始める sbt

目 次

Preface

「始める sbt」で君も sbt を始めよう。

sbt は柔軟で強力なビルド定義 (build definition) を作ることができるけど、そこで使われている概念の数は多くはない。多くはないけども、sbt は他の

ビルドシステムとは一味ちがうから、ドキュメントを読まなければ、いくつかの細かい点でハマっちゃうと思う。

この「始める sbt」で、ビルド定義 (build definition) の作成とメンテナンスに必要な概念を説明する。

「始める sbt」を読むことを強くお勧めしたい！

どうしても忙しいという場合は、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けど、このガイドの他のページを読み飛ばしても大丈夫かは保障しない。

後のページは前のページで紹介された概念をもとにしてるから、順番に読んでいくのがベスト。

sbt を試してくれて、ありがとう。楽しもう！

> 誤訳の報告をする。 sbt0.13 での変更点や新機能に興味がある人は、こちら sbt 0.13.0 の変更点 も一緒に読むといいでしょう。

sbt のインストール

sbt プロジェクトを作るには、以下の手順をたどる:

- 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 の走らせ方を覚える。
- .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習う。

最終的には、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設置という 2 つに絞られるけども、プラットフォームによってもう少し簡単なインストール方法をいくつか提供する。 Mac、Windows、Linux、Typesafe Activator、もしくは手動インストールの手順に進んでほしい。

コツと注意

sbt の実行に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 のオプションにかんする説明を参照する。

Mac への sbt のインストール

サードパーティからのパッケージを使ってのインストール

> 注意: サードパーティからの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連絡してほしい。

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。

Typesafe Activator

Typesafe 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。

Typesafe Activator

Typesafe 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・マネージャがあって (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-get update
sudo apt-get install sbt
```

パッケージ・マネージャは設定されたリポジトリに指定されたパッケージがあるか確認しにいく。 sbt のバイナリは Bintray にて公開されており、Bintray は APT 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。 `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:

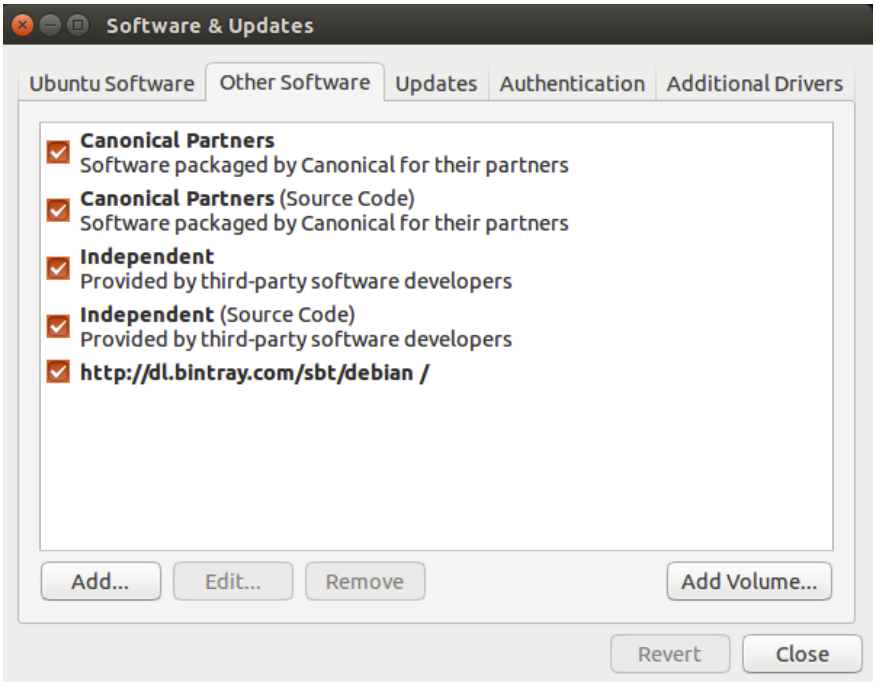

図 1: Ubuntu Software & Updates Screenshot

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
```

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 は提供していないけども、バイナリから sbt をマージする ebuild があるみたいだ。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 に関する問題があればここに報告してほしい。

Typesafe Activator

Typesafe 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 に置く。

~/bin/sbt に以下のスクリプトを作成して JAR を起動する:

```
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 セッティングを調整する。

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 "
```

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。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ので、`-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

Typesafe Activator (sbt を含む) のインストール

Typesafe 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 の上位セットであると言える。

Activator は typesafe.com から取得できる。

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` を含む。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` と打ち込むと `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` と打ち込む。
- `activator new` を使うことで豊富な テンプレートのカタログをもと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、君が sbt をインストールしたことを前提にする。

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` と打ち込む。 Linux か OS X を使っていならば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 は純粋に convention（デフォルトの慣例）だけを使って動作している。 sbt は以下を自動的に検知する:

- ベースディレクトリ内のソース
- `src/main/scala` か `src/main/java` 内のソース
- `src/test/scala` か `src/test/java` 内のテスト
- `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` を用いてプロジェクトを実行したり、`sbt console` を用いて 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から、君のプロジェクトのコードを使った Scala の例をライブ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は何らかの手動設定が必要だ。基本的なビルド設定は `build.sbt` というファイルに書かれ、プロジェクトのベースディレクトリ (base directory) に置かれる。

例えば、君のプロジェクトが `hello` ディレクトリにあるなら、`hello/build.sbt` をこんな感じで書く:

```
lazy val root = (project in file(".")).
  settings(
    name := "hello",
    version := "1.0",
    scalaVersion := "2.11.4"
  )
```

.sbt ビルド定義で、build.sbt の書き方をもっと詳しく説明する。

君のプロジェクトを jar ファイルにパッケージ化する予定なら、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で、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=0.13.8
```

sbt はリリース間で 99% ソースの互換性を持たせてある。 だけど、sbt バージョンを project/build.properties に設定することで混乱を予防することできる。

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 を見たことを前提にする。

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とすると、ベースディレクトリは hello だ。

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。だけど、ほとんどの人は、本物のプロジェクトではそうしない。ゴチャゴチャしすぎるからね。

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 に含むファイル>
    scala/
      <メインの Scala ソース>
    java/
      <メインの Java ソース>
  test/
    resources/
      <テストの jar に含むファイル>
    scala/
      <テストの Scala ソース>
    java/
      <テストの Java ソース>
```

src/ 内の他のディレクトリは無視される。あとは、隠れ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 があるのはもう分かった。他の sbt 関連のファイルは project 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さって一つの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けど、それはプロジェクトのベースディレクトリ下の .sbt とは別物だ。これに関しても、他に前提となる知識が必要なので、後で説明する。

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、パッケージ化された jar ファイル、マネージファイル、キャッシュ、とドキュメンテーション）は、デフォルトで target ディレクトリに置かれる。

バージョン管理の設定

君の .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）は以下を含むべきだ:

```
target/
```

これは（ディレクトリだけにマッチさせるために）語尾の / はつけているけど、 （普通の target/ に加えて project/target/ にもマッチさせるために）先頭の / は意図して 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にする。

インタラクティブモード

プロジェクトのディレクトリで、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が開始する。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 プロンプトに打ち込む:

```
> compile
```

もう一度 compile するには、上矢印を押して、エンターを押す。

君のプログラムを実行するには、run と打ち込む。

インタラクティブモードを終了するには、exit と打ち込むか、Ctrl+D (Unix) か Ctrl+Z (Windows) を用いる。

バッチモード

sbt アクションを空白で区切られたリストとして引数に渡すことで、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 コマンドに関しては、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。アクションは順に実行される（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たびに sbt を使って自動的に再コンパイルすることができる。

ソースファイルが変更されたことを検知してアクションを実行するには、アクションの先頭に ~ を書く。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
```

エンターを押すと、変更の監視を中止できる。

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

以下に、最もよく使われる sbt コマンドを紹介する。より完全な一覧は Command Line Reference にある。

clean

全ての生成されたファイル（target ディレクトリ）を削除する。

compile

メインのソース (src/main/scala と src/main/java ディレクトリにあるをコンパイルする。

test

全てのテストをコンパイルし実行する。

console

コンパイル済み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 と打ち込むか、Ctrl+D (Unix) か Ctrl+Z (Windows) を使う。

run <argument>*

sbt 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。

package

src/main/resources 内のファイルと src/main/scala と src/main/java からコンパイルされたクラスを含む jar を作る。

help <command>

指定されたコマンドの詳しい説明を表示する。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。

reload

ビルド定義（build.sbt、 project/*.scala*、 *project/*.sbt 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、より回りくどい候補が表示される。

履歴コマンド

インタラクティブモードは、たとえ 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:

!

履歴コマンドのヘルプを表示する。

!!

直前のコマンドを再実行する。

!:

全てのコマンド履歴を表示する。

!:n

最後の n コマンドを表示する。

!n

!: で表示されたインデックス n のコマンドを実行する。

!-n

n 個前のコマンドを実行する。

!string

'string' から始まる最近のコマンドを実行する。

!?string

'string' を含む最近のコマンドを実行する。

.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て、「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 `.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 `.sbt` ビルド定義を紹介する。これは、従来あった 2 つのビルド定義の長所を組み合わせたもので、全ての状況において使うことができる。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。

ビルド定義って何？

** 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て、ビルド定義に関わるファイル群を処理した後、sbt は Project の定義群を最終的に作る。

例えば、現在のディレクトリに位置するプロジェクトの Project 定義を以下のようにして build.sbt に書くことができる:

```
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した不可変マップ（キーと値のペア）に関連付けられる。

例えば、name というキーがあり、それは文字列の値、つまり君のプロジェクト名に関連付けられる。

ビルド定義ファイルは直接には *sbt* のマップに影響を与えない。

その代わり、ビルド定義は、型が Setting[T] のオブジェクトを含んだ巨大なリストを作る。 T はマップ内の値の型だ。（Scala の Setting[T] は Java の Setting<T> と同様。） Setting は、新しいキーと値のペアや、既存の値への追加など、マップの変換を記述する。（関数型プログラミングの精神に則り、変換は新しいマップを返し、古いマップは更新されない。）

現在の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マップを変換する。 変換されたマップは新しい sbt のマップとなる。

マップを作るために、sbt はまず、同じキーへの変更が一緒に起き、かつ他のキーに依存する値の処理が依存するキーの後にくるように Setting のリス

トをソートする。 次に、sbt はソートされた Setting のリストを順番にみていって、一つづつマップに適用する。

まとめ: ビルド定義は *Setting[T]* のリストを持った *Project* を定義して、*Setting[T]* は *sbt* のキー・値ペアへの変換を表し、*T* は値の型を指す。

build.sbt はどう設定値を定義するか

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。

以下に具体例で説明しよう:

```
lazy val root = (project in file(".")).
  settings(
    name := "hello",
    version := "1.0",
    scalaVersion := "2.11.4"
  )
```

それぞれの Setting は Scala の式で表される。 settings 内の式は、それぞれ独立しており、完全な Scala 文ではなく、式だ。

build.sbt 内には、他に val、lazy val、def を定義することができる。トップレベルでの object とクラスの定義は build.sbt 内では禁止されている。 それらは project/ ディレクトリ直下の.scala ファイルに置かれる。

左辺値の name、version、および scalaVersion は キーだ。 キーは SettingKey[T]、TaskKey[T]、もしくは InputKey[T] のインスタンスで、T はその値の型だ。キーの種類に関しては後で説明しよう。

キーには := メソッドがあり、Setting[T] を返す。 Java 的な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")
  )
```

だけど、Scala では name := "hello" と書ける（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 の型にもあらわれ、これは、SettingKey[String] となっている。この場合、返された Setting[String] は、キーを追加（もしくは置換）して "hello" という値に設定するマップの変換だ。

間違った型の値を使うと、ビルド定義はコンパイルできない:

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

Keys

種類　キーには三種類ある:

- SettingKey[T]: 値が一度だけ計算されるキー（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受け取るタスクキー。　「初めての 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 として呼ぶことができる。

カスタムキー　カスタムキーは settingKey、 taskKey、inputKey といった関数を用いて定義する。どの関数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。例えば、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いけない。

> 注意 初期化順問題を避けるために val の代わりに lazy val を用いている。

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 と打ち込むなどして、 タスクを実行するたびに、sbt は関連したタスクを一回だけ再実行する。

プロジェクトを記述した sbt のマップは、name のようなセッティング (setting) ならば、その文字列の値をキャッシュすることができるけど、compile のようなタスク（task）の場合は実行可能コードを保存する必要がある（たとえその実行可能コードが最終的に同じ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いけない）。

あるキーがあるとき、それは常にタスクか素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）はキーの特性であり、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、タスキ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打ち込むことで、どのタスクでも実行することができる。それが compile と打ち込むことでコンパイルタスクが起動する仕組みだ。つまり、compile はタスクキーなのだ。

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 の慣例にのっとり、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ためには、sbt インタラクティブモードで inspect <キー名> と打ち込む。 inspect が表示する情報の中にはまだ分からないこともあると思うけど、一番上にはセッティングの値の型と、セッテイングの簡単な説明がある。

build.sbt 内の import 文

build.sbt の一番上に import 文を書くことができ、それらは空行でわけなくてもいい。

自動的に以下のものがデフォルトでインポートされる:

```
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、unmanged 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:

```
lazy val root = (project in file(".")).
  settings(
    name := "hello",
    libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
  )
```

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けど、+= はキーの古い値を上書きするか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しては、このガイドの後ほどまで少しおいておくことにする。後で、一ページ分をさいてちゃんと説明する。

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

これまでは、あたかも `name` のようなキーは単一の sbt のマップのキー・値ペアの項目に対応するフリをして話を進めてきた。それは単純化した話だ。

実のところは、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・値のマップを生成するためにセッティングのリストを処理していくことを考えると、このキー・値マップ内のキーは、スコープ付きキーであることが分かる。また、（`build.sbt` などの）ビルド定義内の、セッティングもスコープ付きキーに適用されるものだ。

スコープは、デフォルトがあったり、暗示されていたりするが、デフォルトが間違っていれば `build.sbt` にてスコープを指定しなければいけない。

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、予想通りのもので、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、複数のより特定な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し（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、現在のプロジェクトが使われる。
- コンフィグ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、現在のプロジェクト、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内で `inspect` コマンドを使ってキーとそのスコープを理解することができる。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、まだ意味不明だろうけど、次のページまで待ってて。

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。コンフィグ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 コンフィグレーション、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、コンフィグ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、現在の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（キーと値のペアのマップで表現される）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。 Setting は、は古いマップをインプットにとり、新たなマップをアウトプットに出す変換だ。新たなマップが 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マップを変換する。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マップに代入する。例えば、マップを name := "hello" というセッティングで変換すると、新しいマップは 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けど、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。（君がどうしても非標準じゃないと気が済まないので）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とすると、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"))
```

このでの 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

タスクやセッティングのキーの値を使って他のタスクやセッティングのキーの値を設定してみる。値をつける関数には:= や += や ++= を用いて、引数に付ける値を入れてやればよい。

最初の例として、プロジェクトの名前と同じ組織名を定義してみよう。

```
// 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
```

他にもディレクトリ名を用いてプロジェクトの名前をつけるとか。

```
// 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
```

これは java.io.File オブジェクトにある getName メソッドを用いて、baseDirectry から取った値へと変換している。

複数の入力値を用いる場合も同様である。

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

ここでも前に宣言された組織名やバージョンのセッティング値を用いて名前が付けられている。

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` に_依存性_（dependency）を持つ。上記を `build.sbt` に書いて、sbt のインタラクティブモードで走らせ、`inspect name` と打ち込むと、以下のように表示される（一部抜粋）：

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

このようにして、sbt はどのセッティングが別のセッティングに依存するかを知っている。タスクを記述するセッティングもあるため、この方法でタスク間の依存性も作ることができる。

例えば、`inspect compile` すると、`compile-inputs` に依存することが分かり、 `compile-input` 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` の計算にインプットとして必要な値が、`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。とにかくちゃんと動く！

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性を作る場合、依存されたキーには値が存在しなくてはならない。 存在しなければ、sbt に怒られる。 例えば、*"Reference to undefined setting"* なんて言われるかもしれない。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。

環状の依存性を作ってしまうことも可能で、これもまたエラーになり、sbt に怒られる。

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` を用いた値を使えば良い。

例として、`sourceGenerators` にプロジェクトのルートディレクトリや `Compile` コンフィギュレーション時のクラスパスを加える事を考えてみよう。

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

依存性を持ったタスク　.sbt ビルド定義でみた通り、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。タスクを定義する際の入力には、セッティングの値を用いることができるが、セッテイングを定義する際にタスクを入力とすることはできない。

（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、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で、お互い全く関係のないキーだ。）

build.sbt で scalacOptions を checksums のエイリアスにすることは出来るのだが、その逆は出来ない。つまり、以下のような例で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、どのようにしても出来ない。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ないためである。逆に、タスクは何度も実行毎に評価される可能性がある。

```
// 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: += and ++=

他のキーを用いて、既存のセッティングやタスクへ値を追加する際は、+= を使えば良い。

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るとして、それを clean が削除するファイルのリストに追加したいとする:

```
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でできる_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

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っている。だけど、始めはアンマネージの方が簡単なので分かりやすい。

アンマネージ依存性を説明すると、こんな感じになる。 jar ファイルを `lib` に入れると、それはプロジェクトのクラスパスに追加される。以上！

例えば、ScalaCheck、specs、ScalaTest などのテスト用の jar を `lib` に加えることもできる。

`lib` の依存性は（`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` には何も書かなくてもいいけど、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` キーを変更することができる。

`lib` のかわりに、`custom_lib` を使うには:

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

`baseDirectory` はプロジェクトのルートディレクトリで、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、`unmangedJars` 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

sbt は、Apache Ivy を使ってマネージ依存性を実装するため、既に Maven か Ivy に慣れていれば、違和感無く入り込めるだろう。

libraryDependencies キー　依存性を libraryDependencies セッティングに列挙するだけで、普通はうまくいく。 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、以下のようになる。ここで、groupId、artifactId、と revision は文字列だ:

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

もしくは、以下のようになる。このときの configuration も文字列だ。

```
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 オブジェクトを作り、その ModuleID を libraryDependencies に追加するだけでいい。

当然、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知らなければいけない。もしモジュールが sbt に最初から入っているデフォルトのリポジトリの一つにあれば、ちゃんと動く。例えば、Apache Derby はデフォルトのリポジトリにある:

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

これを build.sbt に打ち込んで、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、手動で update と打ち込む必要がある状況は普通は無い。）

当然、++= を使って一度に依存ライブラリのリストを追加することもできる:

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

libraryDependencies に対して:=、その他を使う機会があるかもしれないが、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 %%）、 sbt はプロジェクトの Scala 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_2.9.1" % "0.3"
```

ビルドのバージョンが scalaVersion が 2.9.1 だとすると、以下は等価だ:

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

多くの依存ライブラリが複数の Scala バージョンに対してコンパイルされていて、プロジェクトに合ったものを選択したいときに使うというのが考えだ。

実践上での問題として、多くの場合依存ライブラリは少しズレた Scala バージョンが使われることがあるけど、 %% はそこまでは賢くない。そのため、依存ライブラリが 2.10.1 までしか出てなくて、プロジェクトが scalaVersion := "2.10.4" の場合、2.10.1 の依存ライブラリが多分動作するにも関わらず%% を使うことができない。 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じゃなくてもいい。 Ivy は与えられた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 standard Maven2 repository のリポジトリを使う。もし依存ライブラリがこのデフォルトのリポジトリに無ければ、Ivy がそれを見つけられるように *resolver* を追加する必要がある。

リポジトリを追加するには、以下のようにする:

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

例えば:

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

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

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

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
```

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欲しいけど、Compile コンフィギュレーションではいらない場合は、以下のように% "test" と追加する:

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

以下のように書くことで、 Test コンフィギュレーションに合わせた型安全なバーションを用いることが出来るかもしれない:

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

sbt のインタラクティブモードで show compile:dependency-classpath と打ち込んでも、Derby は出てこないはずだ。 だけど、show test:dependency-classpath と打ち込むと、Derby の jar がリストにあるのが確認できる。

普通は、ScalaCheck、specs、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。

このガイドの前のページ、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

単一のビルドに複数の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、プロジェクトが同時に変更されることが多い場合などで便利だ。

ビルド内のサブプロジェクトは、それぞれに独自の src/main/scala 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、 普通のプロジェクト同様に振る舞う。

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
```

変数名はプロジェクトの 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以下の様な関数を呼び出す事で変更出来る。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　複数のプロジェクト間に共通のセッティングを抜き出すには、commonSettings という名前で列を作って、 各プロジェクトから settings メソッドを呼べばいい。可変長引数を受け取るメソッドに列を渡すのに _* が必要なことに注意。

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

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った。

依存関係

個々のプロジェクトのビルドを他のプロジェクトと完全に独立した形で行うこともできる。しかし、大抵プロジェクトというのは他のプロジェクトと何らかの形で依存関係を持つ。ここには集約かクラスパス依存性といった 2 種類の依存関係が存在する:

集約　集約は、集約するプロジェクトを走らせる際に集約されるプロジェクトも同時に走らせる必要がある、といった関係のことである。例えば、以下のような例がある。

```
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、ルートの hello 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　プロジェクトは、他のプロジェクトのコードに依存することができる。これは、dependsOn メソッドを呼び出すことで実現する。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 のクラスを利用することができる。これは、プロジェクトをコンパイルするときの順序も作り出す。この場合、core がコンパイルされる前に、util が更新（update）され、 コンパイルされる必要がある。

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いい。

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」という意味だから、"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。だけど、三つの build.sbt とも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、好きにやっていい。

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 と打ち込む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、現在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、それは現在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、他のプロジェクト 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。

Common code

.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、ビルドルートにある project/ ディレクトリに Scala ファイルを用意すれば良い。

詳細はビルドの整理を見てほしい。

プラグインの使用

このガイドの前のページをまず読んでほしい。 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。

プラグインって何?

プラグインは、新たなセッティングを追加するなどして、ビルド定義を拡張する。そのセッティングは、新しいタスクを加えることもでき、例えば、テストカバレッジレポートを生成する codeCoverage というタスクをプラグインが提供することができる。

プラグインの宣言

プロジェクトが hello ディレクトリにあって、ビルド定義に sbt-site を追加するとした場合、hello/project/site.sbt を新規作成して、 プラグインの Ivy モジュール ID と共に addSbtPlugin を呼び出す:

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

sbt-assembly を追加したければ、hello/project/assembly.sbt を作って以下を書く:

```
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ある訳では無いので、プラグインの説明書にレポジトリの追加する手順が書かれているかもしれない:

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

通常、プラグインは、プロジェクトに追加されるセッティングを提供することで機能を追加する。これを次に説明しよう。

auto plugin の有効化と無効化

プラグインはビルド定義に自動的に追加されるセッティングを宣言することができ、その場合は何もしなくてもいい。

sbt 0.13.5 より新しい auto plugin というプラグインを自動的に追加して、セッティングの依存性がプロジェクトにあることを安全に保証する機構ができた。 auto plugin の多くはデフォルトセッティングを自動的に追加するけども、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:

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

プロジェクトは enablePlugins メソッドを用いて使用したい auto plugin を明示的に定義することができる。

プロジェクトは、disablePlugins メソッドを用い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はそれぞれの auto plugin がドキュメントに書くべきだ。 現行プロジェクトにおいてどの auto plugin が有効化されているかが気になるなら、 sbt コンソールから plugins コマンドを実行してみてほしい。

例えば、

```
> 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。

古い 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を明示的に追加することを必要とする。それぞれのプラグインのドキュメンテーションに設定方法が書いてあると思うけど、典型的にはベースとなるセッティングを追加して、必要に応じてカスタム化というパターンが多い。

sbt-site を用いて具体例で説明すると、site.sbt というファイルを新しく作って

```
site.settings
```

以上を site.sbt に書くことで有効化することができる。

もし、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、sbt プロジェクトで、そのクラスパスは全ての sbt ビルド定義にエクスポートされる。大まかに言うと、~/.sbt/0.13/plugins/ 内の.sbt 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振る舞う。

~/.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 専用プラグイン（sbt プロジェクトを IDE にインポートする）
- xsbt-web-plugin などの、web フレームワークプラグイン

プラグインの作成などプラグインに関する詳細は Plugins とプラグインのベスト・プラクティスを参照。

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。

キーの定義

Keys は、キーの定義の方法で満載だ。多くのキーは、Defaults で実装されている。

キーは三つの型のうちどれかを持つ。 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 source
```

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 ビルド定義でみたとおり、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 ビルド定義でみたとおり、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」のたび（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打ち込まれるたび）に再計算される。

キーは .sbt ファイル、.scala ファイル、もしくは 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。

タスクの実装

キーを定義したら、次はなんらかのタスクからそのキーを使ってみよう。独自のタスクを定義してもいいし、既存のタスクを再定義する予定なのかもしれない。いずれにせよ、やることは同じだ。もしタスクに他のセッティングやタスクへの依存性が無ければ、:= を使ってタスクのキーになんらかのコードを関連付ける:

```
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
    }
  )
```

もしタスクに依存性があれば、他の種類のセッティングで説明したとおり `value` を使って値を参照する。

タスクは、ただの Scala のコードであるため、実装の一番難しい部分は、多くの場合 sbt 特定の問題ではない。難しいのは、実行したい何らかの「本体」の部分を書くことで、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する必要があるかもしれない（その場合は、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く）。

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って、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 を重宝する。

タスクの実行意味論

カスタムタスクの中から `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` を例に取ると、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:

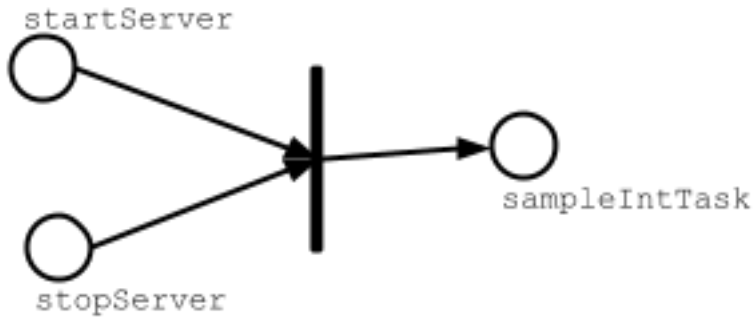

図 2: task-dependency

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。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 の両方に依存して、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、評価は一回だけ行われる。 以下は sampleStringTask の評価を視覚化したものだ:

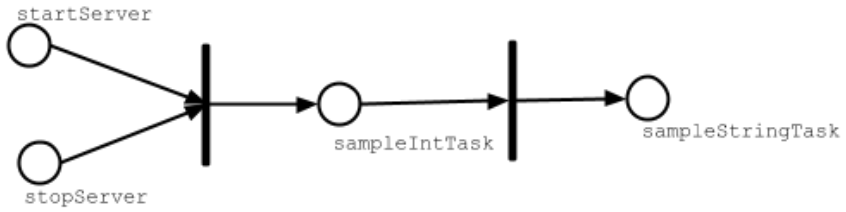

図 3: task-dependency

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　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
```

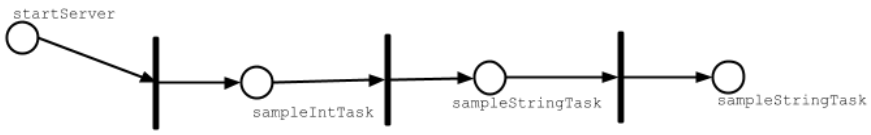

図 4: task-dependency

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けど、プラグインを作るのはすごく簡単だ。

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いけない。当然これも sbt でビルドされる。

project ディレクトリは現行のビルドのビルド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、一番上のビルドはプロパービルド (proper build) と呼んで、 project 内のビルドはメタビルド (meta-build) という用語で呼ぶことがある。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（理論的には）こなすことができる。つまり、 ビルド定義は **sbt** プロジェクトであるということだ。

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で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/            # 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     # 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
```

普通はこういうことをする必要は全く無いので、安心してほしい！だけど、原理を理解すると役立つことがある。

ちなみに、.scala や .sbt で終わる全てのファイルが用いられ、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っていいということも意味する。

ライブラリ依存性をまとめる

project 内の .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ということができる。

```
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 = "2.11.4"
)

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

マルチプロジェクト・ビルドが大きくなってきて、サブプロジェクト間の一貫性を保証したいときにこのようなテクニックが有用になってくる。

いつ .scala ファイルを使うか

.scala ファイルでは、クラスやオブジェクト定義を含む Scala コードを自由に書ける。

推奨される方法はセッティングは基本的にマルチプロジェクト build.sbt ファイル内で行って、 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。.scala ファイルの利用は君または、君のチームがどれだけ Scala 慣れしてるかにもよる。

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、このガイドを総括してみよう。

sbt を使うのに、理解しなければいけない概念は少しの数しかない。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。

もし、この「始める sbt」シリーズをここまで読破したなら、何を知るべきかはもう分かっていると思う。

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

- ビルド定義は、`Setting` オブジェクトが入った一つの大きなリストであり、 `Setting` は、sbt がタスクを実行するのに使うキー・値のペアを変換する。
- `Setting` を作成するには、キーに定義されているメソッドを呼び出す（特に、`:=` と `<<=` メソッドが大切だ）。
- 可変の内部状態は無く、変換があるだけだ。例えば、`Setting` は、sbt のキー・値のペアのコレクションを新たなコレクションへと変換され、上書き更新はされない。
- 全てのセッティングは、キーにより決定された特定の型の値を持つ。
- 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値が一度だけ計算される。
- スコープ
- それぞれのキーは、異なるスコープごとに別の値を取ることができる。
- スコープ付けには、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。
- スコープ付けにより、プロジェクトごと、タスクごと、またはコンフィギュレーションごとに、異なる振る舞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。
- プロジェクト軸は、「ビルド全体」を指すスコープにも設定することができる。
- スコープは、より一般的なスコープにフォールバックし、これを_委譲_（delegate）という。
- ビルドの整理

- build.sbt にセッティングのほとんどを置き、 .scala ビルドファイルは、共通の値、オブジェクト、メソッドなどをくくり出すのに使う。
- ビルド定義そのものも、れっきとした sbt プロジェクトで、project ディレクトリを基とする。
- 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/build.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）

以上のうち、一つでも分からないことがあれば、質問してみるか、もう一度再読してみるか、 sbt のインタラクティブモードで実験してみよう。

じゃ、頑張って！

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。現在はマルチ・プロジェクト .sbt ビルド定義が推奨される。

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 := "2.11.4"
```

(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に方法が無かったけども、 sbt 0.13 になってマルチ・プロジェクト .sbt ビルド定義が追加され、今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。

以下に、具体例で説明する。プロジェクトが hello にあるとすると、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 を以下のように書く:

```
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 と打ち込むと、以下のように表示されるはず（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 と打ち込むと、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意してほしい。 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の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_追加する_ため、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「勝つ」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。

まとめてみると:

- .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の現プロジェクトを 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/proj
> 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 **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。 そうじゃなくて、Build や Project のセッティングリストと build.sbt のセッティングが 連結されて別の不変リストになって、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。