Clojure環境の作り方と始め方

Leiningen、REPL、Clojureライブラリ、
Scala、Java、JRuby

アペリティフ（Aperitif）：食前酒のこと

なにはともあれ、まずはClojureを使ってみましょう。Clojureを使うのに、コンピュータの達人である必要はありません。お気に入りのテキストエディタさえあればスタートできます。

道に迷ったときは

この本では、Clojureについてステップバイステップで一から説明はしません。とはいえ、Clojureのインストールやセットアップが必要な場合もあると思います。ここでは、Web上のリソースを挙げておきますので参考にしてください。

- 更新頻度の高いサイト：
  - Learn Clojure（http://learn-clojure.com/clojure_tutorials.html）
- 詳細なClojure入門ページ：
  - A Brief Beginner's Guide To Clojure（http://www.unexpected-vortices.com/clojure/brief-beginners-guide/index.html）
  - Clojure Koans（https://github.com/functional-koans/clojure-koans）
  - ClojureDocs（http://clojuredocs.org/quickref/Clojure% 20Core）

LeiningenでClojureを操る

REPLがありますか？

Clojureのコードを実行する場合、この本では基本的にREPL（Read-Eval-Print-Loop＝「読んで評価して表示して」を繰り返す）を使います。REPLを使うと、入力されたコマンドはインタプリタによって読み込まれ（Read）、評価され（Eval）、結果が表示（Print）されます。そして、再びコマンドの入力待ち状態になります。入力されたコマンドはすべてメモリ中にあるので、過去に実行したコマンドを再利用したり、コマンドの結果を使ったりすることができます。

Clojureを始めるにあたって、この本ではClojureそのものをインストールするのではなく、代わりにLeiningenというツールをインストールします。Leiningenを使うことによって、依存関係や実行環境の差異による問題に煩わされることが少なくなるので、Clojureを始める前にClojureが嫌になってしまう可能性が低くなります。LeiningenのWebサイトは、https://github.com/technomancy/leiningen#installationにあります。

UNIX系環境へのインストール

UNIX系OSを使っている方は、次の方法でLeiningenをインストールしてください。

① https://raw.github.com/technomancy/leiningen/preview/bin/leinから、leinスクリプトをダウンロードする
② ダウンロードしたスクリプトを$PATHの通った場所に保存する
③ スクリプトを実行可能にする（chmod 755 lein）

Windowsへのインストール

Windowsを使っている方は、バッチファイルlein.bat[注1]をダウンロードします。あらかじめwget.exeかcurl.exeをインストールしておき、PATH環境変数にパスを通しておきましょう。あとは、「lein self-install」を実行するだけです。Cygwinを使っているのであれば、バッチファイルではなく、前述のスクリプトファイルを使用できるはずです。

インストールが終わったら、次のコマンドでleinのバージョンを確認しましょう。

```
lein version
```

次のように表示されるはずです。

```
Leiningen 2.0.0-preview10 on Java 1.7.0_10 Java HotSpot(TM) 64-Bit Server VM
```

もし、うまくいかないようであれば、前述のサイトなどを参考に手順を確認してください。

leinのインストールがうまくいったら、早速REPLを使ってみましょう。コマンドラインから「lein repl」と入力します。

```
[niko@Modrzyks-MacBook-Pro-2][11:42][~/projects/mascarpone/chapter01/] % lein repl
nREPL server started on port 54311
REPL-y 0.1.0-beta10
Clojure 1.4.0
    Exit: Control+D or (exit) or (quit)
Commands: (user/help)
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
          (user/sourcery function-name-here)
Javadoc:  (javadoc java-object-or-class-here)
Examples from clojuredocs.org: [clojuredocs or cdoc]
```

注 1　https://raw.github.com/technomancy/leiningen/preview/bin/lein.bat

```
        (user/clojuredocs name-here)
        (user/clojuredocs "ns-here" "name-here")
user=>
```

このように表示されたでしょうか？　おめでとうございます！　これで、Clojureのレシピを試す準備ができました。最初にお話ししたとおり、実際に自分の手を動かしてみることが大切です。

Column REPLで友達とつながる？

ここではLeiningen 2で可能になった、ちょっとしたトリックについてお話しします。起動したREPLに対して、他のユーザがローカル／リモートから接続できます。REPLを次のように起動します。

```
lein repl :headless
```

すると、次のようなメッセージが表示されます。

```
nREPL server started on port 53337
```

ポート番号が表示されたら、他のターミナルからREPLを実行し、そのポートに接続します。

```
lein repl :connect 53337
```

このトリックを使えば、より性能の高いマシンに何かの処理をさせたり、データを持ってきたりというように、クラウド的なことができますね。

※ 開発の流れ

Lispに慣れている人は、たいていREPLで開発を行う際の独自の作業手順を持っています。ここでは、私の見つけた手順を示しますが、ぜひ自分の手順を見つけてください。

① REPLで、必要な変数や計算結果を取得するための入力をする
② あとから使う情報を順々に変数に保存していく
③ 再利用できそうなコードは小さなファンクションあるいはコードブロックにする。それらを利用することで、反復開発が早く簡単になる
④ それらのファンクションを使ってビルドすると、副作用もなく、間違いも少ない
⑤ 他のユーザが使えるように、ファイルやネームスペースはAPIのように作る
⑥ それぞれのファンクションの機能や副作用について、ちょっとしたドキュメントを書く

誰かを頼りにしよう

フォアグラを注文したとたん、イクラのトッピングされたサーモンサラダを追加で頼めばよかったと思ったことはありませんか？　Clojureの世界で人の書いたコードを取得するには、leinが作成するproject.cljというファイルに記述します。そうすれば、あとはleinがやってくれます。

実際の動きを確認するために、「lein new <appname>」というコマンドを入力します。たとえば、次のように入力します。

```
lein new sample00
```

実際の実行例を次に示します。

```
[~/projects/mascarpone/] % lein new sample
Generating a project called sample based on the 'default' template.
To see other templates (app, lein plugin, etc), try `lein help new`.
```

leinがディレクトリとファイルを作成します。

```
.
├── README.md
├── doc
│   └── intro.md
├── project.clj
├── src
│   └── sample
│       └── core.clj
└── test
    └── sample
        └── core_test.clj
5 directories, 5 files
```

マークダウン形式のドキュメントファイル、コードを記述するclojureファイル、そして次のようなproject.cljファイルが作成されます。

```
(defproject sample "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.4.0"]])
```

これがまさにclojureのコードです。見ればなんとなく意味が読み取れて、読者のなかにはApache AntとかApache Mavenを思い出す人もいるのではないでしょうか？　もしそ

うでなくても、今は気にすることありません。この本を読み進めることで、理解できるようになります。；）

さて、コードの中を見ると、「プロジェクト」に対するメタデータが記述されています。メタデータの詳細は、LeiningenのWebサイト[注2]にあります。

ここでは、ひとまずプロジェクトに必要な依存関係を記述しましょう。cheshireというJSONのパーサーライブラリをプロジェクトに追加します。

```
[cheshire "5.0.1"]
```

変更したproject.cljは次のとおりです。

```
(defproject sample "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [
    [org.clojure/clojure "1.4.0"]
    [cheshire "5.0.1"]
    ])
```

ファイルを変更してREPLを実行すると、leinは1回だけ、次のようなメッセージを出力します。

```
Could not find artifact cheshire:cheshire:pom:5.0.1 in central (http://repo1.
maven.org/maven2)
Retrieving cheshire/cheshire/5.0.1/cheshire-5.0.1.pom (3k)
    from https://clojars.org/repo/
Retrieving com/fasterxml/jackson/core/jackson-core/2.1.1/jackson-core-2.1.1.pom
(6k)
    from http://repo1.maven.org/maven2/
    ……
    ……
Retrieving cheshire/cheshire/5.0.1/cheshire-5.0.1.jar (12k)
    from https://clojars.org/repo/
```

leinが自動的に必要なものをダウンロードします。カリフォルニアの片田舎のホテルで、ワインでも飲みながらのんびりとモデム経由でインターネットをしているのでなければ、おそらくあっという間にダウンロードが終了します。

ダウンロードが完了したら、早速試してみましょう。JSONのコードを生成してみます。

```
(use 'cheshire.core)
```

注２　https://github.com/technomancy/leiningen/blob/master/sample.project.clj

_doc_を使ってドキュメントを取得できます。

```
user=> (doc generate-string)
cheshire.core/generate-string
([obj] [obj opt-map])
  Returns a JSON-encoding String for the given Clojure object. Takes an
  optional date format string that Date objects will be encoded with.
  The default date format (in UTC) is: yyyy-MM-dd'T'HH:mm:ss'Z'
nil
```

まず、通常のClojureシーケンスをJSONに変換します。

```
user=> (generate-string '(1 2 3))
"[1,2,3]"
user=> (generate-string '{:name (1 2 3)})
"{¥"name¥":[1,2,3]}"
```

そして、逆にJSON文字列をパースして、それをClojureシーケンスに置き換えることもできます。

```
user=> (parse-string "{¥"foo¥":¥"bar¥"}")
{"foo" "bar"}
```

いろいろな可能性を感じますね。とりあえず、南フランスのシャルドネ2010で乾杯しますか。

Javaライブラリ、そしてibiblio

もしかしたら、「cheshireの依存関係はどうなっているんだろう？　」と思うかもしれませんね。Javaの世界では、良いか悪いかは別にして、Mavenは必要なライブラリをibiblioと呼ばれるところにあるJavaのアーカイブ[注3]から取得します。

純粋なJavaのライブラリであれば、たいていの場合、Clojureでも同じようにibiblioからダウンロードされます。たとえば、有名なApacheのライブラリにcommons-ioというものがあります。それに対応するpom[注4]と呼ばれるファイルがあります。

pomの中身は大量のXMLですが、その中に次の記述があります。

```
<groupId>commons-io</groupId>
<artifactId>commons-io</artifactId>
<version>2.4</version>
```

注３　http://mirrors.ibiblio.org/maven2/
注４　http://mirrors.ibiblio.org/maven2/commons-io/commons-io/2.4/commons-io-2.4.pom

これをClojureの世界の言葉に置き換えると

```
[commons-io/commons-io "2.4"]
```

ということで、次の記述でJavaの世界とつなげることができます。

```
user=> (org.apache.commons.io.FileUtils/readLines (java.io.File. "README.md")
"UTF-8")
#<ArrayList [# sample, , A Clojure library designed to ... well, that part is
up to you., , ## Usage, , FIXME, , ## License, , Copyright c 2013 FIXME, ,
Distributed under the Eclipse Public License, the same as Clojure.]>
```

この本では、これからもJavaとのいろいろな連携が出てきますが、興味があれば次のサイトを参照するとよいでしょう。

http://clojure.org/java_interop

ClojarsとClojureライブラリ

Clojureに特化したライブラリは、Clojars[注5]で見つけることができます。前のセクションで使ったCheshireをClojarsで見つけるには、次のようにアクセスします。

```
https://clojars.org/search?q=cheshire
```

すると、**図1-1**のように表示されます。

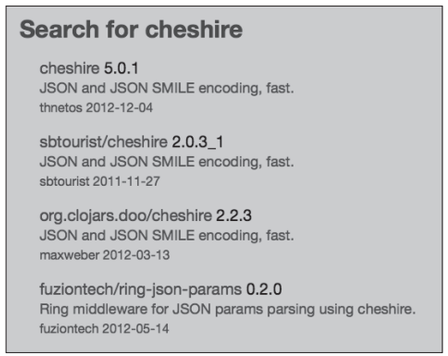

図1-1　CheshireをClojarsで見つけるには

注５　https://clojars.org/

Clojureを使う人のほとんどはClojarsを利用しているので、そこには最新のコードが存在するはずです。

以上、何か材料が必要になったときに探すべき2つの場所を紹介しました。これで、素晴らしい献立を思いついても材料がなくて断念……ということが少なくなりそうですね。

では、ワインをもう一杯？

Leiningen で Clojars を探す

Leiningen でClojarsのライブラリを探すことができますが、少し説明が必要です。次のコマンドは正規表現を使っていて、見た目よりも動作が少し複雑です。

```
lein search "postal"
```

実際にこのコマンドを実行すると、次のような表示が出力されます。

```
Searching over Artifact ID……
   == Showing page 1 / 1
  [paddleguru/postal "1.7-SNAPSHOT"]
  [org.jmatt/postal "1.6-SNAPSHOT"]
  [org.clojars.sethtrain/postal "0.2.0"] Clojure email support
  [org.clojars.sethtrain/postal "0.1.0-SNAPSHOT"] Clojure email support
  [org.clojars.maxweber/postal "1.6-SNAPSHOT"]
  [org.clojars.doo/postal "1.8-SNAPSHOT"]
  [org.clojars.danlarkin/postal "1.3.2-SNAPSHOT"]
  [org.clojars.btw0/postal "1.4.0-SNAPSHOT"]
  [org.clojars.blucas/postal "1.8-SNAPSHOT"]
  [com.draines/postal "1.7.1"]
  ……
  [com.draines/postal "1.9.1"]
  [com.draines/postal "1.9.2"]
```

インターネットの接続状態が悪いと、ローカルの検索インデックスが壊れてしまうことがあります。その場合は、次のフォルダを削除します。

```
~/.lein/indices
```

そうすることで、次回のlein search実行時に検索インデックスが再構築されます。

パンとバター ——Leiningenのプラグインをインストールする

Leiningenにはプラグインの新しい定義方法があります。数年前、プラグインはプロジェクトごとにインストールが必要でした。なので、新しいプロジェクトを始めるたびに必要なプラグインをインストールし、別のプロジェクトを作ったらまた同じプラグインをインストールして……ということを繰り返す必要がありました。ま、ワインを飲みながらであれば、あまり時間も気にならないかもしれませんが……。

今は、次のファイルにClojure流の定義を書けるようになりました。

```
~/.lein/profiles.clj
```

Clojars自体がホストしている、シンプルですがとても有用なlein-pprintというプラグインがあります。このプラグインを使用する場合、前述のファイルには次のように記述します。

```
{:user {:plugins [[lein-pprint "1.1.1"]]}}
```

すると、leinはプラグインを見つけ、ダウンロードして使えるようにします。lein helpを実行すると、次のように表示されます。

```
Several tasks are available:
check               Check syntax and warn on reflection.
......
pprint              Pretty-print a representation of the project map.
repl                Start a repl session either
test                Run the project's tests.
......
```

そして、次のコマンドによって、すべてのプロジェクトからプラグインが使えるようになります。

```
lein pprint
```

すべての関連するメタデータをClojureプロジェクトに出力します。

作ったコードをClojarsで共有する

Leiningen 2はClojarsと連携するのに、もうプラグインを必要としません。


```
lein deploy clojars
```

これだけで、裏ではあなたのコードはアーカイブされ、_jar_ファイルはメタデータと共にClojarsに転送されます。そして、他の開発者は我々が今やったことと同じ手順であなたのコードを使えるようになるのです。

デプロイの状況を見るには

```
lein help deploying
```

と入力します。

依存関係を見る

依存関係を確認するには、次のコマンドを実行します。

```
lein deps :tree
```

ここまでに説明した作業をやっているのであれば、次のように表示されます。

```
[cheshire "5.0.1"]
  [com.fasterxml.jackson.core/jackson-core "2.1.1"]
  [com.fasterxml.jackson.dataformat/jackson-dataformat-smile "2.1.1"]
[commons-io "2.4"]
[org.clojure/clojure "1.4.0"]
```

Eclipseで一仕事

Clojureに最適なIDEは、おそらくEclipseでしょう。Eclipseは次のサイトからダウンロードできます。

http://www.eclipse.org/downloads/index-developer.php

Counterclockwise[注6]はClojureの開発にとても適したEclipseのプラグインで、頻繁にアップデートされています。Counterclockwiseのインストールは、**図1-2**のメニューから行います。

注 6　http://code.google.com/p/counterclockwise/

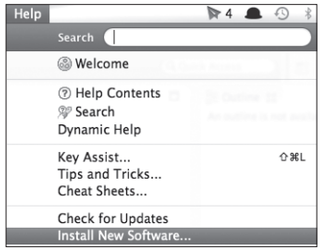

図1-2　Counterclockwiseのインストール

ここから、次のアップデートサイトのURLを追加します（**図1-3**）。

http://ccw.cgrand.net/updatesite/

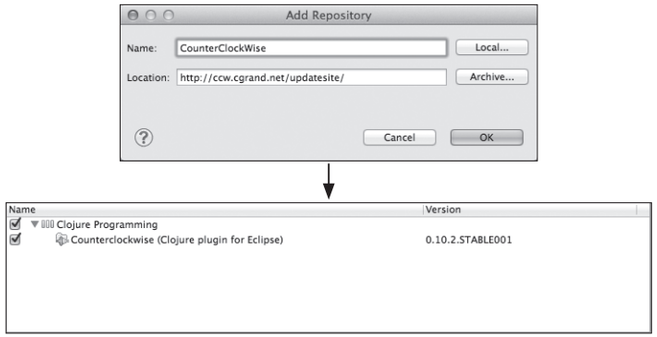

図1-3　アップデートサイトのURLの追加

これでプラグインがインストールされ、生成したプロジェクトをインポートできるようになりましたので、通常のJavaプロジェクトとしてインポートします（**図1-4**）。

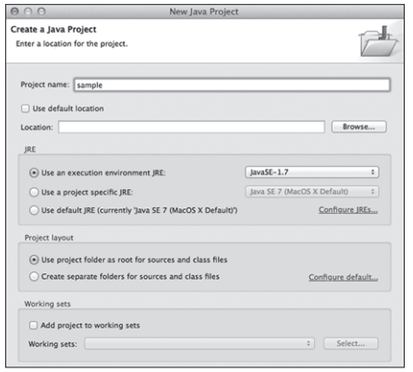

図1-4　Javaプロジェクトとしてインポート

続いて、Leiningenを使用するように設定を変更します（**図1-5**）。

図1-5　Leiningenを使用するための設定

これで**図1-6**のように、Eclipseから直接REPLを開始できるようになりました！

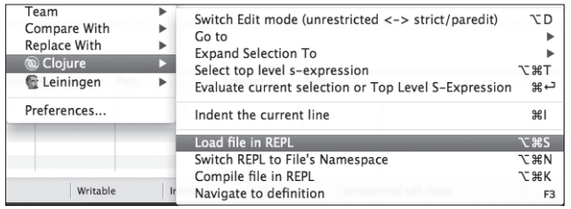

図1-6　Eclipseから直接REPLを開始できる

Eclipseを使う一番の理由は、やはりその素晴らしい補完機能でしょうか（**図1-7**）。早速試してみましょう。

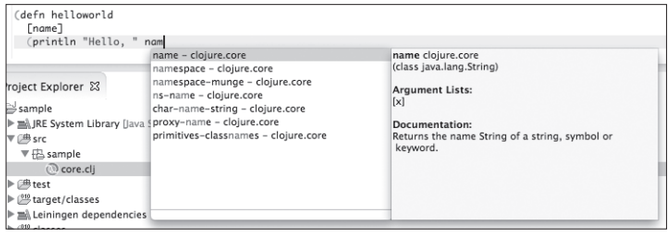

図 1-7　Eclipseの補完機能

新しいコードを入力します（**図 1-8**）。

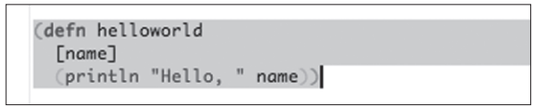

図 1-8　新しいコードを入力する

そして［Command］+［Enter］キーを押すと、REPLから直接補完されます（**図 1-9**）。

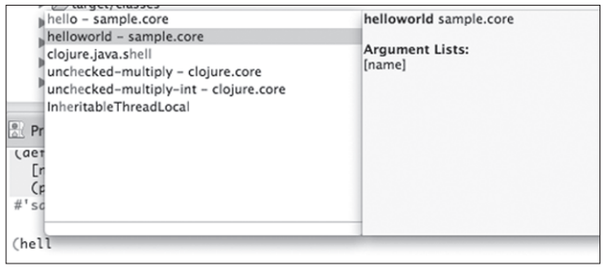

図 1-9　補完されたコード

ちょー簡単ですね。

セッションにライブラリをダイナミックに追加する

Eclipseからコマンドラインに話を戻しましょう。すでにお気づきかもしれませんが、Clojureで何かを変更した場合、毎回REPLを起動し直さないと新しい依存関係が反映されません。Chas Emerickは、こりゃ面倒だと思い、pomegranateという魔法のライブラリにすべてを取り込むようにして、実行時に依存関係を追加できるようにしました。

プロジェクトでずっと利用するライブラリではなく、開発時にちょっと試すようなライブラリを追加してみましょう。まずは、pomegranateを`project.clj`に定義します。

```
:profiles {:dev {:dependencies [
    [com.cemerick/pomegranate "0.0.13"]]}}
```

REPLを起動します。

```
lein repl
```

必要な新しいネームスペースをインポートします。

```
(use '[cemerick.pomegranate :only (add-dependencies)])
```

xyzzyというXMLライブラリをインポートしてみます。

```
(add-dependencies
   :coordinates '[[antler/xyzzy "0.1.0"]]
   :repositories {"clojars" "http://clojars.org/repo"})
```

classpathに追加された依存関係が出力されます。

```
{[org.clojure/data.zip "0.1.0"] nil, [org.clojure/clojure "1.3.0"] nil, [antler/
xyzzy "0.1.0"] #{[org.clojure/clojure "1.3.0"] [org.clojure/data.zip "0.1.0"]}}
```

これで、新たにダウンロードされたライブラリを呼び出すことができます。

```
(require '[xyzzy.core :as xyzzy])
(xyzzy/parse-xml "<html><body><h1>It works</h1></body></html>")
```

ライブラリでパースした結果が表示されます。

```
[{:tag :html, :attrs nil, :content [{:tag :body, :attrs nil, :content [{:tag :h1,
:attrs nil, :content ["It works"]}]}]} nil]
```

もちろん、試して気に入ったら`project.clj`に定義します。

内緒でScalaのコードを走らせる

もし、leinプロジェクトのソースファイルを統合したいのであれば、scalac-plugin[注7]を使うことができます。プラグインのインストールはとても簡単です。

注 7　https://github.com/technomancy/lein-scalac

① [lein-scalac "0.1.0"]をproject.cljの :pluginsに追加する
② project.cljの :scala-source-pathに .scalaソースファイルのある場所（通常、「src/scala」）を設定する
③ .classファイルにコンパイルする

```
lein scalac
```

④ もし、scalacを自動的に実行するのであれば、:prep-tasks ["scalac"]をproject.cljに追加する
⑤ たいていの場合、scala-libraryの設定も必要

```
:dependencies [org.scala-lang/scala-library "2.9.1"]
```

では、実際にscalaのコードを書いてみましょう。

```
// HelloWorld.scala

class HelloWorld {
  def sayHelloToClojure(msg: String) =
    "Here's the second echo message from Scala: " concat msg
}
```

これをClojureから呼び出すには次のようにします。

```
(import HelloWorld)
(.sayHelloToClojure (HelloWorld.) "Hi there")
```

すると、次のように出力されます。

```
sample.core=> (import HelloWorld)
HelloWorld
sample.core=> (.sayHelloToClojure (HelloWorld.) "Hi there")
"Here's the second echo message from Scala: Hi there"
```

お友達にJVM上で別の言語が使えるようになったと自慢しましょう。：）
それか、ウェイトレスさんに別のワインをお願いしましょう！

Javaのコードを走らせるっていうのは、ここだけの話

前のセクションの内容は、昔のスタイルのJavaでも使えます。まず、Javaのクラスを用意します。


```
public class HelloWorldJava {

    public HelloWorldJava() {

    }

    public String sayHello(String who) {
         return "This is a warm welcome from old java to " + who;
    }

}
```

project.cljを設定します。

```
; javaのクラスへのパス
:java-source-paths ["src/java"]
; javacのコンパイルオプション
:javac-opts ["-target" "1.6" "-source" "1.6" "-Xlint:-options"]
```

もう1行、次の修正をします。

```
:prep-tasks ["scalac" "javac"]
```

これで、scalacに加えてjavacが実行されるようになります。REPLを再起動して、次のClojureコードを入力します。

```
sample.core=> (.sayHello (HelloWorldJava.) "World")
"This is a warm welcome from old Java to World"
```

Clojureのメソッドをhookeでラップする

前のセクションで紹介したjavacプラグインのサンプルを探しているときに、偶然ですが関数をフックする素晴らしい方法を見つけました。

project.cljファイルに依存関係を記述しましょう。

```
[robert/hooke "1.3.0"]
```

そして、サンプルを動かしてみましょう（**リスト1-1**）。

リスト1-1　hookeでラップするためのサンプル

```
1 (use 'robert.hooke)
2
3 (defn examine [x]
4   (println x))
5
```

```
(defn microscope
  "The keen powers of observation enabled by Robert Hooke allow
  for a closer look at any object!"
  [f x]
  (f (.toUpperCase x)))

(defn doubler [f & args]
  (apply f args)
  (apply f args))

(defn telescope [f x]
  (f (apply str (interpose " " x))))

(add-hook #'examine #'microscope)
(add-hook #'examine #'doubler)
(add-hook #'examine #'telescope)

(examine "something")
; > S O M E T H I N G
; > S O M E T H I N G
```

これで、手軽に関数を拡張できるようになりました！　しかも美しく……？　：）

もう1つのRuby——JRuby

あるコードを動かしたい。それはRubyで書かれている。でも、わざわざRubyをセットアップするのは面倒だし……。そんなときは、Leiningenにお願いしましょう。次の設定をproject.cljファイルの:pluginsセクションに追加します。

```
[lein-jruby "0.1.0"]
```

あとは、次のようにスクリプトを実行します。

```
lein jruby -S src/ruby/fibonacci.rb 1000
```

fibonacci.rb（**リスト1-2**）は、Rubyでフィボナッチ数列を計算します。

リスト1-2　fibonacci.rb

```
require 'matrix'

FIB_MATRIX = Matrix[[1,1],[1,0]]
def fib(n)
  (FIB_MATRIX**(n-1))[0,0]
end

if(ARGV[0])
  puts fib(ARGV[0].to_i)
else
  puts "Needs an integer to compute fibonacci result"
end
```

繰り返しますが、このコードを実行するのにRubyをインストールする必要はありません。最初のサンプルは依存関係のないものでしたが、もしJRubyにRuby gemをインストールさせる必要がある場合にはこのようにします。

```
lein jruby -S gem install json-jruby
```

続いて、**リスト1-3**のサンプルを

リスト1-3　json.rb

```
#! /usr/bin/env jruby
require "rubygems"
require "json"

# de-serializing:
source_string = '{"sample": "Hello, world!"}'
puts JSON(source_string).inspect
# => {"sample"=>"Hello, world!"}

# serializing:
source_object = ["Just another Ruby Array", {"null value" => nil}]
puts JSON(source_object)
# => ["Just another Ruby Array",{"null value":null}]
```

次のコマンドで実行します。

```
lein jruby -S src/ruby/json.rb
```

ルビー色のワインで乾杯！　でも、この本を読むのを忘れない程度に……。

おいしいプラグインのスープLeiningen仕立て

Groovy、Hadoop、……。Leiningenには、本当にいろいろなプラグインが存在します。ぜひLeiningenのプラグイン一覧をチェックしてみてください。

https://github.com/technomancy/leiningen/wiki/Plugins

私がよく使うプラグインを紹介します（**表1-1**）。

表 1-1　Leiningenの主なプラグイン

Leiningenのプラグイン	機能	備考
lein-midje	テストを実行	インストール必須！　この本でも取り上げている
lein-noir	Webアプリを簡単に作成	この本で取り上げている
lein-deps	プロジェクトの依存関係を表示	lein deps :tree
lein-git-version	Gitタグを使ってversion.cljファイルにバージョンを保存	
lein-webrepl	REPLをWebサーバとして開始。ブラウザからコード編集可能	**図 1-10** を参照

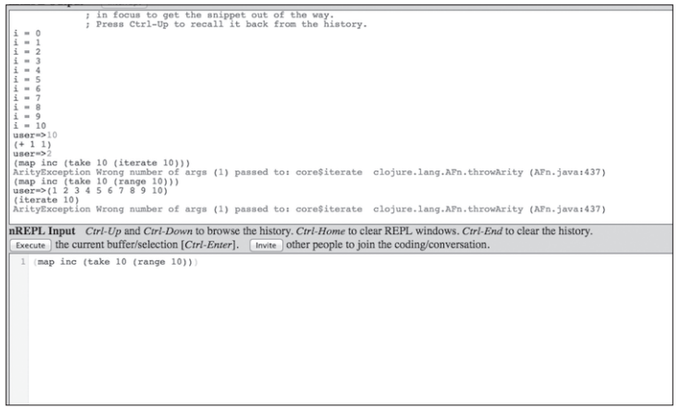

図 1-10　lein-webrepl

まずはプラグイン一覧を眺めて、どんなものがあって何ができそうかを検討してください。leinにはたくさんの素材がありますから、料理の幅がきっと広がると思います。

Leiningen用のプラグインを書いてみる

さて、ついに自分達のプラグインを作ってコミュニティに貢献するときがやってきました！　まずはオンラインからデータを取得してそれを表示する簡単なプラグインを書いてみましょう。

次の記述をproject.cljファイルに追加して、現在のプロジェクトから直接プラグインを書けるようにします。

```
:eval-in-leiningen true
```


タスクの名前はstockとして、次のように呼ぶようにしましょう。

```
lein stock YHOO
```

最後の引数は株のシンボル名です。作成したソースは、stock.clj_というファイル名で_src/leiningen_に置き、ネームスペースは_leiningen.stockとします。ソース中のメソッド名はstockです。

ソース全体は**リスト 1-4**のとおりです。

リスト 1-4　stock.clj_

```
(ns leiningen.stock
    ; to parse xml
    (:use clojure.xml)
    ; to stream the web content
    (:use clojure.java.io))

; base service url
(def base "http://www.google.com/ig/api?stock=")

; method to retrieve the quote from the web
(defn get-quote [key]
    (->
        (get (-> (parse (input-stream (str base key))) :content first :content) 10)
        :attrs
        :data))

; method called from leiningen
(defn stock [project & args]
    (println (get-quote (first args))))
```

実際に実行すると、**表 1-2**のようになります。

表 1-2　stock.clj_の実行結果

Company name	Command line	Result
Apple	lein stock AAPL	520.00
Microsoft	lein stock MSFT	26.83
Yahoo	lein stock YHOO	19.29
Google	lein stock GOOG	520.30

Jark でJVMリロードの待ち時間をなくす

Jarkは、JVM上でリモートのClojureコードをインタラクティブに実行します。もう気がついていると思いますが、Clojureのプログラムを起動すると、毎回JVMがスタートするまでに時間がかかりますよね？　特に開発時はこの待ち時間が馬鹿にならないので

REPLを使うわけですが、それでもとにかく待ち時間をなくしたいことがありますよね。そんなときはJarkの出番です。

まず、Jarkを入手しましょう。

http://icylisper.github.com/jark/

Jarkをダウンロードしてパスの通ったところに置いたら、次のコマンドを実行します。

```
jark server install
```

次にJarkサーバを起動します。

```
jark server start
```

あとは、Jarkクライアントを使ってコードを実行します。

```
jark -e "(+ 2 2)"
echo "(+ 2 2)" | jark -e
jark -e < file.clj
cat file.clj | jark -e
```

コードはJarkサーバに送られ、実行されます。

```
(ns factorial)

(defn compute [n]
   (apply * (take n (iterate inc 1))))

(println "Factorial of 10 :")
(compute 10)
```

また、JarkにはREPLもあります。REPLを使うには次のコマンドを実行します。

```
jark repl
```

私のお気に入りは、次のようなコードで自分のプロジェクトにJarkサーバを埋め込むことができるんです！

```
;At time of writing
; Add [jark "0.4.3-clojure-1.5.0-alpha5" :exclusions [org.clojure]] to project.clj
(require 'clojure.tools.jark.server)
(clojure.tools.jark.server/start PORT)
```

とにかくこれで待ち時間がなくなって、その分ワインを飲む時間が増えるはず。

clojure-contrib って知ってる？ ヤバいよ、それ

かつてClojureが始まってまだ間もない頃は、clojure-core、そしてclojure-contribがありました。Clojureのコアは本当に最小限のコードから構成されていて、clojure-contribにはそれ以外――でも限りなくコアに近いものが含まれていました。しかし、あらゆるプロジェクトがContribライブラリを使ったため、ライブラリはとても大きなものになってしまいました。そのため、コードそのものよりもパッケージが問題となってしまいました。

Clojureのドキュメントには今もなぜかContribがあります。

http://dev.clojure.org/display/doc/Clojure+Contrib

しかし、人々は徐々にそこから離れ、通常の依存関係を使った個別の小さなプロジェクトを使うようになりました。

シェフが手持ちの材料をテーブルの上に広げるように、すべてのものを見渡すことができれば何を使うか決めることができます。いつかはおいしい料理に巡り会うでしょうけれど、それにはたくさんのおいしくないものを食べなくてはいけないかもしれないし、大量のお皿を洗うことになるかもしれないし、いずれにしても時間がかかるでしょう。

現状でモジュール化されているプロジェクトには次のものがあります。

- algo.generic――以前のclojure.contrib.generic
- algo.monads――以前のclojure.contrib.monads
- build.poms――新しいContribライブラリ用のサンプルpom.xml
- core.cache
- core.contracts
- core.incubator――clojure.contrib.strintと共にclojure.contrib.defの一部もここに移った
- core.logic
- core.match
- core.memoize
- core.unify
- data.codec――以前のclojure.contrib.base64
- data.csv
- data.finger-tree
- data.generators――test.generativeからデータジェネレータを取り出したもの
- data.json――以前のclojure.contrib.json

- data.priority-map——以前の clojure.contrib.priority-map
- data.xml——以前の clojure.contrib.lazy-xml
- data.zip——以前の clojure.contrib.zip-filter
- java.classpath——以前の clojure.contrib.classpath
- java.data
- java.jdbc——以前の clojure.contrib.sql
- java.jmx——以前の clojure.contrib.jmx
- math.combinatorics——以前の clojure.contrib.combinatorics
- math.numeric-tower——以前の clojure.contrib.math
- test.benchmark
- test.generative——上記のデータジェネレータ用のテストライブラリ
- tools.cli——clojure.contrib.command-line の置き換え
- tools.logging——以前の clojure.contrib.logging
- tools.macro——以前の clojure.contrib.macro-utils、clojure.contrib.macros
- tools.namespace——以前の clojure.contrib.find-namespaces
- tools.nrepl
- tools.trace——以前の clojure.contrib.trace

これらの各プロジェクトを一通り見てみることをお勧めします。かなりのものが他の人によってすでに作られていることがわかると思います。何があるのかがざっとわかれば、一からコードを書かずに済むはずです。

サンプルとまとめ

以上が最初の章です。ここで説明した内容は次のとおりです。

- Leiningen でプロジェクトを始める
- 依存関係を管理する
- REPL を開始する
- コードを書く——テキストエディタ、REPL、eclipse、Web REPL
- 他の言語と Clojure を使う
- Java を再起動しない方法

次の章ではいよいよ、いろいろな場面で使えるコードを学んでいきます。
でもその前に、もう一杯？