

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
ユニバーサルパッケージからのインストール	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
Windows 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
Windows インストーラ	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	6
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
Hello, World	8
sbt new コマンド	8
アプリの実行	9
sbt シェルの終了	9
ビルド定義	9
ディレクトリ構造	9
ベースディレクトリ	10
ソースコード	10
sbt ビルド定義ファイル	11
ビルドサポートファイル	11
ビルド成果物	11

バージョン管理の設定	11
実行	12
sbt シェル	12
バッチモード	12
継続的ビルドとテスト	13
よく使われるコマンド	13
タブ補完	14
履歴コマンド	14
ビルド定義	15
sbt バージョンの指定	15
ビルド定義とは何か	15
build.sbt はどのように settings を定義するか	16
キー	17
タスクとセッティングの定義	19
sbt シェルにおけるキー	19
build.sbt 内の import 文	20
ライブラリへの依存性を加える	20
タスク・グラフ	21
用語に関して	21
他のタスクへの依存性の宣言	22
.value 呼び出しのインライン化	24
そもそも build.sbt DSL は何のためにある?	28
まとめ	29
スコープ	30
キーに関する本当の話	30
スコープ軸	31
グローバルスコープ	32
委譲	32
sbt 実行中のスコープ付きキーの参照方法	32
スコープ付きキーの表記例	33
スコープの検査	34
ビルド定義からスコープを参照する	36
いつスコープを指定すべきか	37
値の追加	39
既存の値に追加する: += と ++=	39
依存性を用いた追加: += と ++=	40
ライブラリ依存性	40
アンマネージ依存性 (Unmanaged Dependencies)	40
マネージ依存性 (Managed Dependencies)	41

マルチプロジェクト・ビルド	45
複数のサブプロジェクト	45
依存関係	46
デフォルトルートプロジェクト	48
プロジェクトの切り替え	49
共通のコード	49
プラグインの使用	49
プラグインとは何か	50
プラグインの宣言	50
auto plugin の有効化と無効化	50
グローバル・プラグイン	52
利用可能なプラグイン	52
カスタムセッティングとタスク	53
キーを定義する	53
タスクを実装する	54
タスクの実行意味論	55
プラグイン化しよう	59
ビルドの整理	59
sbt は再帰的だ	60
ライブラリ依存性を一箇所にまとめる	61
いつ <code>.scala</code> ファイルを使うべきか	62
auto plugin を定義する	62
まとめ	62
sbt: コア・コンセプト	63
上級者への注意	64
付録: bare .sbt ビルド定義	64
bare .sbt ビルド定義とは何か	64
(0.13.7 以前) 設定は空白行で区切る	64
付録: .scala ビルド定義	65
build.sbt と Build.scala の関係	65
インタラクティブモードにおけるビルド定義	67
注意: 全てが immutable だ	68
各論	68
プラグインとベストプラクティス	68
sbt プラグインをテストする	68
scripted test framework	68
ステップ 1: snapshot	69
ステップ 2: scripted-plugin	69
ステップ 3: src/sbt-test	69

ステップ 4: スクリプトを書く	70
ステップ 5: スクリプトを実行する	71
ステップ 6: カスタムアサーション	71
ステップ 7: テストをテストする	72
ステップ 8: インスパイアされる	73
How to	73
逐次実行	73
Def.sequential を用いて逐次タスクを定義する	74
Def.taskDyn を用いて動的タスクを定義する	75
インプットタスクの後で何かする	77
Def.inputTaskDyn を用いた動的インプットタスクの定義	78
コマンドを用いた逐次実行	79

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 タスク・グラフに書かれている。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 をインストールする。
- 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、もしくは Linux 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

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。

サードパーティパッケージを使っでのインストール

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。

Homebrew

```
$ brew install sbt
```

Macports

```
$ port install sbt
```

Windows 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。

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。

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-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:

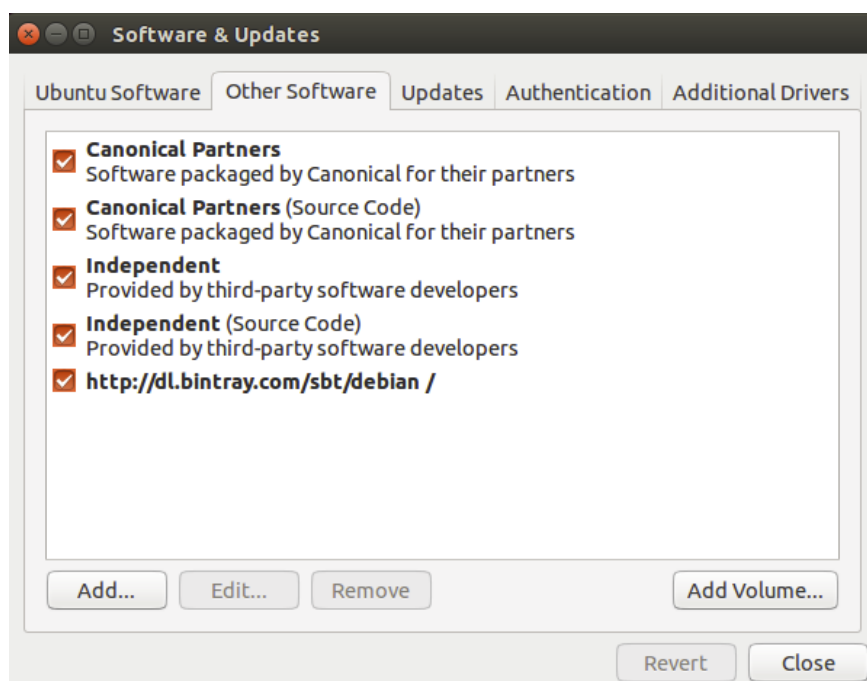


図 1: Ubuntu Software & Updates Screenshot

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。

Hello, World

このページは、既に sbt 0.13.13 以上をインストールしたことを前提とする。

sbt new コマンド

sbt 0.13.13 以降を使っている場合は、sbt **new** コマンドを使って手早く簡単な Hello world ビルドをセットアップすることができる。以下をターミナルから打ち込む。

```
$ sbt new sbt/scala-seed.g8
```

```
....
```

```
Minimum Scala build.
```

```
name [My Something Project]: hello
```


Template applied in ./hello

プロジェクト名を入力するプロンプトが出てきたら `hello` と入力する。

これで、`hello` ディレクトリ以下に新しいプロジェクトができた。

アプリの実行

次に `hello` ディレクトリ内から `sbt` を起動して `sbt` のシェルから `run` と入力する。Linux や OS X の場合、コマンドは以下のようになる：

```
$ cd hello
$ sbt
...
> run
...
[info] Compiling 1 Scala source to /xxx/hello/target/scala-2.12/classes...
[info] Running example.Hello
hello
```

後で他のタスクもみていく。

sbt シェルの終了

`sbt` シェルを終了するには、`exit` と入力するか、`Ctrl+D` (Unix) か `Ctrl+Z` (Windows) を押す。

```
> exit
```

ビルド定義

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。ファイルを読んでみてもいいが、このビルドファイルに書いてあることが分からなくても心配しないでほしい。ビルド定義で、`build.sbt` の書き方を説明する。

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` 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

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。

ソースコードは `hello/app.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が、小さいプロジェクトはともかくとして、通常のプロジェクトでは `src/main/` 以下のディレクトリにソースを入れて整理するのが普通だ。ベースディレクトリに `*.scala` ソースコードを配置できるのは小手先だけのトリックに見えるかもしれないが、この機能は後ほど重要になる。

sbt ビルド定義ファイル

ビルド定義はプロジェクトのベースディレクトリ以下の `build.sbt` (実は `*.sbt` などのファイルでもいい) にて記述する。

```
build.sbt
```

ビルドサポートファイル

`build.sbt` の他に、`project` ディレクトリにはヘルパーオブジェクトや一点物のプラグインを定義した `*.scala` ファイルを含むことができる。詳しくは、ビルドの整理を参照。

```
build.sbt
project/
  Dependencies.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。

sbt シェル

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 `sbt` シェルが起動される。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` シェルに入力する:

```
> 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。

プログラムを実行するには、`run` と入力する。

`sbt` シェルを終了するには、`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`）。

Note: バッチモードでの実行は JVM のスピナップと JIT を毎回行うため、ビルドかなり遅くなる。普段のコーディングでは `sbt` シェル、もしくは以下に説明する継続的ビルドとテストを使うことを推奨する。

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:

```
> ~testQuick
```

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 sbt シェル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 &lt;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

sbt シェルには、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

sbt シェルは、sbt を終了して再起動した後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>履歴コマンドのヘルプを表示する。</td>

<td><tt>!!&br/><td>直前のコマンドを再実行する。</td>

<td><tt>!:  
<td>全てのコマンド履歴を表示する。</td>

<td><tt>!:  
<td>最後の <tt>n</td>

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

```

<td><tt>!:</tt> で表示されたインデックス <tt>n</tt> のコマンドを実行する。</td>

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

<td><tt>n</tt>個前のコマンドを実行する。</td>

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

<td>'string' から始まる最近のコマンドを実行する。</td>

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

<td>'string' を含む最近のコマンドを実行する。</td>

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。sbt 0.13.13 など最近のバージョンをインストール済みで、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。

このページでは build.sbt ビルド定義を紹介する。

sbt バージョンの指定

ビルド定義の一部としてビルドに用いる sbt のバージョンを指定する。これによって異なる sbt ランチャーを持つ複数の人がいても同じプロジェクトを同じようにビルドすることができる。そのためには、project/build.properties という名前のファイルを作成して以下のように sbt バージョンを指定する:

```
sbt.version=0.13.13
```

もしも指定されたバージョンがローカルマシンに無ければ、sbt ランチャーは自動的にダウンロードを行う。このファイルが無ければ、sbt ランチャーは任意のバージョンを選択する。これはビルドの移植性を下げるため、推奨されない。

ビルド定義とは何か

ビルド定義は、build.sbt にて定義され、プロジェクト (型は Project) の集合によって構成される。プロジェクトという用語が曖昧であることがあるため、このガイドではこれらをサブプロジェクトと呼ぶことが多い。

例えば、カレントディレクトリにあるサブプロジェクトは `build.sbt` に以下のように定義できる：

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.1"
  )
```

それぞれのサブプロジェクトは、キーと値のペアによって詳細が設定される。

例えば、`name` というキーがあるが、それはサブプロジェクト名という文字列の値に関連付けられる。キーと値のペア列は `.settings(...)` メソッド内に列挙される：

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.1"
  )
```

`build.sbt` はどのように `settings` を定義するか

`build.sbt` において定義されるサブプロジェクトは、キーと値のペア列を持つと言ったが、このペアはセッティング式 (setting expression) と呼ばれ、`build.sbt DSL` にて記述される。

```
lazy val root = (project in file("."))
  .settings(
    name      := "hello",
    organization := "com.example",
    scalaVersion := "2.12.1",
    version    := "0.1.0-SNAPSHOT"
  )
```

organization	:=	{ "com.example" }
key	operator	(setting/task) body

setting/task expression

`build.sbt DSL` を詳しくみてみよう：

それぞれのエントリーはセッティング式 (setting expression) と呼ばれる。

中にはタスク式と呼ばれるものもある。この違いはこのページの後で説明する。

セッティング式は以下の 3 部から構成される:

1. 左辺項をキー (key) という。
2. 演算子。この場合は `:=`。
3. 右辺項は本文 (body)、もしくはセッティング本文という。

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。

`name` キーは `SettingKey[String]` に型付けされているため、`name` の `:=` 演算子も `String` に型付けされている。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
  )
```

`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 ソースファイル (`.scala`) を置くべきだろう。

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 (値はサブ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。

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。

サブプロジェクトを記述する `sbt` のキーと値の列は、`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、後ほどのタスク・グラフにて詳しくみていく。

sbt シェルにおけるキー

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。)

(さらに、auto plugin があれば autoImport 以下の名前がインポートされる。)

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-SNAPSHOT",
  scalaVersion := "2.12.1"
)
```

```

lazy val root = (project in file("."))
  .settings(
    commonSettings,
    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。後でタスク・グラフで説明するが、`+=` 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。

タスク・グラフ

ビルド定義に引き続き、このページでは `build.sbt` 定義をより詳しく解説する。

`settings` をキーと値のペア群だと考えるよりも、より良いアナロジーは、辺を事前発生 (happens-before) 関係とするタスクの有向非巡回グラフ (DAG) だと考える事だ。これをタスク・グラフと呼ぼう。

用語に関して

重要な用語をおさらいしておく。

- セッティング/タスク式: `.settings(...)` 内のエントリー。
- キー: セッティング式の左辺項。 `SettingKey[A]`、`TaskKey[A]`、もしくは `InputKey[A]` となる。
- セッティング: `SettingKey[A]` を持つセッティング式によって定義される。値はロード時に一度だけ計算される。
- タスク: `TaskKey[A]` を持つタスク式によって定義される。値は呼び出さるたびに計算される。

他のタスクへの依存性の宣言

build.sbt DSL では `.value` メソッドを用いて他のタスクやセッティングへの依存性を表現する。この `value` メソッドは特殊なもので、`:=` (もしくは後に見る `+=` や `++=`) の右辺項内でしか使うことができない。

最初の例として、`update` と `clean` というタスクに依存した形で `scalacOptions` を定義したいとする。(Keys より) 以下の二つのキーを例に説明する。

注意: ここで計算される `scalacOptions` の値はナンセンスなもので、説明のためだけのものだ:

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

以下のように `scalacOptions` を再配線できる:

```
scalacOptions := {
  val ur = update.value // update タスクは scalacOptions よりも事前発生する
  val x = clean.value   // clean タスクは scalacOptions よりも事前発生する
  // ---- scalacOptions はここから始まる ----
  ur.allConfigurations.take(3)
}
```

`update.value` と `clean.value` はタスク依存性を宣言していて、`ur.allConfigurations.take(3)` がタスクの本文となる。

`.value` は普通の Scala のメソッド呼び出しではない。build.sbt DSL はマクロを用いてこれらをタスクの本文から持ち上げる。** `update` と `clean` の両タスクとも、本文内のどの行に現れようと、タスクエンジンが `scalacOptions` の開始中括弧 (`{}`) を評価するときには既に完了済みである。**

具体例で説明しよう:

```
lazy val root = (project in file(".")).
  settings(
    name := "Hello",
    organization := "com.example",
    scalaVersion := "2.12.1",
    version := "0.1.0-SNAPSHOT",
    scalacOptions := {
      val out = streams.value // streams タスクは scalacOptions よりも事前発生する
      val log = out.log
      log.info("123")
    }
  )
```

```

    val ur = update.value // update タスクは scalacOptions よりも事前発生する
    log.info("456")
    ur.allConfigurations.take(3)
  }
)

```

次に、sbt シェル内で scalacOptions と打ち込む:

```

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

val ur = ... は log.info("123") と log.info("456") の間に挟まって
いるが、update タスクは両者よりも事前発生している。

```

もう一つの例:

```

lazy val root = (project in file(".")).
  settings(
    name := "Hello",
    organization := "com.example",
    scalaVersion := "2.12.1",
    version := "0.1.0-SNAPSHOT",
    scalacOptions := {
      val ur = update.value // update task happens-before scalacOptions
      if (false) {
        val x = clean.value // clean task happens-before scalacOptions
      }
      ur.allConfigurations.take(3)
    }
  )

```

sbt シェル内で run それから scalacOptions と打ち込む:

```

> run
[info] Updating {file:/xxx/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[info] Compiling 1 Scala source to /Users/eugene/work/quick-test/task-graph/target/scala-2.12.1/classes
[info] Running example.Hello

```

```
hello
[success] Total time: 0 s, completed Jan 2, 2017 10:45:19 PM
> scalacOptions
[info] Updating {file:/xxx/}root...
[info] Resolving jline#jline;2.14.1 ...
[info] Done updating.
[success] Total time: 0 s, completed Jan 2, 2017 10:45:23 PM
```

ここで `target/scala-2.12/classes/` を探してみしてほしい。 `if (false)` に囲まれていても `clean` タスクが実行されたため、そのディレクトリは存在しないはずだ。

もう一つ重要なのは、`update` と `clean` のタスクの間では順序付けの保証が無いことだ。 `update` してから `clean` が実行されるかもしれないし、 `clean` してから `update` が実行されるかもしれないし、両者が並列に実行される可能性もある。

.value 呼び出しのインライン化

上で解説したように、`.value` は他のタスクやセッティングへの依存性を表現するための特殊なメソッドだ。 `build.sbt` に慣れるまでは、`.value` の呼び出しをタスク本文の一番上にまとめておくことをお勧めする。

しかし、慣れてくると `.value` 呼び出しをインライン化して、タスクやセッティングを簡略に書きたいと思うようになるだろう。変数名をいちいち考えなくてもいいのも楽だ。

インライン化するとこう書ける:

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

`.value` の呼び出しがインライン化されていようが、タスク本文内のどこに書かれていてもタスク本文に入る前に評価は完了する。

タスクのインスペクト

上の例では `scalacOptions` は `update` と `clean` というタスクに依存性 (dependency) を持つ。上のタスクを `build.sbt` に書いて、`sbt` シェル内から `inspect scalacOptions` と打ち込むと以下のように表示される (一部抜粋):


```

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

```

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。

また、`inspect tree compile` と打ち込むと、`compile` は `incCompileSetup` に依存していて、それは `dependencyClasspath` などの他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。

```

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

```

....

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。

他のセッティングに依存したタスクの定義

`scalacOptions` はタスク・キーだ。何らかの値に既に設定されていて、Scala 2.12 以外の場合は `"-Xfatal-warnings"` と `"-deprecation"` を除外したいとする。

```
lazy val root = (project in file(".")).
  settings(
    name := "Hello",
    organization := "com.example",
    scalaVersion := "2.12.1",
    version := "0.1.0-SNAPSHOT",
    scalacOptions := List("-encoding", "utf8", "-Xfatal-warnings", "-deprecation", "-unchecked"),
    scalacOptions := {
      val old = scalacOptions.value
      scalaBinaryVersion.value match {
        case "2.12" => old
        case _       => old filterNot (Set("-Xfatal-warnings", "-deprecation").apply)
      }
    }
  )
```

`sbt` シェルで試すようになるはずだ:

```
> show scalacOptions
[info] * -encoding
[info] * utf8
[info] * -Xfatal-warnings
[info] * -deprecation
[info] * -unchecked
[success] Total time: 0 s, completed Jan 2, 2017 11:44:44 PM
> ++2.11.8
[info] Setting version to 2.11.8
```

```
[info] Reapplying settings...
[info] Set current project to Hello (in build file:/xxx/)
> show scalacOptions
[info] * -encoding
[info] * utf8
[info] * -unchecked
[success] Total time: 0 s, completed Jan 2, 2017 11:44:51 PM
```

次に (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")
```

注意: `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。

```
// 悪い例: checksums セッティングは scalacOptions タスクに関連付けて定義することはできない!
checksums := scalacOptions.value
```

他のセッティングに依存したセッティングの定義

実行のタイミングという観点から見ると、セッティングはロード時に評価される特殊なタスクと考えることができる。

プロジェクトの名前と同じ `organization` を定義してみよう。

```
// プロジェクトの name に基いて organization 名を付ける (どちらも型は SettingKey[String])
organization := name.value
```

実用的な例もみてみる。これは `scalaSource in Compile` というキーを `scalaBinaryVersion` が "2.11" の場合のみ別のディレクトリに再配線する。

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

```

    case _ => old
  }
}

```

そもそも **build.sbt DSL** は何のためにある？

build.sbt DSL は、セッティングやタスクの有向非巡回グラフを構築するためのドメイン特化言語だ。セッティング式はセッティング、タスク、そしてそれらの間の依存性をエンコードする。

この構造は **Make** (1976)、**Ant** (2000)、**Rake** (2003) などにも共通する。

Make 入門

Makefile の基本的な構文は以下のようになる：

```

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

```

対象 (target、デフォルトの target は **all** と呼ばれる) が与えられたとき、

1. **Make** は対象の依存性が既にビルドされたかを調べて、ビルドされていないものをビルドする。
2. **Make** は順番にシステムコマンドを実行する。

Makefile の具体例で説明しよう：

```

CC=g++
CFLAGS=-Wall

all: hello

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

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

```

make を実行すると、デフォルトで **all** という名前の対象を選択する。その対象は **hello** を依存性として列挙するが、それは未だビルドされていないので、**Make** は次に **hello** をビルドする。

次に、**Make** は **hello** という対象の依存性がビルド済みかを調べる。**hello** は **main.o** と **hello.o** という2つの対象を列挙する。これらの対象が最後の

パターンマッチを用いたルールによってビルドされた後でやっと `main.o` と `hello.o` をリンクするシステムコマンドが実行される。

`make` を実行しているだけなら、対象として何がほしいのかだけを考えればよく、中間成果物をビルドするための正確なタイミングやコマンドなどは `Make` がやってくれる。これを依存性指向プログラミングもしくはフローベースプログラミングだと考えることができる。DSL は対象の依存性を記述するが、アクションはシステムコマンドに委譲されるため、正確には `Make` はハイブリッドシステムに分類される。

Rake

このハイブリッド性も実は `Make` の後継である `Ant`、`Rake`、`sbt` といったツールにも受け継がれている。 `Rakefile` の基本的な構文をみてほしい:

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

`Rake` でのブレークスルーは、アクションをシステムコマンドの代わりにプログラミング言語を使って記述したことだ。

ハイブリッド・フローベースプログラミングの利点

ビルドをこのように構成する動機がいくつかある。

第一は非重複化だ。フローベースプログラミングではあるタスクが複数のタスクから依存されていても一度だけしか実行されない。例えば、タスクグラフ上の複数のタスクが `compile in Compile` に依存していたとしても、実際のコンパイルは唯一一回のみ実行される。

第二は並列処理だ。タスクグラフを用いることでタスクエンジンは相互に非依存なタスクを並列にスケジューリングすることができる。

第三は関心事の分離と柔軟さだ。タスクグラフはビルドの作者が複数のタスクを異なる方法で配線することを可能にする。一方、`sbt` やプラグインはコンパイルやライブラリ依存性の管理といった機能を再利用な形で提供できる。

まとめ

ビルド定義のコアなデータ構造は、辺を事前発生 (happens-before) 関係とするタスクの DAG だ。 `build.sbt` は、依存性指向プログラミングもしくはフローベースプログラミングを表現するための DSL で、`Makefile` や `Rakefile` に似ている。

フローベースプログラミングを行う動機は、非重複化、並列処理、とカスタム化の容易さだ。

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、タスク・グラフ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 (サブプロジェクトとも呼ばれる) 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:

- サブプロジェクト
- 依存性コンフィギュレーション
- タスク

サブ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。

依存性コンフィギュレーション軸によるスコープ付け

依存性コンフィギュレーション (dependency configuration) は、ライブラリ依存性のグラフを定義し、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 で使われる代表的な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依存性コンフィグレーションにスコープ付けされているため、依存性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 シェルで `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” に関しては、前のページで解説した。

ここで委譲も見ることができ、もし値が定義されていないならば、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` コンフィギュレーション、`Global` タスクとなる）。

ビルドワイド・セッティング

サブプロジェクト間に共通なセッティングを一度に定義するための上級テクニックとしてセッティングを `ThisBuild` にスコープ付けするという方法がある。

もし特定のサブプロジェクトにスコープ付けされたキーが見つからなかった場合、sbt はフォールバックとして ThisBuild 内を探す。この仕組みを利用して、`version`、`scalaVersion`、`organization` といったよく使われるキーに対してビルドワイドなデフォルトのセッティングを定義することができる。

便宜のため、セッティング式のキーと本文の両方を ThisBuild にスコープ付けする `inThisBuild(...)` という関数が用意されている。セッティング式を渡すと、それに `in ThisBuild` を可能な所に追加したのと同じものが得られる。

```
lazy val root = (project in file("."))
  .settings(
    inThisBuild(List(
      // Same as:
      // organization in ThisBuild := "com.example"
      organization := "com.example",
      scalaVersion := "2.12.1",
      version      := "0.1.0-SNAPSHOT"
    )),
    name := "Hello",
    publish := (),
    publishLocal := ()
  )

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

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

値の追加

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"))
```

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、とタスク・グラフ 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 (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、タスク・グラフで説明したとおり、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わ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も、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、以下のようなになる。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/repository"
```

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、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、一つのビルドで複数のサブ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。

複数のサブプロジェクト

一つのビルドに複数の関連するサブプロジェクトを入れておくと、サブ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 in file("util"))
```

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

`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 `sbt` シェル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名が ID と同じ名前であるときは省略することができる。

```
lazy val util = project
```

```
lazy val core = project
```

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。

```
lazy val commonSettings = Seq(  
  organization := "com.example",  
  version := "0.1.0",  
  scalaVersion := "2.12.1"  
)
```

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

```

settings(
    commonSettings,
    // other settings
)

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

ビルドワイド・セッティング

サブプロジェクト間に共通なセッティングを一度に定義するためのもう一つの方法として、`ThisBuild` にスコープ付けするという少し上級なテクニックがある。（スコープ参照）

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 in file("util"))

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

```

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、好きにやっけていい。

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`、タスク・グラフ、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 とタスク・グラフ 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考え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、[タスク・グラフ][More-About-Settings]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じのものとなる。

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:

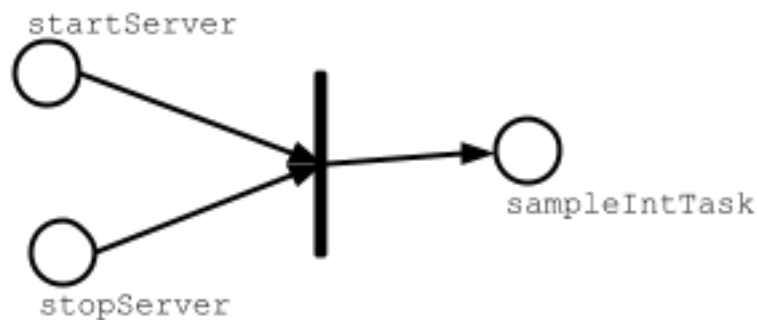


図 2: task-dependency

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、`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:

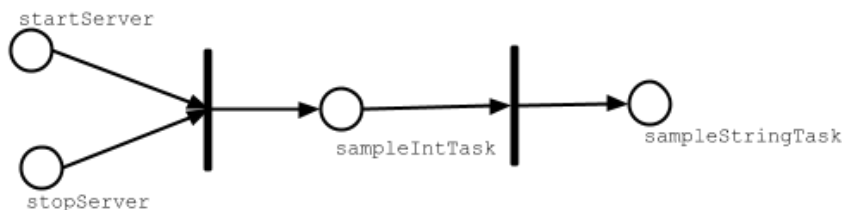


図 3: task-dependency

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

```

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



図 4: task-dependency

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、タスク・グラフ、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/                          # メタビルドのルート・プロジェクトのベ
スディレクトリ

    Dependencies.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

    assembly.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
```

```
MetaDeps.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 `Dependencies.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" % "2.4.17"

  // 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 := "2.12.1"
)

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
- ビルド定義はタスクとタスク間の依存性の大きな DAG 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。
- 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 := "2.12.1"
```

(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 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。
- (プロジェクト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取ると、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いい。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 / simple
[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=0.13.13
```

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=0.13.13
```

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、`compile 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
}
```