある:

1. マルチプロジェクトでの .sbt ビルド定義
2. bare .sbt ビルド定義
3. .scala ビルド定義

このページでは最も新しいマルチプロジェクトでの `.sbt` ビルド定義を紹介する。これは従来の 2 つのビルド定義の長所を組み合わせたもので、全ての用途に適している。以前に書かれたビルド定義を使って作業をするときは、古い種類のものも目にするかもしれない。それらに関しては (このガイドの後ほどの) bare `.sbt` ビルド定義と `.scala` ビルド定義を参照。

さらに、ビルド定義は `project/` ディレクトリ直下に置かれた `.scala` で終わるファイルを含むことができ、そこで共通の関数や値を定義することもできる。

ビルド定義とは何か

**** ここは読んで下さい ****

あらかじめ決められたディレクトリを走査し、ビルド定義に関するファイル群を処理した後、最終的に `sbt` は `Project` の定義群を作る。

例えば、カレントディレクトリにあるプロジェクトの `Project` 定義は `build.sbt` に以下のように記述できる：

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

それぞれのプロジェクトは、そのプロジェクトを記述する不変 `Map` (キーと値のペア群のセット) に関連付けられる。

例えば、`name` というキーがあるが、それはプロジェクト名という文字列の値に関連付けられる。

ビルド定義ファイルは直接的には `sbt` の `Map` に影響を与えない。

その代わり、ビルド定義は `Setting[T]` 型のオブジェクトを含む巨大なリストを作る。 `T` は `Map` における値の型だ。(Scala の `Setting[T]` は Java の `Setting<T>` と同様。) `Setting` は、新しいキーと値のペアの追加、既存の値への付加のような `Map` の変換_を記述する。(不変データ構造と関数型プログラミングの精神に則り、この変換は新しい `Map` を返し、古い `Map` は更新されない。)

カレントディレクトリに位置するプロジェクトに対してプロジェクト名のための `Setting[String]` を関連付けるためには以下のように書く：

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

この `Setting[String]` は `name` キーを追加（もしくは置換）して `"hello"` という値に設定することで `Map` を変換する。変換された `Map` は新しい `sbt` の `Map` となる。

`Map` を作るにあたり、同じキーへのすべての変更が同時に実行されるため、そして、他のキーに依存する値の処理が依存するキーの後に処理されるように、`sbt` はまず `Setting` のリストをソートする。次にソートされた `Setting` のリストを順番にみていって、一つずつ `Map` に適用する。

まとめ: ビルド定義とは `Setting[T]` のリストを持った `Project` を定義するものであり、`Setting[T]` は `sbt` が持つキーと値のペアの `Map` に作用する変換を表し、その `T` は値の型である。

build.sbt はどのように settings を定義するか

`build.sbt` が定義する `Project` は `settings` と呼ばれる `Scala` の式のリストを持つ。

以下に具体例を示す:

```
lazy val commonSettings = Seq(
  organization := "com.example",
  version := "0.1.0",
  scalaVersion := "2.11.8"
)

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

それぞれの `Setting` は `Scala` の式で定義される。`settings` 内の式は、それぞれ独立しており、完全な `Scala` の文ではなく、式である。

`build.sbt` 内には `val`、`lazy val`、`def` を定義することもできる。`build.sbt` において、トップレベルで `object` や `class` を定義することはできない。それらが必要なら `project/` 配下に `.scala` ビルド定義として完全な `Scala` ソースファイルを置くべきだろう。

左辺値の `name`、`version`、および `scalaVersion` は `_キー_` である。キーは `SettingKey[T]`、`TaskKey[T]`、もしくは `InputKey[T]` のインスタンスで、`T` はその値の型である。キーの種類に関しては後述する。

キーには `:=` というメソッドがあり、このメソッドは `Setting[T]` を返す。
Java っぽい構文でこのメソッドを呼び出すこともできる:

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

しかし、Scala なら `name := "hello"` と書くこともできる (Scala では全てのメソッドがどちらの構文でも書ける)。

`name` キーの `:=` メソッドは `Setting` を返すが、特に `Setting[String]` を返す。`String` は `name` 自体の型にも表れていて、そちらは `SettingKey[String]` となる。ここで返された `Setting[String]` は、sbt の Map における `name` というキーを追加または置換して `"hello"` という値に設定する変換である。

誤った型の値を使おうとするとビルド定義はコンパイルエラーになる:

```
lazy val root = (project in file(".")).
  settings(
    name := 42 // コンパイルできない
  )
```

キー

種類

キーには三種類ある:

- `SettingKey[T]`: 一度だけ値が計算されるキー (値はプロジェクトの読み込み時に計算され、保存される)。
- `TaskKey[T]`: 毎回再計算される `_タスク_` を呼び出す、副作用を伴う可能性のある値のキー。
- `InputKey[T]`: コマンドラインの引数を入力として受け取るタスクのキー。「始める sbt」では `InputKey` を説明しないので、このガイドを終えた後で、Input Tasks を読んでみよう。

組み込みのキー

組み込みのキーは `Keys` と呼ばれるオブジェクトのフィールドにすぎない。`build.sbt` は、自動的に `import sbt.Keys._` するため、`sbt.Keys.name` は `name` として参照することができる。

カスタムキー

カスタムキーは `settingKey`、`taskKey`、`inputKey` といった生成メソッドを用いて定義する。どのメソッドでもキーに関連する型パラメータを必要とする。キーの名前は `val` で宣言された変数の名前がそのまま用いられる。例として、新しく `hello` と名づけたキーを定義してみよう。

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

実は `.sbt` ファイルには、設定を記述するのに必要な `val` や `def` を含めることもできる。これらの定義はファイル内のどこで書かれてもプロジェクトの設定より前に評価される。`val` や `def` を用いた定義群は空白行によって他の設定から区切らなければならない。

注意 一般的に、初期化順問題を避けるために `val` の代わりに `lazy val` が用いられることが多い。

タスクキーかセッティングキーか

`TaskKey[T]` は、`_タスク_`を定義しているといわれる。タスクは、`compile` や `package` のような作業だ。タスクは `Unit` を返すかもしれないし (`Unit` は、Scala での `void` だ)、タスクに関連した値を返すかもしれない。例えば、`package` は作成した `jar` ファイルを値として返す `TaskKey[File]` だ。

例えばインタラクティブモードの `sbt` プロンプトに `compile` と入力するなど、何らかのタスクを実行する度に、`sbt` はそのタスクを一回だけ再実行する。

プロジェクトを記述する `sbt` の `Map` は、`name` のようなセッティング (setting) であれば、その文字列の値をキャッシュすることができるが、`compile` のようなタスク (task) の場合は実行可能コードを保持しておく必要がある (たとえその実行可能コードが最終的に文字列を返したとしても、それは毎回再実行されなければならない)。

あるキーがあるとき、それは常にタスクかただのセッティングかのどちらかを参照する。つまり、キーの「タスク性」(毎回再実行するかどうか) はそのキーの特性であり、その値にはよらない。

タスクとセッティングの定義

`:=` を使うことで、タスクに任意の演算を代入することができる。セッティングを定義すると、その値はプロジェクトがロードされた時に一度だけ演算が行われる。タスクを定義すると、その演算はタスクの実行毎に毎回再実行される。

例えば、少し前に宣言した `hello` というタスクはこのように実装できる:

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

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

セッティングの定義は既に何度か見ていると思うが、プロジェクト名の定義はこのようにできる:

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

タスクとセッティングの型

型システムの視点から考えると、タスクキー (task key) から作られた `Setting` は、セッティングキー (setting key) から作られたそれとは少し異なるものだ。 `taskKey := 42` は `Setting[Task[T]]` の戻り値を返すが、 `settingKey := 42` は `Setting[T]` の戻り値を返す。タスクが実行されるとタスクキーは型 `T` の値を返すため、ほとんどの用途において、これによる影響は特にない。

`T` と `Task[T]` の型の違いによる影響が一つある。それは、セッティングキーはキャッシュされていて、再実行されないため、タスクキーに依存できないということだ。このことについては、後ほどの他の種類のセッティングにて詳しくみていく。

sbt インタラクティブモードにおけるキー

sbt のインタラクティブモードからタスクの名前を入力することで、どのタスクでも実行することができる。それが `compile` と入力することでコンパイルタスクが起動する仕組みだ。つまり、`compile` はタスクキーだ。

タスクキーのかわりにセッティングキーの名前を入力すると、セッティングキーの値が表示される。タスクキーの名前を入力すると、タスクを実行するが、その戻り値は表示されないため、タスクの戻り値を表示するには素の `<タスク名>` ではなく、`show <タスク名>` と入力する。Scala の慣例にならない、ビルド定義ファイル内ではキーはキャメルケース (camelCase) で命名する。

あるキーについてより詳しい情報を得るには、sbt インタラクティブモードで `inspect <キー名>` と入力する。 `inspect` が表示する情報の中にはまだよく分からない点もあるかもしれないが、一番上にはセッティングの値の型と、セッティングの簡単な説明が表示されていることだろう。

build.sbt 内の import 文

build.sbt の一番上に import 文を書くことができ、それらは空行で分けなくてもよい。

デフォルトでは以下のものが自動的にインポートされる:

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

(さらに、.scala ファイルがある場合は、それらの全ての Build と Plugin の内容もインポートされる。これに関しては、.scala ビルド定義でさらに詳しく。)

ライブラリへの依存性を加える

サードパーティのライブラリに依存するには二つの方法がある。第一は lib/ に jar ファイルを入れてしまう方法で (アンマネージ依存性、unmanaged dependency)、第二はマネージ依存性 (managed dependency) を加えることで、build.sbt ではこのようになる:

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

```
lazy val commonSettings = Seq(
  organization := "com.example",
  version := "0.1.0",
  scalaVersion := "2.11.4"
)
```

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


これで Apache Derby ライブラリのバージョン 10.4.1.3 へのマネージ依存性を加えることができた。

`libraryDependencies` キーは二つの複雑な点がある: `:=` ではなく `+=` を使うことと、`%` メソッドだ。後で他の種類のセッティングで説明するが、`+=` はキーの古い値を上書きする代わりに新しい値を追加する。`%` メソッドは文字列から Ivy モジュール ID を構築するのに使われ、これはライブラリ依存性で説明する。

ライブラリ依存性に関する詳細については、このガイドの後ろの方までとっておくことにする。後ほど一ページを割いて丁寧に説明する。

スコープ

このページではスコープの説明をする。君が、前のページの `.sbt` ビルド定義を読んで理解したことを前提とする。

キーに関する本当の話

前のページでは、あたかも `name` のようなキーは単一の `sbt` の `Map` のキー・値ペアの項目に対応するかのように説明をしてきた。しかし、それは実際よりも物事を単純化している。

実のところ、全てのキーは「スコープ」と呼ばれる文脈に関連付けられた値を複数もつことができる。

以下に具体例で説明する:

- ビルド定義に複数のプロジェクトがあれば、それぞれのプロジェクトにおいて同じキーが別の値を取ることができる。
- メインのソースとテストとのソースが異なるようにコンパイルしたければ、`compile` キーは別の値をとることができる。
- (jar パッケージの作成のオプションを表す) `packageOption` キーはクラスファイルのパッケージ (`packageBin`) とソースコードのパッケージ (`packageSrc`) で異なる値をとることができる。

スコープによって値が異なる可能性があるため、あるキーへの単一の値は存在しない。

しかし、`_スコープ付き_` キーには単一の値が存在する。

これまで見てきたように `sbt` がプロジェクトを記述するキーと値のマップを生成するためのセッティングキーのリストを処理していると考えるなら、そ

のキーと値の Map におけるキーとは、実は `_` スコープ付き `_` キーである。また、(`build.sbt` などの) ビルド定義内のセッティングもまたスコープ付きキーである。

スコープは、暗黙に存在していたり、デフォルトのものがあったりするが、もしそのデフォルトが適切でなければ `build.sbt` で必要なスコープを指定する必要があるだろう。

スコープ軸

スコープ軸 (scope axis) は、型であり、そのインスタンスは独自のスコープを定義する (つまり、各インスタンスはキーの独自の値を持つことができる)。

スコープ軸は三つある:

- プロジェクト
- コンフィギュレーション
- タスク

プロジェクト軸によるスコープ付け

一つのビルドに複数のプロジェクトを入れる場合、それぞれのプロジェクトにセッティングが必要だ。つまり、キーはプロジェクトによりスコープ付けされる。

プロジェクト軸は「ビルド全体」に設定することもでき、その場合はセッティングは単一のプロジェクトではなくビルド全体に適用される。ビルドレベルでのセッティングは、プロジェクトが特定のセッティングを定義しない場合のフォールバックとして使われることがよくある。

コンフィギュレーション軸によるスコープ付け

コンフィギュレーション (configuration) は、ビルドの種類を定義し、独自のクラスパス、ソース、生成パッケージなどをもつことができる。コンフィギュレーションの概念は、`sbt` がマネージ依存性に使っている `Ivy` と、`MavenScopes` に由来する。

`sbt` で使われるコンフィギュレーションには以下のものがある:

- `Compile` は、メインのビルド (`src/main/scala`) を定義する。
- `Test` は、テスト (`src/test/scala`) のビルド方法を定義する。
- `Runtime` は、`run` タスクのクラスパスを定義する。

デフォルトでは、コンパイル、パッケージ化と実行に関するキーの全てはコンフィグレーションにスコープ付けされているため、コンフィギュレーションごとに異なる動作をする可能性がある。その最たる例が `compile`、`package` と `run` のタスクキーだが、（`sourceDirectories` や `scalacOptions` や `fullClasspath` など）それらのキーに__影響を及ぼす__全てのキーもコンフィグレーションにスコープ付けされている。

タスク軸によるスコープ付け

セッティングはタスクの動作に影響を与えることもできる。例えば、`packageSrc` は `packageOptions` セッティングの影響を受ける。

これをサポートするため、（`packageSrc` のような）タスクキーは、（`packageOption` のような）別のキーのスコープとなりえる。

パッケージを構築するさまざまなタスク（`packageSrc`、`packageBin`、`packageDoc`）は、`artifactName` や `packageOption` などのパッケージ関連のキーを共有することができる。これらのキーはそれぞれのパッケージタスクに対して独自の値を取ることができる。

グローバルスコープ

それぞれのスコープ軸は、その軸の型のインスタンスを代入する（例えば、タスク軸にはタスクを代入する）か、もしくは、`Global` という特殊な値を代入することができる。

`Global` はご想像通りのもので、その軸の全てのインスタンスに対して適用されるセッティングの値だ。例えば、タスク軸が `Global` ならば、全てのタスクに適用される。

委譲

スコープ付きキーは、そのスコープに関連付けられた値がなければ未定義であることもできる。

全てのスコープに対して、`sbt` には他のスコープからなるフォールバック検索パス（`fallback search path`）がある。通常は、より特定のスコープに関連付けられた値が見つからなければ、`sbt` は、`Global` や、ビルド全体スコープなど、より一般的なスコープから値を見つけ出そうとする。

この機能により、より一般的なスコープで一度値を代入することで、複数のより特定のスコープがその値を継承することを可能とする。

以下に、`inspect` を使ったキーのフォールバック検索パス、別名「委譲」(delegate) の探し方を説明する。

sbt 実行中のスコープ付きキーの参照方法

コマンドラインとインタラクティブモードにおいて、sbt はスコープ付きキーを以下のように表示する（そして、パースする）：

`{<ビルド-uri>}<プロジェクト-id>/コンフィギュレーション:タスクキー::キー`

- `{<ビルド-uri>}<プロジェクト-id>` は、プロジェクト軸を特定する。がなければ、プロジェクト軸は「ビルド全体」スコープとなる。
- コンフィギュレーションは、コンフィギュレーション軸を特定する。
- タスクキー は、タスク軸を特定する。
- キー は、スコープ付けされるキーを特定する。

全ての軸において、`*` を使って Global スコープを表すことができる。

スコープ付きキーの一部を省略すると、以下の手順で推論される：

- プロジェクトを省略した場合は、カレントプロジェクトが使われる。
- コンフィギュレーションを省略した場合は、キーに依存したコンフィギュレーションが自動検知される。
- タスクを省略した場合は、Global タスクが使われる。

さらに詳しくは、[Interacting with the Configuration System](#) 参照。

スコープ付きキーの表記例

- `fullClasspath` はキーのみを指定し、デフォルトスコープを用いる。ここでは、カレントプロジェクト、キーに依存したコンフィギュレーション、グローバルタスクスコープとなる。
- `test:fullClasspath` はコンフィギュレーションを指定する。つまりプロジェクト軸とタスク軸はデフォルトを用いつつも `test` コンフィギュレーションにおける `fullClasspath` というキーを表す。
- `*:fullClasspath` はデフォルトコンフィギュレーションを用いずに Global コンフィギュレーションを用いる事を明示している。
- `doc::fullClasspath` はプロジェクト軸とコンフィギュレーション軸はデフォルトを用いつつ、`doc` タスクスコープにおける `fullClasspath` というキーを表す。

- `{file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath` は `{file:/home/hp/checkout/hello/}` をルートディレクトリにビルドした際に含まれる `default-aea33a` というプロジェクトを指定している。さらにこのプロジェクト内の `test` コンフィギュレーションを用いる事も明示している。
- `{file:/home/hp/checkout/hello/}/test:fullClasspath` は `{file:/home/hp/checkout/hello/}` のビルド全体をプロジェクトの軸とする。
- `{.}/test:fullClasspath` は `{.}` で指定されたルートディレクトリのビルド全体をプロジェクト軸に取る。`{.}` は Scala code において `ThisBuild` と記述できる。
- `{file:/home/hp/checkout/hello/}/compile:doc::fullClasspath` は3つのスコープ軸全てを指定している。

スコープの検査

sbt のインタラクティブモードで `inspect` コマンドを使ってキーとそのスコープを把握することができる。例えば、`inspect test:full-classpath` と試してみよう:

```
$ sbt
> inspect test:fullClasspath
[info] Task: scala.collection.Seq[sbt.Attributed[java.io.File]]
[info] Description:
[info]   The exported classpath, consisting of build products and unmanaged and managed, .
[info] Provided by:
[info]   {file:/home/hp/checkout/hello/}default-aea33a/test:fullClasspath
[info] Dependencies:
[info]   test:exportedProducts
[info]   test:dependencyClasspath
[info] Reverse dependencies:
[info]   test:runMain
[info]   test:run
[info]   test:testLoader
[info]   test:console
[info] Delegates:
[info]   test:fullClasspath
[info]   runtime:fullClasspath
[info]   compile:fullClasspath
[info]   *:fullClasspath
```

```

[info] {.}/test:fullClasspath
[info] {.}/runtime:fullClasspath
[info] {.}/compile:fullClasspath
[info] {.}/*:fullClasspath
[info] */test:fullClasspath
[info] */runtime:fullClasspath
[info] */compile:fullClasspath
[info] */*:fullClasspath
[info] Related:
[info] compile:fullClasspath
[info] compile:fullClasspath(for doc)
[info] test:fullClasspath(for doc)
[info] runtime:fullClasspath

```

一行目からこれが（.sbt ビルド定義で説明されているとおり、セッティングではなく）タスクであることが分かる。このタスクの戻り値は `scala.collection.Seq[sbt.Attributed[java.io.File]]` の型をとる。

“Provided by” は、この値を定義するスコープ付きキーを指し、この場合は、`{file:/home/hp/checkout/hello/}default-aea33a/test:full-classpath`（test コンフィギュレーションと `{file:/home/hp/checkout/hello/}default-aea33a` プロジェクトにスコープ付けされた `full-classpath` キー）。

“Dependencies” はまだよく分からないかもしれないが、この説明は次のページまで待ってほしい。

ここで委譲も見ることができ、もし値が定義されていなければ、sbt は以下を検索する：

- 他の二つのコンフィギュレーション（`runtime:full-classpath` と `compile:full-classpath`）。これらのスコープ付きキーは、プロジェクトは特定されていないため「現プロジェクト」で、タスクも特定されていない `Global` だ。
- `Global` に設定されたコンフィギュレーション（`*:full-classpath`）。プロジェクトはまだ特定されていないため「現プロジェクト」で、タスクもまだ特定されていないため `Global` だ。
- `{.}` 別名 `ThisBuild` に設定されたプロジェクト（つまり、特定のプロジェクトではなく、ビルド全体）。
- `Global` に設定されたプロジェクト軸（`*/test:full-classpath`）（プロジェクトが特定されていない場合は、現プロジェクトを意味するため、`Global` を検索することは新しく、`*` と「プロジェクトが未表示」はプロジェクト軸に対して異なる値を持ち、`*/test:full-classpath` と `test:full-classpath` は等価ではない。）

- プロジェクトとコンフィギュレーションの両方とも Global を設定する（`*/*:full-classpath`）（特定されていないタスクは Global であるため、`*/*:full-classpath` は三つの軸全てが Global を取る。）

今度は、（`inspect test:full-class` のかわりに）`inspect full-classpath` を試してみて、違いをみてみよう。コンフィギュレーションが省略されたため、`compile` だと自動検知される。そのため、`inspect compile:full-classpath` は `inspect full-classpath` と同じになるはずだ。

次に、`inspect *:full-classpath` も実行して違いを比べてみよう。`full-classpath` はデフォルトでは、Global コンフィギュレーションには定義されていない。

より詳しくは、[Interacting with the Configuration System](#) 参照。

ビルド定義からスコープを参照する

`build.sbt` で裸のキーを使ってセッティングを作った場合は、現プロジェクト、Global コンフィギュレーション、Global タスクにスコープ付けされる：

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

`sbt` を実行して、`inspect name` と入力して、キーが `{file:/home/hp/checkout/hello/}default-aea33a` により提供されていることを確認しよう。つまり、プロジェクトは、`{file:/home/hp/checkout/hello/}default-aea33a` で、コンフィギュレーションは `*` で、タスクは表示されていない（グローバルを指す）ということだ。

キーにはオーバーロードされた `in` メソッドがあり、それによりスコープを設定できる。`in` への引数として、どのスコープ軸のインスタンスでも渡すことができる。これをやる意味は全くないけど、例として `Compile` コンフィギュレーションでスコープ付けされた `name` の設定を以下に示す：

```
name in Compile := "hello"
```

また、`packageBin` タスクでスコープ付けされた `name` の設定（これも意味なし！ただの例だよ）：

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

もしくは、例えば `Compile` コンフィギュレーションの `packageBin` の `name` など、複数のスコープ軸でスコープ付けする：

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

もしくは、全ての軸に対して Global を使う:

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

(`name in Global` は、スコープ軸である `Global` を全ての軸を `Global` に設定したスコープに暗黙の変換が行われる。タスクとコンフィギュレーションは既にデフォルトで `Global` であるため、事実上行なっているのはプロジェクトを `Global` に指定することだ。つまり、`{file:/home/hp/checkout/hello/}default-aea33a/*:name` ではなく、`*/*:name` が定義される。)

Scala に慣れていない場合に注意して欲しいのは、`in` や `:=` はただのメソッドであって、魔法ではないということだ。Scala ではキレイに書くことができるけど、Java 風に以下のようにも書き下すこともできる:

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

こんな醜い構文で書く必要は一切無いけど、これらが実際にメソッドであることを示している。

いつスコープを指定するべきか

あるキーが、通常スコープ付けされている場合は、スコープを指定してそのキーを使う必要がある。例えば、`compile` タスクは、デフォルトで `Compile` と `Test` コンフィギュレーションにスコープ付けされているけど、これらのスコープ外には存在しない。

そのため、`compile` キーに関連付けられた値を変更するには、`compile in Compile` か `compile in Test` のどちらかを書く必要がある。素の `compile` を使うと、コンフィギュレーションにスコープ付けされた標準のコンパイルタスクをオーバーライドするかわりに、カレントプロジェクトにスコープ付けされた新しいコンパイルタスクを定義してしまう。

“*Reference to undefined setting*” のようなエラーに遭遇した場合は、スコープを指定していないか、間違ったスコープを指定したことによることが多い。君が使っているキーは何か別のスコープの中で定義されている可能性がある。エラーメッセージの一部として `sbt` は、君が意味したであろうものを推測してくれるから、“*Did you mean compile:compile?*” を探そう。

キーの名前はキーの__一部__であるとも考えることもできる。実際の所は、全てのキーは名前と(三つの軸を持つ)スコープによって構成される。つまり、`packageOptions in (Compile, packageBin)` という式全体でキー名だということだ。単に `packageOptions` と言っただけでもキー名だけど、それ

は別のキーだ（`in` 無しのキーのスコープは暗黙で決定され、現プロジェクト、Global コンフィグレーション、Global タスクとなる）。

他の種類のセッティング

このページでは、基本的な `:=` メソッドを超えた、より高度な `Settings` の作り方を説明する。君が `sbt` ビルド定義とスコープを読んだことを前提とする。

復習: セッティング

ビルド定義は `Setting` のリストを作り、それが `sbt` の（キーと値のペアの `Map` で表現される）ビルドの記述を変換するのに使われるということは覚えていると思う。セッティング (`Setting`) は古い `Map` を入力としてとり、新しい `Map` を出力する変換である。そして、新しい `Map` が `sbt` の新しい内部状態となる。

セッティングは種類により異なる方法で `Map` を変換する。これまでは、`:=` メソッドをみてきた。

`:=` が作る `Setting` は、不変の固定値を新たに変換された `Map` に代入する。例えば `Map` を `name := "hello"` というセッティングで変換すると、新しい `Map` は `name` キーの中に `"hello"` を格納する。

既存の値に追加する: `+=` と `++=`

`:=` による置換が最も単純な変換だが、キーには他のメソッドもある。`SettingKey[T]` の `T` が列の場合、つまりキーの値の型が列の場合は、置換のかわりに列に追加することができる。

- `+=` は、列に単一要素を追加する。
- `++=` は、別の列を連結する。

例えば、`sourceDirectories in Compile` というキーの値の型は `Seq[File]` だ。デフォルトで、このキーの値は `src/main/scala` を含む。（どうしても標準的なやり方では気が済まない君が）`source` という名前のディレクトリに入ったソースもコンパイルしたい場合、以下のようにして設定できる:

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

もしくは、`sbt` パッケージに入っている `file()` 関数を使って:

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

(file()) は、単に新しい File 作る)

++= を使って複数のディレクトリを一度に加える事もできる:

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

ここでの Seq(a, b, c, ...) は、列を構築する標準的な Scala の構文だ。

デフォルトのソースディレクトリを完全に置き換えてしまいたい場合は、当然 := を使えばいい:

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

他のキーの値を基に値を計算

タスクキーやセッティングキーの値を使って他のタスクキー、セッティングキーの値を設定してみる。これらの値を返すメソッドは特別なもので、単に := や += や ++= の引数の中で呼び出してやればよい。

まず一つ目の例として、プロジェクトの名前と同じ organization (訳注: Ivy のもの、Maven でいう groupId) を定義してみよう。

```
// プロジェクトの後に組織名を付ける (どちらも型は SettingKey[String])
```

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

次にこれはディレクトリ名を用いてプロジェクトの名前をつける例。

```
// name は Key[String]、 baseDirectory は Key[File]
```

```
// ディレクトリ名を取ってからプロジェクトの名前を付ける
```

```
name := baseDirectory.value.getName
```

この例では標準的な java.io.File オブジェクトの getName メソッドを使って baseDirectory の値を変換している。

複数の入力値を用いる場合も同様である。例えばこのようになる:

```
name := "project " + name.value + " from " + organization.value + " version " + version.value
```

既に宣言されている name の値だけでなく organization や version といったセッティングの値を使って、新たに name というセッティングが設定されている。

依存性を持ったセッティング

name <=> baseDirectory(_.getName) というセッティングにおいて、name は、baseDirectory への_依存性_ (dependency) を持つ。上記の内容を

build.sbt に記述して sbt のインタラクティブモードを立ち上げ、inspect name と入力すると、以下のように表示されるだろう（一部抜粋）：

```
[info] Dependencies:  
[info] *:baseDirectory
```

これは sbt が、どのセッティングが他のセッティングに依存しているかをどう把握しているかを示している。タスクを記述するセッティングを思い出してほしい。そう、タスク間に依存関係を持たせることも可能であるということだ。

例えば inspect compile すれば compile は別のキー compileInputs に依存するということが分かり、compileInputs を inspect すると、それがまた別のキーに依存していることが分かる。依存性の連鎖をたどっていくと、魔法に出会う。例えば compile と打ち込むと、sbt は自動的に update を実行する。これが「とにかくちゃんと動く」理由は、compile の計算に入力として必要な値が sbt に update の計算を先に行うことを強制しているからだ。

このようにして、sbt の全てのビルドの依存性は、明示的には宣言されず、__自動化__されている。あるキーの値を別の計算で使うと、その計算はキーに依存することになる。とにかくちゃんと動いてくれるというわけだ！

セッティングが未定義の場合

セッティングが := や += や ++= を使って自分自身や他のキーへの依存が生まれるとき、その依存されるキーの値が存在しなくてはならない。もしそれが存在しなければ sbt に怒られることになるだろう。例えば、“*Reference to undefined setting*” のようなエラーだ。これが起こった場合は、キーが定義されている正しいスコープで使っているか確認しよう。

これはエラーになるが、循環した依存性を作ってしまうことも起こりうる。sbt が君がそうしてしまったことを教えてくれるだろう。

他のキーの値を基にしたタスク

あるタスクの値を定義するために他のタスクの値を計算する必要があるかもしれない。そのような場合には、:= や += や ++= の引数に Def.task と taskValue を使えばよい。

例として、sourceGenerators にプロジェクトのベースディレクトリやコンパイル時のクラスパスを加える設定をみてみよう。

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

依存性を持ったタスク

.sbt ビルド定義でみたように、タスクキーは:=などでセッティングを作ると `Setting[T]` ではなく、`Setting[Task[T]]` を作る。タスク定義の入力にセッティングの値を用いることができるが、セッティング定義の入力にタスクをとることはできない。

(Keys より) 以下の二つのキーを例に説明する:

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

(`scalacOptions` と `checksums` はお互い何の関係もない、ただ同じ値の型を持つ二つのキーで片方がタスクというだけだ)

`build.sbt` の中で `scalacOptions` を `checksums` のエイリアスにすることはできるが、その逆はできない。例えば、以下の例はコンパイルが通る:

```
// scalacOptions タスクは checksums セッティングの値を用いて定義される
scalacOptions := checksums.value
```

逆方向への依存、つまりタスクの値に依存したセッティングキーの値を定義することはどうしてもできない。なぜなら、セッティングキーの値はプロジェクトのロード時に一度だけしか計算されず、毎回再実行されるべきタスクが毎回実行されなくなってしまうからだ。

```
// checksums セッティングは scalacOptions タスクに関連付けても、値が定まらないかもしれない
checksums := scalacOptions.value
```

依存性を用いた追加: += と ++=

他のキーを使って既存のセッティングキーやタスクキーへ値を追加するには += を使えばよい。

例えば、プロジェクト名を使って名付けたカバレッジレポートがあって、それを `clean` が削除するファイルリストに追加するなら、このようになる:

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

ライブラリ依存性

このページは、このガイドのこれまでのページ、特に .sbt ビルド定義、スコープ、と他の種類のセッティングを読んでいることを前提とする。

ライブラリ依存性は二つの方法で加えることができる:

- lib ディレクトリに jar ファイルを入れることのできる__アンマネージ依存性__ (unmanaged dependencies)
- ビルド定義に設定され、リポジトリから自動でダウンロードされる__マネージ依存性__ (managed dependencies)

アンマネージ依存性 (Unmanaged Dependencies)

ほとんどの人はアンマネージ依存性ではなくマネージ依存性を使う。しかし、アンマネージの方が最初に始めるにあたってはより簡単かもしれない。

アンマネージ依存性はこんな感じのものだ: jar ファイルを lib 配下に置いておけばプロジェクトのクラスパスに追加される、以上！

ScalaCheck、Specs2、ScalaTest のようなテスト用の jar ファイルも lib に配置できる。

lib 配下の依存ライブラリは (compile、test、run、そして console の) 全てのクラスパスに追加される。もし、どれか一つのクラスパスを変えたい場合は、例えば `dependencyClasspath in Compile` や `dependencyClasspath in Runtime` などを適宜調整する必要がある。

アンマネージ依存性を利用するのに、`build.sbt` には何も書く必要はないが、デフォルトの lib 以外のディレクトリを使いたい場合は `unmanagedBase` キーで変更することができる。

lib のかわりに、`custom_lib` を使うならこのようになる:

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

`baseDirectory` はプロジェクトのベースディレクトリで、他の種類のセッティングで説明したとおり、ここでは `unmanagedBase` を `value` を使って取り出した `baseDirectory` の値を用いて変更している。

他には、`unmanagedJars` という `unmanagedBase` ディレクトリに入っている jar ファイルのリストを返すタスクがある。複数のディレクトリを使うとか、何か別の複雑なことを行う場合は、この `unmanagedJar` タスクを何か別のものに変える必要があるかもしれない。例えば `Compile` コンフィギュレーション時に lib ディレクトリのファイルを無視したい、など。

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

マネージ依存性 (Managed Dependencies)

sbt は Apache Ivy を使ってマネージ依存性を実装しているので、既に Maven か Ivy に慣れているなら、違和感無く入り込めるだろう。

libraryDependencies キー

大体的場合、依存性を `libraryDependencies` セットに列挙するだけでうまくいくだろう。Maven POM ファイルや、Ivy コンフィギュレーションファイルを書くなどして、依存性を外部で設定してしまって、sbt にその外部コンフィギュレーションファイルを使わせるということも可能だ。これに関しては、[Library Management] を参照。

依存性の宣言は、以下ようになる。ここで、`groupId`、`artifactId`、と `revision` は文字列だ:

```
libraryDependencies += groupId % artifactID % revision
```

もしくは、以下ようになる。このときの `configuration` は文字列もしくは `Configuration` の値だ。

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

`libraryDependencies` は `Keys` で以下のように定義されている:

```
val libraryDependencies = SettingKey[Seq[ModuleID]]("library-dependencies", "Declares ma
```

% メソッドは、文字列から `ModuleID` オブジェクトを作るので、君はその `ModuleID` を `libraryDependencies` に追加するだけでいい。

当然ながら、sbt は (Ivy を通じて) モジュールをどこからダウンロードしてくるかを覚えていなければならない。もしそのモジュールが sbt に初めから入っているデフォルトのリポジトリの一つに存在していれば、何もしなくてもそのまま動作する。例えば、Apache Derby は Maven2 の標準リポジトリ (訳注: sbt にあらかじめ入っているデフォルトリポジトリの一つ) に存在している:

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

これを `build.sbt` に記述して `update` を実行すると、sbt は Derby を `~/.ivy2/cache/org.apache.derby/` にダウンロードするはずだ。(ちなみに、`update` は `compile` の依存性であるため、ほとんどの場合、手動で `update` と入力する必要はないだろう)

もちろん `++=` を使って依存ライブラリのリストを一度に追加することもできる:

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

`libraryDependencies` に対して `:=` を使う機会があるかもしれないが、おそらくそれは稀だろう。

%% を使って正しい Scala バージョンを入手する

groupID % artifactID % revision のかわりに、groupID %% artifactID % revision を使うと（違いは groupID の後ろの二つ連なった %%）、sbt はプロジェクトの Scala のバイナリバージョンをアーティファクト名に追加する。これはただの略記法なので %% 無しで書くこともできる:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11.1" % "0.3"
```

君のビルドの Scala バージョンが 2.11.1 だとすると、以下の設定は上記と等価だ（“org.scala-tools” の後ろの二つ連なった %% に注意）:

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

多くの依存ライブラリは複数の Scala バイナリバージョンに対してコンパイルされており、ライブラリの利用者はバイナリ互換性のあるものを選択したいと思うはずである。

実際のところの複雑な問題として、依存ライブラリはしばしば少しくらい違った Scala バージョンでも動作するのだが、%% はこれについてそこまで賢くはない。もしある依存ライブラリが Scala 2.10.1 に対してビルドされているとして、君のプロジェクトが `scalaVersion := "2.10.4"` と指定している場合、その 2.10.1 に依存するライブラリがおそらく動作するにも関わらず %% を使うことはできない。もし %% が動かなくなったら、依存ライブラリが使っている実際のバージョンを確認して、動くだろうバージョン（それがあればの話だけど）に決め打ちすればいい。

詳しくは、[Cross Build] を参照。

Ivy revision

groupID % artifactID % revision の revision は、単一の固定されたバージョン番号でなくてもよい。Ivy は指定されたバージョン指定の制限の中でモジュールの最新の revision を選ぶことができる。“1.6.1” のような固定 revision ではなく、“latest.integration”、“2.9.+”、や “[1.0,)” など指定できる。詳しくは、Ivy revisions を参照。

Resolvers

全てのパッケージが一つのサーバに置いてあるとは限らない。sbt は、デフォルトで Maven の標準リポジトリ（訳注：Maven Central Repository）を使う。もし依存ライブラリがデフォルトのリポジトリに存在しないなら、Ivy がそれを見つけられるよう *resolver* を追加する必要がある。

リポジトリを追加するには、以下のように:

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

二つの文字列の間の特別な `at` を使う。

例えばこのようになる:

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

`Keys` で定義されている `resolvers` キーは以下のようにになっている:

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

`at` メソッドは、二つの文字列から `Resolver` オブジェクトを作る。

`sbt` は、リポジトリとして追加すれば、ローカル Maven リポジトリも検索することができる:

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

こんな便利な指定方法もある:

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

他の種類のリポジトリの定義の詳細に関しては、`Resolvers` 参照。

デフォルトの `resolver` のオーバーライド

`resolvers` は、デフォルトの `resolver` を含まず、ビルド定義によって加えられる追加のものだけを含む。

`sbt` は、`resolvers` をデフォルトのリポジトリと組み合わせて `external-resolvers` を形成する。

そのため、デフォルトの `resolver` を変更したり、削除したい場合は、`resolvers` ではなく、`external-resolvers` をオーバーライドする必要がある。

コンフィギュレーションごとの依存性

依存ライブラリをテストコード (`Test` コンフィギュレーションでコンパイルされる `src/test/scala` 内のコード) から使いたいが、メインのコードでは使わないということがよくある。

ある依存ライブラリが `Test` コンフィギュレーションのクラスパスには出てきてほしいが、`Compile` コンフィギュレーションでは要らないという場合は、以下のように `% "test"` と追加する:

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

`Test` コンフィギュレーションの型安全なバージョンを使ってもよい:

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


この状態で sbt のインタラクティブモードで `show compile:dependency-classpath` と入力しても Derby は出てこないはずだ。だが、`show test:dependency-classpath` と入力すると、Derby の jar がリストに含まれていることを確認できるだろう。

普通は、ScalaCheck、Specs2、ScalaTest などのテスト関連の依存ライブラリは `% "test"` と共に定義される。

ライブラリの依存性に関しては、もうこの入門用のページで見つからない情報があれば、このページにもう少し詳細やコツが書いてある。

マルチプロジェクト・ビルド

このページでは、一つのプロジェクトで複数のプロジェクトを管理する方法を紹介する。このガイドのこれまでのページを読んでおいてほしい。特に `build.sbt` を理解していることが必要になる。

複数のプロジェクト

一つのビルドに複数の関連するプロジェクトを入れておくと、プロジェクト間に依存性がある場合や同時に変更されることが多い場合に便利だ。

ビルド内の個々のサブプロジェクトは、それぞれ独自のソースディレクトリを持ち、`package` を実行すると独自の jar ファイルを生成するなど、概ね通常のプロジェクトと同様に動作する。

個々のプロジェクトは `lazy val` を用いて `Project` 型の値を宣言することで定義される。例として、以下のようなものがプロジェクトだ:

```
lazy val util = project
```

```
lazy val core = project
```

`val` で定義された名前はプロジェクトの ID 及びベースディレクトリの名前になる。ID はコマンドラインからプロジェクトを指定する時に用いられる。ベースディレクトリは `in` メソッドを使ってこのデフォルトから変更することができる。上記の例と同じ結果になる記述を明示的に書くと、以下のようになる。

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

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

共通のセッティング

複数プロジェクトに共通なセッティングをくくり出す場合、`commonSettings` という名前のセッティングの `Seq` を作って、それを引数として各プロジェクトの `settings` メソッドを呼び出せばよい。（可変長引数を受け取るメソッドに `Seq` を渡すには `_*` が必要なので注意）

```
lazy val commonSettings = Seq(
  organization := "com.example",
  version := "0.1.0",
  scalaVersion := "2.11.8"
)

lazy val core = (project in file("core")).
  settings(commonSettings: _*).
  settings(
    // other settings
  )

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

これで `version` を一箇所で変更すれば、再読み込み後に全サブプロジェクトに反映されるようになる。

依存関係

一つのビルドの中の個々のプロジェクトはお互いに完全に独立した状態であってもよいが、普通、何らかの形で依存関係を持っているだろう。ここでは集約（`aggregate`）とクラスパス（`classpath`）という二種類の依存関係がある。

集約

集約とは、集約する側のプロジェクトであるタスクを実行するとき、集約される側の複数のプロジェクトでも同じタスクを実行するという関係を意味する。例えば、

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

```
lazy val util = project
```

```
lazy val core = project
```

上の例では、root プロジェクトが util と core を集約している。この状態で sbt を起動してコンパイルしてみよう。3 つのプロジェクトが全てコンパイルされることが分かると思う。

集約プロジェクト内で（この場合は root プロジェクトで）、タスクごとに集約をコントロールすることができる。例えば、update タスクの集約を以下のようにして回避できる:

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

[...]

aggregate in update は、update タスクにスコープ付けされた aggregate キーだ（スコープ参照）。

注意: 集約は、集約されるタスクを順不同に並列実行する。

クラスパス依存性

あるプロジェクトが、他のプロジェクトにあるコードに依存させたい場合、dependsOn メソッドを呼び出して実現すればよい。

例えば、core に util のクラスパスが必要な場合は core の定義を次のように書く:

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

これで core 内のコードから util の class を利用することができるようになった。

また、これにより core がコンパイルされる前に util の update と compile が実行されている必要があるのでプロジェクト間でコンパイル実行が順序付けられることになる。

複数のプロジェクトに依存するには、dependsOn(bar, baz) というふうに、dependsOn に複数の引数を渡せばよい。

コンフィギュレーションごとのクラスパス依存性

`foo dependsOn(bar)` は、`foo` の `Compile` コンフィギュレーションが `bar` の `Compile` コンフィギュレーションに依存することを意味する。これを明示的に書くと、`dependsOn(bar % "compile->compile")` となる。

この `"compile->compile"` 内の `->` は、「依存する」という意味で、`"test->compile"` は、`foo` の `Test` コンフィギュレーションが `bar` の `Compile` コンフィギュレーションに依存することを意味する。

`->config` の部分を省くと、`->compile` だと解釈されるため、`dependsOn(bar % "test")` は、`foo` の `Test` コンフィギュレーションが `bar` の `Compile` コンフィギュレーションに依存することを意味する。

特に、`Test` が `Test` に依存することを意味する `"test->test"` は役に立つ宣言だ。これにより、例えば、`bar/src/test/scala` にテストのためのユーティリティコードを置いておき、それを `foo/src/test/scala` 内のコードから利用することができる。

複数のコンフィギュレーション依存性を宣言する場合は、セミコロンで区切る。例えば、`dependsOn(bar % "test->test;compile->compile")` と書ける。

デフォルトルートプロジェクト

もしプロジェクトがルートディレクトリに定義されてなかったら、`sbt` はビルド時に他のプロジェクトを集約するデフォルトプロジェクトを勝手に生成する。

プロジェクト `hello-foo` は、`base = file("foo")` と共に定義されているため、サブディレクトリ `foo` に置かれる。そのソースは、`foo/Foo.scala` のように `foo` の直下に置かれるか、`foo/src/main/scala` 内に置かれる。ビルド定義ファイルを除いては、通常の `sbt` ディレクトリ構造が `foo` 以下に適用される。

`foo` 内の全ての `.sbt` ファイル、例えば `foo/build.sbt` は、`hello-foo` プロジェクトにスコープ付けされた上で、ビルド全体のビルド定義に取り込まれる。

ルートプロジェクトが `hello` にあるとき、`hello/build.sbt`、`hello/foo/build.sbt`、`hello/bar/build.sbt` においてそれぞれ別々のバージョンを定義してみよう（例: `version := "0.6"`）。次に、インタラクティブプロンプトで `show version` と打ち込んでみる。以下のように表示されるはずだ（定義したバージョンによるが）：

```
> 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` は、`hello/foo/build.sbt` 内で定義され、`hello-bar/*:version` は、`hello/bar/build.sbt` 内で定義され、`hello/*:version` は、`hello/build.sbt` 内で定義される。スコープ付けされたキーの構文を復習しておこう。それぞれの `version` キーは、`build.sbt` の場所により、特定のプロジェクトにスコープ付けされている。だが、三つの `build.sbt` と同じビルド定義の一部だ。

`.scala` ファイルは、上に示したように、単にプロジェクトとそのベースディレクトリを列挙するだけの簡単なものにして、それぞれのプロジェクトのセッティングは、そのプロジェクトのベースディレクトリ直下の `.sbt` ファイル内で宣言することができる。全てのセッティングを `.scala` ファイル内で宣言することは義務付けられているわけではない。

ビルド定義の全てを単一の `project` ディレクトリ内の場所にまとめるために、`.scala` ファイル内にセッティングも含めてしまうほうが洗練されていると思うかもしれない。ただし、これは好みの問題だから、好きにやればいい。

サブプロジェクトは、`project` サブディレクトリや、`project/*:scala` ファイルを持つことができない。`foo/project/Build.scala` は無視される。

プロジェクトの切り替え

`sbt` インタラクティブプロンプトから、`projects` と入力することでプロジェクトの全リストが表示され、`project <プロジェクト名>` で、カレントプロジェクトを選択できる。`compile` のようなタスクを実行すると、それはカレントプロジェクトに対して実行される。これにより、ルートプロジェクトをコンパイルせずに、サブプロジェクトのみをコンパイルすることができる。

また `subProjectID/compile` のように、プロジェクト ID を明示的に指定することで、そのプロジェクトのタスクを実行することもできる。

共通のコード

.sbt ファイルで定義された値は、他の .sbt ファイルからは見えない。
.sbt ファイル間でコードを共有するためには、ベースディレクトリにある project/ 配下に Scala ファイルを用意すればよい。
詳細はビルドの整理を参照。

プラグインの使用

このガイドのこれまでのページを読んでおいてほしい。特に build.sbt とライブラリ依存性を理解していることが必要になる。

プラグインとは何か

sbt のプラグインは、最も一般的には新しいセッティングを追加することでビルド定義を拡張するものである。その新しいセッティングは新しいタスクでもよい。例えば、テストカバレッジレポートを生成する `codeCoverage` というタスクを追加するプラグインなどが考えられる。

プラグインの宣言

プロジェクトが `hello` ディレクトリにあり、ビルド定義に `sbt-site` プラグインを追加する場合、`hello/project/site.sbt` を新しく作成し、Ivy のモジュール ID を `addSbtPlugin` メソッドに渡してプラグイン依存性を定義する:

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

`sbt-assembly` プラグインを追加するなら、以下のような内容で `hello/project/assembly.sbt` をつくる:

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

全てのプラグインがデフォルトのリポジトリに存在するわけではないので、プラグインのドキュメントでそのプラグインが見つかるリポジトリを `resolvers` に追加するよう指示されていることもあるだろう。

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

プラグインは普通、プロジェクトでそのプラグインの機能を有効にするためのセッティング群を提供している。これは次のセクションで説明する。

auto plugin の有効化と無効化

プラグインは、自身が持つセッティング群がビルド定義に自動的に追加されるよう宣言することができ、その場合、プラグインの利用者は何もしなくてもいい。

sbt 0.13.5 から、プラグインを自動的に追加して、そのセッティング群と依存関係がプロジェクトに設定されていることを安全に保証する auto plugin という機能が追加された。

auto plugin の多くはデフォルトのセッティング群を自動的に追加するが、中には明示的な有効化を必要とするものもある。

明示的な有効化が必要な auto plugin を使っている場合は、以下を build.sbt に追加する必要がある:

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

enablePlugins メソッドを使えば、そのプロジェクトで使いたい auto plugin を明示的に定義できる。逆に disablePlugins メソッドを使ってプラグインを除外することもできる。例えば、util から IvyPlugin のセッティングを除外したいとすると、build.sbt を以下のように変更する:

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

明示的な有効化が必要か否かは、それぞれの auto plugin がドキュメントで明記しておくべきだ。あるプロジェクトでどんな auto plugin が有効化されているか気になったら、sbt コンソールから plugins コマンドを実行してみよう。

例えば、このようになる。

```
> 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
```

ここでは、`plugins` の表示によって sbt のデフォルトのプラグインが全て有効化されていることが分かる。sbt のデフォルトセッティングは 3 つのプラグインによって提供される:

1. `CorePlugin`: タスクの並列実行などのコア機能。
2. `IvyPlugin`: モジュールの公開や依存性の解決機能。
3. `JvmPlugin`: Java/Scala プロジェクトのコンパイル/テスト/実行/パッケージ化。

さらに `JUnitXmlReportPlugin` は実験的に `junit-xml` の生成機能を提供する。

古くからある auto plugin ではないプラグインは、マルチプロジェクトビルド内に異なるタイプのプロジェクトを持つことができるように、セッティング群を明示的に追加することを必要とする。

各プラグインのドキュメントに設定方法が明記されているかと思うが、一般的にはベースとなるセッティング群を追加して、必要に応じてカスタマイズするというパターンが多い。

例えば `sbt-site` プラグインの例で説明すると `site.sbt` というファイルを新しく作って

```
site.settings
```

を `site.sbt` に記述することで有効化できる。

ビルド定義がマルチプロジェクトの場合は、プロジェクトに直接追加する:

```
// don't use the site plugin for the `util` project
lazy val util = (project in file("util"))

// enable the site plugin for the `core` project
lazy val core = (project in file("core")).
  settings(site.settings : _*)
```

グローバル・プラグイン

プラグインを `~/.sbt/0.13/plugins/` 以下で宣言することで全てのプロジェクトに対して一括してプラグインをインストールすることができる。`~/.sbt/0.13/plugins/` はそのクラスパスをすべての sbt ビルド定義に対して `export` する sbt プロジェクトだ。大雑把に言えば、`~/.sbt/0.13/plugins/` 内の `.sbt` ファイルや `.scala` ファイルは、それが

全てのプロジェクトの `project/` ディレクトリに入っているかのようにふるまう。

`~/ .sbt/0.13/plugins/build.sbt` を作って、そこに `addSbtPlugin()` 式を書くことで全プロジェクトにプラグインを追加することができる。しかし、これを多用するとマシン環境への依存性を増やしてしまうことになるので、この機能は注意してほどほどに使うべきだ。ベスト・プラクティスも参照してほしい。

利用可能なプラグイン

プラグインのリストがある。

特に人気のプラグインは:

- IDE のためのプラグイン (sbt プロジェクトを好みの IDE にインポートするためのもの)
- `xsbt-web-plugin` のような Web フレームワークをサポートするプラグイン

プラグイン開発の方法など、プラグインに関する詳細は `Plugins` を参照。ベストプラクティスを知りたいなら、ベスト・プラクティスを見てほしい。

カスタムセッティングとタスク

このページでは、独自のセッティングやタスクの作成を紹介する。

このページを理解するには、このガイドの前のページ、特に `build.sbt` と他の種類のセッティングを読んである必要がある。

キーを定義する

`Keys` は、キーをどのように定義するかを示すサンプル例が満載だ。多くのキーは、`Defaults` で実装されている。

キーには 3 つの型がある。 `SettingKey` と `TaskKey` は `sbt` ビルド定義で説明した。 `InputKey` に関しては `Input Tasks` を見てほしい。

以下に `Keys` からの具体例を示す:

```
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")
```

キーのコンストラクタは、二つの文字列のパラメータを取る。キー名 ("scala-version") と 解 説 文 ("The version of scala used for building.") だ。

.sbt ビルド定義でみた通り、`SettingKey[T]` 内の型パラメータ `T` は、セッティングの値の型を表す。`TaskKey[T]` 内の `T` は、タスクの結果の型を表す。

また、.sbt ビルド定義でみた通り、セッティングはプロジェクトが再読み込みされるまでは固定値を持ち、タスクは「タスク実行」の度 (sbt のインタラクティブモードかバッチモードでコマンドが入力される度) に再計算される。

キーは .sbt ファイル、.scala ファイル、または auto plugin 内で定義する事が出来る。有効化された auto plugin の `autoImport` オブジェクト内で定義された `val` は全て .sbt ファイルに自動的にインポートされる。

タスクを実装する

タスクで使えるキーを定義したら、次はそのキーをタスク定義の中で使ってみよう。自前のタスクを定義しようとしているかもしれないし、既存のタスクを再定義しようとしているかもしれないが、いずれにせよ、やることは同じだ。:= を使ってタスクのキーになんらかのコードを関連付けよう:

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

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

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

もしタスクに依存してるものがあれば、他の種類のセッティングで説明したとおり `value` を使ってその値を参照すればよい。

タスクを実装する上で一番難しい点は、多くの場合 sbt 固有の問題ではない。なぜならタスクはただの Scala コードだからだ。難しいのはそのタスクが実行したいことの「本体」部分を書くことだ。

例えば HTML を整形したいとすると、今度は HTML のライブラリを利用したくなるかもしれない（おそらくビルド定義にライブラリ依存性を追加して、その HTML ライブラリに基づいたコードを書けばよいだろう）。

sbt には、いくつかのユーティリティライブラリや便利な関数があり、特にファイルやディレクトリの取り扱いには Scaladocs-IO にある API がしばしば重宝するだろう。

タスクの実行意味論

カスタムタスクから `value` を使って他のタスクに依存するとき、タスクの実行意味論 (execution semantics) に注意する必要がある。ここでいう実行意味論とは、実際どの時点で__タスクが評価されるかを決定するものとする。

`sampleIntTask` を例にとると、タスク本文の各行は一行ずつ正格評価 (strict evaluation) されているはずだ。これは逐次実行の意味論だ:

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

実際には JVM は `sum` を 3 とインライン化したりするかもしれないが、観測可能なタスクの作用は、各行ずつ逐次実行したものと同一のものとなる。

次に、`startServer` と `stopServer` という 2 つのカスタムタスクを定義して、`sampleIntTask` を以下のように書き換えたとする:

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

```

)

lazy val library = (project in file("library")).
  settings(commonSettings: _*).
  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
    }
  )
)

```

sampleIntTask を sbt のインタラクティブ・プロンプトから実行すると以下の結果となる:

```

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

```

何が起こったのかを考察するために、sampleIntTask を視覚化してみよう:

素の Scala のメソッド呼び出しと違って、タスクの value メソッドの呼び出しは正格評価されない。代わりに、sampleIntTask が startServer タスクと stopServer タスクに依存するということを表すマークとして機能する。



図 2: task-dependency

`sampleIntTask` がユーザによって呼び出されると、sbt のタスクエンジンは以下を行う:

- `sampleIntTask` を評価する前にタスク依存性を評価する。(半順序)
- タスク依存性が独立ならば、並列に評価しようとする(並列性)
- 各タスクは一度のコマンド実行に対して 1 回のみ評価される(非重複)

タスク依存性の非重複化

非重複化を説明するために、sbt インタラクティブ・プロンプトから `sampleStringTask` を実行する。

```

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

```

`sampleStringTask` は `startServer` と `sampleIntTask` の両方に依存し、`sampleIntTask` もまた `startServer` タスクに依存するため、`startServer` はタスク依存性として 2 度現れる。しかし、`value` はタスク依存性を表記するだけなので、評価は 1 回だけ行われる。以下は `sampleStringTask` の評価を視覚化したものだ:

もしタスク依存性を非重複化しなければ、`test in Test` のタスク依存性として `compile in Test` が何度も現れるため、テストのソースコードを何度もコンパイルすることになる。

終了処理タスク



図 3: task-dependency

stopServer タスクはどう実装するべきだろうか？ タスクは依存性を保持するものなので、終了処理タスクという考えはタスクの実行モデルにそぐわないものだ。最後の処理そのものもタスクになるべきで、そのタスクが他の中間タスクに依存すればいい。例えば、stopServer が sampleStringTask に依存するべきだが、その時点で stopServer は sampleStringTask と呼ばれるべきだろう。

```

lazy val library = (project in file("library")).
  settings(commonSettings: _*).
  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
    }
  )

```

```
    }
  )
}
```

これが動作することを調べるために、インタラクティブ・プロンプトから `sampleStringTask` を実行してみよう:

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

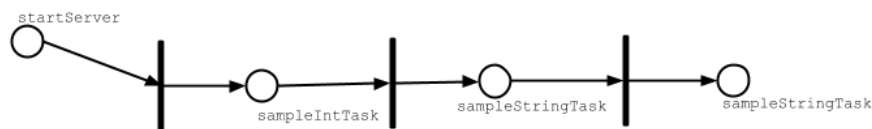


図 4: task-dependency

素の Scala を使おう

何かが起こったその後に別の何かが起こることを保証するもう一つの方法は Scala を使うことだ。例えば `project/ServerUtil.scala` に簡単な関数を書いたとすると、タスクは以下のように書ける:

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

素のメソッド呼び出しは逐次実行の意味論に従うので、全ては順序どおりに実行される。非重複化もされなくなるので、それは気をつける必要がある。

プラグイン化しよう

.scala ファイルに大量のカスタムコードがあることに気づいたら、プラグインを作って複数のプロジェクト間で再利用できないか考えてみよう。

以前にちょっと触れたし、詳しい解説はここにあるが、プラグインを作るのはとても簡単だ。

このページは簡単な味見だけで、カスタムタスクに関しては Tasks ページで詳細に解説されている。

ビルドの整理

このページではビルド構造の整理について説明する。

このガイドの前のページ、特に build.sbt、ライブラリ依存性、そしてマルチプロジェクト・ビルドを理解していることが必要になる。

sbt は再帰的だ

build.sbt は sbt の実際の動作を隠蔽している。sbt のビルドは、Scala コードにより定義されている。そのコード自身もビルドされなければならない。当然これも sbt でビルドされる。sbt でやるより良い方法があるだろうか？

project ディレクトリは、ビルドをビルドする方法を記述したビルドの中のビルドだ。これらのビルドを区別するために、一番上のビルドを**プロパービルド** (proper build)、project 内のビルドを**メタビルド** (meta-build) と呼んだりする。メタビルド内のプロジェクトは、他のプロジェクトができる全てのことをこなすことができる。つまり、**ビルド定義もまた sbt プロジェクトなのだ**。

この入れ子構造は永遠に続く。project/project ディレクトリを作ることによってビルド定義のビルド定義プロジェクトをカスタム化することができる。

以下に具体例で説明する:

```
hello/                                # ビルドのルート・プロジェクトのベースディレクトリ

    Hello.scala                        # ビルドのルート・プロジェクトのソースファイル

                                        # (src/main/scala に入れることもできる)

    build.sbt                          # build.sbt は、project/ 内のメタビルドの
                                        # ルート・プロジェクトのソースの一部となる。
```



```

# つまり、プロパービルドのビルド定義

project/
ディレクトリ      # メタビルドのルート・プロジェクトのベース

Build.scala      # メタビルドのルート・プロジェクトのソース
ファイル、
# つまり、ビルド定義のソースファイル。
# プロパービルドのビルド定義

build.sbt        # これは、project/project 内のメタメタビ
ルドの
# ルート・プロジェクトのソースの一部となり、
# ビルド定義のビルド定義となる

project/        # メタメタビルドのルート・プロジェクトのベー
スディレクトリ

Build.scala # project/project/ 内のメタメタビルドの
# ルート・プロジェクトのソースファイル

```

心配しないでほしい！ 普通はこういうことをする必要は全くない。しかし、原理を理解しておくことはきっと役立つことだろう。

ちなみに、`.scala` や `.sbt` の拡張子で終わっていればどんなファイル名でもよく、`build.sbt` や `Build.scala` と命名するのは慣例にすぎない。これは複数のファイルを使うことができるということも意味する。

ライブラリ依存性を一箇所にまとめる

`project` 内の任意の `.scala` ファイルがビルド定義の一部となることを利用する一つの例として `project/Dependencies.scala` というファイルを作ってライブラリ依存性を一箇所にまとめることができる。

```

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" % "2.4.14"

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

この Dependencies は build.sbt 内で利用可能となる。定義されている val が使いやすいように Dependencies._ を import しておこう。

```
import Dependencies._

lazy val commonSettings = Seq(
  version := "0.1.0",
  scalaVersion := "2.11.8"
)

lazy val backend = (project in file("backend")).
  settings(commonSettings: _*).
  settings(
    libraryDependencies ++= backendDeps
  )
```

マルチプロジェクトでのビルド定義が肥大化して、サブプロジェクト間で同じ依存ライブラリを持っているかを保証したくなったとき、このようなテクニックは有効だ。

いつ .scala ファイルを使うべきか

.scala ファイルでは、トップレベルの class や object 定義を含む Scala コードを自由に記述できる。

推奨される方法はマルチプロジェクトを定義する build.sbt ファイル内にほとんどのセッティングを定義し、project/*.scala ファイルはタスクの実装や、共有したい値やキーを定義するのに使うことだ。また .scala ファイルを使うかどうかの判断には、君や君のチームがどれくらい Scala に慣れているかということも関係するだろう。

auto plugin を定義する

上級ユーザ向けのビルドの整理方法として、`project/*.scala` 内に専用の auto plugin を書くという方法がある。連鎖プラグイン (triggered plugin) を定義することで auto plugin を全サブプロジェクトにカスタムタスクやコマンドを追加する手段として使うことができる。

まとめ

このページではこのガイドを総括する。

sbt を使うのに、理解すべき概念の数はさほど多くない。確かに、これらには多少の学習曲線があるが、sbt にはこれらの概念_以外_のことは特にないとも考えることもできる。sbt は、強力なコア・コンセプトだけを用いて全てを実現している。

この「始める sbt」シリーズをここまで読破したのであれば、知るべきことが何かはもう分かっているはずだ。

sbt: コア・コンセプト

- Scala の基本。Scala の構文に慣れていると役立つのは言うまでもない。Scala の設計者自身による Scala スケーラブルプログラミング (原著) は、素晴らしい入門書だ。
- .sbt ビルド定義
- ビルド定義とは、一つの大きな **Setting** オブジェクトのリストであり、**Setting** とは、sbt がタスクを実行するために使うキーと値のペアを変換するものだ。
- **Setting** を作成するために `:=`、`+=`、`++=` のようなキーに定義されたメソッドを呼び出す。
- mutable な状態は存在せず、ただ変換があるだけだ。例えば **Setting** は sbt のキーと値のペアのコレクションを新しいコレクションに変換する、上書き更新はしない。
- 各セッティングは、キーにより決定された固有の型の値を持つ。
- `_タスク_` は、特殊なセッティングで、タスクを実行するたびに、キーの値を生成する計算が再実行される。非タスクのセッティングは、ビルド定義の読み込み時に一度だけ値が計算される。
- スコープ

- それぞれのキーは、異なるスコープごとにそれぞれ別の値を持つことができる。
- スコープ付けは、コンフィギュレーション、プロジェクト、タスクの三つの軸を用いることができる。
- スコープ付けによって、プロジェクト毎、タスク毎、コンフィギュレーション毎に、異なるふるまいを持たせることができる。
- コンフィギュレーションは、メインのもの（`Compile`）や、テスト用のもの（`Test`）のようなビルドの種類だ。
- プロジェクト軸には（個々のサブプロジェクトだけでなく）「ビルド全体」を指すスコープもある。
- スコープはより一般的なスコープにフォールバックする、これを__委譲__（`delegate`）という。
- `build.sbt` にほとんどの設定を置くが、`class` 定義や大きめのタスク実装などは `.scala` ビルド定義を使う。
- ビルド定義はそれ自体も `project` ディレクトリをルートとする `sbt` プロジェクトである。
- プラグインはビルド定義の拡張だ。
- プラグインは、`addSbtPlugin` メソッドを用いて `project/plugins.sbt` に追加する。（プロジェクトのベースディレクトリにある `build.sbt` ではないことに注意）

上記のうち、一つでも分からないことがあれば、質問してみるか、このガイドをもう一度読み返すか、`sbt` のインタラクティブモードで実験してみよう。

健闘を祈る！

上級者への注意

`sbt` はオープンソースであるため、いつでもソースを見れることも忘れずに！

付録: bare .sbt ビルド定義

このページでは旧式の `.sbt` ビルド定義の説明をする。現在の推奨はマルチプロジェクト `.sbt` ビルド定義だ。

bare .sbt ビルド定義とは何か

明示的に `Project` を定義するマルチプロジェクト `.sbt` ビルド定義や `.scala` ビルド定義と違って `bare` ビルド定義は `.sbt` ファイルの位置から暗黙にプロ

ジェクトが定義される。

`Project` を定義する代わりに、bare `.sbt` ビルド定義は `Setting[_]` 式のリストから構成される。

```
name := "hello"
```

```
version := "1.0"
```

```
scalaVersion := "2.11.8"
```

(0.13.7 以前) 設定は空白行で区切る

注意: 0.13.7 以降は空白行の区切りを必要としない。

こんな風に `build.sbt` を書くことはできない。

// 空白行がない場合はコンパイルしない

```
name := "hello"
```

```
version := "1.0"
```

```
scalaVersion := "2.10.3"
```

sbt はどこまでで式が終わってどこからが次の式なのかを判別するために、何らかの区切りを必要とする。

付録: `.scala` ビルド定義

このページでは、旧式の `.scala` ビルド定義の説明をする。以前のバージョンの sbt で複数のプロジェクトを扱うには `.scala` ビルド定義を使う以外しかなかったが、sbt 0.13 からはマルチプロジェクト `.sbt` ビルド定義が追加され、現在はそのスタイルが推奨されている。

このページは、このガイドのこれまでのページ、特に `.sbt` ビルド定義 と他の種類のセッティングを読んでいることを前提とする。

`build.sbt` と `Build.scala` の関係

ビルド定義の中で、`.sbt` と `.scala` を混ぜて使うには、両者の関係を理解する必要がある。

実際に 2 つのファイルを使って説明しよう。まず、プロジェクトが `hello` というディレクトリにあるなら `hello/project/Build.scala` を以下のように作る:

```

import sbt._
import Keys._

object HelloBuild extends Build {
  val sampleKeyA = settingKey[String]("demo key A")
  val sampleKeyB = settingKey[String]("demo key B")
  val sampleKeyC = settingKey[String]("demo key C")
  val sampleKeyD = settingKey[String]("demo key D")

  override lazy val settings = super.settings ++
    Seq(
      sampleKeyA := "A: in Build.settings in Build.scala",
      resolvers := Seq()
    )

  lazy val root = Project(id = "hello",
    base = file("."),
    settings = Seq(
      sampleKeyB := "B: in the root project settings in Build.scala"
    ))
}

```

次に hello/build.sbt を以下のような内容で作成する:

```
sampleKeyC in ThisBuild := "C: in build.sbt scoped to ThisBuild"
```

```
sampleKeyD := "D: in build.sbt"
```

sbt のインタラクティブプロンプトを起動する。inspect sampleKeyA と入力すると、以下のように表示されるはずだ (一部抜粋) :

```

[info] Setting: java.lang.String = A: in Build.settings in Build.scala
[info] Provided by:
[info] {file:/home/hp/checkout/hello/}/*:sampleKeyA

```

次に inspect sampleKeyC と入力すると、以下のように表示される:

```

[info] Setting: java.lang.String = C: in build.sbt scoped to ThisBuild
[info] Provided by:
[info] {file:/home/hp/checkout/hello/}/*:sampleKeyC

```

二つの値とも “Provided by” は同じスコープを表示していることに注目してほしい。つまり、.sbt ファイルの sampleKeyC in ThisBuild は、.scala ファイルの Build.settings リストにセッティングを追加することと等価と

いうことだ。sbt はビルド全体にスコープ付けされたセッティングを両者から取り込んでビルド定義を作成する。

次は、inspect sampleKeyB:

```
[info] Setting: java.lang.String = B: in the root project settings in Build.scala
[info] Provided by:
[info] {file:/home/hp/checkout/hello/}hello/*:sampleKeyB
```

sampleKeyB は、ビルド全体 ({file:/home/hp/checkout/hello/}) ではなく、特定のプロジェクト ({file:/home/hp/checkout/hello/}hello) にスコープ付けされていることに注意してほしい。

もうお分かりだと思うが、inspect sampleKeyD は sampleKeyB に対応する:

```
[info] Setting: java.lang.String = D: in build.sbt
[info] Provided by:
[info] {file:/home/hp/checkout/hello/}hello/*:sampleKeyD
```

sbt は .sbt ファイルからのセッティングを Build.settings と Project.settings に__追加し__、これは .sbt 内のセッティングの優先順位が高いことを意味する。Build.scala を変更して、build.sbt でも設定されている sampleKeyC か sampleKeyD キーを設定してみよう。build.sbt 内のセッティングが Build.scala 内のそれに「勝って」優先されるはずだ。

もう一つ気づいたかもしれないが、sampleKeyC と sampleKeyD は build.sbt でそのまま使うことができる。これは sbt が Build オブジェクトのコンテンツを自動的に .sbt ファイルにインポートすることにより実現されている。具体的には build.sbt ファイル内で import HelloBuild._ が暗黙に呼ばれている。

まとめると:

- .scala ファイル内で、Build.settings にセッティングを追加すると、自動的にビルド全体にスコープ付けされる。
- .scala ファイル内で、Project.settings にセッティングを追加すると、自動的にプロジェクトにスコープ付けされる。
- .scala ファイルに書いた全ての Build オブジェクトのコンテンツは .sbt ファイルにインポートされる。
- .sbt ファイル内のセッティングは .scala ファイルのセッティングに__追加__される。
- .sbt ファイル内のセッティングは、明示的に指定されない限りプロジェクトにスコープ付けされる。

インタラクティブモードにおけるビルド定義

sbt のインタラクティブプロンプトでカレントプロジェクトを `project/` 内のビルド定義プロジェクトに切り替えることができる。 `reload plugins` と打ち込むことで切り替わる:

```
> reload plugins
[info] Set current project to default-a0e8e4 (in build file:/home/hp/checkout/hello/project)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/project/Build.scala)
> reload return
[info] Loading project definition from /home/hp/checkout/hello/project
[info] Set current project to hello (in build file:/home/hp/checkout/hello/)
> show sources
[info] ArrayBuffer(/home/hp/checkout/hello/hw.scala)
>
```

上記にあるとおり、`reload return` を使ってビルド定義プロジェクトから普通のプロジェクトに戻る。

注意: 全てが immutable だ

`build.sbt` 内のセッティングが、`Build` や `Project` オブジェクトの `settings` フィールドに追加されると考えるのは間違っている。そうではなく、`Build` や `Project` のセッティングリストと `build.sbt` のセッティングが連結されて別の immutable なリストになって、それが sbt に使われるというのが正しい。`Build` と `Project` オブジェクトは、immutable なコンフィギュレーションであり、ビルド定義の全体からすると、たった一部にすぎない。

事実、セッティングには他にも出どころがある。具体的には、以下の順で追加される:

- `.scala` ファイル内の `Build.settings` と `Project.settings`。
- ユーザ定義のグローバルセッティング。例えば、`~/sbt/build.sbt` に 全て のプロジェクトに影響するセッティングを定義できる。
- プラグインによって注入されるセッティング、次のプラグインの使用参照。
- プロジェクトの `.sbt` ファイル内のセッティング。
- (`project` 内のプロジェクトである) ビルド定義プロジェクトの場合は、グローバルプラグイン (`~/sbt/plugins`) が追加される。プラグインの使用で詳細が説明される。

後続のセッティングは古いものをオーバーライドする。このリスト全体でビルド定義が構成される。

各論

プラグインとベストプラクティス

sbt プラグインをテストする

テストの話をしよう。一度プラグインを書いてしまうと、どうしても長期的なものになってしまう。新しい機能を加え続ける（もしくはバグを直し続ける）ためにはテストを書くのが合理的だ。

scripted test framework

sbt は、scripted test framework というものが付いてきて、ビルドの筋書きをスクリプトに書くことができる。これは、もともと変更の自動検知や、部分コンパイルなどの複雑な状況下で sbt 自体をテストするために書かれたものだ:

ここで、仮に B.scala を削除するが、A.scala には変更を加えないものとする。ここで、再コンパイルすると、A から参照される B が存在しないために、エラーが得られるはずだ。[中略 (非常に複雑なことが書いてある)]

scripted test framework は、sbt が以上に書かれたようなケースを的確に処理しているかを確認するために使われている。

このフレームワークは scripted-plugin 経由で利用可能だ。このページはプラグインにどのようにして scripted-plugin を導入するかを解説する。

ステップ 1: snapshot

scripted-plugin はプラグインをローカルに publish するため、まずは version を **-SNAPSHOT** なものに設定しよう。ここで SNAPSHOT を使わないと、あなたと世界のあなた以外の人が別々のアーティファクトを観測するといった酷い不整合な状態に入り込む場合があるからだ。

ステップ 2: scripted-plugin

次に、scripted-plugin をプラグインのビルドに加える。project/scripted.sbt:

```
libraryDependencies += { "org.scala-sbt" % "scripted-plugin" % sbtVersion.value }
```

以下のセッティングを scripted.sbt に加える:

```
ScriptedPlugin.scriptedSettings
scriptedLaunchOpts := { scriptedLaunchOpts.value ++
  Seq("-Xmx1024M", "-XX:MaxPermSize=256M", "-Dplugin.version=" + version.value)
}
scriptedBufferLog := false
```

ステップ 3: src/sbt-test

src/sbt-test/<テストグループ>/<テスト名> というディレクトリ構造を作る。とりあえず、src/sbt-test/<プラグイン名>/simple から始めるとする。

ここがポイントなんだけど、simple 下にビルドを作成する。プラグインを使った普通のビルド。手動でテストするために、いくつか既にあると思うけど。以下に、build.sbt の例を示す:

```
lazy val root = (project in file(".")).
  settings(
    version := "0.1",
    scalaVersion := "2.10.6",
    assemblyJarName in assembly := "foo.jar"
  )
```

これが、project/plugins.sbt:

```
sys.props.get("plugin.version") match {
  case Some(x) => addSbtPlugin("com.eed3si9n" % "sbt-assembly" % x)
  case _ => sys.error("""|The system property 'plugin.version' is not defined.
Specify this property using the scriptedLaunchOpts -D.""|.stripMargin)
}
```

これは JamesEarlDouglas/xsbt-web-plugin@feabb2 から拝借してきた技で、これで scripted テストに version を渡すことができる。

他に、src/main/scala/hello.scala も用意した:

```
object Main extends App {  
  println("hello")  
}
```

ステップ 4: スクリプトを書く

次に、好きな筋書きを記述したスクリプトを、テストビルドのルート下に置いた `test` というファイルに書く。

```
# ファイルが作成されたかを確認  
> assembly  
$ exists target/scala-2.10/foo.jar
```

スクリプトの文法は以下の通り:

1. `#` は一行コメントを開始する
2. `> name` はタスクを `sbt` に送信する（そして結果が成功したかをテストする）
3. `$ name arg*` はファイルコマンドを実行する（そして結果が成功したかをテストする）
4. `-> name` タスクを `sbt` に送信するが、失敗することを期待する
5. `-$ name arg*` ファイルコマンドを実行するが、失敗することを期待する

ファイルコマンドは以下のとおり:

- `touch path+` は、ファイルを作成するかタイムスタンプを更新する
- `delete path+` は、ファイルを削除する
- `exists path+` は、ファイルが存在するか確認する
- `mkdir path+` は、ディレクトリを作成する
- `absent path+` は、はファイルが存在しないことを確認する
- `newer source target` は、`source` の方が新しいことを確認する
- `must-mirror source target` は、`source` が同一であることを確認する
- `pause` は、`enter` が押されるまで待つ
- `sleep time` は、スリープする
- `exec command args*` は、別のプロセスでコマンドを実行する
- `copy-file fromPath toPath` は、ファイルをコピーする
- `copy fromPath+ toDir` は、パスを相対構造を保ったまま `toDir` 下にコピーする
- `copy-flat fromPath+ toDir` は、パスをフラットに `toDir` 下にコピーする

ということで、僕のスクリプトは、`assembly` タスクを実行して、`foo.jar` が作成されたかをチェックする。もっと複雑なテストは後ほど。

ステップ 5: スクリプトを実行する

スクリプトを実行するためには、プラグインのプロジェクトに戻って、以下を実行する:

```
> scripted
```

これはテストビルドをテンポラリディレクトリにコピーして、`test` スクリプトを実行する。もし全て順調にいけば、まず `publishLocal` の様子が表示され、以下のようなものが表示される:

```
Running sbt-assembly / simple
[success] Total time: 18 s, completed Sep 17, 2011 3:00:58 AM
```

ステップ 6: カスタムアサーション

ファイルコマンドは便利だけど、実際のコンテンツをテストしないため、それだけでは不十分だ。コンテンツをテストする簡単な方法は、テストビルドにカスタムのタスクを実装してしまうことだ。

上記の `hello` プロジェクトを例にとると、生成された `jar` が “hello” と表示するかを確認したいとする。`sbt.Process` を用いて `jar` を走らせることができる。失敗を表すには、単にエラーを投げればよい。以下に `build.sbt` を示す:

```
lazy val root = (project in file(".")).
  settings(
    version := "0.1",
    scalaVersion := "2.10.6",
    assemblyJarName in assembly := "foo.jar",
    TaskKey[Unit]("check") := {
      val process = sbt.Process("java", Seq("-jar", (crossTarget.value / "foo.jar").toString))
      val out = (process!!)
      if (out.trim != "bye") error("unexpected output: " + out)
    }
  )
```

ここでは、テストが失敗するのを確認するため、わざと “bye” とマッチするかテストしている。

これが test:

```
# ファイルが作成されたかを確認
> assembly
$ exists target/foo.jar
```

```
# hello って言うか確認
> check
```

scripted を走らせると、意図通りテストは失敗する:

```
[info] [error] {file:/private/var/folders/Ab/AbC1EFghIj4LMNOPqrStUV+++XX/-Tmp-/sbt_cdd1b...
[info] [error] Total time: 0 s, completed Sep 21, 2011 8:43:03 PM
[error] x sbt-assembly / simple
[error]      {line 6} Command failed: check failed
[error] {file:/Users/foo/work/sbt-assembly/}default-373f46/*:scripted: sbt-assembly / si...
[error] Total time: 14 s, completed Sep 21, 2011 8:00:00 PM
```

ステップ 7: テストをテストする

慣れるまでは、テスト自体がちゃんと振る舞うのに少し時間がかかるかもしれない。ここで使える便利なテクニックがいくつかある。

まず最初に試すべきなのは、ログバッファリングを切ることだ。

```
> set scriptedBufferLog := false
```

これにより、例えばテンポラリディレクトリの場所などが分かるようになる:

```
[info] [info] Set current project to default-c6500b (in build file:/private/var/folders/...
...
```

テスト中にテンポラリディレクトリを見たいような状況があるかもしれない。
test スクリプトに以下の一行を加えると、scripted はエンターキーを押すまで一時停止する:

```
$ pause
```

もしうまくいなくて、sbt/sbt-test/sbt-foo/simple から直接 sbt を実行しようと思っているなら、それは止めたほうがいい。正しいやり方はディレクトリごと別の場所にコピーしてから走らせることだ。

ステップ 8: インスパイアされる

sbt プロジェクト下には文字通り 100+ の scripted テストがある。色々眺めてみて、インスパイアされよう。

例えば、以下に by-name と呼ばれるものを示す:

```
> compile
```

```
# change => Int to Function0
$ copy-file changes/A.scala A.scala
```

```
# Both A.scala and B.scala need to be recompiled because the type has changed
-> compile
```

xsbt-web-plugin や sbt-assembly にも scripted テストがある。

これでおしまい！プラグインをテストしてみた経験などを聞かせて下さい！

How to

How to 記事の一覧は目次を参照してください。

逐次実行

sbt で最もよくある質問の一つに「X をやった後で Y をするにはどうすればいいのか?」というものがある。

一般論としては、sbt のタスクはそうのように作られていない。なぜなら、build.sbt はタスクの依存グラフを作るための DSL だからだ。これに関してはタスクの実行意味論で解説してある。そのため、理想的にはタスク Y を自分で定義して、そこからタスク X に依存させるべきだ。

```
taskY := {
  val x = taskX.value
  x + 1
}
```

これは、以下のような、副作用のあるメソッド呼び出しを続けて行っているような命令型の素の Scala と比べるとより制限されていると言える:

```
def foo(): Unit = {
  doX()
}
```

```
doY()  
}
```

この依存指向なプログラミング・モデルの利点は sbt のタスク・エンジンがタスクの実行の順序を入れ替えることができることにある。実際、可能な限り sbt は依存タスクを並列に実行する。もう一つの利点は、グラフを非重複化して一回のコマンド実行に対して `compile in Compile` などのタスクは一度だけ実行することで、同じソースを何度もコンパイルすることを回避している。

タスク・システムがこのような設計になっているため、何かを逐次実行させるというのは一応可能ではあるけども、システムの流れに反する行為であり、簡単だとは言えない。

- `Def.sequential` を用いて逐次タスクを定義する
- `Def.taskDyn` を用いて動的タスクを定義する
- タスクの後で何かする
- `Def.inputTaskDyn` を用いた動的インプットタスクの定義
- コマンドを用いた逐次実行

Def.sequential を用いて逐次タスクを定義する

sbt 0.13.8 で `Def.sequential` という関数が追加されて、準逐次の意味論でタスクを実行できるようになった。逐次タスクの説明として `compilecheck` というカスタムタスクを定義してみよう。これは、まず `compile in Compile` を実行して、その後で `scalastyle-sbt-plugin` の `scalastyle in Compile` を呼び出す。

セットアップはこのようになる。

project/build.properties

```
sbt.version=0.13.12
```

project/style.sbt

```
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.8.0")
```

build.sbt

```
lazy val compilecheck = taskKey[Unit]("compile and then scalastyle")
```

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

```
settings(
  compilecheck in Compile := Def.sequential(
    compile in Compile,
    (scalastyle in Compile).toTask("")
  ).value
)
```

このタスクを呼び出すには、シェルから `compilecheck` と打ち込む。もしコンパイルが失敗すると、`compilecheck` はそこで実行を中止する。

```
root> compilecheck
[info] Compiling 1 Scala source to /Users/x/proj/target/scala-2.10/classes...
[error] /Users/x/proj/src/main/scala/Foo.scala:3: Unmatched closing brace '}' ignored here
[error] }
[error] ^
[error] one error found
[error] (compile:compileIncremental) Compilation failed
```

これで、タスクを逐次実行できた。

Def.taskDyn を用いて動的タスクを定義する

逐次タスクだけで十分じゃなければ、次のステップは動的タスクだ。純粋な型 `A` の値を返すことを期待する `Def.task` と違って、`Def.taskDyn` は `sbt.Def.Initialize[sbt.Task[A]]` という型のタスク・エンジンが残りの計算を継続するタスクを返す。

`compile in Compile` を実行した後で `scalastyle-sbt-plugin` の `scalastyle in Compile` タスクを実行するカスタムタスク、`compilecheck` を実装してみよう。

project/build.properties

```
sbt.version=0.13.12
```

project/style.sbt

```
addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.8.0")
```

build.sbt v1


```

lazy val compilecheck = taskKey[sbt.inc.Analysis]("compile and then scalastyle")

lazy val root = (project in file(".")).
  settings(
    compilecheck := (Def.taskDyn {
      val c = (compile in Compile).value
      Def.task {
        val x = (scalastyle in Compile).toTask("").value
        c
      }
    }).value
  )

```

これで逐次タスクと同じものができたけども、違いは最初のタスクの結果である `c` を返していることだ。

build.sbt v2

`compile in Compile` の戻り値と同じ型を返せるようになったので、もとのキーをこの動的タスクで再配線 (rewire) できるかもしれない。

```

lazy val root = (project in file(".")).
  settings(
    compile in Compile := (Def.taskDyn {
      val c = (compile in Compile).value
      Def.task {
        val x = (scalastyle in Compile).toTask("").value
        c
      }
    }).value
  )

```

これで、`compild in Compile` をシェルから呼び出してやりたかったことをやらせれるようになった。

インプットタスクの後で何かする

ここまでタスクに焦点を当ててみてきた。タスクには他にインプットタスクというものがある、これはユーザからの入力をシェル上で受け取る。典型的な例としては `run in Compile` タスクがある。`scalastyle` タスクも実はインプットタスクだ。インプットタスクの詳細は [Input Task](#) 参照。

ここで、`run in Compile` タスクの実行後にテスト用にブラウザを開く方法を考えてみる。

`src/main/scala/Greeting.scala`

```
object Greeting extends App {  
  println("hello " + args.toList)  
}
```

`build.sbt v1`

```
lazy val runopen = inputKey[Unit]("run and then open the browser")
```

```
lazy val root = (project in file(".")).  
  settings(  
    runopen := {  
      (run in Compile).evaluated  
      println("open browser!")  
    }  
  )
```

ここでは、ブラウザを本当に開く代わりに副作用のある `println` で例示した。シェルからこのタスクを呼び出してみよう:

```
> runopen foo  
[info] Compiling 1 Scala source to /x/proj/...  
[info] Running Greeting foo  
hello List(foo)  
open browser!
```

`build.sbt v2`

この新しいインプットタスクを `run in Compile` に再配線することで、実は `runopen` キーを外すことができる:

```
lazy val root = (project in file(".")).  
  settings(  
    run in Compile := {  
      (run in Compile).evaluated  
      println("open browser!")  
    }  
  )
```

Def.inputTaskDyn を用いた動的インプットタスクの定義

ここで、プラグインが `openbrowser` というブラウザを開くタスクを既に提供していると仮定する。それをインプットタスクの後で呼び出す方法を考察する。

build.sbt v1

```
lazy val runopen = inputKey[Unit]("run and then open the browser")
lazy val openbrowser = taskKey[Unit]("open the browser")

lazy val root = (project in file(".")).
  settings(
    runopen := (Def.inputTaskDyn {
      import sbt.complete.Parsers.spaceDelimited
      val args = spaceDelimited("<args>").parsed
      Def.taskDyn {
        (run in Compile).toTask(" " + args.mkString(" ")).value
        openbrowser
      }
    }).evaluated,
    openbrowser := {
      println("open browser!")
    }
  )
```

build.sbt v2

この動的インプットタスクを `run in Compile` に再配線するのは複雑な作業だ。内側の `run in Compile` は既に継続タスクの中に入ってしまったので、単純に再配線しただけだと循環参照を作ってしまうことになる。この循環を断ち切るためには、`run in Compile` のクローンである `actualRun in Compile` を導入する必要がある:

```
lazy val actualRun = inputKey[Unit]("The actual run task")
lazy val openbrowser = taskKey[Unit]("open the browser")

lazy val root = (project in file(".")).
  settings(
    run in Compile := (Def.inputTaskDyn {
      import sbt.complete.Parsers.spaceDelimited
```

```

val args = spaceDelimited("<args>").parsed
Def.taskDyn {
  (actualRun in Compile).toTask(" " + args.mkString(" ")).value
  openbrowser
}
}).evaluated,
actualRun in Compile := Defaults.runTask(
  fullClasspath in Runtime,
  mainClass in (Compile, run),
  runner in (Compile, run)
).evaluated,
openbrowser := {
  println("open browser!")
}
)

```

この actualRun in Compile の実装は Defaults.scala にある run の実装からコピペしてきた。

これで run foo をシェルから打ち込むと、actualRun in Compile を引数とともに評価して、その後で openbrowser タスクを評価するようになった。

コマンドを用いた逐次実行

副作用にしか使っていないくて、人間がコマンドを打ち込んでいるのを真似したいだけならば、カスタムコマンドを作れば済むことかもしれない。これは例えば、リリース手順とかに役立つ。

これは sbt そのもののビルドスクリプトから抜粋だ:

```

commands += Command.command("releaseNightly") { state =>
  "stampVersion" ::
  "clean" ::
  "compile" ::
  "publish" ::
  "bintrayRelease" ::
  state
}

```