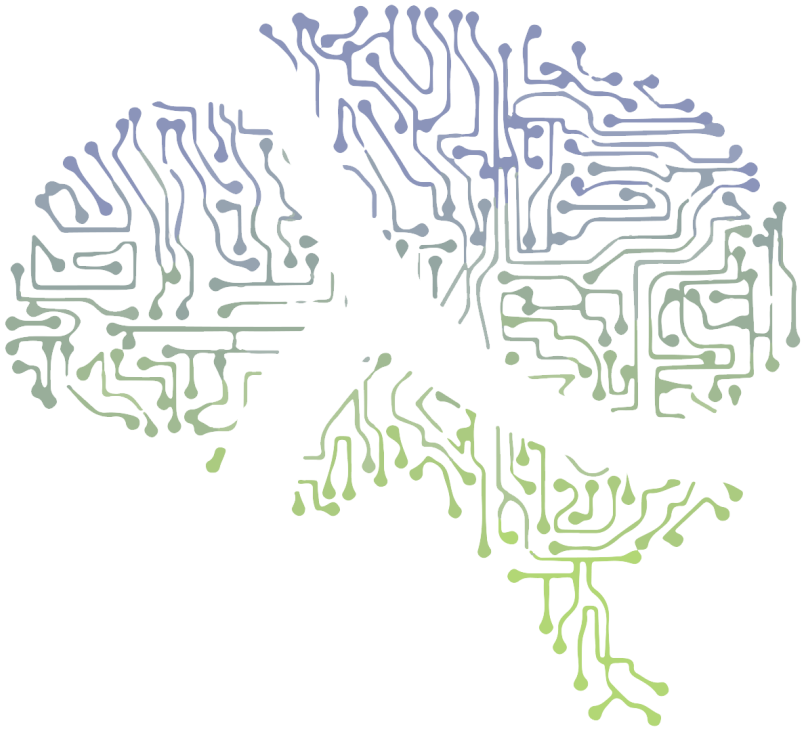

明解 ClojureScript

明解 ClojureScript

Andrey Antukh and Alejandro Gomez　著
Hiroki Noguchi　訳

2021-08-04 版　　発行

目次

第 1 章　本書について　7

第 2 章　はじめに　8

第 3 章　ClojureScript 入門　10

3.1　Lisp 構文 の第一歩 . . . 10
3.2　基本的なデータ型 . . . 11
数値 . . . 11
キーワード . . . 12
シンボル . . . 13
文字列 . . . 13
文字 . . . 13
コレクション . . . 14
3.3　vars . . . 16
3.4　関数 . . . 16
はじめの一歩 . . . 16
自作関数の定義 . . . 17
多様な Arity をもつ関数 . . . 18
可変長引数関数 . . . 18
無名関数の短い構文 . . . 19
3.5　フロー制御 . . . 19
if による条件分岐 . . . 19
cond による条件分岐 . . . 20
case による条件分岐 . . . 21
3.6　真偽判定 . . . 21
3.7　ローカル、ブロック、ループ . . . 22
ローカル . . . 22
ブロック . . . 22
ループ . . . 23
3.8　コレクションの型 . . . 28
イミュータブルと永続性 . . . 28
シーケンスの抽象化 . . . 29

コレクション詳細 35
3.9 Destructuring 43
3.10 スレッディングマクロ (Threading Macro) 47
thread-first マクロ (`->`) 47
thread-last マクロ (`->>`) 47
thread-as マクロ (`as->`) 48
thread-some マクロ (`some->` と `some->>`) 48
thread-cond マクロ (`cond->` と `cond->>`) 49
追加資料 49
3.11 リーダーコンディショナル (Reader Conditionals) 50
Standard (`#?`) 50
Splicing (`#?@`) 50
参考文献 51
3.12 名前空間 52
名前空間の定義 52
他の名前空間の読み込み 52
名前空間とファイル名 54
3.13 抽象化とポリモーフィズム 54
プロトコル 54
既存の型の拡張 55
ClojureScript での抽象化 57
プロトコルを用いた内部分析 58
マルチメソッド 58
ヒエラルキー 59
3.14 データ型 62
Deftype 62
Defrecord 63
プロトコルの実装 65
Reify 65
Specify 66
3.15 ホスト環境の相互運用性 67
型 67
プラットホームの型との連携 67
プラットホームへのアクセス 67
新たにインスタンスを生成する 68
インスタンスのメソッド呼び出し 68
オブジェクトのプロパティへのアクセス 69
プロパティへアクセスするための略記法 69
JavaScript のオブジェクト 69
変換 70

配列 . . . 71
3.16 状態管理 . . . 72
var . . . 73
アトム . . . 73
監視 . . . 74
Volatile . . . 74

第 4 章 ツールとコンパイラ 75

4.1 コンパイラの概要 . . . 75
実行環境 . . . 75
コンパイラのダウンロード . . . 75
Node.js へのコンパイル . . . 76
ブラウザへのコンパイル . . . 78
監視プロセス . . . 80
最適化レベル . . . 81
4.2 REPL での作業 . . . 82
導入 . . . 82
Nashorn REPL . . . 82
Node.js REPL . . . 83
Browser REPL . . . 84
4.3 Google Closure Library . . . 86
4.4 依存関係の管理 . . . 86
Leiningen のインストール . . . 87
初めてのプロジェクト . . . 87
依存関係の管理 . . . 90
外部の依存関係 . . . 90
4.5 Unit テスト . . . 94
初めの一歩 . . . 94
非同期のテスト . . . 97
4.6 CI との連携 . . . 97

第 5 章 ClojureScript 発展編 99

5.1 Transducer . . . 99
データの変換 . . . 99
プロセス変換への一般化 . . . 100
ClojureScript コアにおける Transducer . . . 104
初期化 (Initialisation) . . . 105
ステートフルな transducer . . . 106
Eduction . . . 110
ClojureScript コアにおける他の transducer . . . 110
transducer を定義する . . . 110

トランスデゥシブルなプロセス 111
5.2 一時性 115
5.3 メタデータ 118
Vars 118
値 119
メタデータのための構文 121
メタデータを扱う関数 122
5.4 中心的なプロトコル 124
関数 124
Printing 124
シーケンス 125
コレクション 126
連想性 128
比較 131
メタデータ 132
JavaScript との相互運用 133
リダクション 135
遅延計算 137
状態 138
変更 142
5.5 CSP (core.async を用いた場合) 144
チャンネル 144
プロセス 151
コンビネーター 156
高度な抽象化 161

第 1 章

本書について

本書はプログラミング言語 ClojureScript を扱います。ClojureScript での開発ツールの手引書として利用できます。ClojureScript を用いた日常的な開発で役に立つ内容を紹介します。

本書はプログラミングの入門書ではありません。少なくとも 1 つ以上プログラミング言語の経験がある読者を想定しています。ただし、Clojure や関数型言語の経験は想定していません。誰にも馴染みのない ClojureScript の理論的な基盤について言及するときは参考文献を紹介します。

ClojureScript の公式ドキュメントは良質ですが少ないので、私たちは ClojureScript 入門者に役立つ参考情報と広範な例、そして一連の実用的な手法をまとめた全集を書こうと思いました。本書は ClojureScript 言語と共に進化します。言語機能の参考文献として、また実用的なプログラミング例を含む手引き書として利用できます。

次の条件を満たしていれば、この本から多く学べるでしょう。

- ClojureScript か関数型プログラミングに関心があり、経験が少しある。

- JavaScript または JavaScript にコンパイルされる言語を書いていて、ClojureScript が提供する機能について知りたい。

- Clojure の経験があり、ClojureScript と Clojure の違いを学びたい。また、同じコードから 2 つの言語をターゲットにする実践的な方法を学びたい。

どの条件にも合わなかったとしても大丈夫です。読み進められるようにサポートします。本書が読みやすいものになるように、フィードバックも受けつけています。私たちは ClojureScript が入門者の方々が学びやすいものにすること、また Clojure が進めてきたプログラミング手法を普及させることを目標にしています。私たちはこの目標に大きな価値があると信じています。

翻訳版としては、ウクライナ語版[*1]と日本語版があります。

[*1] https://lambdabooks.github.io/clojurescript-unraveled

第 2 章

はじめに

> なぜ ClojureScript をしてるかだって？　Clojure がロックで JavaScript が流行っているからさ。
> - Rich Hickey

ClojureScript は JavaScript をターゲットにした Clojure の実装です。JavaScript をターゲットにすることにより、ClojureScript で書かれたプログラムは、Web ブラウザや Node.js、また io.js や Nashorn など、様々な実行環境で動作します。

他の JavaScript へコンパイルされる言語（TypeScript、FunScript、CoffeeScript 等）とは異なり、ClojureScript は JavaScript をバイトコードのように使います。ClojureScript は関数型プログラミングを受け入れており、デフォルトでとても安全で一貫性があります。そのセマンティクスは、JavaScript と全く異なります。

もう一つの大きな違いは (私たちの見解では長所ですが)、Clojure がゲスト言語になるように設計されていることです。Clojure はそれ自体に仮想マシンを持たない言語であり、実行環境の違いに応じて順応させることができます。これにより、Clojure（ClojureScript）はホスト言語で書かれた全ての既存ライブラリにアクセスすることができます。

詳細に入る前に、ClojureScript が提供する素晴らしいアイデアのうち、中心的なアイデアをまとめます。全てを理解できなかったとしても大丈夫です。本書を読み進めるにつれて、各々の内容が明確になっていきます。

- ClojureScript では、デザインの決定とイディオムにおいて、関数型プログラミングのパラダイムに従う必要があります。ClojureScript は関数型プログラミングにこだわりますが、理論的な純粋さを追求するような言語ではなく、実用的な言語です。

- ClojureScript では、不変なデータでプログラムを書くことが推奨されます。この不変なデータはパフォーマンスが高く、最新の手法で実装されています。

- ClojureScript では、identity と state を明確に区別します。変更を不変な値の一連の流れとして管理するために、明示的に状態を生成します。

- ClojureScript には、値に基づくポリモーフィズムと、型に基づくポリモーフィズムがあり、Expression Problem をシンプルに解決することができます。

- ClojureScript は Lisp 方言なので、プログラムはプログラミング言語自体のデータ構造で書くことができます。同図像性として知られる性質により、メタプログラミング (つまりプログラムを書くプログラム) を可能な限りシンプルに行うことができます。

以上の概念を合わせることにより、たとえ ClojureScript を使わないとしても、ソフトウェアの実装方法は大きな影響を受けるでしょう。関数型プログラミング、(不変な) データと値の変更の分離、変更を管理する明示的なイディオム、抽象化へのプログラミングのためのポリモーフィズムの構成により、私たちが書くシステムをよりシンプルにすることができます。

> シンプルなプログラミング言語、ツール、テクニック、アプローチを使うことで、日々開発しているソフトウェアと全く同じものを作ることができるのだ。
> - Rich Hickey

私たちが ClojureScript から感じた楽しみやインスピレーションをこの本から受けていただければ幸いです。

第 3 章

ClojureScript 入門

本章では、ClojureScript の概要を見ていきます。Clojure の知識は想定していません。4 章以降を理解するために必要なことを簡潔に説明します。オンラインの REPL[*1] でコードの動作を確認することができます。

3.1 Lisp 構文 の第一歩

Lisp は 1958 年に John McCarthy 氏 によって発明された最も古いプログラミング言語の 1 つです。現在も様々な分野で利用されており、Lisp 方言と呼ばれる多くの派生が進化してきました。CloureScript も Lisp 方言の 1 つです。元々 Lisp ではリストは丸括弧で囲われますが、Clojure(Script) では独自のデータ構造が実装されており、読み書きが容易です。

ClojureScript では、リストの 1 番目に関数がある場合、関数を呼び出すために使われます。次の例では加算する関数を 3 つの引数に適用させています。他の言語とは異なり、+ は演算子ではなく関数であることに注意してください。Lisp には演算子はなく、関数しかありません。

```
(+ 1 2 3)
;; => 6
```

上の例では、加算するための関数である + を 1, 2, 3 に適用しています。ClojureScript では ? や - のような文字をシンボルの名前として利用できます。このことにより、ClojureScript で書かれたプログラムは読みやすくなっています。

```
(zero? 0)
;; => true
```

関数の呼び出しとデータとを区別するために、リストの前にクオート ’ をつけます。クオートがつけられたリストは、関数の呼び出しではなく、データとして処理されます。

[*1] https://www.ClojureScript.io

```
'(+ 1 2 3)  ;; => (+ 1 2 3)
```

ClojureScript ではリスト以外の構文も使います。詳細な説明は後の章で行いますが、ベクタの使い方を例として取り上げます。ベクタは角括弧 [ ] で囲まれて、ローカルな束縛を定義します。

```
(let [x 1
      y 2
      z 3]
  (+ x y z))
;; => 6
```

以上が ClojureScript でプログラムを書く上で必要となる構文の全てです。他の Lisp 方言でプログラムを書く場合も同様です。独自のデータ構造でプログラミング言語が実装されていることは、構文に統一性があり simple なので、素晴らしい特徴だといえます。マクロでコードを生成することも他の言語と比べて簡単であり、私たちのニーズに合わせて言語を拡張するのに十分な力を与えてくれます。

3.2 基本的なデータ型

他のプログラミング言語と同様、ClojureScript には多くのデータ型が用意されています。ClojureScript には、数値、文字列、浮動小数などのスカラー型のデータ構造があります。これら以外にも、シンボル、キーワード、正規表現、変数、アトム、Volatile などのデータ構造もあります。

ClojureScript はホスト言語を採用し、可能な限りホスト言が提供する型を使用します。例えば、数値と文字列は、JavaScript の数値と文字列と同じように利用して、結果を得ることができます。

数値

ClojureScript において、数値は整数と浮動小数点数を含みます。ClojureScript は JavaScript にコンパイルされる言語だということを覚えておいてください。整数は JavaScript では浮動小数点数になります。他の言語と同じように、ClojureScript で数値は次のように表現されます。

```
23
+23
-100
1.7
-2
33e8
12e-14
3.2e-4
```

キーワード

ClojureScript においてキーワードは、常にキーワードへ評価されるオブジェクトです。マップのデータ構造でキーを効率よく表すために使われます。

```
:foobar    ;; => :foobar
:2         ;; => :2
:?         ;; => :?
```

キーワードはリテラルとしてコロン : を使います。コロンはオブジェクトの名前の一部ではありません。keyword 関数によりキーワードを作成することも可能です。以下の例は、今は理解できなくても構いません。後に関数の章で取り上げます。

```
(keyword "foo")
;; => :foo
```

キーワードをダブルコロン :: で始めると、キーワードの前に現在の名前空間が付きます。名前空間付きのキーワードは、等価性の比較に影響を与えるので注意してください。

```
::foo
;; => :cljs.user/foo

(= ::foo :foo)
;; => false
```

別の方法としては、リテラルに名前空間を含める方法もあります。この方法は、他の名前空間に名前空間付きのキーワードを作成する際に役に立ちます。

```
:cljs.unraveled/foo
;; => :cljs.unraveled/foo
```

keyword 関数は 2 つの引数をとることができます。第 1 引数で名前空間を指定します。

```
(keyword "cljs.unraveled" "foo")
;; => :cljs.unraveled/foo
```

シンボル

ClojureScript のシンボルはキーワードとよく似ていますが、自分自身に対して評価されるのではなく、シンボルが参照している対象 (関数や変数等) に対して評価されます。

シンボルの名前はアルファベットで始まり、記号 (*, +, !, -, _, ', ?) を含むことができますが、数字で始めることはできません。

```
sample-symbol
othersymbol
f1
my-special-swap!
```

理解できなかったとしても気にしないでください。シンボルは様々な例で使われており、読み進むにつれて理解が深まります。

文字列

文字列については特に説明することはありません。ClojureScript は、他の言語と同じように動作します。ただし、文字列が不変である点に注意してください。この点は JavaScript と同じです。

```
"An example of a string"
```

ClojureScript の文字列で特異な点は、Lisp の構文に由来するものです。文字列の構文は 1 行でも複数行でも同じです。

```
"This is a multiline
     string in ClojureScript."
```

文字

リテラルで文字を書くこともできます。

```
\a        ; 小文字の文字
\newline  ; 改行文字
```

ホスト言語の JavaScript では文字のリテラルを使えないので、ClojureScript の文字は、JavaScript の文字列の 1 文字に変換されます。

コレクション

あるプログラミング言語について詳しく知るためには、その言語でコレクションがどのように抽象化されているかを知る必要があります。ClojureScript にも当てはまります。ClojureScript のコレクションには多くの型があります。ClojureScript が他の言語のコレクションと違うのは、コレクションが永続的で不変であることです。詳しい説明に入る前に ClojureScript のコレクションに存在する型について概要を見ましょう。

■リスト リストは Lisp 系言語で典型的なコレクションの型です。リストは ClojureScript で最もシンプルなコレクションです。リストはどの型の要素を含むことができます。リストに他のコレクションを含むこともできます。リストは要素を丸括弧で囲みます。

```
'(1 2 3 4 5)
'(:foo :bar 2)
```

上記の例では、リストの先頭にクオート ' がついています。これは Lisp 系言語では、リストが関数やマクロの呼び出しに使われるためです。リストを関数やマクロの呼び出しに使う場合、初めの要素は関数やマクロのシンボル、残りの要素はその引数である必要があります。上の例ではリストの先頭にクオートがついているため、要素のリストとして扱われます。
リスト先頭にクオート ' があるかどうかで、動作がどう違うかを確認しましょう。

```
(inc 1)
;; => 2

'(inc 1)
;; => (inc 1)
```

先頭にクオートをつけずに `(inc 1)` として評価した場合、`inc` シンボルは inc 関数として評価されます。`1` が第 1 引数として評価されて、`2` という値が返ります。
リストを作るために、明示的に `list` 関数を使うことも可能です。

```
(list 1 2 3 4 5)
;; => (1 2 3 4 5)

(list :foo :bar 2)
;; => (:foo :bar 2)
```

リストは、前から順に要素にアクセスするには効率よく処理できますが、リスト内の要素に対してランダムに (もしくは特定のインデックスに) アクセスするのは不得意です。

■ベクタ リストのようにベクタも連続する値を保存しますが、インデックスを用いて効率よく要素にアクセスできます。要素が前から順に評価されるリストと対照的です。後に詳しく説明しますが、現状ではこの程度の理解で十分です。ベクタはリテラルのために角括弧を使います。

```
[:foo :bar]
[3 4 5 nil]
```

リストと同様に、ベクタはどの型のオブジェクトを含むことができます。vector 関数を用いてベクタを作成できますが、あまり一般的ではありません。

```
(vector 1 2 3)
;; => [1 2 3]

(vector "blah" 3.5 nil)
;; => ["blah" 3.5 nil]
```

■マップ マップはコレクションの 1 つであり、キーと値をセットで保存します。他の言語では、連想配列や辞書と言われるデータ構造です。

```
{:foo "bar", :baz 2 }
{:alphabet [:a :b :c]}
```

キーと値を分けるためにコンマを使うかどうかは任意です。ClojureScript の構文では、コンマはスペースとして処理されます。

ベクタのように、マップリテラルの各々の要素は、結果がマップに保存される前に毎回評価されます。ただし、要素が順に評価されるかは保証されません。

■セット 最後はセットです。セットは任意の型のデータを 0 個以上保存することができますが、要素間に順序はありません。セットのためには、マップのように波括弧 { } を使います。波括弧の前に シャープ # をつける点がマップとは異なります。set 関数を用いてコレクションをセットに変換することもできます。

```
#{1 2 3 :foo :bar}
;; => #{1 :bar 3 :foo 2}

(set [1 2 1 3 1 4 1 5])
;; => #{1 2 3 4 5}
```

後のセクションでは、セットと他のコレクションの型について詳しくみます。

3.3 vars

ClojureScript は関数型プログラミング言語であり、不変性に重点をおいています。そのため、他の言語の変数にあたる概念がありません。代数学における変数にあたる概念はあります。つまり数学で `x = 6` というときには、`x` というシンボルが 6 を表すということを意味します。

ClojureScript において var はシンボルで表現されて、メタデータと一緒に 1 つの値を保存します。特殊形式の `def` を使うことで、var を 1 つ定義することができます。

```
(def x 2 2)
(def y [1 2 3])
```

var は名前空間において常にトップレベルです。名前空間については後に詳しく説明します。もし `def` を関数の呼び出しの中で使う場合、var は名前空間のレベルに定義されますが、これは推奨されません。関数の中で変数を定義するには `let` を使うことが推奨されます。

3.4 関数

はじめの一歩

ClojureScript には第一級関数と呼ばれるものがあります。第一級の関数は、他の型と同じように使えます。例えば、常にレキシカルスコープを重視しながら、関数の引数として渡せたり、関数の返り値として利用することができます。ClojureScript におけるダイナミックスコープについては別のセクションで扱います。

スコープについて詳しく知るには、http://en.wikipedia.org/wiki/Scope を参考にしてください。様々なタイプのスコープについて詳細を説明しています。

ClojureScript は Lisp 方言の 1 つであり、関数の呼び出しは、次のように前置記法を用います。

```
(inc 1)
;; => 2
```

この例では `inc` は関数であり、実行時に利用できる関数の 1 つです。この場合、`1` は `inc` 関数の第 1 引数です。

```
(+ 1 2 3)
;; => 6
```

+ シンボルは加算を意味します。複数の引数をとることができる点ができ、ALGOL 型のプログラミング言語が 2 つの引数しかとることができないことと対照的です。

前置記法には大きな利点があり、その中には必ずしも明白でないものもあります。ClojureScript では関数とオペレータを区別しません。全てが関数です。すぐに分かる長所としては、前置記法ではオペレータが任意の数の引数を許容することです。また、演算子の優先順位の問題も解消されます。

自作関数の定義

特殊形式の `fn` を用いると、無名関数を定義することができます。次の例では、関数は 2 つの引数を取り、それらの平均を返します。

```
(fn [param1 param2]
  (/ (+ param1 param2) 2.0))
```

関数を定義して即時に呼び出すことができます（1 つの式で）。

```
((fn [x] (* x x)) 5)
;; => 25
```

次に名前付き関数を作成しましょう。名前付き関数とはどういうことでしょうか。ClojureScript ではすごくシンプルです。関数は第一級クラスであり、他の全ての値と同じように振る舞います。関数をシンボルに束縛することにより、関数に名前をつけます。

```
(def square (fn [x] (* x x)))

(square 12)
;; => 144
```

ClojureScript には、関数定義の糖衣構文として `defn` マクロがあります。

```
(defn square
  "特定の数字の 2 乗を返す"
  [x]
  (* x x))
```

関数名と引数の間にある文字列は docstring(documentation string) と呼ばれます。Web でドキュメントを自動生成するときに docstring が使われます。

多様な Arity をもつ関数

ClojureScript は任意の数の引数をとる関数を定義することができます。(Arity という用語は関数が受ける引数の数を意味します) そのための構文は普通の関数定義と大体同じですが、複数の本体をもつ点が異なります。例をみてみましょう。

```
(defn myinc
  "自己定義版のパラメータ化された inc"
  ([x] (myinc x 1))
  ([x increment] (+ x increment)))
```

`([x] (myinc x 1))` は、もし引数が 1 つの場合、`myinc` 関数に `x` を第 1 引数、`1` を第 2 引数として渡すということを表しています。`([x increment] (+ x increment))` は、もし引数が 2 つの場合、2 つの引数を足した結果を返します。ここで定義した関数を使ってみましょう。引数の数を間違えて関数を呼び出すと、コンパイラからエラーメッセージが出力されることに注目してください。

```
(myinc 1)
;; => 2

(myinc 1 3)
;; => 4

(myinc 1 3 3)
;; Compiler error
```

Arity の概念を本書で説明することは、本書の範囲を超えています。詳しくは Wikipedia の Arity のページを参照してください。

可変長引数関数

可変長引数をとる関数を定義することでも、複数の引数を受け取る関数を定義することができます。可変長引数をとる関数は、任意の数の引数をとることができます。

可変長引数関数を定義するには `&` シンボルを引数のベクタにつけます。

```
(defn my-variadic-set
  [& params]
  (set params))

(my-variadic-set 1 2 3 1)
;; => #{1 2 3}
```

無名関数の短い構文

ClojureScript では、無名関数のためのシンプルな構文としてリーダマクロの `#()` を使うことができます (ワンライナーを書くためによく用いられます)。リーダーマクロとは、コンパイル時に適当なフォームに変換される「特別な」表現方法です。この場合、`fn` のフォームに変換されます。

```
(def average #(/ (+ %1 %2) 2))

(average 3 4)
;; => 3.5
```

この `average` の定義は、次のように展開されます。

```
(def average-longer (fn [a b] (/ (+ a b) 2)))

(average-longer 7 8)
;; => 7.5
```

`%1` や `%2` は引数の位置を示す印であり、リーダーマクロが `fn` の式に変換される時に暗黙的に宣言されます。

もし関数が 1 つしか引数を取らない場合、`%` シンボルの後の数字は省略できます。例えば、`#(* %1 %1)` は `#(* % %)` と書くことができます。

さらに `%&` シンボルにより可変長引数にすることができます。

```
(def my-variadic-set #(set %&))

(my-variadic-set 1 2 2)
;; => #{1 2}
```

3.5 フロー制御

ClojureScript のフロー制御は、JavaScript や C などとアプローチが異なります。

if による条件分岐

ClojureScript において `if` は文ではなく式であり、3 つの引数をとります。1 つ目は条件式、2 つ目は条件が真の場合に評価される式、3 つ目は条件が偽の場合に評価される式です。

```
(defn discount
  "1 点以上の購入で 5% 割引されます"
  [quantity]
  (if (>= quantity 100)
    0.05
    0))

(discount 30)
;; => 0

(discount 130)
;; => 0.05
```

if の条件分岐の中で複数の式を実行するには do を使います。do については次のセクションで説明します。

cond による条件分岐

if 式には複数の条件を追加するための else if の部分がないため、多少制限されますが、cond マクロでこの点を解決できます。cond では複数の条件を定義できます。

```
(defn mypos?
  [x]
  (cond
    (> x 0) "positive"
    (< x 0) "negative"
    :else "zero"))

(mypos? 0)    ;; => "zero"

(mypos? -2)  ;; => "negative"
```

また、cond には condp と呼ばれる別の形式もあり、単純な cond と非常によく似た働きをしますが、条件 (述語とも呼ばれます) が全ての条件に対して同じ場合は、より綺麗に見えます。

以下の各行では、ClojureScript は (keyword code) を評価した結果に = 関数を適用します。

```
(defn translate-lang-code
  [code]
  (condp = (keyword code)
    :es "Spanish"
    :en "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

case による条件分岐

case は、前例の condp と同様の使い方をします。主な違いは、常に = 述語 (関数) が使用されて、その分岐値がコンパイル時に評価されることです。そのため cond や condp よりもパフォーマンスが良いですが、条件の値が静的でなければならないという欠点があります。

```
(defn translate-lang-code
  [code]
  (case code
    "es" "Spanish"
    "en" "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

3.6 真偽判定

真偽判定はプログラミング言語によりセマンティクスが異なります。多くの言語では、空のコレクションや整数の 0 、またこれらと同様の値を偽と見なします。ClojureScript では nil と false だけが偽と見なされ、その他は論理的に真として扱われます。

callable プロトコル (IFn については後述) を実装する能力と一緒に、集合のようなデータ構造は、関数の中でそれらを追加的にラップする必要なしに、述語として使うことができます。

```
(def valid? #{1 2 3})

(filter valid? (range 110))
;; => (1 2 3)
```

このように動作するのは、セットが全ての要素の値自体か nil を返すためです。

```
(valid? 1)
;; => 1

(valid? 4)
;; => nil
```

3.7 ローカル、ブロック、ループ

ローカル

ClojureScript には ALGOL のような変数の概念がありませんが、ローカル (local) はあります。ローカルはイミュータブルであり、変更しようとするとエラーが発生します。

ローカルは `let` 式で定義されます。`let` 式は、最初のパラメータとしてベクタで始まり、その後に任意の数の式が続きます。最初のパラメータのベクタには束縛フォームを与えて、その `let` 内のローカルで有効な名前と値のペアを宣言します。

```
(let [x (inc 1)
      y (+ x 1)]
  (println "Simple message from the body of a let")
  (* x y))
```

上の例では、シンボル `x` が `(inc 1)` の値に束縛されて、シンボル `y` が `x` と 1 の合計 (つまり 3) に束縛されます。これらの束縛を受けて、`(println "Simple message from the body of a let")` と `(* x y)` が評価されます。

ブロック

JavaScript において波括弧 `{ }` は「共に属する」コードのブロックを決めます。ClojureScript では `do` を用いてブロックを作成します。`do` はコンソールの何かの結果を出力したり、ログを出力したりするような、副作用を伴う場合に使います。副作用とは、戻り値には不要なものをいいます。

`do` は任意の数の式を含むことができますが、最後に評価された式の値が返り値となります。

```
(do
  (println "hello world")
  (println "hola mundo")
  ;; この値は返却されずに捨てられます。
  (* 3 5)
  (+ 1 2))
;; hello world
;; hola mundo
;; => 3
```

先ほど説明した `let` の本体は、複数の式をもつことができる点において `do` とよく似ています。実際、`let` は暗黙の `do` をもちます。

ループ

ClojureScript は関数型のアプローチを採用しているので、JavaScript での `for` のような一般的なループがありません。ClojureScript のループは再帰を使って処理されます。再帰を用いてプログラムを書くためには、命令型のプログラミング言語とは少し違った方法で問題をモデル化する必要があります。

他の言語で `for` が使われるパターンの多くは、高階関数を用いて置き換えることができます。高階関数とは関数を引数として受け取る関数です。

■loop/recur によるループ　では、`loop` と `recur` を用いた再帰でループを表現する方法を見てみましょう。`loop` は、空の可能性のある束縛のリスト (`let` との対称性に注目してください) を定義して、繰り返し実行すると、それらの束縛の新しい値を使ってループの開始点に戻ります。例を見てみましょう。

```
(loop [x 0]
  (println "Looping with " x)
  (if (= x 2)
    (println "Done looping!")
    (recur (inc x))))
;; Looping with 0
;; Looping with 1
;; Looping with 2
;; Done looping!
;; => nil
```

上の例では、まず `x` を `0` に束縛して本体を実行します。`if` の条件が満たされていないので、`(recur (inc x))` で `inc` で `x` に `1` が加算されてループが再実行されます。条件が満たされると `recur` の呼び出しが止まり、ループが終了します。

`recur` を使うことができるのは `loop` だけではありません。関数の中で `recur` を使用すると、新しい束縛値で本体が再帰的に実行されます。

```
(defn recursive-function
  [x]
  (println "Looping with" x)
  (if (== x 2)
    (println "Done looping!")
    (recur (inc x))))

(recursive-function 0)
;; Looping with 0
;; Looping with 1
;; Looping with 2
;; Done looping!
;; => nil
```

■高階関数によるループの置き換え　命令型プログラミング言語では、for で反復的にデータを変形しますが、次のような目的で利用することが多いです。命令型プログラミング言語では、for ループを使用してデータを繰り返し変換することが一般的です。通常、次のいずれかを目的とします。

- イテラブルの全ての値を、別のイテラブルに変形する
- ある条件でイテラブルの要素をフィルターする
- イテラブルの要素を順に処理をして 1 つの値に変換する
- イテラブルにある各々の値を計算する

上記の実行は、高階関数と ClojureScript の構文を用いて書くことができます。まず最初の 3 つの例を見ていきましょう。

イテラブルにある全ての値を変換するには map 関数を使います。map 関数は、関数とシーケンスを引数にとり、関数をシーケンスの各要素に適用します。

```
(map inc [0 1 2])
;; => (1 2 3)
```

map 関数の初めのパラメータには、1 つの引数をとり 1 つの値を返す関数を指定します。例えば、グラフ作成のアプリケーションがあるとします。y = 3x + 5 の式を x の値のセットに対して適用して y の値のセットをえるには、次のように書きます。

```
(defn y-value [x] (+ (* 3 x) 5))

(map y-value [1 2 3 4 5])
;; => (8 11 14 17 20)
```

高階関数に渡す関数が短い場合は、無名関数を使うこともできます。無名関数には fn か # のどちらを用いても構いません。

```
(map (fn [x] (+ (* 3 x) 5)) [1 2 3 4 5])
;; => (8 1 1 1 4 1 7 2 0)

(map #(+ (* 3 %) 5) [1 2 3 4 5])
;; => (8 1 1 1 4 1 7 2 0)
```

データ構造の値をフィルターするには、filter 関数を使います。filter 関数は、述語とシーケンスをとり、述語に対して真を返す要素だけを持つ新しいシーケンスを生成します。

```
(filter odd? [1 2 3 4])
;; => (1 3)
```

ここでも、filter 関数に適用する最初の引数として true または false を返す任意の関数を使用できます。5 文字未満の単語のみを保つ例をみてみましょう (count 関数は引数の長さを返します)。

```
(filter (fn [word] (< (count word) 5)) ["ant" "baboon" "crab" "duck" "echidna" "fox"])
;; => ("ant" "crab" "duck" "fox")
```

イテラブルの要素を 1 つずつ処理をして蓄積しながら 1 の値を返すには reduce を用います。reduce は値を蓄積するための関数、初期値、コレクションを取りますが、初期値については任意です。

```
(reduce + 0 [1 2 3 4])
;; => 10
```

また、reduce の最初の引数として自作の関数を指定することもできますが、その関数には 2 つの引数が必要です。最初の引数は「蓄積値」で、2 番目の引数は処理されるコレクションのアイテムです。この関数は、リスト内の次のアイテムのアクミュレータになる値を返します。例えば、数値の集合の平方和を求めましょう（統計では重要な計算です)。関数を分ける場合は次のようにします。

```
(defn sum-squares
  [accumulator item]
  (+ accumulator (* item item)))

(reduce sum-squares 0 [3 4 5])
;; => 50
```

無名関数を使う場合は次のようにします。

```
(reduce (fn [acc item] (+ acc (* item item))) 0 [3 4 5])
;; => 50
```

単語セット内の総文字数を検出する reduce 関数を次に示します。

```
(reduce (fn [acc word] (+ acc (count word)))
        0 ["ant" "bee" "crab" "duck"])
;; => 14
```

上の例では、短い構文を使っていません。なぜなら、短い構文を使うとタイピング数を減らせますが、コードの可読性が低下する可能性があるからです。新しい言語を始めるときは、自分が書いたものを後で読めることは重要です。もし短い構文に慣れている場合は、この構文を使用してください。

アキュムレータに渡す開始値は慎重に選ぶ必要があることを覚えておいてください。一連の数の積を求めるために reduce を使うときは、0 ではなく 1 を開始値として使う必要があります。0 を用いて掛け算をすることができないためです。

```
;; wrong starting value
(reduce * 0 [3 4 5])    ;; => 0

;; correct starting accumulator
(reduce * 1 [3 4 5])    ;; => 60
```

■for によるシーケンス内包表記　ClojureScript の for は、反復のためではなくシーケンスを生成するために用いられます。これはシーケンス内包表記として知られています。このセクションでは、シーケンス内包表記がどのように動作するかを学び、シーケンスを宣言的に構築するためにシーケンス内包表記を使います。

for は束縛のためのベクタと式をとり、式が評価された結果をシーケンスとして生成します。

```
(for [x [1 2 3]]
  [x (* x x)])
;; => ([1 1] [2 4] [3 9])
```

この例では、x は [1 2 3] の各要素に順に束縛され、元の要素と それを 2 乗した値をもつ 2 要素ベクタの新しいシーケンスを返します。

for は複数の束縛をサポートしているため、命令型言語での for ループのネストと同様に、コレクションがネストされて反復されます。最も内側の束縛が「最も速く」繰り返されます。

```
(for [x [1 2 3]
      y [4 5]]
  [x y])
;; => ([1 4] [1 5] [2 4] [2 5] [3 4] [3 5])
```

さらに 3 つの修飾子があります。:let はローカルな束縛の作成のために、:while はシーケンス生成からの脱出のために、:when は値のフィルタリングのために用います。

次は :let を用いたローカル束縛の例です。:let で定義された束縛はその式でのみ利用可能です。

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]]
  z)
;; => (5 6 6 7 7 8)
```

:while 修飾子は、条件が合わなくなったときにシーケンスの生成を止める条件を表すために使います。次の例を見てください。

```
(for [x [1 2 3]
      y [4 5]
      :while (= y 4)]
  [x y])
;; => ([1 4] [2 4] [3 4])
```

生成された値をフィルターするには、次のように :when 修飾子を使います。

```
(for [x [1 2 3]
      y [4 5]
      :when (= (+ x y) 6)]
  [x y])
;; => ([1 5] [ 2  4])
```

上記のような修飾語を組み合わせることで、複雑な配列の生成を表現したり、理解の意図をより明確に表現することができます。

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]
      :when (= z 6)]
  [x y])
;; => ([1 5] [2 4])
```

命令型プログラミング言語における for 構文の最も一般的な使用方法を説明した際に、結果を気にせずにシーケンスのすべての値に対して計算を実行したい場合があると述べました。おそらく、シーケンスの値を用いて何らかの副次的効果を得るために、このようなことをするのでしょう。

同様の目的を達成するために、 ClojureScript には doseq 構文があります。doseq は for と似ていますが、式を実行した後、結果の値を捨てて nil を返します。

```
(doseq [x [1 2 3]
        y [4 5]
       :let [z (+ x y)]]
;; 1 + 4 = 5
;; 1 + 5 = 6
;; 2 + 4 = 6
;; 2 + 5 = 7
;; 3 + 4 = 7
;; 3 + 5 = 8
;; => nil
```

コレクション内の各アイテムに対して副作用のある操作 (println 等) を効率よく繰り返し適用したい場合は、それに特化した関数 run! を使います。内部的には高速に reduce を使用します。

```
(run! println [1 2 3])
;; 1
;; 2
;; 3
;; => nil
```

この関数は明示的に nil を返します。

3.8 コレクションの型

イミュータブルと永続性

ClojureScript のコレクションは永続的でイミュータブルだと言及しましたが、詳しく説明していませんでした。イミュータブルなデータ構造は、変更ができないデータ構造です。イミュータブルなデータ構造において、部分的な更新は許されません。conj を用いてベクタに値を付け加える例を見てみましょう。

```
(let [xs [1 2 3]
      ys (conj xs 4)]
  (println "xs:" xs)
  (println "ys:" ys))
;; xs: [1 2 3]
;; ys: [1 2 3 4]
;; => nil
```

上の例では xs に要素を 1 つ追加して新たなベクタ ys を作成しています。ベクタ xs はイミュータブルなので変更されません。

永続的なデータ構造とは、変換時に元のバージョンを残したまま新しいバージョンを返すデータ構造のことです。ClojureScript では、構造共有 (structural sharing) と呼ばれる実装技術を用いて、メモリと時間の効率化を図っており、2 つのバージョンの値の間で共有されるデータのほとんどは重複しません。

構造共有がどのように動作しているかを知りたい場合は読み進めてください。もし興味がない場合は、読み飛ばして次のセクション (シーケンスの抽象化) に進んでください。

ClojureScript におけるデータ構造の構造共有を説明するために、古いデータ構造と新しいデータ構造の一部が同じオブジェクトかどうかを identical? を用いて比較してみます。ここではリストを例に説明します。

```
(let [xs (list 1 2 3)
      ys (cons 0 xs)]
  (println "xs:" xs)
  (println "ys:" ys)
  (println "(rest ys):" (rest ys))
  (identical? xs (rest ys)))
;; xs: (1 2 3)
;; ys: (0 1 2 3)
;; (rest ys): (1 2 3)
;; => true
```

リストの xs に cons 関数を使って値を追加して新たなリスト ys を作成しています。リスト ys の rest はリスト xs とメモリでは等しくなります。この挙動を見ると、リスト xs と リスト ys がデータ構造を共有していることがわかります。

シーケンスの抽象化

ClojureScript による抽象化で中心的な概念の 1 つに シーケンス (sequence) があります。シーケンスはリストとして見なされ、どのコレクションの型もシーケンスとして見なすことができます。シーケンスは全てのコレクションの型のように永続的でイミュータブルです。大半の ClojureScript の関数はシーケンスを返します。

シーケンスを生成するために使うことができる型はシーカブル (seqables) と呼ばれます。seq をシーカブルに対して呼び出して、シーケンスをえることができます。シーケンスは 基本的な関数 first と rest をサポートします。どちらも first と rest に与える引数に対して seq を呼び出します。

```
(first [1 2 3])
;; => 1

(rest [1 2 3])
;; => (2 3)
```

seq 関数をシーカブルに対して呼ぶとき、シーカブルが空かどうかで結果が異なります。空の場合には nil を、そうでなければシーケンスを返します。

```
(seq [])
;; => nil

(seq [1 2 3])
;; => (1 2 3)
```

next は rest と似たシーケンス操作関数ですが、シーケンスの要素が空か 1 つの場合に nil を返す点が rest とは異なります。前述のシーケンスの 1 つが与えられたとき、rest により返される空のシーケンスは論理値で真を返します。一方、next により返される nil の値は偽と評価されます。真偽判定の章で詳しく説明します。

```
(rest [])
;; => ()

(next [])
;; => nil

(rest [1 2 3])
;; => (2 3)

(next [1 2 3])
;; => (2 3)
```

■nil パンニング seq はコレクションが空のときに nil を返し、nil は論理値として偽と評価されるので、コレクションが空かどうかを seq 関数を用いて確認することができます。このようなテクニックは nil パンニングと呼ばれます。

```
(defn print-coll
  [coll]
  (when (seq coll)
    (println "Saw " (first coll))
    (recur (rest coll))))

(print-coll [1 2 3])
;; Saw 1
;; Saw 2
;; Saw 3
;; => nil

(print-coll #{1 2 3})
;; Saw 1
;; Saw 3
;; Saw 2
;; => nil
```

nil は シーカブル でもシーケンスでもありませんが、これまで見てきた全ての関数でサポートされています。

```
(seq nil)
;; => nil

(first nil)
;; => nil

(rest nil)
;; => ()
```

■シーケンス操作関数 ClojureScript コレクションを変換するための中心的な関数は、引数からシーケンスの生成を行い、前章で学んだ一般的なシーケンス操作関数において使えるように実装されています。このことにより、それらの関数は、全ての seqable なデータ型に対して使うことができます。次の例で map 関数が様々な seqables に対して使えることを見てみましょう。

```
(map inc [1 2 3])
;; => (2 3 4)

(map inc #{1 2 3})
;; => (2 4 3)

(map count {:a 41 :b 40})

(map inc '(1 2 3))
;; => (2 3 4)
```

map 関数をマップのコレクションに対して使うとき、高階関数はキーと値を含む 2 つのアイテムからなるベクタを受け取ります。次の例では、キーと値にアクセスするために destructuring を使います。

```
(map (fn [[key value]] (* value value))
     {:ten 10 :seven 7 :four 4})
;; => (100 49 16)
```

値のシーケンスを取得するために限れば、同じ操作をより慣用的な方法で行うことも可能です。

```
(map (fn [value] (* value value))
     (vals {:ten 10 :seven 7 :four 4}))
;; => (100 49 16)
```

気がついたかもしれませんが、シーケンスを処理する関数は、空のシーケンスを返すだけでいいので、空のコレクションや nil 値であっても安全に使用することができます。

```
(map inc [])   ; ==> ()

(map inc #{})  ; ==> ()

(map inc nil)  ; ==> ()
```

これまで map, filter, reduce などの一般的な使用例はすでに紹介しましたが、 ClojureScript の core 名前空間には、さらに多くの汎用のシーケンス操作関数が用意されています。ここで説明する操作の多くは、シーカブルで動作するか、もしくはユーザ定義型に拡張可能です。

coll? 述語を用いて、データがコレクション型かどうかを判別できます。

```
(coll? nil)
;; => false

(coll? [1 2 3])
;; => true

(coll? {:language "ClojureScript" :file-extension "cljs"})
;; => true

(coll? "ClojureScript")
;; => false
```

同様に、データ構造がシーケンスかどうかを判別する seq? 述語、シーカブルかどうかを判別したりする seqable? 述語があります。

```
(seq? nil)
;; => false
(seqable? nil)
;; => false

(seq? [])
;; => false
(seqable? [])
;; => true

(seq? #{1 2 3})
;; => false
(seqable? #{1 2 3})
;; => true

(seq? "ClojureScript")
;; => false
(seqable? "ClojureScript")
;; => false
```

一定時間内にカウントできるコレクションの場合は、count 操作を使用できます。この操作は文字列に対しても機能しますが、これまで見てきたように、文字列はコレクション、シーケンス、またはシーカブルではありません。

```
(count nil)
;; => 0

(count [1 2 3])
;; => 3

(count {:language "ClojureScript" :file-extension "cljs"})
;; => 2

(count "ClojureScript")
;; => 13
```

また empty 関数を使うことで、与えられたコレクションの空の variant を得ることもできます。

```
(empty nil)
;; => nil

(empty [1 2 3])
;; => []

(empty #{1 2 3})
;; => #{}
```

empty? 述語は、コレクションが空の場合に true を返します。

```
(empty? nil)
;; => true

(empty? [])
;; => true

(empty? #{1 2 3})
;; => false
```

conj 操作は要素をコレクションに追加しますが、コレクションの型に応じて異なる「場所」に追加することができます。コレクションの型で最もパフォーマンスが高い場所に要素を追加します。ただし、全てのコレクションに順序が定義されているわけではありません。

conj には追加する要素をいくつでも渡すことができます。実際の動作を見てみましょう。

```
(conj nil 4 2)
;; => (4 2)
```

```
(conj [1 2] 3)
;; => [1 2 3]

(conj [1 2] 3 4 5)
;; => [1 2 3 4 5]

(conj '(1 2) 0)
;; => (0 1 2)

(conj #{1 2 3} 4)
;; => #{1 3 2 4}

(conj {:language "ClojureScript"} [:file-extension "cljs"])
;; => {:language "ClojureScript", :file-extension "cljs"}
```

■遅延 ClojureScript のシーケンスを返す大半の関数は、全く新たなシーケンスを生成するのではなく、遅延シーケンスを生成します。遅延シーケンスは、要求がされたときに内容を生成します。通常は、それらに反復作業を行うときに生成されます。遅延により、必要以上のことをしないことが保証して、潜在的な無限シーケンスを正規のシーケンスとして扱うことができます。

整数の範囲を生成する range 関数を見てみましょう。

```
(range 5)
;; => (0 1 2 3 4)

(range 1 10)
;; => (1 2 3 4 5 6 7 8 9)

(range 10 100 15)
;; (10 25 40 55 70 85)
```

(range) だけで呼び出した場合、整数の無限シーケンスを生成します。REPL で (range) を試さないでください。REPL はその式を完全に評価しようとするので、とても長い間待たされます。

少し例を見てみましょう。グラフ作成のプログラムを書いていて、y = 2 x ^ 2 + 5 のグラフを書くとします。y の値が 100 未満の場合の x の値を求めます。0 から 100 までの全ての数を生成して、take-while を用いて条件に合うものを保ちます。ここに例を見てみましょう。グラフ作成プログラムを作成していて、方程式 y = 2 x ^ 2 + 5 をグラフ化していて、x の値のうち、 y の値が 100 未満の値だけが必要だとします。0 から 100 までの全ての数を生成して、条件が満たされている間だけ値をとります。

```
(take-while (fn [x] (< (+ (* 2 x x) 5) 100))
            (range 0 100))
;; => (0 1 2 3 4 5 6)
```

コレクション詳細

ここまで、ClojureScript におけるシーケンスの抽象化とシーケンス操作関数について理解を深めました。次は、コレクション型の詳細と、それらがサポートする関数について見ていきます。

■リスト ClojureScript では、シンボルをグループ化してプログラムにするためのデータ構造として、主にリストが使用されます。他の Lisp 方言とは異なり、ClojureScript ではリストとは異なるデータ構造体を使用します (ベクタとマップ)。このことによりコードの統一感が低下しますが、コードの読みやすさは向上します。

ClojureScript のリストは、各ノードに値とリストの残りの部分へのポインタが含まれる連結リストだと考えることができます。リストの最後に項目を追加するにはリスト全体を横断する必要がありますが、リストの先頭に項目を追加するのは自然に (高速に) できます。リストの最初に要素を追加するには、`cons` 関数を用います。

```
(cons 0 (cons 1 (cons 2 ())))
;; => (0 1 2)
```

リストのリテラル表現として `()` を使いました。シンボルを 1 つも含んでいないので、関数の呼び出しとしては見なされません。もし要素がある場合には、式が評価されないようにクオート ’ をつける必要があります。

```
(cons 0 ’(1 2))
;; => (0 1 2)
```

リストコレクションにおいては先頭部は一定の時間で追加できる場所なので、リストに対する `conj` 操作は要素を自然に先頭部に追加します。

```
(conj ’(1 2) 0)
;; => (0 1 2)
```

リストと ClojureScript の他のデータ構造において、`peek`, `pop`, `conj` 関数はスタック操作のために使えます。スタックの最上部は `conj` が要素を追加する場所であることから、`conj` 操作はスタックへのプッシュ操作と同等です。リストの場合、`conj` は要素をリストの先頭に追加して、`peek` はリストの最初の要素を返し、`pop` は先頭の要素以外の全ての要素を返します。

スタックを操作する `conj` と `pop` がスタックのために使われるコレクションの型に変更を加えないことに注意してください。

```
(def list-stack '(0 1 2))

(peek list-stack)
;; => 0

(pop list-stack)
;; => (1 2)

(type (pop list-stack))
;; => cljs.core/List

(conj list-stack -1)
;; => (-1 0 1 2)

(type (conj list-stack -1))
;; => cljs.core/List
```

リストが特に不得意なのは、インデックスによるランダムアクセスです。リストはメモリ内で連結されたリストのような構造体に格納されるので、要求されたアイテムを取り出すためにも、またインデックスが有効範囲にない場合にエラーを投げるためにも、ランダムに要素にアクセスするには先頭から順に検索する必要があります。遅延シーケンスのようなインデックス付けされていない順序付きコレクションも、この制限を受けます。

■ベクタ ベクタは ClojureScript で最も一般的なデータ構造の 1 つです。ベクタは、伝統的な Lisp 方言がリストを使用する多くの場所、例えば、関数の引数宣言や `let` 束縛での構文内で使われます。

ClojureScript のベクタは、リテラル表現として角括弧 `[]` を使います。`vector` 関数で生成することも可能です。他のコレクションからベクタを作成するには `vec` 関数を使います。

```
(vector? [0 1 2])
;; => true

(vector 0 1 2)
;; => [0 1 2]

(vec '(0 1 2))
;; => [0 1 2]
```

ベクタは様々な種類の値が順に並べられたコレクションです。リストとは異なり、ベクタは末尾から自然に拡張することができ、`conj` 関数はアイテムをベクタの末尾に追加します。ベクタの末尾へ要素を追加する操作は、実質的に一定時間で行えます。

```
(conj [0 1] 2)
;; => [0 1 2]
```

リストとベクタを区別するもう 1 つの点は、ベクタはインデックス付きコレクションであり、効率的なランダムアクセスや非破壊更新をサポートする点です。nth 関数を使うと、指定されたインデックスの値を取得できます。

```
(nth [0 1 2] 0)
;; => 0
```

ベクタは連続した数値キー (インデックス) を値に関連付けるので、連想データ構造として扱うことができます。ClojureScript が提供する assoc 関数は、連想データの構造体とキーと値のペアのセットを指定すると、変更されたキーに対応する値を持つ新しいデータの構造体を生成します。インデックスは、ベクタの最初の要素の 0 から始まります。

```
(assoc ["cero" "uno" "two"] 2 "dos")
;; => ["cero" "uno" "dos"]
```

ベクタにすでに含まれているキー、またはベクタの最後の位置にあるキーにのみ assoc を使うことができます。

```
(assoc ["cero" "uno" "dos"] 3 "tres")
;; => ["cero" "uno" "dos" "tres"]

(assoc ["cero" "uno" "dos"] 4 "cuatro")
;; Error: Index 4 out of bounds [0,3]
```

連想データ構造は関数として使うこともできます。これらは、関連付けられている値に対するキーの関数です。ベクタの場合、指定されたキーが存在しないと例外が発生します。

```
(["cero" "uno" "dos"] 0)
;; => "cero"

(["cero" "uno" "dos"] 2)
;; => "dos"

(["cero" "uno" "dos"] 3)
;; Error: Not item 3 in vector of length 3
```

ベクタはリストと同様にスタックとして使うことが可能であり、peek, pop, conj を使うことができます。ベクタとリストは要素を追加するときに、先頭と末尾が逆であることに注意してください。

```
(def vector-stack [0 1 2])

(peek vector-stack)
;; => 2

(pop vector-stack)
;; => [0 1]

(type (pop vector-stack))
;; => cljs.core/PersistentVector

(conj vector-stack 3)
;; => [0 1 2 3]

(type (conj vector-stack 3))
;; => cljs.core/PersistentVector
```

map と filter の操作は遅延シーケンスを返しますが、これらの操作の後には完全に実現されたシーケンスが必要なことが一般的なため、ベクタを返す同等のものとして mapv や filterv 等があります。これらの関数には、遅延シーケンスからベクタを構築するよりも高速であり、意図をより明確にするという利点があります。

```
(map inc [0 1 2])
;; => (1 2 3)

(type (map inc [0 1 2]))
;; => cljs.core/LazySeq

(mapv inc [0 1 2])
;; => [1 2 3]

(type (mapv inc [0 1 2]))
;; => cljs.core/PersistentVector
```

■マップ マップは ClojureScript で広く使われています。ベクタと同様に、マップはメタデータを vars に付与するために使われます。ClojureScript のデータ構造体は全てマップのキーとして使えますが、キーワードは関数としても呼び出すことができるため、キーワードを使うことが一般的です。ClojureScript のマップは、リテラル表現として、キーと値のペアを中括弧 { } で囲んで記述します。代わりに hash-map 関数で生成することもできます。

```
(map? {:name "Cirilla"})
;; => true

(hash-map :name "Cirilla")
;; => {:name "Cirilla"}
```

```
(hash-map :name "Cirilla" :surname "Fiona")
;; => {:name "Cirilla" :surname "Fiona"}
```

通常のマップには特定の順番がないため、conj の操作は 1 つ以上のキーと値のペアをマップに追加するだけです。conj をマップに使うには、最後の引数として 1 つ以上のキーと値のペアのシーケンスを想定しています。

```
(def ciri {:name "Cirilla"})

(conj ciri [:surname "Fiona"])
;; => {:name "Cirilla", :surname "Fiona"}

(conj ciri [:surname "Fiona"] [:occupation "Wizard"])
;; => {:name "Cirilla", :surname "Fiona", :occupation "Wizard"}
```

上の例では要素の順が保たれていますが、もし多くのキーを含む場合、順序が保持されていないことがわかります。マップはキーと値を関連付ける連想型のデータ構造です。マップに要素を追加するには assoc を使い、マップから要素を取り除くには dissoc を使います。assoc はすでにあるキーの値を更新することができます。これらの関数を試してみましょう。

```
(assoc {:name "Cirilla"} :surname "Fiona")
;; => {:name "Cirilla", :surname "Fiona"}

(assoc {:name "Cirilla"} :name "Alfonso")
;; => {:name "Alfonso"}

(dissoc {:name "Cirilla"} :name)
;; => {}
```

マップはキーの関数でもあり、指定されたキーに関連する値を返します。ベクタとは異なり、マップに存在しないキーを指定すると、nil が返されます。

```
({:name "Cirilla"} :name)
;; => "Cirilla"

({:name "Cirilla"} :surname)
;; => nil
```

ClojureScript はソートされたハッシュマップも提供しています。ソートされていないバージョンと同じように動作しますが、繰り返しの際に順序を保持します。sorted-map を使用すると、デフォルトの順序でソートされたマップを作成できます。

```
(def sm (sorted-map :c 2 :b 1 :a 0))
;; => {:a 0, :b 1 , :c 2 }

(keys sm)
;; => (:a :b :c)
```

独自の順序付けが必要な場合は、sorted-map-by に比較関数を与えられます。組み込みの compare 関数から返される値を反転する例を見てみましょう。compare 関数は 2 つの要素を比較して、最初の項目が 2 番目の項目より小さい場合は -1 、等しい場合は 0 、最初の項目が 2 番目の項目よりも大きい場合 1 を返します。

```
(defn reverse-compare [a b] (compare b a))

(def sm (sorted-map-by reverse-compare :a 0 :b 1 :c 2))
;; => {:c 2 , :b 1 , :a 0}

(keys sm)
;; => (:c :b :a)
```

■セット ClojureScript のセットは #{} のリテラル表現を用いて作成されます。set コンストラクタを用いて作成することもできます。セットは、要素間の順番がないコレクションであり、同じ要素の重複は許されません。

```
(set? #{\a \e \i \o \u})
;; => true

(set [1 1 2 3])
;; => #{1 2 3}
```

セットのリテラル表現は同じ値を重複して含むことができません。もし要素が重複する場合、エラーが発生します。

```
#{1 1 2 3}
;; clojure.lang.ExceptionInfo: Duplicate key: 1
```

セットを操作するための関数は多くあります。clojure.set の名前空間にあるため、インポートする必要があります。名前空間については後に詳しく学びます。現時点では、clojure.set の名前空間をインポート後、シンボル s に束縛していることが分かれば十分です。

```
(require '[clojure.set :as s])

(def danish-vowels #{\a \e \i \o \u \æ \ø  \å })
;; => #{"a" "e" "å " "æ" "i" "o" "u" "ø "}

(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(s/difference danish-vowels spanish-vowels)
;; => #{"å " "æ" "ø "}

(s/union danish-vowels spanish-vowels)
;; => #{"a" "e" "å " "æ" "i" "o" "u" "ø "}

(s/intersection danish-vowels spanish-vowels)
;; => #{"a" "e" "i" "o" "u"}
```

イミュータブルなセットの優れた特性はネストできることです。ミュータブルなセットをもつプログラミング言語では重複した値を含みますが、ClojureSript では重複を許しません。実際、全ての ClojureScript のデータ構造はイミュータブルの性質のおかげで任意にネストすることができます。セットは他のコレクションのデータ型と同様に `conj` を使うことも可能です。

```
(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(def danish-vowels (conj spanish-vowels \æ \ø  \å ))
;; => #{"a" "e" "i" "o" "u" "æ" "ø " "å "}

(conj #{1 2 3} 1)
;; => #{1 3 2}
```

セットは、読み取り専用の連想型データとして動作して、セットに含まれる値をセット自体に関連づけます。ClojureScript では `nil` と `false` 以外の全ての値が真であるため、述語関数としてセットを使用できます。

```
(def vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(get vowels \b)
;; => nil

(contains? vowels \b)
;; => false

(vowels \a)
;; => "a"
```

```
(vowels \z)
;; => nil

(filter vowels "Hound dog")
;; => ("o" "u" "o")
```

セットの要素に順序をつけて扱うには、マップの場合と同様に、sorted-set 関数と sorted-set-by 関数が用意されています。マップの sorted-map と sorted-map-by に似ています。

```
(def unordered-set #{[0] [1] [2]})
;; => #{[0] [2] [1]}

(seq unordered-set)
;; => ([0] [2] [1])

(def ordered-set (sorted-set [0] [1] [2]))
;; =>#  {[0] [1] [2]}

(seq ordered-set)
;; => ([0] [1] [2])
```

■キュー さらに ClojureScript では、永続的でイミュータブルなキューを使うことができます。キューは他のコレクションのように広くは使われていません。キューを作成するためのリテラル表現として #queue [] が用意されています。キューを作成するためのコンストラクタ関数はありません。

```
(def pq #queue [1 2 3])
;; => #queue [1 2 3]
```

値をキューに追加するために conj を使うと、アイテムは末尾に追加されます。

```
(def pq #queue [1 2 3])
;; => #queue [1 2 3]

(conj pq 4 5)
;; => #queue [1 2 3 4 5]
```

キューに関して留意すべき点は、スタック操作が通常のスタックのセマンティクス (同じ終点から出し入れすること) に従わないことです。pop は前の位置から値を取り、conj は要素を後ろに push(もしくは append) します。

```
(def pq #queue [1 2 3])
;; => #queue [1 2 3]

(peek pq)
;; => 1

(pop pq)
;; => #queue [2 3]

(conj pq 4)
;; => #queue [1 2 3 4]
```

キューはリストやベクタほど頻繁には使われませんが、ClojureScript では手軽に利用できることを知っておくと便利です。

3.9 Destructuring

destructuring(分割) という言葉が示すように、destructuring はコレクションのような構造化されたデータを分割して、個別の要素に焦点を当てる方法です。ClojureScript は分割のための構文はシンプルです。destructuring は、インデックスがついたシーケンスに対しても、連想型のデータ構造に対しても、束縛が宣言された場所であればどこでも使うことができます。

前の説明を理解するために、destructuring の使用例を見てみましょう。シーケンスがあり、1 番目と 3 番目の要素だけに関心があるとします。これらへの参照は、`nth` 関数で簡単に取得できます。

```
(let [v [0 1 2]
      fst (nth v 0)
      thrd (nth v 2)]
  [thrd fst])
;; => [2 0]
```

しかし、前のコードは冗長すぎます。destructuring では、束縛の左側のベクタを使用して、インデックスがついたシーケンスからよりシンプルに値を取り出すことができます。

```
(let [[fst _ thrd] [0 1 2]]
  [thrd fst])
;; => [2 0]
```

上の例では、`[fst _ thrd]` が destructuring の箇所です。これはベクタとして表され、`fst` と `thrd` に対応するインデックス `0` と `2` の値がそれぞれ束縛されます。シンボルの `_` は、興味のない値 (この場合は `1`) のプレースホルダとして使われます。

destructuring は `let` での束縛に限定されないことに注意してください。`for` や `doseq` の特殊形式、また関数の引数部など、シンボルに値を束縛する全ての場所で動作します。destructuring 構文を関数の引数部に使うことで、ペアをとり、その位置をスワップする関数を作成できます。

```
(defn swap-pair [[fst snd]]
  [snd fst])

(swap-pair [1 2])
;; => [2 1]

(swap-pair '(3 4))
;; => [4 3]
```

ベクタを用いた位置の destructuring は、シーケンスからインデックス付きの値を取り出すために非常に便利ですが、シーケンス内の残りの要素を破棄したくない場合があります。可変長引数関数の引数を `&` で受け取る方法と同様に、`&` をベクタの destructuring 内で使用して、シーケンスの残りの部分をグループ化することができます。

```
(let [[fst snd & more] (range 10)]
  {:first fst
   :snd snd
   :rest more})
;; => {:first 0, :snd 1, :rest (2 3 4 5 6 7 8 9)}
```

0 のインデックスの値が `fst` に束縛され、1 のインデックスの値が `snd` に束縛され、2 以降のインデックスのシーケンスがシンボル `more` に束縛されていることに注目してください。

データ構造を destructutring する際に、データ構造全体に興味がある場合があります。これには `:as` キーワードを使うことで、任意の名前でデータ構造をシンボルに束縛することができます。destructuring 内で使用される場合、元のデータ構造は `:as` の後のシンボルに束縛されます。

```
(let [[fst snd & more :as original] (range 10)]
  {:first fst
   :snd snd
   :rest more
   :original original})
;; => {:first 0, :snd 1,
;;     :rest (2 3 4 5 6 7 8 9),
;;     :original (0 1 2 3 4 5 6 7 8 9)}
```

インデックス付きのシーケンスだけでなく、連想型のデータも destructuring することができますが、ベクタではなくマップを用いて束縛を行います。キーは値を束縛するためのシンボルであり、値は連想型データ構造で検索するキーです。例を見てみましょう。

```
(let [{language :language} {:language "ClojureScript"}]
  language)
;; => "ClojureScript"
```

上の例では、:language キーと関連づけられた値を取り出して、language シンボルに束縛しています。もし探しているキーが存在しない場合は nil が束縛されます。

```
(let [{name :name} {:language "ClojureScript"}]
  name)
;; => nil
```

連想型での destructuring では、束縛にデフォルト値を指定できます。このデフォルト値は、destructuring を行うデータ構造でキーが見つからない場合に使用されます。:or キーワードをマップの後につけることで、デフォルトの値が束縛されます。次の例を見てください。

```
(let [{name :name :or {name "Anonymous"}}
      {:language "ClojureScript"}]
  name)
;; => "Anonymous"

(let [{name :name :or {name "Anonymous"}} {:name "Cirilla"}]
  name)
;; => "Cirilla"
```

連想型の destructuring では、元のデータ構造を :as キーワードの後に置かれたシンボルに束縛することも可能です。

```
(let [{name :name :as person} {:name "Cirilla" :age 49}]
  [name person])
;; => ["Cirilla" {:name "Cirilla" :age 49}]
```

連想型データ構造のキーとして使えるのは、キーワードだけではありません。数字、文字列、シンボル、およびその他の多くのデータ構造もキーとして使えるため、これらを使用して構造を変更することもできます。シンボルが var 検索として解決されないようにするには、シンボルをクオートする必要があります。

```
(let [{one 1} {0 "zero" 1 "one"}]
  one)
;; => "one"
```

```
(let [{name "name"} {"name" "Cirilla"}]
  name)
;; => "Cirilla"

(let [{lang 'language} {'language "ClojureScript"}]
  lang)
;; => "ClojureScript"
```

キーに対応する値はシンボルと等しいことが多く (例えば :language の値を language シンボルに束縛する等)、キーはキーワード、文字列、シンボルであることが多いため、ClojureScript にはこれらのために略記法があります。

:keys を用いて キーワードの destructuring を始めます。以下に全ての例を示します。

```
(let [{:keys [name surname]} {:name "Cirilla" :surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

例の通り、:keys を用いて束縛内でシンボルのベクタと関連づけた場合、シンボルのキーワードに対応する値はそれらに対応づけられます。つまり、{:keys [name surname]} は {name :name surname :surname} の略記法であり、同じ意味です。

```
(let [{:strs [name surname]} {"name" "Cirilla" "surname" "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]

(let [{:syms [name surname]} {'name "Cirilla" 'surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

また、destructuring の興味深い特性は、任意で destructuring をネストできることです。これにより、コレクション上のネストされたデータにアクセスするコードは、コレクションの構造を模倣するため、非常に理解しやすくなります。

```
(let [{[fst snd] :languages} {:languages ["ClojureScript" "Clojure"]}]
  [snd fst])
;; => ["Clojure" "ClojureScript"]
```

3.10　スレッディングマクロ (Threading Macro)

スレッドマクロはアロー関数として知られています。スレッドマクロを使うことで、ネストされた複数の関数呼び出しを実行するときに、読みやすいコードを記述できます。

例えば (f (g (h x))) というコードにおいて、f は g の結果を第 1 引数として受け取ります。スレッドマクロ -> を使うことで、(-> x (h) (g) (f)) と書くことができます。こちらのほうが読みやすいですね。

スレッドマクロはマクロとして定義されているため、実行時のパフォーマンスに影響を与えません。(-> x (h) (g) (f)) はコンパイル時に (f (g (h x))) に変換されます。

h, g, f の括弧はなくてもかまいません。(f (g (h x))) は (-> x h g f) と省略して書くこともできます。

thread-first マクロ (->)

-> は thread-first マクロ と呼ばれますが、複数の式が実行されていく中で、最初の引数に値を挿入していくことから、そのような名前がつけられています。

では具体例として、まずはスレッドマクロを使わない場合を見てみましょう。

```
(def book {:name "Lady of the Lake"
           :readers 0})

(update (assoc book :age 1999) :readers inc)
;; => {:name "Lady of the lake" :age 1999 :readers 1}
```

thread-first マクロ (->) を使うと、次のように書くことができます。

```
(-> book
    (assoc :age 1999)
    (update :readers inc))
;; => {:name "Lady of the lake" :age 1 999 :readers 1}
```

このリーダマクロはデータ構造を変換するときに特に役にたちます。これは ClojureScript (Clojure) のデータ構造を変換する関数は、一貫して最初の引数でデータ構造を受け取るからです。

thread-last マクロ (->>)

thread-last マクロ (->>) と thread-first マクロ (->) の主な違いは、最初の引数にデータを入れるのではなく、最後の引数にデータを入れる点です。例を見てみましょう。

```
(def numbers [1 2 3 4 5 6 7 8 9 0])

(take 2 (filter odd? (map inc numbers)))
;; => (3 5)
```

上の例は thread-last マクロ (->>) を用いて、次のように書き直すことができます。

```
(->> numbers
     (map inc)
     (filter odd?)
     (take 2))
;; => (3 5)
```

thread-last マクロ (->>) は、シーケンスのデータを変形するときに役にたちます。ClojureScript のシーケンスとコレクションを操作する関数は、シーケンスやコレクションを最後の引数として受け取るためです。

thread-as マクロ (as->)

最後に -> と ->> どちらにも適用できない場合があります。この場合は、as-> マクロを使います。as-> マクロは、最初と最後に限らず、好きな場所にデータを入れることができます。

as-> マクロは 2 つの引数を固定でもち、その後に任意の数の式が続きます。-> と同様に、最初の引数は以降の式に渡されていく値です。2 番目の引数は束縛の名前です。以降の各々の式で、前の式の結果のためにその束縛の名前が使われます。

例を見てみましょう。

```
(as-> numbers $
  (map inc $)
  (filter odd? $)
  (first $)
  (hash-map :result $ :id 1))
;; => {:result 3 :id 1}
```

thread-some マクロ (some-> と some->>)

ClojureScript のより特殊なスレッド系マクロを 2 つ紹介します。これらは -> と ->> と同じように動作します。式の 1 つが nil と評価された場合に式を短絡することをサポートします。

例を見てみましょう。

```
(some-> (rand-nth [1 nil])
        (inc))
;; => 2

(some-> (rand-nth [1 nil])
        (inc))
;; => nil
```

これにより、ヌル・ポインタ・エクセプションを簡単に回避することができます。

thread-cond マクロ (cond-> と cond->>)

cond-> と cond->> マクロは -> と ->> と似ており、パイプラインからの処理を条件によりスキップできます。例を見てみましょう。

```
(defn describe-number
  [n]
  (cond-> []
    (odd? n) (conj "odd")
    (even? n) (conj "even")
    (zero? n) (conj "zero")
    (pos? n) (conj "positive")))

(describe-number 3)
;; => ["odd" "positive"]

(describe-number 4)
;; => ["even" "positive"]
```

対応する条件が論理的に真と評価された場合のみ、値が引き渡されます。

追加資料

詳しい情報は次の文献を参照してください。

- Lesser known Clojure: variants of threading macro[*2]
- Threading Macros Guide[*3]

[*2] http://www.spacjer.com/blog/2015/11/09/lesser-known-clojure-variants-of-threading-macro
[*3] http://clojure.org/guides/threading_macros

3.11　リーダーコンディショナル (Reader Conditionals)

この言語の特徴として、Clojure の異なる方言ごとに、コードを共有したり、プラットフォームに依存するコードを書くことができます。

リーダーコンディショナル (Reader Conditionals) を使うには、ファイルの拡張子を `.cljs` から `.cljc` に変更する必要があります。リーダの条件分けは拡張子が `.cljc` の場合にのみ動作します。

Standard (`#?`)

リーダーコンディショナルには、standard と splicing の 2 タイプがあります。standard の場合は、`cond` のように書くことができます。

```
(defn parse-int
  [v]
  #?(:clj  (Integer/parseInt v)
     :cljs (js/parseInt v)))
```

リーダマクロの `#?` は `cond` と同じように見えますが、条件部にキーワードを用いて、ClojureScript には `:cljs` 、Clojure には `:clj` をプラットフォームの特定のために使います。リーダマクロはコンパイル時に評価されるので、実行時にオーバーヘッドが発生しません。

Splicing (`#?@`)

リーダーコンディショナルの splicing は、standard と同じように使いますが、リストを結合 (splice) できます。このためにリーダマクロ `#?@` を使います。

```
(defn make-list
  []
  (list #?@(:clj  [5 6 7 8]
            :cljs [1 2 3 4])))

;; ClojureScript の場合
(make-list)
;; => (1 2 3 4)
```

ClojureScript のコンパイラは、上のコードをこのように解釈します。

```
(defn make-list
  []
  (list 1 2 3 4))
```

リーダーコンディショナルの splicing では、複数のトップレベルのフォームを結合することはできないため、次のコードは不正です。

```
#?@(:cljs [(defn func-a [] :a)
           (defn func-b [] :b)])
;; => #error "Reader conditional splicing not allowed at the top level."
```

そのようにするには、複数のフォームを使うか、do ブロックを使う必要があります。

```
#?(:cljs (defn func-a [] :a))
#?(:cljs (defn func-b [] :b))

;; または、

#?(:cljs
   (do
     (defn func-a [] :a)
     (defn func-b [] :b)))
```

参考文献

詳しい内容は以下の文献を参考にしてください。

- Reader Conditionals Guide[*4]
- Clojure Reader Conditionals by Example[*5]
- cuerdas - string manipulation library[*6]

[*4] http://clojure.org/guides/reader_conditionals
[*5] https://danielcompton.net/2015/06/10/clojure-reader-conditionals-by-example
[*6] https://github.com/funcool/cuerdas

3.12 名前空間

名前空間の定義

名前空間は、ClojureScript でコードをモジュール化する基本単位です。ClojureScript の名前空間は、Java のパッケージ、Ruby や Python のモジュールに似ており、ns マクロで定義できます。ClojureScript のソースを少し見たことがある人は、ファイルの最初に次のように書かれていることに気付いたかもしれません。

```
(ns myapp.core
  "Some docstring for the namespace.")

(def x "hello")
```

名前空間は動的であり、いつでも生成することができます。ただし、ファイルごとに 1 つの名前空間を使用するのが規則です。通常、名前空間の定義はファイルの先頭にあり、その後に オプションの docstring が続きます。

var とシンボルについては前に説明しました。定義する全ての var は、その名前空間に関連づけられます。具体的な名前空間を定義しない場合は、デフォルトの cljs.user が名前空間として使われます。

```
(def x "hello")
;; => #'cljs.user/x
```

他の名前空間の読み込み

名前空間を定義して、その中で変数を定義するのは非常に簡単ですが、他の名前空間のシンボルが使えなければ便利でありません。この目的のために、ns マクロは他の名前空間をロードする簡単な方法を提供します。

```
(ns myapp.main
  (:require myapp.core
            clojure.string))

(clojure.string/upper-case myapp.core/x)
;; => "HELLO"
```

ご覧のとおり、別の名前空間から var や関数にアクセスするためには、完全な修飾名 (名前空間 + 変数名) を使用します。

これにより他の名前空間にアクセスできますが、繰り返しが多く、冗長になりすぎます。名前空間の名前が長い場合は特に不便です。これを解決するには、:as ディレクティブを利用して、名前空間に追加の (通常より短い) エイリアスを作成します。その方法は次のとおりです。

```
(ns myapp.main
  (:require [myapp.core :as core]
            [clojure.string :as str]))

(str/upper-case core/x)
;; => "HELLO"
```

さらに ClojureScript では、:refer ディレクティブを用いることで、特定の名前空間から var や関数を参照することが容易になります。:refer の後には、特定の名前空間内の var を参照するシンボルのシーケンスを書きます。事実上、これらの var と関数はあなたの名前空間の一部のようになるため、これらに修飾子をつける必要はありません。

```
(ns myapp.main
  (:require [clojure.string :refer [upper-case]]))

(upper-case x)
;; => "HELLO"
```

最後に、cljs.core の中にある全てのものを把握する必要があります。cljs.core の名前空間は自動的にロードされるので、明示的に require すべきではありません。cljs.core で定義されている var と衝突する var を宣言する場合、ns マクロは特定のシンボルを除外して自動的にロードされないようにする別のディレクティブを提供します。

次の例を見てください。

```
(ns myapp.main
  (:refer-clojure :exclude [min]))

(defn min
  [x y]
  (if (> x y)
    y
    x))
```

ns マクロは、ホストのクラスを読み込むために :import ディレクティブ、マクロを読み込むために :refer-macros ディレクティブを用意しています。これらについては、別のセクションで説明します。

名前空間とファイル名

myapp.core のような名前空間がある場合、コードは myapp ディレクトリ内で core.cljs という名前のファイルにしなければなりません。上記の例では myapp.core と myapp.main を使用していますが、ファイル構造は次のようになります。

```
myapp
└── src
    └── myapp
        ├── core.cljs
        └── main.cljs
```

3.13 抽象化とポリモーフィズム

「インターフェイス等を用いてビジネスロジックの抽象化を上手く定義できたが、コントロールできない別のモジュールを処理する必要がでてきた」

過去にこのような状況を経験したことがあるかと思います。アダプターやプロキシ等の手法で対処することが考えられますが、複雑な作業を追加で行う必要があります。

一部の動的言語では「モンキーパッチ」が可能です。そのような言語では、クラスは開かれていて、いつでもメソッドを定義したり再定義したりできます。このテクニックは非常に悪い慣習であることもよく知られています。

サードパーティのライブラリをインポートするときに、使用しているメソッドを暗黙のうちに上書きできる言語は信頼できません。この場合、一貫した動作は期待できません。

これらの問題は Expression Problem として知られています。

詳細は、http://en.wikipedia.org/wiki/Expression_problem を参照してください。

プロトコル

「インターフェイス」を定義するための ClojureScript のプリミティブは、プロトコルと呼ばれています。プロトコルは名前と関数のセットで構成されます。全ての関数には、JavaScript の this、Python の self に対応する引数が少なくとも 1 つはあります。

プロトコルは、型に基づくポリモーフィズムを提供して、常に最初の引数によりディスパッチされます（前述の通り、JavaScript の this に相当します）。プロトコルは次のように記述します。

```
(ns myapp.testproto)

(defprotocol IProtocolName
  "プロトコルを記述する docstring"
  (sample-method [this] "この関数に関連付けられている doc 文字列"))
```

プロトコルを型と区別するために、接頭辞に I をつける慣習があります。Clojure のコミュニティでは、この接頭辞の使い方について様々な意見があります。私たちの意見では、名前衝突と混乱の可能性を避けるために許容できる解決策です。しかし、この接頭辞を使用しないことは悪しき慣行とはみなされません。

ユーザーからすると、プロトコル関数は、プロトコルが定義されている名前空間で定義された単純な関数です。これを用いると、競合する関数名を持つ同じ型のために実装された異なるプロトコル間での競合を回避するために、容易でシンプルなアプローチをとることができます。

次の例では、IInvertible という名前で、反転できるデータのためのプロトコルを作成します。

```
(defprotocol IInvertible
  "これは「反転可能な」データ型のためのプロトコルです。"
  (invert [this] "指定した項目を反転します"))
```

既存の型の拡張

プロトコルの大きな強みの 1 つは、既存の型やサードパーティの型を拡張できることです。この操作は、様々な方法で実行できます。ほとんどの場合、extend-type マクロか extend-protocol マクロを使います。extend-type の構文は次のように記述します。

```
(extend-type TypeA
  ProtocolA
  (function-from-protocol-a [this]
    ;; ここに実装を記述する
    )

  ProtocolB
  (function-from-protocol-b-1 [this parameter 1]
    ;; ここに実装を記述する
    )
  (function-from-protocol-b-2  [this parameter 1 parameter 2]
    ;; ここに実装を記述する
    ))
```

extend-type を使用すると、異なるプロトコルを使用して 1 つの型を 1 つの式で拡張できます。

先ほど定義した IInvertible プロトコルを試してみましょう。

```
(extend-type string
  IInvertible
  (invert [this] (apply str (reverse this))))
```

```
(extend-type cljs.core.List
  IInvertible
  (invert [this] (reverse this)))

(extend-type cljs.core.PersistentVector
  IInvertible
  (invert [this] (into [] (reverse this))))
```

プロトコルを文字列に対して拡張するために、js/String を使わずに、特別なシンボル string が使われていることに注目してください。これは、組み込みの JavaScript の型には特別な扱いがあり、string を js/String に置き換えると、コンパイラは警告を発生するからです。

プロトコルを JavaScript のプリミティブ型に拡張する場合、js/Number, js/String, js/Object, js/Array, js/Boolean, js/Function を使う代わりに、それぞれに特別なシンボル (number, string, object, array, boolean, function) を使う必要があります。

では、私たちのプロトコル実装を試してみましょう。

```
(invert "abc")
;; => "cba"

(invert 0)
;; => 0

(invert '(1 2 3))
;; => (3 2 1)

(invert [1 2 3])
;; => [3 2 1]
```

比較すると、extend-protocol はその逆を行います。つまり、あるプロトコルに対して、複数の型の実装を追加します。構文は次のようになります。

```
(extend-protocol ProtocolA
  TypeA
  (function-from-protocol-a [this]
    ;; ここに実装を記述する
    )

  TypeB
  (function-from-protocol-a [this]
    ;; ここに実装を記述する
    ))
```

したがって、前例は次のように書くこともできます。

```
(extend-protocol IInvertible
  string
  (invert [this] (apply str (reverse this)))

  cljs.core.List
  (invert [this] (reverse this))

  cljs.core.PersistentVector
  (invert [this] (into [] (reverse this))))
```

ClojureScript での抽象化

ClojureScript 自体は、プロトコルとして定義された抽象に基づいて構築されています。ClojureScript 言語自体のほぼ全ての動作は、サードパーティ製のライブラリに適用できます。実際の例を見てみましょう。

前のセクションでは、様々な組み込みのコレクションについて説明しました。次の例では、セットを使用します。次のコードを見てください。

```
(def mynums #{1 2})

(filter mynums [1 2 4 5 1 3 4 5])
;; => (1 2 1)
```

何が起こったのでしょうか。セット型は ClojureScript 内部の `IFn` プロトコルを実装しています。`IFn` プロトコルは、関数や呼び出し可能なもののための抽象化を表現します。このように、`filter` で呼び出し可能な述部のように使用できます。では、文字列のコレクションを `filter` する述語関数として正規表現を使用したい場合はどうなるでしょうか。

```
(filter #"^foo" ["haha" "foobar" "baz" "foobaz"])
;; TypeError: Cannot call undefined
```

`RegExp` の型は `IFn` プロトコルを実装していないので、例外が発生します。正規表現は呼び出せませんが、次のように簡単に修正することができます。

```
(extend-type js/RegExp
  IFn
  (-invoke
   ([this a]
     (re-find this a))))
```

この例を分析してみましょう。IFn プロトコルで invoke 関数を実装するように js/RegExp 型を拡張しています。正規表現を関数のように呼び出すには、関数のオブジェクトとパターンを指定して re-find 関数を呼び出します。

これで、正規表現のインスタンスを filter 操作の述語として使用できるようになります。

```
(filter #"^foo" ["haha" "foobar" "baz" "foobaz"])
;; => ("foobar" "foobaz")
```

プロトコルを用いた内部分析

ClojureScript には、実行時のイントロスペクション (introspection) を便利にする関数 satisfies? が用意されています。この関数の目的は、実行時に任意のオブジェクト (任意の型のインスタンス) が特定のプロトコルの基準を満たすかを判定することです。先ほどの例では、セットのインスタンスが IFn プロトコルを満たす場合、true を返します。

```
(satisfies? IFn #{1})
;; => true
```

マルチメソッド

ポリモーフィズムの非常に一般的なユースケースを解決するプロトコルについて前に説明しました。つまり、型によるディスパッチです。しかし、いくつかの状況では、プロトコルを用いたアプローチには制限があります。そのような場合、マルチメソッドが救いの手を差し伸べてくれます。

マルチメソッドは、型によるディスパッチに限定されません。代わりに、複数の引数の型と値によるディスパッチを提供します。また、特定の目的のために階層性を定義できます。また、プロトコルと同様にマルチメソッドは「開かれたシステム」なので、ユーザやサードパーティは新しい型のためにマルチメソッドを拡張できます。

マルチメソッドのための基本的な構文は defmulti と defmethod です。defmulti は、イニシャル・ディスパッチ関数 (initial dispatch function) を用いてマルチメソッドを作成するために使います。次はそのモデルです。

```
(defmulti say-hello
  "言語キーに応じて挨拶のメッセージを返すポリモーフィズム的な関数。
   デフォルトの言語キーを en に設定します。"
  (fn [param] (:locale param))
  :default :en)
```

defmulti 内で定義されてい無名関数は、ディスパッチ関数です。この関数は say-hello 関数を呼び出すたびに呼び出されて、ディスパッチに使用される何らかのマーカーオブジェクトを返します。この例では、最初の引数の :locale キーの内容を返します。

最後に、実装を追加します。defmethod で実装を定義します。

```
(defmethod say-hello :en
  [person]
  (str "Hello " (:name person "Anonymous")))

(defmethod say-hello :es
  [person]
  (str "Hola " (:name person "An ó nimo")))
```

もし :locale と (オプションの):name キーを含むハッシュマップに対してこの関数を実行すると、マルチメソッドはまずディスパッチ関数を呼び出してディスパッチ値を決定し、次にその値の実装を検索します。実装が見つかった場合、ディスパッチャはそれを実行します。それ以外の場合、ディスパッチはデフォルトの実装 (指定されている場合) を検索して実行します。

```
(say-hello {:locale :es})
;; => "Hola An ó nimo"

(say-hello {:locale :en :name "Ciri"})
;; => "Hello Ciri"

(say-hello {:locale :fr})
;; => "Hello Anonymous"
```

デフォルトの実装が指定されていない場合は、一部の値がそのマルチメソッドの実装をもたないことを通知する例外が発生します。

ヒエラルキー

ヒエラルキーは、ClojureScript において、ドメインに必要なリレーションを構築することができます。ヒエラルキーは、シンボル、キーワード、型のような名前付きのオブジェクト間のリレーションとして定義されます。

ヒエラルキーは、必要に応じてグローバルまたはローカルに定義できます。マルチメソッドのように、ヒエラルキーは単一の名前空間に限定されません。ヒエラルキーは、定義されている名前空間だけでなく、任意の名前空間から拡張できます。

グローバルの名前空間はより制限されていますが、これは正当な理由によるものです。名前空間を持たないキーワードまたはシンボルは、グローバルのヒエラルキーでは使用できません。この動作は、複数のサードパーティ製のライブラリが異なるセマンティクスに対して同じシンボルを使用する場合、予期しない状況が発生するのを防ぐのに役立ちます。

■ヒエラルキーの定義 ヒエラルキーのリレーションは、derive 関数を用いて確立する必要があります。

```
(derive ::circle ::shape)
(derive ::box ::shape)
```

ここでは、名前空間を持つキーワード間のリレーションを定義しました。この場合、::circle は ::shape の子であり、::box も ::shape の子です。

::circle は :current.ns/circle の省略形なので、REPL で実行する場合、::circle は :current.ns/circle として評価されます。

■ヒエラルキーとイントロスペクション ClojureScript には、グローバルあるいはローカルに定義されたヒエラルキーのイントロスペクションを実行時に可能にする小さなツールセットが用意されています。このツールセットは 3 つの関数で構成されています。isa?, ancestors, descendants です。前例で定義したヒエラルキーを用いて、このツールセットの使い方を見てみましょう。

```
(ancestors ::box)
;; => #{:cljs.user/shape}

(descendants ::shape)
;; => #{:cljs.user/circle :cljs.user/box}

(isa? ::box ::shape)
;; => true

(isa? ::rect ::shape)
;; => false
```

■ローカル環境に定義されたヒエラルキー 前に説明した通り、ClojureScript はローカルのヒエラルキーを定義することもできます。これは、make-hierarchy 関数により行うことができます。次に、前例をローカルのヒエラルキーを用いて書き直してみましょう。

```
(def h (-> (make-hierarchy)
           (derive :box :shape)
           (derive :circle :shape)))
```

これで、ローカルに定義されたヒエラルキーで同じ関数を使用できます。

```
(isa? h :box :shape)
;; => true
```

```
(isa? :box :shape)
;; => false
```

ご覧のように、ローカルのヒエラルキーでは、通常の (名前空間が修飾されない) キーワードを使用できます。もし isa? をローカルのヒエラルキーの引数を渡さずに実行すると、予想した通りに false を返します。

■マルチスレッドにおけるヒエラルキー ヒエラルキーの大きな利点の 1 つは、マルチメソッドと上手く働くことです。これは、マルチメソッドがデフォルトで isa? 関数をディスパッチの最終ステップで使うからです。

その意味を明確に理解するために例を見てみましょう。初めに、defmulti を用いてマルチメソッドを定義します。

```
(defmulti stringify-shape
  "shape キーワードを人が読めるような表現で出力する関数"
  identity
  :hierarchy #'h)
```

:hierarchy キーワードパラメータを使用して、使用するヒエラルキーをマルチメソッドに指定します。指定しない場合は、グローバルのヒエラルキーが使用されます。

次に、defmethod を用いて、マルチメソッドのための実装を定義します。

```
(defmethod stringify-shape :box
  [_]
  "A box shape")

(defmethod stringify-shape :shape
  [_]
  "A generic shape")

(defmethod stringify-shape :default
  [_]
  "Unexpected object")
```

では、関数を :box で呼び出したときに何が起こるかを見てみましょう。

```
(stringify-shape :box)
;; => "A box shape"
```

全て予想通りに、マルチメソッドは与えられた引数に適する実装を直接実行しました。次に :circle を用いて同じ関数を呼び出してみましょう。この場合、:circle に合致する実装はありません。

```
(stringify-shape :circle)
;; => "A generic shape"
```

マルチメソッドは指定されたヒエラルキーを用いて自動的に解決されます。:circle は :shape の子孫 (descendant) であるため、:shape の実装が呼び出されます。

ヒエラルキーの一部ではないキーワードを指定すると、:default の実装が呼び出されます。

```
(stringify-shape :triangle)
;; => "Unexpected object"
```

3.14 データ型

ここまでは、マップ、セット、リスト、ベクタをデータを表現するために使ってきました。多くの場合、これは本当に素晴らしいアプローチです。しかし、自分自身でこのような型を定義する必要があるときもあります。この本では、このような型のことを「データ型」と呼びます。

これまでは、マップ、セット、リスト、ベクタを使ってデータを表現してきました。ほとんどの場合、これは本当に素晴らしいアプローチです。しかし、場合によっては、独自に型を定義する必要があります。本書ではそれらをデータ型と呼ぶことにします。

データ型は、次の機能を提供します。

- ホスト環境が支援する独自の型 (名前付きまたは無名)
- プロトコルを実装する機能 (インライン)
- フィールドまたはクロージャを用いた明示的に宣言された構造
- マップのような動作 (レコードを経由。詳細は後ほど。)

Deftype

ClojureScript で独自の型を作成するための最も低レベルな構文は deftype マクロです。例として、User という型を定義してみましょう。

```
(deftype User [firstname lastname])
```

型を定義すると、User のインスタンスを作成できます。次の例では、User の後のドット . はコンストラクタを呼び出していることを示します。

```
(def person (User. "Triss" "Merigold"))
```

フィールドには .- でアクセスできます。

```
(.-firstname person)
;; => "Triss"
```

deftype で定義された型は、ホスト環境に支援されるクラスのようなオブジェクトを現在の名前空間と関連づけて作成します。これは defrecord で定義される型も同様です。defrecord については後に取り上げます。利便性のために ClojureScript は、:require ディレクティブでインポートできる ->User というコンストラクタ関数も定義しています。

私たちは個人的にはこのタイプの関数は好きではないため、より慣用的な名前で独自のコンストラクターを定義したいと思います。

```
(defn make-user
  [firstname lastname]
  (User. firstname lastname))
```

私たちのコードでは ->User の代わりにこちらを使います。

Defrecord

レコード (record) は、ClojureScript で型を定義するための少し高いレベルの抽象化であり、型の定義のために推奨されるべき方法です。

私たちが知っているように、ClojureScript ではマップなどのプレーンなデータ型を使用する傾向がありますが、ほとんどの場合、アプリケーションの実体を表す名前付きの型が必要です。そこでレコードの出番です。

レコードはマッププロトコルを実装するデータ型であるため、マップと同様に使用できます。また、レコードも適切な型であるため、プロトコルを介して型ベースのポリモーフィズムをサポートします。要約すると、レコードを用いると、様々な抽象化で働くマップという、両方の世界の良い点をもつことができます。

レコードを使用して User 型の定義を始めましょう。

```
(defrecord User [firstname lastname])
```

deftype の構文とよく似ています。実際には、型を定義するための下位レベルのプリミティブとして、舞台裏では deftype を使用しています。

ここで、フィールドへのアクセスについて、素の型の場合との違いを見てみましょう。

```
(def person (User. "Yennefer" "of Vengerberg"))

(:firstname person)
;; => "Yennefer"

(get person :firstname)
;; => "Yennefer"
```

レコードはマップであり、マップのように扱うことができます。前述の通り、レコードはマップであり、マップのように動作します。

```
(map? person)
;; => true
```

また、マップと同様に、レコードは最初に定義されていないフィールドを追加できます。

```
(def person2 (assoc person :age 92))

(:age person2)
;; => 92
```

このように、assoc 関数は期待どおりに動作し、同じ型の新しいインスタンスを、新しいキーと値のペアで返します。ただし、dissoc には注意してください。レコードでの動作は、マップでの動作とは少し異なります。外すフィールドがオプションのフィールドである場合は新しいレコードを返しますが、必須のフィールドを外す場合はマップを返します。マップとのもう 1 つの違いは、レコードが関数のように動作しないことです。

```
(def plain-person {:firstname "Yennefer", :lastname "of Vengerberg"})

(plain-person :firstname)  ;; => "Yennefer"

(person :firstname)        ;; => person.User does not implement IFn protocol.
```

利便性のために、deftype と同様に、defrecord マクロは ->User 関数と追加の map->User コンストラクタ関数を公開しています。このコンストラクタについては、deftype で定義したコンストラクタと同じ意見です。他のコンストラクタを使用する代わりに、独自のコンストラクタを定義することをお勧めします。それらが存在する場合、どのように使用できるかを見てみましょう。

```
(def cirilla (->User "Cirilla" "Fiona"))
(def yen (map->User {:firstname "Yennefer"
                     :lastname "of Vengerberg"}))
```

プロトコルの実装

これまで見てきた両方の型定義のプリミティブは、(前に説明した通り) プロトコルのインライン実装を可能にします。例として、次のように定義します。

```
(defprotocol IUser
  "User 型を操作するための共通の抽象化"
  (full-name [_] "Get the full name of the user."))
```

ここでは、抽象化のために、インライン実装 (この場合は IUser) を使用して型を定義できます。

```
(defrecord User [firstname lastname]
  IUser
  (full-name [_]
    (str firstname " " lastname)))

;; Create an instance.
(def user (User. "Yennefer" "of Vengerberg"))

(full-name user)
;; => "Yennefer of Vengerberg"
```

Reify

reify マクロは、型を事前に定義せずにオブジェクトを作成するために使用できる特別なコンストラクタです。プロトコルの実装は deftype や defrecord と同じように提供されますが、対照的に reify はアクセス可能なフィールドをもちません。

IUser の抽象化でうまく機能するユーザ型のインスタンスをエミュレートするには、次のようにします。

```
(defn user
  [firstname lastname]
  (reify
    IUser
    (full-name [_]
      (str firstname " " lastname))))

(def yen (user "Yennefer" "of Vengerberg"))
(full-name yen)
;; => "Yennefer of Vengerberg"
```

Specify

specify! は reify の高度な代替手段であり、既存の JavaScript オブジェクトにプロトコルの実装を追加できます。これは、JavaScript のライブラリのコンポーネントにプロトコルを移植する場合に便利です。

```
(def obj #js {})

(specify! obj
  IUser
  (full-name [_]
    "my full name"))

(full-name obj)
;; => "my full name"
```

specify は specify! のイミュータブル版です。これは ICloneable を実装しているイミュータブルでコピー可能な値 (ClojureScript のコレクション等) に対して使えます。

```
(def a {})

(def b (specify a
         IUser
         (full-name [_]
           "my full name")))

(full-name a)
;; Error: No protocol method IUser.full-name
;; defined for type cljs.core/PersistentArrayMap: {}

(full-name b)
;; => "my full name"
```

3.15 ホスト環境の相互運用性

ClojureScript は、その兄弟である Clojure と同様に、ゲスト言語として設計されています。これは、ClojureScript にとっては JavaScript、Clojure にとっては JVM といった既存のエコシステムの上で、言語の設計がうまく機能することを意味します。

型

ClojureScript は、予想と異なるかもしれませんが、プラットホームが提供する全ての型を利用しようとします。次は、ClojureScript の基礎となるプラットフォームから継承して再利用する機能の一覧です。

- ClojureScript の string は、JavaScript の String である。

- ClojureScript の number は、JavaScript の Number である。

- ClojureScript の `nil` は、JavaScript の null である。

- ClojureScript の正規表現は、JavaScript の `RegExp` のインスタンスである。

- ClojureScript はインタープリタで実行されず、常に JavaScript にコンパイルされる。

- ClojureScript はプラットフォームの API を同じセマンティクスで簡単に呼び出せる。

- ClojureScript のデータ型は、内部的には JavaScript のオブジェクトにコンパイルされる。

ClojureScript はその上に、ベクタ、マップ、セットなど、本章のこれまでのセクションで説明したプラットフォームに存在しない独自の抽象化や型を構築します。

プラットホームの型との連携

ClojureScript には、オブジェクトメソッドの呼び出し、新しいインスタンスの作成、オブジェクトプロパティへのアクセスなど、プラットフォームの型との対話を可能にするスペシャルフォームの小さなセットが付属しています。

プラットホームへのアクセス

ClojureScript には、特殊な名前空間 `js/` を通してプラットフォームの環境全体にアクセスできます。JavaScript の組み込み関数 `parseInt` を実行する式は次の通りです。

```
(js/parseInt "222")
;; => 222
```

新たにインスタンスを生成する

ClojureScript にはインスタンスを作成する方法が 2 通りあります。まずは、new を使う方法です。

```
(new js/RegExp "^foo$")
```

ドット . を使うこともできます。

```
(js/RegExp. "^foo$")
```

後者の方法でインスタンスを作成することをお勧めします。2 つの形式の間に違いはありませんが、ClojureScript コミュニティでは後者の形式が最も頻繁に使用されています。

インスタンスのメソッド呼び出し

インスタンスメソッドを呼び出すには、JavaScript の方法 (obj.method() の形式) とは異なり、Lisp 系言語の他の標準関数と同じように最初に来ますが、関数名は特殊な形式 . で始まります。

正規表現のインスタンスの .test() メソッドを呼び出すには次のようにします。

```
(def re (js/RegExp "^Clojure"))

(.test re "ClojureScript")
;; => true
```

JavaScript のオブジェクトに対してインスタンスメソッドを呼び出すことができます。以下の例において、前者はこれまで見てきたパターンに従っています。後者は前者のショートカットです。

```
(.sqrt js/Math 2)
;; => 1.4142135623730951
(js/Math.sqrt 2)
;; => 1.4142135623730951
```

オブジェクトのプロパティへのアクセス

オブジェクトのプロパティにアクセスする方法は、メソッドを呼び出す方法とほぼ同じです。違いは、ドット . の代わりに ドットハイフン .- を使うことです。例を見てみましょう。

```
(.-multiline re)
;; => false
(.-PI js/Math)
;; => 3.141592653589793
```

プロパティへアクセスするための略記法

js/ で始まるシンボルは、ネストされたプロパティのアクセスを表示するためにドットを含むことができます。次の例では、両者とも同じ関数を呼び出します。

```
(.log js/console "Hello World")

(js/console.log "Hello World")
```

次の式は同じプロパティにアクセスします。

```
(.-PI js/Math)
;; => 3.141592653589793

js/Math.PI
;; => 3.141592653589793
```

JavaScript のオブジェクト

ClojureScript には、プレーンな JavaScript のオブジェクトを作成する様々な方法があり、各々に目的があります。基本的なものは `js-obj` 関数です。このメソッドは、可変数のキーと値のペアを受け入れて、JavaScript のオブジェクトを返します。

```
(js-obj "country" "FR")
;; => #js {:country "FR"}
```

返り値は、プレーンな JavaScript のオブジェクトを受け入れる何らかのサードパーティのライブラリに渡すことができますが、この関数の戻り値の実際の表現を見ることができます。まったく同じことをするための別の形式です。

リーダーマクロの `#js` を使うことも可能です。マップかベクタの先頭に `#js` をつけることで、結果が JavaScript のオブジェクトに変換されます。

```
(def myobj #js {:country "FR"})
```

上の例で変換されるものは、次のコードのようになります。

```
var myobj = {country: "FR"};
```

前のセクションで説明した通り、オブジェクトのプロパティにはドットハイフン `.-` を用いてアクセスできます。

```
(.-country myobj)
;; => "FR"
```

JavaScript のオブジェクトはミュータブルなので、`set!` 関数で新たな値を設定できます。

```
(set! (.-country myobj) "KR")
```

変換

前述のフォームの不便な点は、再帰的な変換を行わないため、ネストされたオブジェクトがある場合、ネストされたオブジェクトは変換されないことです。ClojureScript のマップを使う場合と、JavaScript のマップを使う場合を比べてみましょう。

```
(def clj-map {:country {:code "FR" :name "France"}})
;; => {:country {:code "FR", :name "France"}}
(:code (:country clj-map))
;; => "FR"
```

```
(def js-obj #js {:country {:code "FR" :name "France"}})
;; => #js {:country {:code "FR", :name "France"}
(.-country js-obj)
;; => {:code "FR", :name "France"}
(.-code (.-country js-obj))
;; => nil
```

このユースケースを解決するために、clj->js と js->clj が用意されています。それぞれ、ClojureScript と JavaScript のコレクションを相互に変換します。ClojureScript への変換で :country キーワードが文字列に変換されている点に注意してください。

```
(clj->js {:foo {:bar "baz"}})
;; => #js {:foo #js {:bar "baz"}}
(js->clj #js {:country {:code "FR" :name "France"}}))
;; => {"country" {:code "FR", :name "France"}}
```

配列の場合、期待通りに動作する into-array という特別な関数があります。

```
(into-array ["France" "Korea" "Peru"])
;; => #js ["France" "Korea" "Peru"]
```

配列

前例では、既存の ClojureScript コレクションから配列を作成しましたが、別の方法として make-array 関数を使う方法もあります。次の例では、長さが 10 の事前に割り当てられた配列を作成します。

```
(def a (make-array 10))
;; => #js [nil nil nil nil nil nil nil nil nil nil]
```

ClojureScript では、配列はシーケンスの抽象化にも適しているため、配列を繰り返し処理したり、count 関数で単純に要素の数を取得したりすることができます。

```
(count a)
;; => 10
```

JavaScript のプラットフォームでは、配列はミュータブルなコレクション型であるため、特定のインデックスにアクセスし、その位置に値を設定できます。

```
(aset a 0 2)
;; => 2
a
;; => #js [2 nil nil nil nil nil nil nil nil nil]
```

もしくは、値を得るために、インデックスを指定してアクセスできます。

```
(aget a 0)
;; => 2
```

JavaScript では、配列においてインデックスでアクセスすることは、オブジェクトのプロパティへのアクセスと同じであるため、プレーンなオブジェクトと連携するために同じ関数を使うことができます。

```
(def b #js {:hour 16})
;; => #js {:hour 16}

(aget b "hour")
;; => 16

(aset b "minute" 22)
;; => 22

b
;; => #js {:hour 16, :minute 22}
```

3.16 状態管理

ClojureScript の基本的なアイデアの 1 つがイミュータブルな性質であることを学びました。ClojureScript ではスカラー値もコレクションもイミュータブルですが、`Date` のように JS のホスト環境に存在する可変型は例外です。

イミュータブルな性質は多くの優れた特性がありますが、時とともに変化する値をモデル化する必要に迫られることもあります。データ構造を適切に変更できない場合、これを実現するにはどうすればよいでしょうか。

var

var は名前空間内で自由に再定義できますが、いつ変更されるかを知る方法はありません。var を他の名前空間から再定義できないことは、やや制限的です。また、状態を変更している場合は、それがいつ発生するかを知りたいと思うでしょう。

アトム

自由に変更できる値を含むオブジェクトとして、ClojureScript は `Atom` 型を提供します。値を変更するだけでなく、付加したり切り離したりできる watcher 関数による監視や、アトムに含まれる値が常に有効であることを確認するバリデーションもサポートします。

もし Ciri という名前の人に対応するアイデンティティをモデル化するとすれば、Ciri のデータを含むイミュータブルな値を 1 つのアトムでラップすることができます。アトムの値は、`deref` 関数またはその短縮表記 `@` を使用して取得できます。

```
(def ciri (atom {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Atom: {:name "Cirilla", :lastname "Fiona", :age 20}>

(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 20}
```

`swap!` 関数を値を変更するためにアトムに使うことができます。Ciri の誕生日が今日なので、彼女の年齢 age に加算しましょう。

```
(swap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 21}
```

`reset!` 関数は、アトムに含まれる値を新しい値で置き換えます。

```
(reset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

監視

アトムの監視関数を追加したり削除したりできます。アトムの値が swap! や reset! によって変更されるたびに、アトムの監視関数を呼び出されます。監視は、add-watch 関数を用いて追加できます。それぞれの watcher には、後でアトムから監視を削除するために使用されるキー (例では :logger) が関連付けられていることに注意してください。

```
(def a (atom))
(add-watch a :logger (fn [key the-atom old-value new-value]
                       (println "Key:" key "Old:" old-value "New:" new-value)))

(reset! a 42)
;; Key: :logger Old: nil New: 42
;; => 42

(swap! a inc)
;; Key: :logger Old: 42  New: 43
;; => 43

(remove-watch a :logger)
```

Volatile

Volatile は、アトムと同様に、変更可能な値を含むオブジェクトです。しかし、Volatile はアトムが提供するような監視やバリデーションの機能をもっていません。これにより、多少パフォーマンスが向上するため、監視やバリデーションが不要なミュータブルなコンテナに向いています。

Volatile の API はアトムのものとよく似ています。それらは、含まれる値を取得するために、参照を解除して値を取得できます。vswap! でスワップを、vreset! でリセットをサポートします。

```
(def ciri (volatile! {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Volatile: {:name "Cirilla", :lastname "Fiona", :age 20}>
(volatile? ciri)
;; => true
(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}

(vswap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}
(vreset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

アトムとのもう 1 つの違いは、Volatiles のコンストラクタは、最後に ! をつける点です。Volatile を作成するには volatile! を、アトムを作るには atom を使います。

第 4 章

ツールとコンパイラ

本章では、ClojureScript を用いた開発を容易にするための既存のツールについて簡単に紹介します。前章とは異なり、本章の内容は、それぞれが独立した内容になっています。

4.1 コンパイラの概要

前章の言語自体の理論的な説明にうんざりしてしまって、何かコードを書いて実行したいと思われているかもしれません。このセクションの目的は、ClojureScript コンパイラの実用的な紹介を少し行うことです。

ClojureScript のコンパイラは、多くのディレクトリと名前空間で分けられたソースコードを取得して、JavaScript にコンパイルします。今日では JavaScript を実行できる環境は多くあり、それぞれに固有の特性があります。

この章では、ツールを追加せずに ClojureScript を使う方法を説明します。これは、他のツール (Leiningen[*1] や [*2]) が利用できない場合において、コンパイラの動作や利用法を理解するのに役立ちます。

実行環境

実行環境とは何でしょうか。実行環境は、JavaScript を実行できるエンジンです。例えば、最も一般的な実行環境はブラウザ (Chrome、Firefox 等) であり、次に Node.js[*3] が続きます。

他にも、Rhino(JDK6 以降)、Nashorn(JDK8 以降)、QtQuick(QT) 等の実行環境がありますが、どれも最初の 2 つと大きな違いはありません。ですので、今のところ ClojureScript は、ブラウザや Node.js ですぐに実行できるコードにコンパイルするでしょう。

コンパイラのダウンロード

ClojureScript はセルフホストですが、ClojureScript を使用する最善の方法は JVM 実装を使用することです。これを使用するには、JDK8 がインストールされている必要があります。ClojureScript

[*1] http://leiningen.org
[*2] http://boot-clj.com
[*3] https://Node.js.org

自体に必要なのは JDK7 のみですが、この章で使用するスタンドアロンコンパイラには JDK8 が必要です。JDK8 は http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html にあります。

最新の ClojureScript コンパイラは、wget を使用してダウンロードできます。

```
wget https://github.com/clojure/clojurescript/releases/download/r1.9.36/cljs.jar
```

ClojureScript コンパイラはスタンドアロンの実行可能な jar ファイルにパッケージされているため、ClojureScript のソースコードを JavaScript にコンパイルするために必要なファイルはこれと JDK8 のみです。

Node.js へのコンパイル

まず、Node.js をターゲットにコードをコンパイルする例から始めましょう。この例では、Node.js をインストールする必要があります。

Node.js をインストールする方法はいくつかありますが、Node.js のバージョンマネージャーの nvm を使用することをお勧めします。インストール方法と nvm の使用法については、nvm のドキュメント[*4]を参照してください。

nvm をインストールしたら、最新バージョンの Node.js をインストールします。

```
$ nvm install v6.2.0
$ nvm alias default v6.2.0
```

次のコマンドで Node.js がインストールされているかをテストしましょう。

```
$ node --version
v6.2.0
```

■アプリケーションの作成 実用的な例を作成する最初のステップとして、アプリケーションのためのディレクトリを作成します。実用例に使うコードをそこに取り込んでいきます。

まず、"Hello world" アプリケーションのためのディレクトリを作成しましょう。

```
$ mkdir -p myapp/src/myapp
$ touch myapp/src/myapp/core.cljs
```

[*4] https://github.com/creationix/nvm

次のようなディレクトリ構造にします。

```
myapp
└── src
    └── myapp
        └── core.cljs
```

次に、前に作成した myapp/src/myapp/core.cljs に次のコードを書き込みます。

```
(ns myapp.core
  (:require [cljs.nodejs :as nodejs]))

(nodejs/enable-util-print!)

(defn -main
  [& args]
  (println "Hello world!"))

(set! *main-cli-fn* -main)
```

ファイル内で宣言された名前空間が、ディレクトリの構造と正確に一致することが非常に重要です。これは、ClojureScript がソースコードを構築する方法です。

■アプリケーションのコンパイル このソース・コードをコンパイルするためには、ClojureScript コンパイラにソース・ディレクトリと出力ファイルを伝える単純なビルド・スクリプトが必要です。ClojureScript には他にも多くの選択肢がありますが、現時点では取り上げません。

myapp/build.clj を作成します。このファイルの内容は次のとおりです。

```
(require '[cljs.build.api :as b])

(b/build "src"
 {:main 'myapp.core
  :output-to "main.js"
  :output-dir "out"
  :target :nodejs
  :verbose true})
```

ここでは、この例で使用されているコンパイラのオプションについて簡単に説明します。

- :output-to は、コンパイルされたコードの出力先を指定します。この場合 main.js です。
- :main は、実行時にエントリーポイントとして動作する名前空間を指定します。
- :target は、コンパイルされたコードを実行するプラットフォームを示します。この例では、Node.js を使用します。このパラメーターを省略した場合、ソースはコンパイルされてブラウザの環境で実行されます。

コンパイルを実行するには、次のコマンドを実行します。

```
$ cd myapp
$ java -cp ../cljs.jar:src clojure.main build.clj
```

コンパイル後、node コマンドでコンパイルされたファイルを実行します。

```
$ node main.js
Hello world!
```

ブラウザへのコンパイル

このセクションでは、前のセクションの "Hello world" の例と同様のアプリケーションを作成し、ブラウザの環境で実行します。このアプリケーションの最小要件は、JavaScript を実行できるブラウザだけです。

プロセスはほぼ同じで、ディレクトリの構造も同じです。変更するのは、アプリケーションのエントリーポイントとビルド・スクリプトだけです。前の例のディレクトリ・ツリーを元に、別のディレクトリを再生成します。

```
$ mkdir -p mywebapp/src/mywebapp
$ touch mywebapp/src/mywebapp/core.cljs
```

結果、次のディレクトリ構造になります。

```
mywebapp
└── src
    └── mywebapp
        └── core.cljs
```

次に、新しいコンテンツを `mywebapp/src/mywebapp/core.cljs` に書き込みます。

```
(ns mywebapp.core)

(enable-console-print!)

(println "Hello world!")
```

ブラウザの環境では、アプリケーションのための特定のエントリーポイントは必要ないため、全体の名前空間がエントリーポイントになります。

■アプリケーションのコンパイル ソースコードをコンパイルしてブラウザで正しく動作させるには、mywebapp/build.clj を次のように上書きします。

```
(require '[cljs.build.api :as b])

(b/build "src" {:output-to "main.js"
                :source-map true
                :output-dir "out/"
                :main 'mywebapp.core
                :verbose true
                :optimizations :none})
```

ここで使用するコンパイラオプションについて簡単に説明します。

- :output-to は、コンパイルされたコードの出力先を指定します。この場合は main.js です。
- :main は、実行時にエントリーポイントとして動作する名前空間を指定します。
- :source-map は、ソースマップの出力先を指定します。ソースマップは、ClojureScript のソースを、生成された JavaScript に接続して、エラー・メッセージが元のソースを示すようにします。
- :output-dir は、コンパイルで使われる全てのソースを保存するディレクトリを指定します。ソースマップをソースだけでなく、残り全てのコードと正しく動作させるためのものです。
- :optimizations はコンパイルの最適化を示します。このオプションには異なる値がありますが、これについては後のセクションで詳しく説明します。

コンパイルを実行するには、次のコマンドを実行します。

```
cd mywebapp
java -cp ../cljs.jar:src clojure.main build.clj
```

この処理には時間がかかることがありますが、心配せずに少し待ってください。Clojure コンパイラを使用した JVM のブートストラップは少し遅いです。以降のセクションでは、このような遅いプロセスの起動と停止を頻繁に行わないように、監視プロセスを開始する方法を説明します。

コンパイルを待つ間に、ダミーの HTML ファイルを作成して、ブラウザでサンプル・アプリケーションを簡単に実行できるように、ダミーの HTML ファイルを作成しましょう。以下の内容を mywebapp ディレクトリの index.html に書き込みます。

```
<!DOCTYPE html>
<html>
  <header>
    <meta charset="utf-8" />
    <title>Hello World from ClojureScript</title>
  </header>
  <body>
    <script src="main.js"></script>
  </body>
</html>
```

コンパイルが完了して基本的な HTML ファイルができたら、好きなブラウザで開いて、開発ツールのコンソールで調べることができます。"Hello world!" というメッセージが表示されます。

監視プロセス

ClojureScript コンパイラの起動時間が遅いことに既に気づいているかもしれません。この問題を解決するために、ClojureScript のコンパイラには、ソースコードの変更を監視して、ファイルの変更がディスクに書き込まれると同時に再コンパイルするツールが付属しています。

別のビルド・スクリプトを watch.clj という名前で作成します。

```
(require '[cljs.build.api :as b])

(b/watch "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map true
  :main 'mywebapp.core
  :optimizations :none})
```

ここで、前のセクションと同じようにスクリプトを実行します。

```
$ java -cp ../cljs.jar:src clojure.main watch.clj
Building ...
Reading analysis cache for jar:file:/home/niwi/cljsbook/playground/cljs.jar!/cljs/core.cl
Compiling src/mywebapp/core.cljs
Compiling out/cljs/core.cljs
Using cached cljs.core out/cljs/core.cljs
... done. Elapsed 0.754487937 seconds
Watching paths: /home/niwi/cljsbook/playground/mywebapp/src
```

名前空間を mywebapp.core に戻して、出力テキストを Hello World, Again! に変更すると、src/mywebapp/core.cljs がすぐにコンパイルされます。ブラウザで index.html をリロードすると、新しい出力がコンソールに表示されます。この方法の別の利点は、出力が少し増えることです。

最適化レベル

ClojureScript コンパイラにはさまざまな最適化レベルがあります。これらのコンパイルレベルは舞台裏で行われますが、Google Closure Compiler によります。

コンパイルのプロセスは次の通りです。

1. リーダーはコードを読み、何らかの分析を行います。この段階でコンパイラは警告を発する場合があります。
2. その後、ClojureScript コンパイラは ClojureScript の 1 ファイルごとに 1 つの JavaScript の出力ファイルを作成します。
3. 生成された JavaScript のファイルは Google Closure コンパイラに渡され、最適化レベルや他のオプション (ソースマップや output デイレクトリ) に応じて最終的な出力ファイルが生成されます。

最終的な出力形式は、選択した最適化レベルによって異なります。

■none　この最適化レベルでは、生成された JavaScript は名前空間ごとに別々の出力ファイルに書き込まれ、コードへの追加の変換はありません。

■whitespace　この最適化レベルでは、生成された JavaScript ファイルが依存関係の順で出力ファイルに連結されます。改行やその他の空白は削除されます。

これによりコンパイル速度が多少低下し、コンパイルが遅くなります。ただし、処理速度はそれほど遅くはなく、中小規模のアプリケーションには十分に使用できます。

■simple　単純なコンパイルレベルは、`whitespace` 最適化レベルからの作業に基づいて構築されます。ローカル変数や関数パラメータの名前をより短い名前に変更するなど、式や関数内でさらに最適化を実行します。

`:simple` 最適化によるコンパイルは、常に構文的に有効な JavaScript の機能を保持するため、コンパイルされた ClojureScript と他の JavaScript との間の相互作用に影響はありません。

■advanced　高度なコンパイルレベルは `simple` 最適化レベルの上に構築され、さらに積極的な最適化とデッドコードの削除を実行します。これにより、出力ファイルが大幅に小さくなります。

`:advanced` 最適化は、Google Closure Compiler ルールに従った JavaScript の厳密なサブセットでのみ動作します。ClojureScript はこの厳密なサブセット内に有効な JavaScript を生成しますが、サード・パーティーの JavaScript コードを操作する場合には、全てを期待どおりに動作させるために追加作業が必要になります。

サードパーティの JavaScript ライブラリとのやり取りについては、後のセクションで説明します。

4.2 REPL での作業

導入

ソースファイルを作成して、ClojureScript で何かを試したいときに毎回コンパイルすることもできますが、REPL を使用するのが簡単です。REPL とは、Read(キーボードかた入力を読みとり)、Evaluate(入力を評価して)、Print(結果を出力する)、Loop(入力に戻り繰り返す) の頭文字をとったものです。言い換えると、REPL によって ClojureScript のコンセプトを試して、即座にフィードバックを得ることができます。

ClojureScript には、様々な実行環境で REPL を実行するためのサポートが用意されていますが、それぞれに長所と短所があります。たとえば、Node.js では REPL を実行できますが DOM にアクセスできません。どの REPL 環境が最適かは、個々のニーズと要件によって異なります。

Nashorn REPL

Nashorn REPL は、特別なものを必要とせず、ClojureScript コンパイラを実行するために、以前の例で使用した JVM(JDK8) のみを必要とするため、最も簡単で、おそらく最も苦痛のない REPL 環境です

REPL の Playground のために新規でスクリプトファイル repl.clj を作成して、次のように編集します。

```
(require '[cljs.repl]
         '[cljs.repl.nashorn])

(cljs.repl/repl
 (cljs.repl.nashorn/repl-env)
 :output-dir "out"
 :cache-analysis true)
```

次のコマンドにより REPL を起動して実行します。

```
$ java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

お気付きかもしれませんが、REPL には、ヒストリーやその他のシェルのような機能がサポートされていません。これは、デフォルトの REPL では readline がサポートされていないためです。この問題は rlwrap という名前の簡単なツールを使えば解決できます。このツールはオペレーティングシステムのパッケージマネージャで見つけることができるはずです (Ubuntu の場合、sudo apt install -y rlwrap でインストールします)。

rlwrap により REPL で readline が機能して、コマンドの履歴やコードナビゲーションなどのシェルのようなユーティリティが使うことができ、REPL をより快適に使うことができます。これを使用するには、REPL を開始するためには、使用した前のコマンドの前に rlwrap を追加します。

```
$ rlwrap java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

Node.js REPL

まず初めに Node.js が必要です。この REPL を使用するには、Node.js をシステムにインストールしてください。

外部依存関係を持たない Nashorn REPL がすでに使用可能になっているのに、なぜ Node.js が必要なのか不思議に思うかもしれません。答えは非常に単純です。Node.js はバックエンドで最もよく使われている JavaScript 実行環境であり、その上に構築された数多くのコミュニティーによるパッケージが存在するからです。

幸いなことに、Node.js の REPL は、一度システムにインストールすれば非常に簡単に起動できます。次の内容を `tools.clj` に追加します。

```
(require '[cljs.repl]
         '[cljs.repl.node])

(cljs.repl/repl
 (cljs.repl.node/repl-env)
 :output-dir "out"
 :cache-analysis true)
```

以前 Nashorn REPL で行ったように REPL を開始します。

```
$ rlwrap java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

Browser REPL

この REPL は、起動と実行に最も手間がかかります。これは実行環境のためにブラウザを使用しており、追加の要件があるためです。

まず、brepl.clj という名前のファイルを作成して、次のように編集します。

```
(require
  '[cljs.build.api :as b]
  '[cljs.repl :as repl]
  '[cljs.repl.browser :as browser])

(b/build "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map true
  :main 'myapp.core
  :verbose true
  :optimizations :none})

(repl/repl (browser/repl-env)
  :output-dir "out")
```

このスクリプトは、先ほどと同じようにソースをビルドして REPL を開始します。

ブラウザの REPL は、REPL が動作する前にブラウザ内でコードを実行することも要求します。そのために、前のセクションで使用したものとよく似たアプリケーションの構造を再作成します。

```
$ mkdir -p src/myapp
$ touch src/myapp/core.cljs
```

次に、新しいコンテンツを src/myapp/core.cljs に書き込みます。

```
(ns myapp.core
 (:require [clojure.browser.repl :as repl]))

(defonce conn
  (repl/connect "http://localhost:9000/repl"))

(enable-console-print!)

(println "Hello, world!")
```

最後に、ブラウザ側のコードを実行するためのエントリーポイントとして使用する index.html を作成します。

```
<!DOCTYPE html>
<html>
  <header>
    <meta charset="utf-8" />
    <title>Hello World from ClojureScript</title>
  </header>
  <body>
    <script src="main.js"></script>
  </body>
</html>
```

多くの設定をしましたが、実際に動いているところを見てみると、それだけの価値があることは間違いありません。前例と同じ方法で brepl.clj を実行します。

```
$ rlwrap java -cp cljs.jar:src clojure.main brepl.clj
Compiling client js ...
Waiting for browser to connect ...
```

最後に、お気に入りのブラウザを開き、http://localhost:9000/. にアクセスします。ページがロードされるた後 (ページは空白です)、REPL を実行したコンソールに戻り、REPL が実行されていることを確認しましょう。

```
[...]
To quit, type: :cljs/quit
cljs.user=> (+ 14 28)
42
```

ブラウザ REPL の大きな利点の 1 つは、ブラウザ環境のすべてにアクセスできることです。たとえば、REPL に (js/alert "hello world") と入力すると、ブラウザに警告ボックスが表示されます。

4.3 Google Closure Library

Google Closure Library は Google が開発している JavaScript ライブラリです。モジュールのアーキテクチャを備えており、DOM 操作、イベント処理、Ajax 、JSON を扱うためにクロスブラウザの関数を提供します。

Google Closure Library は、ClojureScript コンパイラで内部的に使用されている Closure Compiler を利用するために書かれています。

ClojureScript は Google Closure コンパイラと Closure Library の上に構築されています。実際、ClojureScript の名前空間は Closure モジュールです。これは、Closure Library を非常に簡単に操作できることを意味します。

```
(ns yourapp.core
  (:require [goog.dom :as dom]))

(def element (dom/getElement "body"))
```

このコードは、Closure Library の dom モジュールをインポートして、そのモジュールで宣言された関数を使用する方法を示しています。

さらに、Closure Library は、クラスまたはオブジェクトのように動作する「特別な」モジュールを公開します。これらの機能を使用するには、`(ns ...)` の形式で `:import` ディレクティブを使用する必要があります。

```
(ns yourapp.core
  (:import goog.History))

(def instance (History.))
```

Clojure のプログラムでは、Java クラスをインポートするためにホスト環境である Java と相互運用するために、`:import` ディレクティブを使用します。ただし、ClojureScript で型 (クラス) を定義する場合は、`:import` ディレクティブではなく、標準の `:require` ディレクティブを使用する必要があります。

4.4 依存関係の管理

ここまでは、組み込みの ClojureScript のツールを使ってソース・ファイルを JavaScript にコンパイルしてきました。次は、外部またはサード・パーティ (あるいはその両方) の依存関係を管理する方法について説明します。

このため、この章の残りの部分では、ClojureScript プロジェクトを構築するために、事実上ビルドと依存関係管理ツールのデファクトである Leiningen の使い方を説明します。boot の人気も高まっていますが、本書では Leiningen に限定します。

Leiningen のインストール

Leiningen のインストール方法は非常に簡単です。次の手順に従ってください。

```
$ mkdir ~/bin
$ cd ~/bin
$ wget https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein
$ chmod a+x ./lein
$ export PATH=$PATH:~/bin
```

~/bin ディレクトリが常にパスに設定されていることを確認します。永続的にするには、export で始まる行を ~/.bashrc に追加します。(bash を使用していると仮定します)。

ターミナルを別で開いて lein version を実行します。次のように表示されます。

```
$ lein version
Leiningen 2.5.1 on Java 1.8.0_45 OpenJDK 64-Bit Server VM
```

ここでは、Linux や BSD のような Unix 系のシステムを使っていると仮定しています。Windows ユーザの方は、Leiningen のホームページの説明を参照してください。Linux、Mac OS X、BSD 版の Leiningen スクリプトは Web サイトから入手することもできます。

初めてのプロジェクト

ツールの動作を紹介する最良の方法は、ツールを使用して toy プロジェクトを作成することです。この例では、閏 (うるう) 年かどうかを判別する小さなアプリケーションを作成します。

まず、Leiningen の mies テンプレートを使用します。

テンプレートは、初めのプロジェクト構造を作成するための Leiningen の機能です。Clojure コミュニティには非常に多くのものがあります。この例では、ClojureScript の中心的な開発者が作成した mies テンプレートを使用します。テンプレートの詳細については、Leiningen のドキュメントを参照してください。

プロジェクトのレイアウトを作成します。

```
$ lein new mies leapyears
$ cd leapyears # 生成したプロジェクトのディレクトリに移動する
```

プロジェクトの構造は次のとおりです。

```
leapyears
├── index.html
├── project.clj
├── README.md
├── scripts
│   ├── build
│   ├── release
│   ├── watch
│   ├── repl
│   └── brepl
└── src
    └── leapyears
        └── core.cljs
```

project.clj には、Leiningen が依存関係をダウンロードしてプロジェクトをビルドするために使用する情報が含まれています。今のところ、project.clj はこういうものだと思ってください。

index.html ファイルを開き、body の先頭に次の内容を追加します。

```
<section class="viewport">
  <div id="result">
    ----
  </div>
  <form action="" method="">
    <label for="year">Enter a year</label>
    <input id="year" name="year" />
  </form>
</section>
```

次の手順では、フォームをインタラクティブにするコードを追加します。次のコードを src/leapyears/core.cljs に追加します。

```
(ns leapyears.core
  (:require [goog.dom :as dom]
            [goog.events :as events]
            [cljs.reader :refer (read-string)]))

(enable-console-print!)

(def input (dom/getElement "year"))
(def result (dom/getElement "result"))

(defn leap?
  [year]
  (or (zero? (js-mod year 400))
      (and (pos? (js-mod year 100))
           (zero? (js-mod year 4)))))
```

```
(defn on-change
  [event]
  (let [target (.-target event)
        value (read-string (.-value target))]
    (if (leap? value)
      (set! (.-innerHTML result) "YES")
      (set! (.-innerHTML result) "NO"))))

(events/listen input "keyup" on-change)
```

次のようにして、ClojureScript コードをコンパイルします。

```
$ ./scripts/watch
```

バックグラウンドでは、監視スクリプトは lein ビルドツールを使用して、前のセクションで説明した Java の build コマンドに似たコマンドを実行します。

```
rlwrap lein trampoline run -m clojure.main scripts/watch.clj
```

システムに rlwrap がインストールされている必要があります。

最後に、ブラウザで index.html を開きます。テキストボックスに年を入力すると、うるう年の状態が表示されます。

scripts ディレクトリには、build や release などのファイルがあります。これらは、前のセクションで言及したビルド・スクリプトと同じですが、ここでは watch を使います。

依存関係の管理

ClojureScript のコンパイルプロセスに Leiningen を使用する本当の目的は、依存関係の取得を自動化することです。これは、手動で取得するよりも大幅に簡単です。

依存関係は、ほかのパラメータと一緒に project.clj で宣言されます。このファイルの形式は次の通りです (mies テンプレートより)。

```
(defproject leapyears "0.1.0-SNAPSHOT"
  :description "FIXME: write this!"
  :url "http://example.com/FIXME"
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [org.clojure/clojurescript "1.9.36"]
                 [org.clojure/data.json "0.2.6"]]
  :jvm-opts ^:replace ["-Xmx1g" "-server"]
  :node-dependencies [[source-map-support "0.3.2"]]
  :plugins [[lein-npm "0.5.0"]]
  :source-paths ["src" "target/classes"]
  :clean-targets ["out" "release"]
  :target-path "target")
```

ClojureScript に関連するプロパティについて簡単に説明します。

:dependencies は、プロジェクトに必要な依存関係を含むベクタです。

:clean-targets は、lein clean により削除されるパスを含むベクタです。

ClojureScript の依存関係は jar ファイルを使ってパッケージ化されます。Clojure やその他の JVM 言語を使用している場合、jar ファイルは非常になじみ深いものです。しかし、それらに慣れていなくても心配する必要はありません。jar ファイルは、プロジェクトを含む単純な zip ファイルのようなものであり、ライブラリ用の project.clj、メタデータの一部、および ClojureScript のソースを含みます。パッケージングについては別のセクションで説明します。

Clojure パッケージは Clojars で公開されることが多いです。また、ClojureScript Wiki には多くのサードパーティのライブラリがあります。

外部の依存関係

ClojureScript には存在しませんが、すでに JavaScript に実装されていて、プロジェクトで使いたいライブラリがある場合があります。

含めるライブラリによって、多くの方法があります。いくつか見ていきましょう。

Closure 対応のライブラリ

Google Closure モジュールとの互換性を持つように作成されたライブラリをプロジェクトに含める場合には、そのライブラリをソース (クラスパス) に配置して、他の Clojure 名前空間と同じようにアクセスします。

これは最も単純なケースです。Google Closure モジュールは直接互換性があり、Google Closure モジュールシステムを使って書かれた JavaScript コードは、あなたの Clojure のコードと、追加の手順なしでミックスできます。

mies テンプレートを使用して、新しいプロジェクトを作成してみましょう。

```
$ lein new mies myextmods
$ cd myextmods
```

実験用に簡単な Google Closure モジュールを作成します。

src/myextmods/myclosuremodule.js

```
goog.provide("myextmods.myclosuremodule");

goog.scope(function() {
  var module = myextmods.myclosuremodule;
  module.get_greetings = function() {
    return "Hello from google closure module.";
  };
});
```

では、REPL を開き、名前空間を reqire して、expose された関数を使用してみます。

you can open the nodejs repl just executing ./scripts/repl on the root of the repository. レポジトリのルートで `./scripts/repl` を実行するだけで、Node.js の REPL を開くことができます。

```
(require '[myextmods.myclosuremodule :as cm])
(cm/get_greetings)
;; => "Hello from google closure module."
```

CommonJS モジュール互換のライブラリ

Node.js の人気により、Node.js で用いられる CommonJS の人気は、現在 JavaScript のライブラリで最も使用されているモジュール形式です。サーバサイド開発で使用されるか、ブラウザ側のアプリケーションで使用されるかにより、利用法は別々です。

CommonJS のモジュール形式 (Google Closure モジュールを用いた前例と類似点が多い) を使用した単純なファイルを作成します。

src/myextmods/mycommonjsmodule.js

```
function getGreetings() {
  return "Hello from commonjs module.";
}

exports.getGreetings = getGreetings;
```

後に、このシンプルな pet ライブラリを使うために、ClojureScript コンパイラにファイルのパスと、使用するモジュールの型を :foreign-libs 属性で指定する必要があります。

scripts/repl.clj を開いて、次のように変更します。

```
(require
  '[cljs.repl :as repl]
  '[cljs.repl.node :as node])

(repl/repl
 (node/repl-env)
 :language-in  :ecmascript5
 :language-out :ecmascript5
 :foreign-libs [{:file "myextmods/mycommonjsmodule.js"
                 :provides ["myextmods.mycommonjsmodule"]
                 :module-type :commonjs}]
 :output-dir "out"
 :cache-analysis false)
```

直接的なパスはこの pet ライブラリを指すために使用されますが、リモートリソースへの完全な URI を指定することができ、自動的にダウンロードされます。

ここで、./scripts/repl.clj を用いた ./scripts/repl を実行して、REPL 内で moment を使ってみます。

```
(require '[myextmods.mycommonjsmodule :as cm])
(cm/getGreetings)
;; => "Hello from commonjs module."
```

従来のモジュール化されていない (グローバルスコープの) ライブラリ

今日では、何らかのモジュールを使用してライブラリをパッケージ化することが非常に一般的になっていますが、グローバルオブジェクトを公開するだけで、モジュールを全く使用しないライブラリも数多くありますが、ClojureScript から使うことができます。

グローバルオブジェクトを公開するライブラリを使用するには、CommoJS モジュールと同様の手順を実行する必要がありますが、:module-type 属性を省略する点が異なります。

これにより、名前空間の js/ を介してグローバルオブジェクトにアクセスするために必要な名前空間の合成が作成されます。名前空間は、その背後にあるオブジェクトを公開せず、その依存関係が必要であることをコンパイラに示すだけなので、synthetic と呼ばれます。

グローバル関数のみを宣言する単純なファイルを作成します。

src/myextmods/myglobalmodule.js

```
function getGreetings() {
  return "Hello from global scope.";
}
```

scripts/repl.clj を開いて、次のように変更します。

```
(require
  '[cljs.repl :as repl]
  '[cljs.repl.node :as node])

(repl/repl
 (node/repl-env)
 :language-in  :ecmascript5
 :language-out :ecmascript5
 :foreign-libs [{:file "myextmods/mycommonjsmodule.js"
                 :provides ["myextmods.mycommonjsmodule"]
                 :module-type :commonjs}
                {:file "myextmods/myglobalmodule.js"
                 :provides ["myextmods.myglobalmodule"]}]
 :output-dir "out"
 :cache-analysis false)
```

前の例と同様に、REPL 内で評価してみましょう。

```
(require 'myextmods.myglobalmodule)
(js/getGreetings)
;; => "Hello from global scope."
```

4.5 Unit テスト

ご想像のとおり、ClojureScript でのテストは、Clojure、Java、Python、JavaScript などの他の言語で広く使用されているものと同じ概念で構成されています。

言語に関係なく、Unit テストの主な目的は、いくつかのテスト・ケースを実行し、テスト対象のコードが予期したとおりに動作し、予期しない例外を発生させずに戻ることを検証することです。

ClojureScript のデータ構造はイミュータブルであるため、プログラムのエラーが発生しにくく、テストが少し容易になります。ClojureScript のもう 1 つの利点は、複雑なオブジェクトではなく、プレーン・データを使用する傾向があることです。したがって、テスト用の「モック」オブジェクトの構築が大幅に簡略化されます。

初めの一歩

公式の ClojureScript のテスト・フレームワークは、名前空間 cljs.test にあります。これは非常に単純なライブラリですが、私たちの目的には十分すぎるはずです。

他にも、追加の機能を提供したり、テストに対して直接的に異なるアプローチを提供するライブラリがあります (例えば test.check など)。ただし、ここでは説明しません。

テストを試すために mies テンプレートを使用して、新しいプロジェクトを作成します。

```
$ lein new mies mytestingapp
$ cd mytestingapp
```

このプロジェクトには、依存関係管理のセクションで説明したのと同じレイアウトなので、ここでは説明しません。次に、テスト用のディレクトリツリーを作成します。

```
$ mkdir -p test/mytestingapp
$ touch test/mytestingapp/core_tests.cljs
```

また、この新しく作成されたテストディレクトリで動作するように、既存の watch.clj を改変する必要があります。

```
(require '[cljs.build.api :as b])

(b/watch (b/inputs "test" "src")
  {:main 'mytestingapp.core_tests
   :target :nodejs
   :output-to "out/mytestingapp.js"
   :output-dir "out"
   :verbose true})
```

この新しいスクリプトは、src ディレクトリと test ディレクトリの両方をコンパイルして監視し、新しいエントリーポイントを 名前空間 mytestingapp.core_tests に設定します。

次に、core_tests.cljs にテスト・コードを追加します。

```
(ns mytestingapp.core-tests
  (:require [cljs.test :as t]))

(enable-console-print!)

(t/deftest my-first-test
  (t/is (= 1 2)))

(set! *main-cli-fn* #(t/run-tests))
```

このコードスニペットに関連する部分は以下の通りです。

```
(t/deftest my-first-test
  (t/is (= 1 2)))
```

deftest マクロは、テストを定義するための基本的な要素です。最初のパラメータとして名前を取り、続いて is マクロを使用した 1 つ以上のアサーションを取ります。この例では、(=1 2) が真であることを assert しようとします。

これを実行してみましょう。まず、監視プロセスを開始します。

```
$ ./scripts/watch
Building ...
Copying jar:file:/home/niwi/.m2/repository/org/clojure/clojurescript/1.9.36/clojurescript
Reading analysis cache for jar:file:/home/niwi/.m2/repository/org/clojure/clojurescript/1
Compiling out/cljs/core.cljs
... done. Elapsed 3.862126827 seconds
Watching paths: /home/niwi/cljsbook/playground/mytestingapp/test, /home/niwi/cljsbook/pla
```

コンパイルが完了したら、Node.js を使用して、コンパイルされたファイルの実行を試みます。

```
$ node out/mytestingapp.js
 Testing mytestingapp.core-tests
FAIL in (my-first-test) (cljs/test.js:374:14)
expected: (= 1 2)
  actual: (not (= 1 2))
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

予想される assert のエラーがコンソールに正常に出力されたことがわかります。テストを修正するには、not= を付けて = を変更して、ファイルを再度実行します。

```
$ node out/mytestingapp.js

Testing mytestingapp.core-tests

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

このようなアサーションのテストはあまり役に立ちません。アプリケーション・コードをテストしてみましょう。そのために、うるう年かどうかを調べる関数を使います。次の内容を src/mytestingapp/core.clj に書き込みます。

```
(defn leap?
  [year]
  (and (zero? (js-mod year 4))
       (pos? (js-mod year 100))
       (pos? (js-mod year 400))))
```

次に、新しい leap? 関数が正しく動作するかを確認するために、新しいテスト・ケースを作成します。core_tests.cljs は次のようになります。

```
(ns mytestingapp.core-tests
  (:require [cljs.test :as t]
            [mytestingapp.core :as core]))

(enable-console-print!)

(t/deftest my-first-test
  (t/is (not= 1 2)))

(t/deftest my-second-test
  (t/is (core/leap? 1980))
  (t/is (not (core/leap? 1981))))

(set! *main-cli-fn* #(t/run-tests))
```

コンパイル済みのファイルを再度実行して、2 つのテストが実行されていることを確認します。最初のテストは以前と同じように合格し、2 つの新しいうるう年のテストも合格します。

非同期のテスト

ClojureScript の特徴の 1 つは、非同期のシングルスレッド実行環境で実行されることですが、これには課題があります。

非同期実行環境では、非同期関数をテストできるはずです。このため、ClojureScript のテストライブラリには非同期マクロが用意されており、非同期コードでうまく動作するテストを作成することができます。

まず、非同期的に動作する関数を作成する必要があります。この目的のために、同じ操作を行うけども、コールバックを使用して非同期に結果を返す述部 async-leap? を作成します。

```
(defn async-leap?
  [year callback]
  (js/setImmediate
   (fn []
     (let [result (or (zero? (js-mod year 400))
                      (and (pos? (js-mod year 100))
                           (zero? (js-mod year 4))))]
       (callback result)))))
```

JavaScript の関数 setImmediate は非同期タスクをエミュレートするために使用され、コールバックはその述部の結果によって実行されます。

これをテストするには、前述の async マクロを使用してテストケースを作成する必要があります。

```
(t/deftest my-async-test
  (t/async done
    (core/async-leap? 1980 (fn [result]
                             (t/is (true? result))
                             (done)))))
```

非同期マクロによって公開される done 関数は、非同期操作が終了して、すべてのアサーションが実行された後に呼び出す必要があります。

done 関数は一度だけ実行することが非常に重要です。省略、または 2 回実行すると、異常な動作が発生する可能性があるため、使用しないでください。

4.6 CI との連携

大半の CI ツールおよびサービスは、提供するテストスクリプトが標準の終了コードを返すことを前提としています。しかし、JavaScript には ClojureScript が使用できる汎用の終了コード API がないため、ClojureScript のテストフレームワークでは、設定なしにこの終了コードをカスタマイズすることができません。

この問題を解決するため、ClojureScript のテストフレームワークは、テストが完了した後にカスタムコードを実行する手段を提供します。ここでは、最終テストのステータス (成功の場合は 0、失敗の場合は 1) に応じて、環境固有の終了コードを設定する必要があります。

このコードを core_tests.cljs の最後に挿入します。

```
(defmethod t/report [::t/default :end-run-tests]
  [m]
  (if (t/successful? m)
    (set! (.-exitCode js/process) 0)
    (set! (.-exitCode js/process) 1)))
```

これで、テストスクリプトの終了コードを実行後に確認できます。

```
$ node out/mytestingapp.js
$ echo $?
```

このコード・スニペットは明らかに、Node.js を使用してテストを実行していることを前提としています。別の実行環境でスクリプトを実行している場合は、その環境で終了コードを設定する方法と、それに応じて前のスニペットを変更する方法を理解しておく必要があります。

第 5 章

ClojureScript 発展編

5.1 Transducer

データの変換

ClojureScript では、シーケンス抽象化の観点から、データ変換のための豊富な語彙が用意されています。コレクション処理の関数を組み合わせて新しい関数を作成する方法を見てみましょう。この章では、ブドウの房 (cluster) を分け、腐った (rotten) 房を分別して、きれいにするという単純な問題を扱います。次のようなブドウの房のコレクションがあります。

```
(def grape-clusters
  [{:grapes [{:rotten? false :clean? false}
             {:rotten? true :clean? false}]
    :color :green}
   {:grapes [{:rotten? true :clean? false}
             {:rotten? false :clean? false}]
    :color :black}])
```

私たちは、ブドウの房を個々のブドウに分け、腐ったものを捨て、残りのブドウを食べられるようにきれいにしようとしています。このデータ変換の作業を行うための装備が ClojureScript には十分に備わっています。おなじみの `map` 、`filter`、`mapcat` 関数を使って実装できます。

```
(defn split-cluster
  [c]
  (:grapes c))

(defn not-rotten
  [g]
  (not (:rotten? g)))

(defn clean-grape
  [g]
  (assoc g :clean? true))
```

```
(->> grape-clusters
     (mapcat split-cluster)
     (filter not-rotten)
     (map clean-grape))
     ;; => ({rotten? false :clean? true}
     ;;     {:rotten? false :clean? true})
```

上記の例では、ブドウの選択と洗浄の問題を上手に解決していますが、partial application と 関数合成を使いながら mapcat、filter、map を組み合わせることによって、このような変換を抽象化することもできます。

```
(def process-clusters
  (comp
   (partial map clean-grape)
   (partial filter not-rotten)
   (partial mapcat split-cluster)))

(process-clusters grape-clusters)
;; => ({rotten? false :clean? true}
;;     {:rotten? false :clean? true})
```

コードは非常にきれいですが、いくつか問題があります。たとえば、map filter mapcat の各呼び出しは、遅延はあっても破棄される中間結果を生成するシーケンスを消費して生成します。各シーケンスは次のステップに渡され、シーケンスを返します。ブドウの房のコレクションを 1 回の横断で変換できるとしたら素晴らしいと思いませんか。

process-clusters 関数は、どんなシーケンスにも機能しますが、シーケンス以外には再利用できないことも問題です。ブドウの房のコレクションをメモリ上で利用可能にするのではなく、ストリームの中で非同期にプッシュされることを想像してみてください。このような状況では、map filter mapcat などは、型に対応した具体的な実装があるため、process-clusters 関数を再利用できなかったのです。

プロセス変換への一般化

map filter mapcat を行うプロセスは、必ずしも具体的な型に結び付けられているわけではありませんが、各々の型に対して再実装していきます。文脈に依存しないように、このようなプロセスを一般化する方法を見てみましょう。まず、map と filter の単純なバージョンを実装して、内部でどのように動作するかを見てみましょう。

```
(defn my-map
  [f coll]
  (when-let [s (seq coll)]
    (cons (f (first s)) (my-map f (rest s)))))

(my-map inc [0 1 2])
;; => (1 2 3)

(defn my-filter
  [pred coll]
  (when-let [s (seq coll)]
    (let [f (first s)
          r (rest s)]
      (if (pred f)
        (cons f (my-filter pred r))
        (my-filter pred r)))))

(my-filter odd? [0 1 2])
;; => (1)
```

どちらも seqable を受け取り、シーケンスを返すことを前提としています。多くの再帰関数と同様に、これらはおなじみの `reduce` 関数として実装できます。`reduce` 関数はアキュムレータと入力を受け取り、次のアキュムレータを返します。今後は、このような関数を reducing 関数 (reduce を行う関数) と呼びます。

```
(defn my-mapr
  [f coll]
  ;; reducing function
  (reduce (fn [acc input]
            (conj acc (f input)))
          ;; initial value
          []
          ;; collection to reduce
          coll))

(my-mapr inc [0 1 2])
;; => [1 2 3]

(defn my-filterr
  [pred coll]
  ;; reducing function
  (reduce (fn [acc input]
            (if (pred input)
              (conj acc input)
              acc))
          ;; initial value
          []
          ;; collection to reduce
          coll))

(my-filterr odd? [0 1 2])
;; => [1]
```

シーケンスだけでなく reduce を実行できる全てのものに対して関数が機能するようになったため、以前のバージョンをより一般的にすることができました。ただし、my-mapr と my-filterr は、ソース (coll) について何も知りませんが、reduce の初期値 ([]) と reducing 関数の本体内でハードコーディングされた conj と一緒に自ら生成する出力 (ベクタ) に結び付けられます。遅延シーケンス等の別のデータ構造に結果を蓄積することもできますが、そのためには関数を書き直さなければなりません。

これらの関数を本当の意味で汎用的にするにはどうすればよいのでしょうか。これらの関数は、変換中の入力のソースや、生成された出力について知るべきではありません。conj は reducing 関数の 1 つであることにお気づきでしょうか。アキュムレータと入力を受け取り、別のアキュムレータを返します。したがって、my-mapr と my-filterr が使用する reducing 関数をパラメータ化しても、ビルドする結果の型については何も知りません。試してみましょう。

```
(defn my-mapt
  ;; function to map over inputs
  [f]
  ;; parameterised reducing function
  (fn [rfn]
    ;; transformed reducing function, now it maps  f !
    (fn [acc input]
      (rfn acc (f input)))))

(def incer (my-mapt inc))

(reduce (incer conj) [] [0 1 2])
;; => [1 2 3]

(defn my-filtert
  ;; predicate to filter out inputs
  [pred]
  ;; parameterised reducing function
  (fn [rfn]
    ;; transformed reducing function,
    ;; now it discards values based on  pred !
    (fn [acc input]
      (if (pred input)
        (rfn acc input)
        acc))))

(def only-odds (my-filtert odd?))

(reduce (only-odds conj) [] [0 1 2])
;; => [1]
```

高階関数が多いので、何が起こっているのかを理解するために分解してみましょう。my-mapt がどのように機能するかを段階的に検証していきます。my-filtert の仕組みは似ているので、ここでは省略します。

まず、my-mapt はマッピング関数を取ります。この例では、inc を指定して別の関数を返しています。f を inc に置き換えて、build しているものを確認しましょう。

```
(def incer (my-mapt inc))
;; (fn [rfn]
;;   (fn [acc input]
;;     (rfn acc (inc input))))
;;                ^^^
```

結果として得られる関数は、それがデリゲートする reducing 関数を受け取るようにパラメータ化されています。それを cond と一緒に呼んだときに何が起こるか見てみましょう。

```
(incer conj)
;; (fn [acc input]
;;   (conj acc (inc input)))
;;    ^^^^
```

入力の変換のために `inc` 、結果を蓄積するために `conj` を使用する reducing 関数を取り戻します。本質的に、reducing 関数の変換として map 定義しました。ある reducing 関数を別の reducing 関数に変換する関数のことを ClojureScript では transducers と呼びます。

transducer の一般性を示すために、reduce の呼び出しにおいて、異なるソースと出力先を使用してみましょう。

```
(reduce (incer str) "" [0 1 2])
;; => "123"

(reduce (only-odds str) "" '(0 1 2))
;; => "1"
```

`map` 及び `filter` の transducer 版は、入力をソースから出力先に運ぶプロセスを変換しますが、入力がどこから来てどこで終わるかは知りません。これらの実装においては、文脈に関係なく目的を達成することの本質が含まれています。

transducer についての知識が増えたので、独自のバージョンの `mapcat` を実装できます。私たちはすでにその基本的な部分を持っています。`map` transducer です。`mapcat` が行うことは、関数を入力上にマップし、結果の構造を 1 レベルの平らさにすることです。結合する箇所を transducer として実装してみましょう。

```
(defn my-cat
  [rfn]
  (fn [acc input]
    (reduce rfn acc input)))

(reduce (my-cat conj) [] [[0 1 2] [3 4 5]])
;; => [0 1 2 3 4 5]
```

my-cat transducer は、その入力をアキュムレータに結合する reducing 関数を返します。このようにして、rfn 関数で reduce 可能な入力を reduce して、このような reduce のための初期値としてアキュムレータ (acc) を使用します。mapcat は map と cat の合成です。transducer が構成される順序は逆向きに見えるかもしれませんが、この点については後に明らかにします。

```
(defn my-mapcat
  [f]
  (comp (my-mapt f) my-cat))

(defn dupe
  [x]
  [x x])

(def duper (my-mapcat dupe))

(reduce (duper conj) [] [0 1 2])
;; => [0 0 1 1 2 2]
```

ClojureScript コアにおける Transducer

`map` `filter` `mapcat` のような ClojureScript の コア関数は、transducer を返す 1 バージョンの項数をサポートします。`process-cluster` の定義を再考して、transducer の観点から定義してみましょう。

```
(def process-clusters
  (comp
   (mapcat split-cluster)
   (filter not-rotten)
   (map clean-grape)))
```

前の `process-clusters` の定義からいくつかの点が変わりました。まず、シーケンスの処理に部分的に適用する代わりに、transducer を返すバージョンの `mapcat` `filter` `map` を使用します。

合成される順序が逆であり、実行された順序で表示されることにお気づきでしょうか。`map`、`filter`、`mapcat` は全て transducer を返すことに注目してください。`filter` は `map` が返す reducing 関数を変換して、次に進む前にフィルタリングを適用します。`mapcat` は `filter` が返す reducing 関数を変換して、マッピングと結合を適用してから処理を進めます。

transducer の強力な特性の一つは、それらが規則的な関数合成を用いて合成されることです。さらにエレガントなのは、様々な transducer の合成そのものが transducer であることです。これは、定義した `process-cluster` も transducer であることを意味するので、合成可能で文脈に依存しないアルゴリズムの変換を定義したことになります。

ClojureScript のコア関数の多くは transducer を受けとりますが、新しく作成した process-cluster でいくつか例を見てみましょう。

```
(into [] process-clusters grape-clusters)
;; => [{:rotten? false, :clean? true}
;;     {:rotten? false, :clean? true}]

(sequence process-clusters grape-clusters)
;; => ({:rotten? false, :clean? true} {:rotten? false, :clean? true})

(reduce (process-clusters conj) [] grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

transducer から返される reducing 関数を用いて `reduce` を使用することは非常に一般的であるため、transduce と呼ばれる変換 (transformation) で reduce を行う関数が存在します。これで、reduce への前の呼び出しを `transduce` を用いて書き直すことができます。

```
(transduce process-clusters conj [] grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

初期化 (Initialisation)

最後の例では、`transduce` 関数 (`[]`) に初期値を指定しましたが、省略しても同じ結果が得られます。

```
(transduce process-clusters conj grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

何が起こっているのでしょうか。アキュムレータとして使用する初期値を指定していないのに、その値が何であるかを `transduce` はどのようにして知ることができるのでしょうか。引数なしで `conj` を呼び出して何が起こるかを確認してみます。

```
(conj)
;; => []
```

`conj` 関数には空のベクタを返す引数が必要ないバージョンがありますが、これだけが引数なしを許す reducing 関数という訳ではありません。他の関数も見てみましょう。

```
(+)
;; => 0

(*)
;; => 1

(str)
;; => ""

(= identity (comp))
;; => true
```

transducer によって返される reducing 関数は、同様に引数がない場合をサポートしなければならず、これは通常、変換された reducing 関数にデリゲートされます。これまでに実装した transducer には、引数なし版の実用的な実装はありませんでした。引数なし版で単に reducing 関数を呼び出してみます。修正した my-mapt は以下のようになります。

```
(defn my-mapt
  [f]
  (fn [rfn]
    (fn
      ;; arity 0 that delegates to the reducing fn
      ([] (rfn))
      ([acc input]
       (rfn acc (f input))))))
```

transducer によって返される reducing 関数の引数なし版の呼び出しは、ネストされたすべての reducing 関数の引数なし版を呼び出し、最終的に最も外側の reducing 関数を呼び出します。定義済みの process-clusters transducer の例を見てみましょう。

```
((process-clusters conj))
;; => []
```

引数がない場合の呼び出しは transducer のスタックを経由して、最終的に `(conj)` を呼び出します。

ステートフルな transducer

ここまでは純粋に関数的な transducer だけを見てきました。つまり、暗黙的な状態を持たず、状態の予測は容易です。しかし、take のような本質的に状態をもつデータを変換する関数が多くあります。take は保持する n 個の要素と 1 つのコレクションを受け取り、最大で n 個の要素を持つコレクションを返します。

```
(take 10 (range 100))
;; => (0 1 2 3 4 5 6 7 8 9)
```

少し話題を変えて、reduce 関数の早期終了について学びましょう。アキュムレータを reduced と呼ばれる型にラップして、reduce の処理をすぐに終了するように指示することができます。コレクション内の入力を集約し、アキュムレータに 10 個の要素が入ると終了する reduce の例を見てみましょう。

```
(reduce (fn [acc input]
          (if (= (count acc) 10)
            (reduced acc)
            (conj acc input)))
        []
        (range 100))
;; => [0 1 2 3 4 5 6 7 8 9]
```

transducer は reducing 関数の変形であるため、早期終了のために reduced を使用します。状態をもつ transducer はプロセスが終了する前に、何らかの掃除を行う必要があるかもしれないので、完成のためのステップとして、引数を 1 つとる場合をサポートしなければならないことに注意してください。通常、引数を取らない場合と同様に引数は、引数が 1 つである変形された reducing 関数にデリゲートされます。

これを知っていると、take のような状態をもつ transducer を書くことができます。これまでに見てきたような入力の数を追跡するために、可変な状態を内部的に使用して、必要な要素数を確認したらすぐに、アキュムレータを reduced にラップします。

```
(defn my-take
  [n]
  (fn [rfn]
    (let [remaining (volatile! n)]
      (fn
        ([] (rfn))
        ([acc] (rfn acc))
        ([acc input]
         (let [rem @remaining
               nr (vswap! remaining dec)
               ;; we still have items to take
               result (if (pos? rem)
                        ;; we're done,
                        ;; acc becomes the result
                        (rfn acc input)
                        acc)]
           ;; wrap result in reduced if not already
           (if (not (pos? nr))
             (ensure-reduced result)
             result)))))))
```

これは ClojureScript コアにある take 関数を単純化したものです。注意すべき点がいくつかあるので、少しずつ分けて見ていきましょう。

まず最初に注意しなければならないのは、transducer 内で変更可能な値を作成している点です。変換を行う reducing 関数を受け取るまでは値を作成していないことに注意してください。transducer を返す前に作ってしまうと、my-take 関数を 2 回以上使うことはできません。transducer は、使用の度に変換を行う reducing 関数を渡されるので、何度でも使うことができます。また、変更可能な変数は、毎回使うたびに作成されます。

```
(fn [rfn]
  ;; make sure to create mutable variables
  ;; inside the transducer
  (let [remaining (volatile! n)]
    (fn
      ;; ...
)))

(def take-five (my-take 5))

(transduce take-five conj (range 100))
;; => [0 1 2 3 4]

(transduce take-five conj (range 100))
;; => [0 1 2 3 4]
```

ここで、my-take から返される reducing 関数について詳しく見ていきましょう。まず volatile に deref を行い、取得されるべき残りの要素の数を取得して、それをデクリメントして次の残りの値を取得します。もし、まだ取るべきアイテムが残っているなら、アキュムレータと入力を渡して rfn を呼びます。そうでなければ、すでに最終的な結果をもっています。

```
([acc input]
  (let [rem @remaining
        nr (vswap! remaining dec)
        result (if (pos? rem)
                 (rfn acc input)
                 acc)]
    ;; ...
))
```

my-take の本体は明らかになっているはずです。次の剰余 (nr) を使用して、処理されるアイテムがまだ存在するかどうかをチェックします。存在しない場合は、ensure-reduced 関数を使用して、結果を reduced にラップします。ensure-reduced は、まだ reduce されていない場合は、値を reduced でラップして、すでに削減されている場合には単に値を返します。まだ完了していない場合は、累積した結果を返して処理します。

```
(if (not (pos? nr))
  (ensure-reduced result)
  result)
```

ステートフルな transducer の例を見てきましたが、完了の段階では何もしませんでした。累積値を表示するために完了のステップを使用する transducer の例を見てみましょう。要素の数が n の要素あれば、サイズが n 個のベクタの入力を変換する partition-all の簡素版を実装しましょう。目的をよく理解するために、引数を 2 つとる場合で、数字とコレクションを与えると何が得られるかを見てみましょう。

```
(partition-all 3 (range 10))
;; => ((0 1 2) (3 4 5) (6 7 8) (9))
```

partition-all の transducer を返す関数は、数値 n を取り、n 個のサイズのベクタで入力をグループ化する transducer を返しまる。完了する段階では、累積結果があるかどうかを確認して、ある場合は結果に追加します。以下は、ClojureScript のコア関数である partition-all を単純化したもので、array-list は変更可能な JavaScript の配列のラッパーです。

```
(defn my-partition-all
  [n]
  (fn [rfn]
    (let [a (array-list)]
      (fn
        ([] (rfn))
        ([result]
         ;; no inputs accumulated, don't have to modify result
         (let [result (if (.isEmpty a)
                        result
                        (let [v (vec (.toArray a))]
                          ;; flush array contents for garbage collection
                          (.clear a)
                          ;; pass to  rfn , removing the reduced wrapper if present
                          (unreduced (rfn result v))))]
           (rfn result)))
        ([acc input]
         (.add a input)
         ;; got enough results for a chunk
         (if (== n (.size a))
           (let [v (vec (.toArray a))]
             (.clear a)
             ;; the accumulated chunk
             ;; becomes input to  rfn
             (rfn acc v))
           acc))))))
```

```
(def triples (my-partition-all 3))

(transduce triples conj (range 10))
;; => [[0 1 2] [3 4 5] [6 7 8] [9]]
```

Eduction

Eduction とは、コレクションと 1 つ以上の変換を組み合わせる方法であり、それらに対して reduce したり、繰り返したりすることができ、そのたびに変換を適用します。処理したいコレクションとそれに対する変換があり、他に拡張してほしい場合、私たちはソースコレクションと私たちの変換をカプセル化した Eduction をそれらに渡すことができます。eduction 関数を使用して eduction を作成できます。

```
(def ed (eduction (filter odd?) (take 5) (range 100)))

(reduce + 0 ed)
;; => 25

(transduce (partition-all 2) conj ed)
;; => [[1 3] [5 7] [9]]
```

ClojureScript コアにおける他の transducer

`map` 、`filter` 、`mapcat` 、`take` 、`partition-all` について学びましたが、ClojureScript にはもっと多くの transducer があります。以下は、他の興味深いものの一部です。

- `drop` は take の双対であり、入力を reducing 関数に渡す前に `n` の値まで小さくします
- `distinct` は、入力を 1 回だけ行うことを許可します。
- `dedupe` は入力値の連続する重複を削除します

他にどんな transducer があるのかを知るために、ClojureScript コアを調べてみることをお勧めします。

transducer を定義する

独自の transducer を作成する前に考慮すべき点がいくつかあるので、この章では、transducer を適切に実装する方法を学びます。まず、transducer の一般的な構造は次の通りであることを学びました。

```
(fn [xf]
  (fn
    ;; init
    ([]
     ...)
    ;; completion
    ([r]
     ...)
    ;; step
    ([acc input]
     ...)))
```

通常、transducer の間では ... の箇所のコードのみが変化します。これらは、結果として得られる関数のそれぞれの引数で保存されなければならない不変の式です。

- 0 番目の引数 (init): ネストされた変換 `xf` の 0 番目の引数を呼び出す必要があります。
- 1 番目の引数 (completion): 最終値を生成し、潜在的にフラッシュ状態を生成するために使用され、ネストされたトランスフォーム `xf` の 1 番目の引数を正確に 1 度だけ呼び出す必要があります。
- 2 番目の引数 (ステップ): 結果を導く reducing 関数であり、ネストされたトランスフォーム `xf` の 2 番目の引数を 0 回以上呼び出します。

トランスデゥシブルなプロセス

トランスデゥシブルなプロセスとは、入力値を取り込む一連のステップによって定義される任意のプロセスです。入力のソースは、プロセスによって異なります。私たちの例の大半は、コレクションまたは遅延シーケンスからの入力を扱っていますが、非同期の値のストリーム、または core.async のチャンネルである可能性があります。各ステップで生成される出力もプロセスごとに異なります。into は transducer のすべての出力を持つコレクションを作成して、シーケンスは遅延シーケンスを生成し、非同期ストリームは出力をリスナーに通知します。

変換可能なプロセスについての理解を深めるために、無限の列 (unbounded queue) を実装します。列に値を追加することは、入力を取り込む一連のステップと捉えることができるため、まず、無限の列を実装するプロトコルとデータ型を定義します。

```
(defprotocol Queue
  (put! [q item] "put an item into the queue")
  (take! [q] "take an item from the queue")
  (shutdown! [q] "stop accepting puts in the queue"))
```

```
(deftype UnboundedQueue [^:mutable arr ^:mutable closed]
  Queue
  (put! [_ item]
    (assert (not closed))
    (assert (not (nil? item)))
    (.push arr item)
    item)
  (take! [_]
    (aget (.splice arr 0 1) 0))
  (shutdown! [_]
    (set! closed true)))
```

私たちは Queue プロトコルを定義しましたが、お気付きかもしれませんが、UnboundedQueue の実装は transducer をまったく認識しません。そのステップ関数として put! があるので、このインターフェースの上に transducible プロセスを実装します。

```
(defn unbounded-queue
  ([]
   (unbounded-queue nil))
  ([xform]
   (let [put! (completing put!)
         xput! (if xform (xform put!) put!)
         q (UnboundedQueue. #js [] false)]
     (reify
       Queue
       (put! [_ item]
         (when-not (.-closed q)
           (let [val (xput! q item)]
             (if (reduced? val)
               (do
                 ;; call completion step
                 (xput! @val)
                 ;; respect reduced
                 (shutdown! q)
                 @val)
               val))))
       (take! [_]
         (take! q))
       (shutdown! [_]
         (shutdown! q))))))
```

unbounded-queue コンストラクタは、内部的に UnboundedQueue インスタンスを使用して、take! と shutdown! の呼び出しをプロキシして、put! 関数の中のトランスデゥシブルなロジックを実装します。何が起こっているのか理解するために、少しずつ見ていきましょう。

```
(let [put! (completing put!)
      xput! (if xform (xform put!) put!)
      q (UnboundedQueue. #js [] false)]
  ;; ...
)
```

まず、Queue プロトコルの put! 関数に、引数をとらない場合と引数を 1 つとる場合を追加するために `completing` を使います。これにより、この reducing 関数を `xform` に与えて別のものを引き出す場合に、transducer とうまく動作するようになります。その後、もし transducer(`xform`) が与えられると、transducer を put! に適用しながら reducing 関数を引き出します。transducer が渡されなければ、`put!` を使います。`q` は `UnboundedQueue` の内部インスタンスです。

```
(reify
  Queue
  (put! [_ item]
    (when-not (.-closed q)
      (let [val (xput! q item)]
        (if (reduced? val)
          (do
            ;; call completion step
            (xput! @val)
            ;; respect reduced
            (shutdown! q)
            @val)
          val))))
  ;; ...
)
```

expose された `put!` 操作は queue がシャットダウンされていない場合にのみ実行されます。UnboundedQueue の `put!` の実装では、新たな値を入れることができるか、不変性を壊さないかを検証するために assert を用います。もし queue が閉じられておらず、値を入れることができる場合は、変換される可能性がある `xput!` を用います。

put の操作が reduce された値を返した場合、transducible なプロセスを終了する必要があることを示しています。この場合、queue をシャットダウンして、追加の値を受け入れないようにします。もし reduce された値をえられなければ、puts を引き受け続けることができます。

queue が transducer がない状態でどう動作するかを見てみましょう。

```
(def q (unbounded-queue))
;; => #<[object Object]>

(put! q 1)
;; => 1
(put! q 2)
;; => 2
```

```
(take! q)
;; => 1
(take! q)
;; => 2
(take! q)
;; => nil
```

期待していた通りですね。今度はステートレスな transducer で試してみましょう。

```
(def incq (unbounded-queue (map inc)))
;; => #<[object Object]>

(put! incq 1)
;; => 2
(put! incq 2)
;; => 3

(take! incq)
;; => 2
(take! incq)
;; => 3
(take! incq)
;; => nil
```

transducible なプロセスを実装したかどうかを確認するために、ステートフルな transducer を使いましょう。4 と等しくない間は値を受け入れ、入力を 2 つの要素のチャンクに分割する transducer を使用します。

```
(def xq (unbounded-queue (comp
                          (take-while #(not= % 4))
                          (partition-all 2))))

(put! xq 1)
(put! xq 2)
;; => [1 2]
(put! xq 3)
(put! xq 4) ;; shouldn't accept more values from here on
(put! xq 5)
;; => nil

(take! xq)
;; => [1 2]
;; seems like partition-all flushed correctly!
(take! xq)
;; => [3]
(take! xq)
;; => nil
```

queue の例は、core.async チャンネルが内部ステップで transducer を使う方法にに大きく影響を受けています。チャンネルと transducer を使ったチャンネルの使い方については、後のセクションで説明します。

Transducible なプロセスは、早期終了のシグナルの伝達手段として reduced を反映しなければいけません。たとえば、コレクションのビルドは、reduced に遭遇したときや、transducer をもつ core.async チャンネルが閉じられた場合に停止します。reduce された値は必ず deref で開封されて、完了のステップに渡されます。完了ステップは 1 度しか呼び出されてはいけません。

transducible なプロセスは、それ自身のステップ関数を用いて transducer を呼び出す際に生成する reducing 関数を expose すべきではありません。なぜなら、それは状態をもち、他の場所から使用することは安全でないからです。

5.2 一時性

ClojureScript の不変で永続的なデータ構造は、それなりのパフォーマンスを発揮しますが、最終的な結果を共有するだけのために、複数のステップを使って大きなデータ構造を変換している状況もあります。たとえば、core にある into 関数は、コレクションを取得して、シーケンスの内容を続々と取り込みます。

```
(into [] (range 100))
;; => [0 1 2 ... 98 99]
```

上の例では、100 個の要素からなるベクタを一度に一つずつ conj しながら生成しています。最終的な結果でない全ての中間ベクタは into 関数以外では見ることができず、永続化のために必要な配列のコピーは不要なオーバーヘッドです。

このような状況のために、ClojureScript は transient と呼ばれる永続的なデータ構造を提供しています。マップ、ベクタ、セットには、transient に相当するものがあります。Transient は、常に transient 関数を使用して、永続的なデータ構造から生成されます。これにより、一定の時間内に、transient のバージョンが作成されます。

```
(def tv (transient [1 2 3]))
;; => #<[object Object]>
```

Transient は、対応する永続的な読取り API をサポートします。

```
(def tv (transient [1 2 3]))

(nth tv 0)
;; => 1
```

```
(get tv 2)
;; => 3

(def tm (transient {:language "ClojureScript"}))

(:language tm)
;; => "ClojureScript"

(def ts (transient #{:a :b :c}))

(contains? ts :a)
;; => true

(:a ts)
;; => :a
```

transient には更新のための永続的で不変なセマンティクスがないので、conj や assoc 関数を使って変換できません。その代わり、transient 上で機能する変換関数は感嘆符！ をつけます。conj! を使った例を見てみましょう。

```
(def tv (transient [1 2 3]))

(conj! tv 4)
;; => #<[object Object]>

(nth tv 3)
;; => 4
```

このように、ベクタの transient バージョンは不変でも永続でもありません。その代わり、ベクタはその場で変異します。conj! を使って繰り返し tv を変換することもできますが、永続的なデータ構造で使用されるイディオムを放棄してはいけません。transient のデータを変換する場合は、次の例のように、その戻されたバージョンを使用して変更を加えます。

```
(-> [1 2 3]
  transient
  (conj! 4)
  (conj! 5))
;; => #<[object Object]>
```

transient! を呼ぶことで、transient を元の永続的で不変なデータ構造に変換することもできます。この操作は、transient を永続的なデータ構造から派生させる操作と同様に、一定の時間で実行されます。

```
(-> [1 2 3]
  transient
  (conj! 4)
  (conj! 5)
  persistent!)
;; => [1 2 3 4 5]
```

transient を永続的な構造に変換する特徴は、一時的なものが永続的なデータ構造に変換された後に無効になり、それ以上の変換ができなくなることです。これは、派生した永続的なデータ構造が transient の内部的な node を使用しており、それらを変更することにより不変性と永続的な保証が壊れるためです。

```
(def tm (transient {}))
;; => #<[object Object]>

(assoc! tm :foo :bar)
;; => #<[object Object]>

(persistent! tm)
;; => {:foo :bar}

(assoc! tm :baz :frob)
;; Error: assoc! after persistent!
```

最初の `into` の例に戻りましょう。ここではパフォーマンスのために transient を使用してシンプルに実装します。内部的には変異を使用しますが、永続的なデータ構造を返す純粋な関数型のインタフェースを公開します。

```
(defn my-into
  [to from]
  (persistent! (reduce conj! (transient to) from)))

(my-into [] (range 100))
;; => [0 1 2 ... 98 99]
```

5.3 メタデータ

ClojureScript のシンボル、変数、永続的なコレクションは、メタデータの添付 (attach) をサポートしています。メタデータは、添付される実体 (entity) に関する情報を含むマップです。ClojureScript のコンパイラは、型のヒントのような目的のためにメタデータを使用します。また、メタデータのシステムは、ツールやライブラリ及びアプリケーションの開発者も利用できます。

ClojureScript を日常的に書いているなかでは、メタデータが必要になることはあまりないかもしれませんが、メタデータは知っておくと便利な機能なので、いつか役に立つかもしれません。実行時のコード分析やドキュメントの生成などが非常に簡単になります。次のセクションでは、その理由を説明します。

Vars

var を定義して、デフォルトでどのメタデータが付加されているかを見てみましょう。このコードは REPL で実行されるため、ソースファイルで定義されている var のメタデータは変化するかもしれません。meta 関数を使用して、指定された値のメタデータを取得します。

```
(def answer-to-everything 42)
;; => 42

#'answer-to-everything
;; => #'cljs.user/answer-to-everyhing

(meta #'answer-to-everything)
;; => {:ns cljs.user,
;;     :name answer-to-everything,
;;     :file "NO_SOURCE_FILE",
;;     :source "answer-to-everything",
;;     :column 6,
;;     :end-column 26,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (),
;;     :doc nil,
;;     :test nil}
```

上記のコードにおいて、`#'answer-to-everything` は、`answer-to-everything` シンボルの値を保持する var への参照を渡します。これには、定義された名前空間 (`:ns`)、その名前、ファイル (ここでは REPL で定義されているため、ソースファイルはありません)、ソース、定義されたファイルでの位置、引数の一覧 (関数の場合のみ意味があります)、ドキュメントの文字列、およびテスト関数に関する情報が含まれていることがわかります。

関数における var のメタデータを見てみましょう。

```
(defn add
  "A function that adds two numbers."
  [x y]
  (+ x y))

(meta #'add)
;; => {:ns cljs.user,
;;     :name add,
;;     :file "NO_SOURCE_FILE",
;;     :source "add",
;;     :column 7,
;;     :end-column 10,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (quote ([x y])),
;;     :doc "A function that adds two numbers.",
;;     :test nil}
```

引数のリストは :arglists フィールド、マニュアルは :doc フィールドに格納されます。ではテスト関数を定義して、何のために :test フィールドを使うかを見ていきましょう。

```
(require '[cljs.test :as t])

(t/deftest i-pass
  (t/is true))

(meta #'i-pass)
;; => {:ns cljs.user,
;;     :name i-pass,
;;     :file "NO_SOURCE_FILE",
;;     :source "i-pass",
;;     :column 12,
;;     :end-column 18,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (),
;;     :doc "A function that adds two numbers.",
;;     :test #<function (){ ... }>}
```

i-pass var のメタデータ内の :test フィールドがテスト関数です。テスト関数は、指定した名前空間でテストを検出して実行するために cljs.test ライブラリによって使用されます。

値

var はメタデータを持つことができ、コンパイラと cljs.test ライブラリが利用するために、どのようなメタデータを付与されるかを学びました。永続コレクションがメタデータをもつことは可能ですが、デフォルトではメタデータはありません。with-meta 関数を使用すると、指定したメタデータが付加された同じ値と型を持つオブジェクトを派生させることができます。例を見てみましょう。

```
(def map-without-metadata {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-without-metadata)
;; => nil

(def map-with-metadata (with-meta map-without-metadata
                                  {:answer-to-everything 42}))
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:answer-to-everything 42}

(= map-with-metadata
   map-without-metadata)
;; => true

(identical? map-with-metadata
            map-without-metadata)
;; => false
```

ClojureScript の等価性は値に基づいているため、メタデータが 2 つのデータ構造の等価性に影響しないことは驚くべきことではありません。もう 1 つ興味深い点は、with-meta を使用すると、指定されたものと同じ型と値をもつ別のオブジェクトが作成され、指定されたメタデータがそれに付与されることです。

永続データ構造から新しい値を派生させるときにメタデータで何が起こるかを確認してみましょう。

```
(def derived-map (assoc map-with-metadata :language "Clojure"))
;; => {:language "Clojure"}

(meta derived-map)
;; => {:answer-to-everything 42}
```

この例でわかるように、メタデータは永続的なデータ構造の派生バージョンで保持されます。微妙なところもありますが、新しいデータ構造を派生する関数が同じ型のコレクションを返す限り、メタデータは保存されます。これは、変換によって型が変更される場合には当てはまりません。この点を理解するために、ベクタから seq や subvector を派生させると何が起こるかを見てみましょう。

```
(def v (with-meta [0 1 2 3] {:foo :bar}))
;; => [0 1 2 3]

(def sv (subvec v 0 2))
;; => [0 1]

(meta sv)
;; => nil

(meta (seq v))
;; => nil
```

メタデータのための構文

ClojureScript の reader はメタデータの注釈 (annotation) を構文としてサポートしますが、様々な方法で記述できます。var 定義やコレクションの先頭にキャレット (^) とその後にマップをつけて、指定したメタデータのマップで注釈をつけることができます。

```
(def ^{:doc "The answer to Life, Universe and Everything."} answer-to-everything 42)
;; => 42

(meta #'answer-to-everything)
;; => {:ns cljs.user,
;;     :name answer-to-everything,
;;     :file "NO_SOURCE_FILE",
;;     :source "answer-to-everything",
;;     :column 6,
;;     :end-column 26,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (),
;;     :doc "The answer to Life, Universe and Everything.",
;;     :test nil}

(def map-with-metadata ^{:answer-to-everything 42} {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:answer-to-everything 42}
```

`answer-to-everything` の var の定義で指定されたメタデータが var のメタデータとどのようにマージされるかに注意してください。

メタデータの一般的な使用法は、特定のキーの値を `true` に設定することです。たとえば、変数が動的まか定数かを var のメタデータに追加できます。このような場合には、キャレットに続けてキーワードを使用する省略表記があります。次に例を示します。

```
(def ^:dynamic *foo* 42)
;; => 42

(:dynamic (meta #'*foo*))
;; => true

(def ^:foo ^:bar answer 42)
;; => 42

(select-keys (meta #'answer) [:foo :bar])
;; => {:foo true, :bar true}
```

メタデータの付与には、別の簡略表記があります。キャレットに続けてシンボルを使用すると、:tag キーの下のメタデータのマップに追加されます。^boolean などのタグを使用すると、ClojureScript コンパイラに、式の型や関数の戻り値の型に関するヒントを与えます。

```
(defn ^boolean will-it-blend? [_] true)
;; => #<function ... >

(:tag (meta #'will-it-blend?))
;; => boolean

(not ^boolean (js/isNaN js/NaN))
;; => false
```

メタデータを扱う関数

これまで meta と with-meta について学んできましたが、ClojureScript にはメタデータを変換するための関数がいくつか用意されています。var-meta は、元のオブジェクトと同じ型と値を持つ新しいオブジェクトを派生させるという点で with-meta と似ていますが、直接的に付与するメタデータをとりません。代わりに、var-meta は、変換する指定のオブジェクトのメタデータに適用する関数をとり、新たなメタデータを派生させます。例を見てみましょう。

```
(def map-with-metadata ^{:foo 40} {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:foo 40}

(def derived-map (vary-meta map-with-metadata update :foo + 2))
;; => {:language "ClojureScript"}

(meta derived-map)
;; => {:foo 42}
```

既存の var や値のメタデータを変更する場合、変更のためには alter-meta! 関数を、別のマップに置き換えるには reset-meta! を使うことができます。

```
(def map-with-metadata ^{:foo 40} {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:foo 40}

(alter-meta! map-with-metadata update :foo + 2)
;; => {:foo 42}

(meta map-with-metadata)
;; => {:foo 42}

(reset-meta! map-with-metadata {:foo 40})
;; => {:foo 40}

(meta map-with-metadata)
;; => {:foo 40}
```

5.4 中心的なプロトコル

ClojureScript の core に定義されている関数群がプロトコルを中心に実装されていることも素晴らしい特徴の 1 つだといえます。これは、そのようなプロトコルで拡張したどんな型、また私たちや第三者によって定義された型でも動作するように開かれています。

関数

これまでの章で学んだように、関数だけが呼び出されるものとは限りません。ベクタはインデックス、マップはキー、セットは値を関数のように使うことができます。

IFn プロトコルを実装する関数として呼び出し可能な型に拡張することができます。関数としての呼び出しをサポートしていないコレクションが queue です。PersistentQueue 型に IFn を実装して、queue をインデックスの関数として呼び出すことができるようにします。

```
(extend-type PersistentQueue
  IFn
  (-invoke
    ([this idx]
      (nth this idx))))

(def q #queue[:a :b :c])
;; => #queue [:a :b :c]

(q 0)
;; => :a

(q 1)
;; => :b

(q 2)
;; => :c
```

Printing

いくつかの中心的なプロトコルについて学ぶために、値のペアを保持する Pair 型を定義します。

```
(deftype Pair [fst snd])
```

型の印字方法をカスタマイズしたい場合は、IPrintWithWriter プロトコルを実装できます。出力値、writer オブジェクト、オプションを受け取る -pr-writer と呼ばれる関数を定義します。この関数は、求められる Pair 型 の文字列表現を書くために、writer オブジェクトの -write 関数を使用します。

```
(extend-type Pair
  IPrintWithWriter
  (-pr-writer [p writer _]
    (-write writer (str "#<Pair " (.-fst p) "," (.-snd p) ">"))))
```

シーケンス

前のセクションでは、ClojureScript の主要な抽象概念の 1 つであるシーケンスについて学びました。シーケンスを操作するための `first` 関数と `rest` 関数を覚えているでしょうか。これらは `ISeq` プロトコルで定義されているので、このような関数に対応する型を拡張することができます。

```
(extend-type Pair
  ISeq
  (-first [p]
    (.-fst p))

  (-rest [p]
    (list (.-snd p))))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(first p)
;; => 1

(rest p)
;; => (2)
```

シーケンスを操作する便利な関数として `next` があります。`next` は、与えられた引数がシーケンスである限り動作しますが、`INext` プロトコルを使って明示的に実装することができます。

```
(def p (Pair. 1 2))

(next p)
;; => (2)

(extend-type Pair
  INext
  (-next [p]
    (println "Our next")
    (list (.-snd p))))

(next p)
;; Our next
;; => (2)
```

最後に、ISeqable プロトコルを実装する独自の型をシーカブルにすることができます。これは、シーケンスを元に戻すために seq に渡すことができることを意味します。

```
(def p (Pair. 1 2))

(extend-type Pair
  ISeqable
  (-seq [p]
    (list (.-fst p) (.-snd p))))

(seq p)
;; => (1 2)
```

これで Pair 型は、ClojureScript のシーケンス操作関数の大半で動作します。

```
(def p (Pair. 1 2))
;; => #<Pair 1,2>

(map inc p)
;; => (2 3)

(filter odd? p)
;; => (1)

(reduce + p)
;; => 3
```

コレクション

コレクションの関数もプロトコルで定義されています。このセクションの例では、native な JavaScript の文字列をコレクションのように動作させます。コレクションを扱う上で最も重要な関数は、ICollection プロトコルで定義されている conj です。文字列は、文字列への conj に意味を持つ唯一の型であるため、文字列に対する conj の操作は単なる連結になります。

```
(extend-type string
  ICollection
  (-conj [this o]
    (str this o)))

(conj "foo" "bar")
;; => "foobar"

(conj "foo" "bar" "baz")
;; => "foobarbaz"
```

コレクションを操作する empty は便利な関数です。これは IEmptyable Collection プロトコルの一部です。文字列の型に実装してみましょう。

```
(extend-type string
  IEmptyableCollection
  (-empty [_]
    ""))

(empty "foo")
;; => ""
```

ここでは、native な JavaScript の文字列を拡張するための特別なシンボルである string を使いました。詳細については、JavaScript の型を拡張するセクションを参照してください。

■コレクションの特性 すべてのコレクションに備わっているわけではない特性がいくつかあります。たとえば、一定の時間で数えられることや、可逆的であることなどです。これらの特性がすべてのコレクションに対して意味をもつわけではないので、これらの特性は異なるプロトコルに分割されます。これらのプロトコルを説明するために、前に定義した Pair 型を使用します。

count 関数を使用して一定の時間でカウントできるコレクションの場合、ICounted プロトコルを実装できます。Pair 型には容易に実装できるはずです。

```
(extend-type Pair
  ICounted
  (-count [_]
    2))

(def p (Pair. 1 2))

(count p)
;; => 2
```

ベクタやリストなどのコレクション型は、nth 関数でインデックス番号を付けることができます。もし私たちの型がインデックスされるなら、私たちは IIndexed プロトコルを実装することができます:

```
(extend-type Pair
  IIndexed
  (-nth
    ([p idx]
     (case idx
       0 (.-fst p)
       1 (.-snd p)
       (throw (js/Error. "Index out of bounds"))))
    ([p idx default]
     (case idx
       0 (.-fst p)
       1 (.-snd p)
       default))))

(nth p 0)
;; => 1

(nth p 1)
;; => 2

(nth p 2)
;; Error: Index out of bounds

(nth p 2 :default)
;; => :default
```

連想性

キーを値にマップする連想的なデータ構造が数多くあります。私たちはすでにいくつかの関数に遭遇しており、get、assoc、dissoc 等の関数を知っています。これらの関数を構築しているプロトコルについて説明します。

まず最初に、連想的なデータ構造においてキーを検索する方法が必要です。ILookup プロトコルは、そのための関数を定義します。ここでは、Pair 型にキーを検索する機能を追加します。これは、Pair 型がインデックス 0 と 1 を値にマップする連想的なデータ構造であるためです。

```
(extend-type Pair
  ILookup
  (-lookup
    ([p k]
     (-lookup p k nil))
    ([p k default]
     (case k
       0 (.-fst p)
       1 (.-snd p)
       default))))
```

```
(get p 0)
;; => 1

(get p 1)
;; => 2

(get p :foo)
;; => nil

(get p 2 :default)
;; => :default
```

データ構造で assoc を使用するには、IAssociative プロトコルを実装する必要があります。Pair 型では、値を関連づけるためのキーとして 0 と 1 だけが許可されます。IAssociative には、キーの有無を問い合わせる機能もあります。

```
(extend-type Pair
  IAssociative
  (-contains-key? [_ k]
    (contains? #{0 1} k))

  (-assoc [p k v]
    (case k
      0 (Pair. v (.-snd p))
      1 (Pair. (.-fst p) v)
      (throw (js/Error. "Can only assoc to 0 and 1 keys")))))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(assoc p 0 2)
;; => #<Pair 2,2>

(assoc p 1 1)
;; => #<Pair 1,1>

(assoc p 0 0 1 1)
;; => #<Pair 0,1>

(assoc p 2 3)
;; Error: Can only assoc to 0 and 1 keys
```

assoc を補完する関数として dissoc があり、dissoc は IMap プロトコルの一部です。Pair 型ではあまり意味がありませんが、それでも実装してみましょう。0 または 1 の関連付けを解除すると、そのような位置に nil が配置され、無効なキーは無視されます。

```
(extend-type Pair
  IMap
  (-dissoc [p k]
    (case k
      0 (Pair. nil (.-snd p))
      1 (Pair. (.-fst p) nil)
      p)))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(dissoc p 0)
;; => (nil 2)

(dissoc p 1)
;; => (1 nil)

(dissoc p 2)
;; => (1 2)

(dissoc p 0 1)
;; => (nil nil)
```

連想的なデータ構造は、エントリー (entries) と呼ばれるキーと値のペアで構成されます。key 関数と val 関数を使用すると、このようなエントリーのキーと値を参照できます。これらの関数は IMapEntry プロトコルに基づいて構築されています。key 関数と val 関数の例と、マップのエントリーを使用してマップを構築する方法をいくつか見てみましょう。

```
(key [:foo :bar])
;; => :foo

(val [:foo :bar])
;; => :bar

(into {} [[:foo :bar] [:baz :xyz]])
;; => {:foo :bar, :baz :xyz}
```

Pair 型もマップのエントリーにできます。最初の要素をキー、2 番目の要素を値として扱います。

```
(extend-type Pair
  IMapEntry
  (-key [p]
    (.-fst p))

  (-val [p]
    (.-snd p)))
```

```
(def p (Pair. 1 2))
;; => #<Pair 1,2>

(key p)
;; => 1

(val p)
;; => 2

(into {} [p])
;; => {1 2}
```

比較

2 つ以上の値が等しいかどうかを = を用いて判定するには、IEquiv プロトコルを実装する必要があります。Pair 型に、IEquiv プロトコルを実装してみましょう:

```
(def p  (Pair. 1 2))
(def p' (Pair. 1 2))
(def p'' (Pair. 1 2))

(= p p')
;; => false

(= p p' p'')
;; => false

(extend-type Pair
  IEquiv
  (-equiv [p other]
    (and (instance? Pair other)
         (= (.-fst p) (.-fst other))
         (= (.-snd p) (.-snd other)))))

(= p p')
;; => true

(= p p' p'')
;; => true
```

型を比較できるようにすることもできます。compare 関数は 2 つの値を取り、最初の値が 2 番目の値より小さい場合は負の数を返し、両方が等しい場合は 0 を返し、最初の値が 2 番目の値より大きい場合は 1 を返します。型を比較できるようにするには、IComparable プロトコルを実装する必要があります。

Pair 型の場合、比較とは最初の 2 つの値が等しいかどうかをチェックすることを意味します。等しい場合は、結果は 2 番目の値になります。そうでない場合は、最初の比較の結果を返します。

```
(extend-type Pair
  IComparable
  (-compare [p other]
    (let [fc (compare (.-fst p) (.-fst other))]
      (if (zero? fc)
        (compare (.-snd p) (.-snd other))
        fc))))

(compare (Pair. 0 1) (Pair. 0 1))
;; => 0

(compare (Pair. 0 1) (Pair. 0 2))
;; => -1

(compare (Pair. 1 1) (Pair. 0 2))
;; => 1

(sort [(Pair. 1 1) (Pair. 0 2) (Pair. 0 1)])
;; => ((0 1) (0 2) (1 1))
```

メタデータ

meta 関数と with-meta 関数も、それぞれ IMeta と IWithMeta という 2 つのプロトコルに基づいています。追加のフィールドをメタデータを保持するために加えて、両プロトコルを実装することにより、メタデータを運ぶ独自の型を作成できます。

メタデータをもつことができる Pair を実装して見ましょう。

```
(deftype Pair [fst snd meta]
  IMeta
  (-meta [p] meta)

  IWithMeta
  (-with-meta [p new-meta]
    (Pair. fst snd new-meta)))


(def p (Pair. 1 2 {:foo :bar}))
;; => #<Pair 1,2>

(meta p)
;; => {:foo :bar}

(def p' (with-meta p {:bar :baz}))
;; => #<Pair 1,2>

(meta p')
;; => {:bar :baz}
```

JavaScript との相互運用

ClojureScript は JavaScript の VM でホストされるため、ClojureScript のデータ構造を JavaScript のデータ構造に変換したり、また、その逆を行う必要があります。native の JavaScript の型をプロトコルで表現される抽象化に加えることもできます。

■JavaScript の型を拡張する JavaScript のオブジェクトを拡張する場合は、`js/String` や `js/Date` などの JavaScript のグローバル環境を使用せずに、特別なシンボルが使用されます。これは、グローバル環境にある JavaScript オブジェクトの変更を避けるために行われます。

JavaScript の型を拡張するシンボルには、`object`、`array`、`number`、`string`、`function`、`boolean`、`nil` があります。プロトコルを native オブジェクトにディスパッチするには、Google Closure の goog.typeOf 関数を使います。全ての型のためのプロトコルを実装するために使われる特別なデフォルトのシンボルがあります。

JavaScript の型の拡張を説明するために、`mutable?` 関数を用いて `MaybeMutable` プロトコルを実装します。JavaScript の可変性はデフォルトなので、`mutable?` から true を返すデフォルトの JavaScript の型を拡張します。:

```
(defprotocol MaybeMutable
  (mutable? [this]
    "Returns true if the value is mutable."))

(extend-type default
  MaybeMutable
  (mutable? [_] true))

;; object
(mutable? #js {})
;; => true

;; array
(mutable? #js [])
;; => true

;; string
(mutable? "")
;; => true

;; function
(mutable? (fn [x] x))
;; => true
```

幸い、全ての JavaScript オブジェクトの値が可変という訳ではないため、文字列と関数に対して `false` を返すように `MaybeMutable` の実装を改良することができます。

```
(extend-protocol MaybeMutable
  string
  (mutable? [_] false)

  function
  (mutable? [_] false))

;; object
(mutable? #js {})
;; => true

;; array
(mutable? #js [])
;; => true

;; string
(mutable? "")
;; => false

;; function
(mutable? (fn [x] x))
;; => false
```

JavaScript では date に特別な記号はないので、`js/Date` を直接拡張する必要があります。JavaScript のグローバルな名前空間にある残りの型にも同じことが当てはまります。

■データの変換 ClojureScript の型から JavaScript の型、あるいはその逆を行うために、`IEncodeJS` と `IEncodeClojure` プロトコルにそれぞれ基づいた `clj->js` と `js->clj` 関数を使います。

この例では、ES6 で導入された Set 型を使用します。これは、全ての JavaScript の実行環境で使用できるわけではありません。

■ClojureScript から JavaScript まず、ClojureScript の set 型を拡張して JavaScript に変換できるようにします。デフォルトでは、set は JavaScript の配列に変換されます。

```
(clj->js #{1 2 3})
;; => #js [1 3 2]
```

これを修正しましょう。`clj->js` は値を再帰的に変換するので、set の要素を全て JavaScript に変換して、変換された値を使って set を作成します。

```
(extend-type PersistentHashSet
  IEncodeJS
  (-clj->js [s]
    (js/Set. (into-array (map clj->js s)))))
```

```
(def s (clj->js #{1 2 3}))
(es6-iterator-seq (.values s))
;; => (1 3 2)

(instance? js/Set s)
;; => true

(.has s 1)
;; => true
(.has s 2)
;; => true
(.has s 3)
;; => true
(.has s 4)
;; => false
```

`es6-iterator-seq` は、ES6 のイテレブルから seq を取得するための ClojureScript core にある実験的な関数です。

■JavaScript から ClojureScript 今度は JavaScript のセットを拡張して ClojureScript に変換します。`clj->js` と同様に、`js->clj` はデータ構造の値を再帰的に変換します。

```
(extend-type js/Set
  IEncodeClojure
  (-js->clj [s options]
    (into #{} (map js->clj (es6-iterator-seq (.values s))))))

(= #{1 2 3}
   (js->clj (clj->js #{1 2 3})))
;; => true

(= #{[1 2 3] [4 5] [6]}
   (js->clj (clj->js #{[1 2 3] [4 5] [6]})))
;; => true
```

ClojureScript と JavaScript の値は、一対一のマッピングはないことに注意してください。たとえば、ClojureScript のキーワードは、`clj->js` に渡されると JavaScript の文字列に変換されます。

リダクション

`reduce` 関数は `IReduce` プロトコルに基づいており、このプロトコルを使用すると、独自の型やサードパーティの型を reducible にすることができます。`reduce` と一緒に使う以外にも、自動的に `transduce` と一緒に動作するので、transducer を使って reduce を行うことができます。

JavaScript の配列は ClojureScript でも reduce できます。

```
(reduce + #js [1 2 3])
;; => 6

(transduce (map inc) conj [] [1 2 3])
;; => [2 3 4]
```

しかし、新しい ES6 に Set 型はそうではないので、IReduce プロトコルを実装しましょう。Set の values メソッドを使ってイテレーターを取得して、これを es6-iterator-seq 関数を使って seq に変換します。その後、元の reduce 関数にデリゲートして、取得したシーケンスを reduce します。

```
(extend-type js/Set
  IReduce
  (-reduce
    ([s f]
     (let [it (.values s)]
       (reduce f (es6-iterator-seq it))))
    ([s f init]
     (let [it (.values s)]
       (reduce f init (es6-iterator-seq it))))))

(reduce + (js/Set. #js [1 2 3]))
;; => 6

(transduce (map inc) conj [] (js/Set. #js [1 2 3]))
;; => [2 3 4]
```

連想的なデータ構造は IKVReduce プロトコルに基づく reduce - kv 関数を用いて reduce できます。reduce と reduce-kv の主な違いは、reduce-kv は 3 つの引数をとる関数であり、reducer、アキュムレータの受け取り、各アイテムのキーと値をとることです。

例を見てみましょう。マップをペアからなるベクタに変換します。ベクタはインデックスを値に関連付けるので、reduce-kv を使って値を reduce することができます。

```
(reduce-kv (fn [acc k v]
             (conj acc [k v]))
           []
           {:foo :bar
            :baz :xyz})
;; => [[:foo :bar] [:baz :xyz]]
```

新しい ES6 のマップ型を拡張して、reduce-kv をサポートするようにします。これを行うには、キーと値のペアのシーケンスを取得し、アキュムレータ、キー、値を位置を示す引数として使用して reducing 関数を呼び出します。

```
(extend-type js/Map
  IKVReduce
  (-kv-reduce [m f init]
    (let [it (.entries m)]
      (reduce (fn [acc [k v]]
                (f acc k v))
              init
              (es6-iterator-seq it)))))

(def m (js/Map.))
(.set m "foo" "bar")
(.set m "baz" "xyz")

(reduce-kv (fn [acc k v]
             (conj acc [k v]))
           []
           m)
;; => [["foo" "bar"] ["baz" "xyz"]]
```

どちらの例でも、最終的には reduce 関数にデリゲートすることになりました。reduce 関数は、値の reduce された値を認識して、値が見つかると終了します。これらのプロトコルを reduce に関して実装しない場合は、早期終了のために reduce された値をチェックする必要があることを覚えておいてください。

遅延計算

非同期処理の概念をもついくつかの型があり、それらが表す値はまだ認識 (realize) されていないかもしれません。realized? を使って値が認識されたかどうかを確認することができます。ここでは、計算を行い、結果が必要になったときにそれを実行する Delay 型を使って見ていきましょう。遅延を逆に参照すると、計算が実行されて、遅延が認識されます。

```
(defn computation []
  (println "running!")
  42)

(def d (delay (computation)))

(realized? d)    ;; => false

(deref d)
;; running!
;; => 42

(realized? d)    ;; => true

@d    ;; => 42
```

realized? と deref は IPending と IDeref のプロトコルの上にあります。

状態

Atom や Volatile のような ClojureScript の状態に関する構造体 (state constructs) は、異なる特性とセマンティクスを持ち、add-watch や reset! または swap! のような操作は、プロトコルによってサポートされています。

■Atom このようなプロトコルを理解するために、私たちは独自の簡素版の `Atom` を実装します。バリデータやメタデータはサポートしませんが、次のようなことは可能です。

- deref で現在の値を取得するためのアトムの参照解除する
- reset でアトムに含まれる値をリセットする
- swap! で atom を状態を変える関数を用いて変更する

`deref` は `IDeref` プロトコル、`reset!` は `IReset` プロトコル、`swap!` は `ISwap` プロトコルに基づいています。まず、atom の実装のために、データ型とコンストラクターを定義します。

```
(deftype MyAtom [^:mutable state ^:mutable watches]
  IPrintWithWriter
  (-pr-writer [p writer _]
    (-write writer (str "#<MyAtom " (pr-str state) ">"))))

(defn my-atom
  ([]
   (my-atom nil))
  ([init]
   (MyAtom. init {})))

(my-atom)        ;; => #<MyAtom nil>
(my-atom 42)    ;; => #<MyAtom 42>
```

ここでは、atom の現在の状態と watcher のマップ (watches) の両方を {:mutable true} のメタデータでマークしていることに注目してください。これらを修正して、アノテーションを用いて明示的にします。

この `MyAtom` 型はまだあまり有用ではありませんが、まず `IDeref` プロトコルを実装して、現在の値を逆参照できるようにします。

```
(extend-type MyAtom
  IDeref
  (-deref [a]
    (.-state a)))

(def a (my-atom 42))

@a    ;; => 42
```

参照を解除できるようになったので、IWatchable プロトコルを実装して、独自の atom に watches を追加したり削除したりできるようにします。watches を MyAtom のマップに格納して、キーをコールバックに関連づけます。

```
(extend-type MyAtom
  IWatchable
  (-add-watch [a key f]
    (let [ws (.-watches a)]
      (set! (.-watches a) (assoc ws key f))))

  (-remove-watch [a key]
    (let [ws (.-watches a)]
      (set! (.-watches a) (dissoc ws key))))

  (-notify-watches [a oldval newval]
    (doseq [[key f] (.-watches a)]
      (f key a oldval newval))))
```

atom に watches を追加できるようになりましたが、まだ変更できないのであまり役に立ちません。変更を取り込むためには、IReset プロトコルを実装して、atom の値をリセットするたびに watches に通知する必要があります。

```
(extend-type MyAtom
  IReset
  (-reset! [a newval]
    (let [oldval (.-state a)]
      (set! (.-state a) newval)
      (-notify-watches a oldval newval)
      newval)))
```

ここで、正しく動作するか確認しましょう。watches を追加してから atom の値を変更して、watches が呼び出されることを確認してから削除します。

```
(def a (my-atom 41))
;; => #<MyAtom 41>

(add-watch a :log (fn [key a oldval newval]
                    (println {:key key
                              :old oldval
                              :new newval})))
;; => #<MyAtom 41>

(reset! a 42)
;; {:key :log, :old 41, :new 42}
;; => 42
```

```
(remove-watch a :log)
;; => #<MyAtom 42>

(reset! a 43)
;; => 43
```

私たちが実装を進めている atom にはまだスワップの機能がないので、ここで ISwap プロトコルを実装します。-swap! には 4 つの arity があります。swap! に渡される関数には、1 つ以上の引数をとることができます。

```
(extend-type MyAtom
  ISwap
  (-swap!
    ([a f]
     (let [oldval (.-state a)
           newval (f oldval)]
       (reset! a newval)))

    ([a f x]
     (let [oldval (.-state a)
           newval (f oldval x)]
       (reset! a newval)))

    ([a f x y]
     (let [oldval (.-state a)
           newval (f oldval x y)]
       (reset! a newval)))

    ([a f x y more]
     (let [oldval (.-state a)
           newval (apply f oldval x y more)]
       (reset! a newval)))))
```

これで、独自版の atom 抽象の実装ができました。これを REPL でテストし、期待どおりに動作するかどうかを確認しましょう。

```
(def a (my-atom 0))    ;; => #<MyAtom 0>

(add-watch a :log (fn [key a oldval newval]
                    (println {:key key
                              :old oldval
                              :new newval})))
;; => #<MyAtom 0>

(swap! a inc)
;; {:key :log, :old 0, :new 1}
;; => 1
```

```
(swap! a + 2)
;; {:key :log, :old 1, :new 3}
;; => 3

(swap! a - 2)
;; {:key :log, :old 3, :new 1}
;; => 1

(swap! a + 2 3)
;; {:key :log, :old 1, :new 6}
;; => 6


(swap! a + 4 5 6)
;; {:key :log, :old 6, :new 21}
;; => 21

(swap! a * 2)
;; {:key :log, :old 21, :new 42}
;; => 42

(remove-watch a :log)  ;; => #<MyAtom 42>
```

成功しました! メタデータやバリデータをサポートしないバージョンの Atom を実装しましたが、このような機能をサポートするように拡張することは、読者の皆さんの課題とします。メタデータとバリデーターを保存できるようにするには、MyAtom 型を変更する必要があります。

■Volatile Volatile は atom よりも単純で、変化の監視をサポートしません。大半のプログラミング言語に存在する可変変数のように、全ての変更は前の値を上書きします。Volatile は IVolatile プロトコルに基づいています。IVolatile プロトコルは、vreset! のためのメソッドのみを定義します。これは vswap! がマクロとして実装されているからです。まず、独自の volatile 型とコンストラクタを作成します。

```
(deftype MyVolatile [^:mutable state]
  IPrintWithWriter
  (-pr-writer [p writer _]
    (-write writer (str "#<MyVolatile " (pr-str state) ">"))))

(defn my-volatile
  ([]
    (my-volatile nil))
  ([v]
    (MyVolatile. v)))

(my-volatile)     ;; => #<MyVolatile nil>

(my-volatile 42)  ;; => #<MyVolatile 42>
```

私たちの MyVolatile は、参照の解除と再設定をサポートする必要があります。では、IDeref と IVolatile を実装しましょう。これにより、MyVolatile で deref、vreset!、vswap! が使えるようになります。

```
(extend-type MyVolatile
  IDeref
  (-deref [v]
    (.-state v))

  IVolatile
  (-vreset! [v newval]
    (set! (.-state v) newval)
    newval))

(def v (my-volatile 0))
;; => #<MyVolatile 42>

(vreset! v 1)
;; => 1

@v
;; => 1

(vswap! v + 2 3)
;; => 6

@v
;; => 6
```

変更

transient のセクションでは、ClojureScript が提供する不変で永続的なデータ構造に対応する可変のデータ構造について学びました。これらのデータ構造は可変であり、それらに対する関数は、可変であることを明示するために感嘆符 (!) で終わります。あなたの想像通り、transient に対する全ての操作はプロトコルに基づいています。

■永続性から一時性 (一時性から永続性) IEditableCollection プロトコルに基づく transient 関数を使って、永続的なデータ構造を変換できることを学びました。transient を永続的なデータ構造に変換するには、ITransientCollection に基づく persistent! を使います。

不変で永続的なデータ構造とそれらの一時的な対応物を実装することはこの本の範囲外ですが、興味があれば ClojureScript のデータ構造の実装を見ることをお勧めします。

■Transient のベクタとセット transient のデータ構造のためのプロトコルの大部分は学びましたが、assoc! を transient のベクタに使うための ITransientVector と、disj! を transient のセットに使うための ITransientSet についてまだ取り上げていません。

ITransientVector プロトコルを説明するために、連想的な transient のデータ構造にするために、JavaScript の配列型を拡張してみます。

```
(extend-type array
  ITransientAssociative
  (-assoc! [arr key val]
    (if (number? key)
      (-assoc-n! arr key val)
      (throw (js/Error.
              "Array's key for assoc! must be a number."))))

  ITransientVector
  (-assoc-n! [arr n val]
    (.splice arr n 1 val)
    arr))

(def a #js [1 2 3])   ;; => #js [1 2 3]
(assoc! a 0 42)        ;; => #js [42 2 3]
(assoc! a 1 43)        ;; => #js [42 43 3]
(assoc! a 2 44)        ;; => #js [42 43 44]
```

ITransientSet プロトコルを説明するために、ES6 の Set 型を拡張して、conj!、disj!、persistent! をサポートします。以前に Set 型を拡張して ClojureScript に変換できるようにしたことを思い出してください。

```
(extend-type js/Set
  ITransientCollection
  (-conj! [s v]
    (.add s v)
    s)

  (-persistent! [s]
   (js->clj s))

  ITransientSet
  (-disjoin! [s v]
    (.delete s v)
    s))

(def s (js/Set.))

(conj! s 1)
(conj! s 1)
(conj! s 2)
(conj! s 2)

(persistent! s)    ;; => #{1 2}

(disj! s 1)

(persistent! s)    ;; => #{2}
```

5.5 CSP (core.async を用いた場合)

CSP は Communicating Sequential Processes の略で、1978 年に C.A.R.Hoare によって開発された並行システムを記述するための形式であり、チャネルを介したメッセージの受け渡しと同期に基づく並行性のモデルです。CSP の背後にある理論的モデルの詳細については、この本では説明しません。その代わりに、core.async が提供する並行性に関わるプリミティブに焦点を当てます。

`core.async` は ClojureScript core の一部ではありませんが、ライブラリとして実装されています。言語の一部ではありませんが、広く使用されています。多くのライブラリは core.asynx のプリミティブの上に構築されています。そのため、本書で取り上げる価値はあると思います。これは、ClojureScript のマクロでコードを変換することにより実現できる構文的な抽象化の良い例でもあります。このセクションで説明する例を実行するには、core.async をインストールする必要があります。

チャンネル

チャンネルはベルトコンベアのようなもので、一度に 1 つの値を入れたり出したりできます。複数の reader と writer を持つことができ、core.async の基本的なメッセージパッシングのメカニズムです。これがどのように動作するかを確認するために、いくつかの操作を実行するチャンネルを作成します。

```
(require '[cljs.core.async :refer [chan put! take!]])

(enable-console-print!)

(def ch (chan))

(take! ch #(println "Got a value:" %))
;; => nil

;; there is a now a pending take operation,
;; let's put something on the channel
(put! ch 42)
;; Got a value: 42
;; => 42
```

上記の例では、`chan` コンストラクタを使用してチャネル `ch` を作成しました。その後、チャンネルで take の操作を実行して、take の操作が成功したとき、呼び出されるコールバックを提供します。値をチャンネル上で設定するために put!　を使った後、take の操作が完了して、"Got a value: 42" という文字列が出力されます。put! は、チャンネルに設定されたばかりの値を返します。

`take!` とは異なり、`put!` 関数はコールバックを受け取りませんが、前の例では何も提供していません。put の場合、コールバックは指定した値が取得されるたびに呼び出されます。put と take は任意の順序で発生しますが。ポイントを説明するために、いくつか put と take を実行してみます。

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan))

(put! ch 42 #(println "Just put 42"))  ;; => true

(put! ch 43 #(println "Just put 43"))  ;; => true

(take! ch #(println "Got" %))
;; Got 42
;; Just put 42
;; => nil

(take! ch #(println "Got" %))
;; Got 43
;; Just put 43
;; => nil
```

あなたはなぜ put! が true を返したか疑問に思っているかもしれません。まだ値が取得されていない場合でも、put の操作を実行できることを示します。チャンネルは閉じることができ、これにより put の操作が失敗します。

```
(require '[cljs.core.async :refer [chan put! close!]])

(def ch (chan))

(close! ch)    ;; => nil

(put! ch 42)   ;; => false
```

上記の例は最も単純な状況ですが、チャンネルが close されたとき、保留中の操作では何が起こるのでしょうか。いくつか take を実行してチャンネルを閉じて、何が起こるか見てみましょう。

```
(require '[cljs.core.async :refer [chan put! take! close!]])

(def ch (chan))

(take! ch #(println "Got value:" %))    ;; => nil
(take! ch #(println "Got value:" %))    ;; => nil

(close! ch)
;; Got value: nil
;; Got value: nil
;; => nil
```

チャンネルが閉じている場合は、すべての take! の操作は nil の値を受け取ります。チャンネルにおいて nil は、チャネルがクローズされたことを受信者に知らせるための指標値です。そのため、チャネルに nil の値を設定することはできません。

```
(require '[cljs.core.async :refer [chan put!]])

(def ch (chan))

(put! ch nil)
;; Error: Assert failed: Can't put nil in on a channel
```

■バッファ 保留中の take と put の操作は、チャンネルで enqueue されていますが、保留中の take の操作や put の操作が多数ある場合はどうなるでしょうか。put と take の多いチャンネルを叩いて確認してみましょう。

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan))

(dotimes [n 1025]
  (put! ch n))
;; Error: Assert failed: No more than 1024
;; pending puts are allowed on a single channel.

(def ch (chan))

(dotimes [n 1025]
  (take! ch #(println "Got" %)))
;; Error: Assert failed: No more than 1024
;; pending takes are allowed on a single channel.
```

上記の例が示しているように、チャンネルには保留中の put または take には制限があり、現在は 1024 ですが、これは変更される可能性のある実装の詳細です。チャンネルには保留中の put と take の両方が存在することはできません。保留中の take がある場合、またはその逆の場合、put はすぐに成功するためです。

チャンネルは put の操作のバッファリングをサポートします。バッファーを持つチャンネルを作成する場合、バッファーに空きがあれば put の操作は即座に成功して、そうでなければキューに入れられます。1 つの要素のバッファでチャンネルを作成するポイントを説明します。chan コンストラクタは最初の引数として数値を受け取り、指定されたサイズのバッファをもちます。

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan 1))

(put! ch 42 #(println "Put succeeded!"))
;; Put succeeded!
;; => true

(dotimes [n 1024]
  (put! ch n))
;; => nil

(put! ch 42)
;; Error: Assert failed: No more than 1024
;; pending puts are allowed on a single channel.
```

上の例では何が起こったのでしょうか。サイズ 1 のバッファーをもつチャンネルを作成して、そのチャンネルに対して put の操作を実行しましたが、値がバッファーに入れられたため、すぐに成功しました。その後、保留中の put の queue を埋めるためにさらに put の操作を 1024 回行い、さらに 1 つの値を put しようとすると、チャンネルはそれ以上の put を enqueue できないと警告を発しました。

チャンネルがどのように機能して、どのバッファがどのように使われるわかったところで、次に core.async が実装している異なるバッファについて見ていきましょう。それぞれのバッファには異なるポリシーがあり、それぞれのバッファがいつ何を使うべきかを知るのは興味深いことです。デフォルトでは、チャンネルはバッファされません。

■Fixed 固定サイズのバッファは、chan コンストラクタに数値を指定したときに作成され、その数値で指定されたサイズになります。これは可能な限り最も単純なバッファで、いっぱいになると put が queue に入れられます。

chan コンストラクタは、最初の引数として数値またはバッファのいずれかを受け入れます。次の例で作成した 2 つのチャンネルは、どちらもサイズ 32 の固定バッファを使用します。

```
(require '[cljs.core.async :refer [chan buffer]])

(def a-ch (chan 32))

(def another-ch (chan (buffer 32)))
```

■Dropping 固定バッファーを使用すると、put の操作を enqueue できます。しかし前に見たように、固定バッファがいっぱいになっても put は queue に入れられます。バッファーがいっぱいになったときに発生する put 操作を破棄したくない場合は、dropping バッファを使用できます。

dropping バッファのサイズは固定されており、いっぱいになると書き込みは完了しますが、その値は破棄されます。例を見てみましょう。

```
(require '[cljs.core.async :refer [chan dropping-buffer put! take!]])

(def ch (chan (dropping-buffer 2)))

(put! ch 40)
;; => true
(put! ch 41)
;; => true
(put! ch 42)
;; => true

(take! ch #(println "Got" %))
;; Got 40
;; => nil
(take! ch #(println "Got" %))
;; Got 41
;; => nil
(take! ch #(println "Got" %))
;; => nil
```

3 つの put の操作を実行して、3 つとも成功しましたが、チャンネルの dropping バッファのサイズ が 2 であるため、最初の 2 つの値だけが taker に渡されました。利用できる値がないため 3 番目の take が enqueue されているのがわかりますが、3 番目の出力値 (42) は破棄されています。

■スライド sliding バッファは、dropping バッファとは逆のポリシーを持ちます。許容値の put が完了すると、最も古い値は新しい値に置き換えられます。sliding バッファは、最後の put だけを処理して、古い値を破棄できる場合に便利です。

```
(require '[cljs.core.async :refer [chan sliding-buffer put! take!]])

(def ch (chan (sliding-buffer 2)))

(put! ch 40)
;; => true
(put! ch 41)
;; => true
(put! ch 42)
;; => true

(take! ch #(println "Got" %))
;; Got 41
;; => nil
(take! ch #(println "Got" %))
;; Got 42
;; => nil
(take! ch #(println "Got" %))
;; => nil
```

3 つの put の操作を行い、そのうちの 3 つは成功しましたが、チャンネルの sliding バッファのサイズが 2 であるため、最後の 2 つの値だけが取得側に渡されました。ご覧のとおり、最初の put の値が破棄されてから値が使用できないため、3 番目の take が enqueue されます。

■Transducer　transducer の節でも見たように、チャンネルに値を入れることは transducible なプロセスと捉えることができます。つまり、チャンネルを作成して transducer を渡すことで、入力値を変換してからチャンネルに入れることができます。

チャンネルを持つ transducer を使用したい場合、transducer によって修正される reducing 関数がバッファの add 関数となるので、バッファを供給しなければなりません。バッファの add 関数は、バッファと入力を受け取り、そのような入力を組み込みながらバッファを返すため reducing 関数です。

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan 1 (map inc)))

(put! ch 41)    ;; => true

(take! ch #(println "Got" %))
;; Got 42
;; => nil
```

reducing 関数が reduce された値を返したとき、チャンネルはどうなるのでしょうか。チャンネルを終了する概念はクローズされているため、reduce された値が検出されるとチャンネルはクローズされます。

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan 1 (take 2)))

(take! ch #(println "Got" %))    ;; => nil
(take! ch #(println "Got" %))    ;; => nil
(take! ch #(println "Got" %))    ;; => nil

(put! ch 41)
;; => true
(put! ch 42)
;; Got 41
;; => true
(put! ch 43)
;; Got 42
;; Got nil
;; => false
```

チャンネルに最大 2 回の入力を許可するので、take、stateful、transducer を使用しました。次に、チャンネルで 3 つの take の操作を実行して、値を受け取るのは 2 つだけだと予想します。上の例でわかるように、3 番目の take はチャンネルが閉じていることを示す sentinel nil 値を取得します。また、3 回目の put の操作は false を返して、失敗したことを示します。

■例外処理　バッファに値を追加するとして例外が発生した場合、core.async では操作が失敗し、例外がコンソールに記録されます。しかし、チャンネルのコンストラクターは 3 番目の引数 (例外を処理する関数) を受け入れます。

例外ハンドラを持つチャンネルを作成すると、例外が発生するたびに例外ハンドラが呼び出されます。ハンドラが nil を返した場合は操作は暗黙的に失敗して、別の値を返した場合はその値で add の操作が再試行されます。

```
(require '[cljs.core.async :refer [chan put! take!]])

(enable-console-print!)

(defn exception-xform
  [rfn]
  (fn [acc input]
    (throw (js/Error. "I fail!"))))

(defn handle-exception
  [ex]
  (println "Exception message:" (.-message ex))
  42)

(def ch (chan 1 exception-xform handle-exception))

(put! ch 0)
;; Exception message: I fail!
;; => true

(take! ch #(println "Got:" %))
;; Got: 42
;; => nil
```

■Offer と Poll　これまでに、チャンネルに関する 2 つの基本的な操作である put! と take!について学びました。それらは、値を取得または設定しますが、すぐに実行できない場合は enqueue されます。どちらの関数もその性質的に非同期であり、すぐに成功することができますが、後で完了することもできます。

core.async には値の設定と取得のために 2 つの同期的な操作があります。offer! と poll! です。それらがどのように機能するかを例を見てみましょう。

offer! はすぐに値を設定できる場合は、チャンネルに値を設定します。チャンネルが値を受け取った場合は true を返し、それ以外の場合は false を返します。put! とは異なり、offer! は閉じたチャンネルと開いたチャンネルを区別できない点に注意してください。

```
(require '[cljs.core.async :refer [chan offer!]])

(def ch (chan 1))

(offer! ch 42)
;; => true

(offer! ch 43)
;; => false
```

poll! がすぐに値を取得できる場合は、チャンネルから値を取得します。成功した場合は値を返し、それ以外の場合は nil を返します。take! とは異なり、poll! は閉じたチャンネルと開いたチャンネルを区別できません。

```
(require '[cljs.core.async :refer [chan offer! poll!]])

(def ch (chan 1))

(poll! ch)
;; => nil

(offer! ch 42)
;; => true

(poll! ch)
;; => 42
```

プロセス

チャンネルについてはすべて学びましたが、プロセスについては詳しく取り上げていませんでした。プロセスは、独立して実行されて、通信と調整のためにチャンネルを使用するロジックです。プロセス内での put および take は、操作が完了するとプロセスが停止します。プロセスの停止により ClojureScript が実行される環境にあるスレッドだけがブロックされるわけではありません。代わりに、操作の実行を待機しているときに再開されます。

プロセスは go マクロを使って起動されて、put と take には <! および >! のプレースホルダをつけます。go マクロはコールバックを使用するようにコードを書き換えますが、go の内部ではすべてが同期的なコードに見えるため、理解が容易です。

```
(require '[cljs.core.async :refer [chan <! >!]])
(require-macros '[cljs.core.async.macros :refer [go]])
```

```
(enable-console-print!)

(def ch (chan))

(go
  (println [:a] "Gonna take from channel")
  (println [:a] "Got" (<! ch)))

(go
  (println [:b] "Gonna put on channel")
  (>! ch 42)
  (println [:b] "Just put 42"))

;; [:a] Gonna take from channel
;; [:b] Gonna put on channel
;; [:b] Just put 42
;; [:a] Got 42
```

上記の例では、ch から値を取得してコンソールに出力するプロセスを go で起動しています。値がすぐに使用可能になるわけではないので、値が使用可能になるまで一時停止します。その後、チャンネルに価値をもたらす別のプロセスを開始します。

保留中の take があるため、put の操作は成功します。値が最初のプロセスに渡されると、両方のプロセスが終了します。

どちらの go ブロックも独立して実行されて非同期に実行されますが、同期的なコードに見えます。上記の go ブロックはかなり単純ですが、チャンネルを介して調整する並行プロセスを書けることは、複雑な非同期のワークフローを実装するために非常に強力なツールとなります。チャンネルはまた、producer と consumer を分離します。

プロセスは任意の時間だけ待つことができ、与えられたミリ秒後に閉じられるチャンネルを返す timeout 関数があります。go ブロック内で timeout チャンネルと take の操作を組み合わせると、スリープさせることができます。

```
(require '[cljs.core.async :refer [<! timeout]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(defn seconds
  []
  (.getSeconds (js/Date.)))

(println "Launching go block")

(go
  (println [:a] "Gonna take a nap" (seconds))
  (<! (timeout 1000))
  (println [:a] "I slept one second, bye!" (seconds)))
```

```
(println "Block launched")

;; Launching go block
;; Block launched
;; [:a] Gonna take a nap 9
;; [:a] I slept one second, bye! 10
```

出力されたメッセージからわかるように、プロセスは、timeout チャンネルの take の操作でブロックされたときには、1 秒間何もしません。プログラムは続行されて、1 秒後にプロセスが再開するか終了します。

■選択 go ブロック内で一度に 1 つの値を設定や取得するだけでなく、alts! を使って複数のチャンネル操作に非決定的な選択をすることもできます。一連のチャンネルの put または take の操作が与えられ (チャンネルで put と take は同時に行うこともできます)、準備が整うとすぐに実行されます。複数の操作が alts! を呼び出すときに実行できれば、擬似的なランダムの選択を行います。

timeout 関数と ats! を組み合わせることで、簡単にチャンネルの操作を試して、一定時間後に取り消すことができます。その方法を見てみましょう。

```
(require '[cljs.core.async :refer [chan <! timeout alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def ch (chan))

(go
  (println [:a] "Gonna take a nap")
  (<! (timeout 1000))
  (println [:a] "I slept one second, trying to put a value on channel")
  (>! ch 42)
  (println [:a] "I'm done!"))

(go
  (println [:b] "Gonna try taking from channel")
  (let [cancel (timeout 300)
        [value ch] (alts! [ch cancel])]
    (if (= ch cancel)
      (println [:b] "Too slow, take from channel cancelled")
      (println [:b] "Got" value))))

;; [:a] Gonna take a nap
;; [:b] Gonna try taking from channel
;; [:b] Too slow, take from channel cancelled
;; [:a] I slept one second, trying to put a value on channel
```

上の例では、1 秒待ってから ch チャンネルに値を入れる go ブロックを起動しました。もう一方の go ブロックは、300 ミリ秒後に閉じる cancel チャンネルを作成します。その後、alts! を使って ch からの読み込みとキャンセルを同時に行おうとします。これらいずれかのチャンネルから値を取得できる場合は、常に成功します。cancel は 300 ミリ秒後にクローズされて、閉じたチャンネルからの take が nil sentinel を返すため、alts! は成功します。alts! が 2 つの要素からなるベクタを返すことに注目してください。それらは、操作による返り値と、実行されたチャンネルを含みます。

このため、candel チャンネルで read の操作が行われたか、ch チャンネルで行われたかを検出することができます。この例をコピーして、最初のプロセスの timeout を 100 ミリ秒に設定して、ch チャンネルに対する read の操作がどのように成功するかを確認することをお勧めします。

read の操作間での選択方法を学んだので、alt! で条件付きの read の操作を表現する方法を見てみましょう。チャンネルとその上に置こうとする値を提供する必要があるため、チャンネルと write の操作を表す値を持つ 2 つの要素のベクタを使用します。

```
(require '[cljs.core.async :refer [chan <! alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def a-ch (chan))
(def another-ch (chan))

(go
  (println [:a] "Take a value from  a-ch ")
  (println [:a] "Got" (<! a-ch))
  (println [:a] "I'm done!"))

(go
  (println [:b] "Take a value from  another-ch ")
  (println [:a] "Got" (<! another-ch))
  (println [:b] "I'm done!"))

(go
  (println [:c] "Gonna try putting in both channels simultaneously")
  (let [[value ch] (alts! [[a-ch 42]
                           [another-ch 99]])]
    (if (= ch a-ch)
      (println [:c] "Put a value in  a-ch ")
      (println [:c] "Put a value in  another-ch "))))

;; [:a] Take a value from  a-ch
;; [:b] Take a value from  another-ch
;; [:c] Gonna try putting in both channels simultaneously
;; [:c] Put a value in  a-ch
;; [:a] Got 42
;; [:a] I'm done!
```

上記の例を実行すると、a-ch チャンネルでの put の操作だけが成功します。両方のチャンネルは、値を取得する準備ができています。このコードを実行すると異なる結果が得られることがあります。

■優先順位　デフォルトで alt! は、複数の操作を実行する準備ができている場合は、常に非決定的な選択を行いますが、代わりに alts! に :priority のオプションを渡す操作を優先させることができます。:priority が true の場合、もし 複数の操作が準備できていれば、順番に試行されます。

```
(require '[cljs.core.async :refer [chan >! alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def a-ch (chan))
(def another-ch (chan))

(go
  (println [:a] "Put a value on  a-ch ")
  (>! a-ch 42)
  (println [:a] "I'm done!"))

(go
  (println [:b] "Put a value on  another-ch ")
  (>! another-ch 99)
  (println [:b] "I'm done!"))

(go
  (println [:c] "Gonna try taking from both channels with priority")
  (let [[value ch] (alts! [a-ch another-ch] :priority true)]
    (if (= ch a-ch)
      (println [:c] "Got" value "from  a-ch ")
      (println [:c] "Got" value "from  another-ch "))))

;; [:a] Put a value on  a-ch
;; [:a] I'm done!
;; [:b] Put a value on  another-ch
;; [:b] I'm done!
;; [:c] Gonna try taking from both channels with priority
;; [:c] Got 42 from  a-ch
```

a-ch と another-ch は両方とも、いつ alts! を read するかの値を持っており、:priority オプションを true に設定すると a-ch が優先されます。:priority オプションを削除してこの例を複数回実行すると、alts! は priority なしで非決定論的な選択をします。

■デフォルト　alts! についてもう一つの興味深いことに、操作の準備ができておらず、デフォルト値を指定すれば、すぐに戻ることができます。いずれかの操作が準備できている場合に限り、条件付きで操作を選択することができます。準備できていない場合にはデフォルト値を返します。

```
(require '[cljs.core.async :refer [chan alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(def a-ch (chan))
```

```
(def another-ch (chan))

(go
  (println [:a] "Gonna try taking from any of the channels without blocking")
  (let [[value ch] (alts! [a-ch another-ch] :default :not-ready)]
    (if (and (= value :not-ready)
             (= ch :default))
      (println [:a] "No operation is ready, aborting")
      (println [:a] "Got" value))))
;; [:a] Gonna try taking from any of the channels without blocking
;; [:a] No operation is ready, aborting
```

上の例でわかるように、操作の準備ができていない場合は、alts! による返される値は、呼び出し時に :default キーの後に指定したものであり、channel は :default キーワード自体です。

コンビネーター

チャンネルとプロセスを理解したところで、次に、core.syanc に存在するチャンネルを扱うための興味深いコンビネーターについて検討します。このセクションでは、これらすべてについて簡単に説明して、簡単な使用例を示します。

■パイプ pipe は入力チャンネルと出力チャンネルを取り、入力チャンネルに設定されたすべての値を出力チャンネルに渡します。3 番目の引数に false を指定しない限り、ソースが閉じられるたびに出力チャンネルが閉じられます。

```
(require '[cljs.core.async :refer [chan pipe put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))

(pipe in out)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 0)  ;; => true
(put! in 1)  ;; => true
(close! in)

;; [:a] Got 0  ;; [:a] Waiting for a value
;; [:a] Got 1  ;; [:a] Waiting for a value
;; [:a] I'm done!
```

上記の例では、go-loop マクロを使用して、out チャンネルが閉じられるまで値を繰り返し読み取ります。in チャンネルを閉じると、out チャンネルも閉じられて go-loop が終了します。

■pipeline-async　pipeline-async は ある数字を受け取りますが、その数字は、並列性の制御、出力チャンネル、非同期の関数、入力チャンネルのためのものです。非同期の関数には 2 つの引数があり、入力チャンネルに格納された値と、非同期操作の結果を格納するチャンネルのために使います。result チャンネルは終了後に閉じられます。その数は、入力を使用して非同期関数を呼び出すために使用される並行的に動作する go ブロックの数を制御します。

出力チャンネルは、非同期関数の呼び出しが完了するまでに要する時間に関係なく、入力チャンネルに相対的な順序で出力を受け取ります。これにはオプションの last パラメータがあり、入力チャンネルが閉じられたときに出力チャンネルを閉じるかどうかを制御します。デフォルトは true です。

```
(require '[cljs.core.async :refer [chan pipeline-async put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))
(def parallelism 3)

(defn wait-and-put [value ch]
  (let [wait (rand-int 1000)]
    (js/setTimeout (fn []
                     (println "Waiting" wait "miliseconds for value" value)
                     (put! ch wait)
                     (close! ch))
                   wait)))

(pipeline-async parallelism out wait-and-put in)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 1)
(put! in 2)
(put! in 3)
(close! in)
;; Waiting 164 miliseconds for value 3
;; Waiting 304 miliseconds for value 2
;; Waiting 908 miliseconds for value 1
;; [:a] Got 908
;; [:a] Waiting for a value
;; [:a] Got 304
;; [:a] Waiting for a value
;; [:a] Got 164
;; [:a] Waiting for a value
;; [:a] I'm done!
```

■pipeline pipeline `pipeline-async` に似ていますが、非同期の関数を取る代わりに transducer を使います。 transducer は各入力に独立して適用されます。

```
(require '[cljs.core.async :refer [chan pipeline put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))
(def parallelism 3)

(pipeline parallelism out (map inc) in)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 1)
(put! in 2)
(put! in 3)
(close! in)

;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] Got 3
;; [:a] Waiting for a value
;; [:a] Got 4
;; [:a] Waiting for a value
;; [:a] I'm done!
```

■split `split` は述部とチャンネルを取り、2 つのチャンネルを持つベクタを返します。最初のチャンネルは述部が true の値を受け取り、2 番目のチャンネルは述部が false の値を受け取ります。オプションで 3 番目 (true チャンネル) と 4 番目 (false チャンネル) の引数を使って、チャンネルのバッファまたは数字を渡すことができます。

```
(require '[cljs.core.async :refer [chan split put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def chans (split even? in))
(def even-ch (first chans))
(def odd-ch (second chans))
```

```
(go-loop [value (<! even-ch)]
  (if (nil? value)
    (println [:evens] "I'm done!")
    (do
      (println [:evens] "Got" value)
      (println [:evens] "Waiting for a value")
      (recur (<! even-ch)))))

(go-loop [value (<! odd-ch)]
  (if (nil? value)
    (println [:odds] "I'm done!")
    (do
      (println [:odds] "Got" value)
      (println [:odds] "Waiting for a value")
      (recur (<! odd-ch)))))

(put! in 0)
(put! in 1)
(put! in 2)
(put! in 3)
(close! in)
;; [:evens] Got 0
;; [:evens] Waiting for a value
;; [:odds] Got 1
;; [:odds] Waiting for a value
;; [:odds] Got 3
;; [:odds] Waiting for a value
;; [:evens] Got 2
;; [:evens] Waiting for a value
;; [:evens] I'm done!
;; [:odds] I'm done!
```

■reduce `reduce` は、reducing 関数、初期値、および input チャンネルをとります。指定された初期値を seed として使用して、input チャンネルに設定されたすべての値に reduce を行い、閉じたチャンネルを返します。

```
(require '[cljs.core.async :as async :refer [chan put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(def in (chan))

(go
  (println "Result" (<! (async/reduce + (+) in))))

(put! in 0)
(put! in 1)
(put! in 2)
(put! in 3)
(close! in)
;; Result: 6
```

■onto-chan　onto-chan はチャンネルとコレクションを取得して、コレクションの内容をチャンネルに格納します。終了後にチャンネルを閉じますが、チャンネルを閉じるかどうかを指定する 3 番目の引数を受け入れます。onto-chan を用いて reduce の例を書きかえてみましょう。

```
(require '[cljs.core.async :as async :refer [chan put! <! close! onto-chan]])
(require-macros '[cljs.core.async.macros :refer [go]])

(def in (chan))

(go
  (println "Result" (<! (async/reduce + (+) in))))

(onto-chan in [0 1 2 3])

;; Result: 6
```

■to-chan　to-chan はコレクションを取得して、コレクション内のすべての値を格納するチャンネルを返し、その後、チャンネルを閉じます。

```
(require '[cljs.core.async :refer [chan put! <! close! to-chan]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def ch (to-chan (range 3)))

(go-loop [value (<! ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! ch)))))

;; [:a] Got 0
;; [:a] Waiting for a value
;; [:a] Got 1
;; [:a] Waiting for a value
;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] I'm done!
```

■merge　merge は入力チャンネルのコレクションを取得して、入力チャンネルに設定されたすべての値を格納するチャンネルを返します。すべての入力チャンネルを閉じると、返されたチャンネルが閉じます。返されるチャンネルはデフォルトではバッファされませんが、最後の引数として数値またはバッファを指定できます。

```
(require '[cljs.core.async :refer [chan put! <! close! merge]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in1 (chan))
(def in2 (chan))
(def in3 (chan))

(def out (merge [in1 in2 in3]))

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in1 1)
(close! in1)
(put! in2 2)
(close! in2)
(put! in3 3)
(close! in3)

;; [:a] Got 3
;; [:a] Waiting for a value
;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] Got 1
;; [:a] Waiting for a value
;; [:a] I'm done!
```

高度な抽象化

core.async の低レベルなプリミティブと、それらがチャンネルと動作するために提供されているコンビネータについて学びました。さらに core.async は、より高度な機能の構成要素として機能するチャンネルの上に、役に立つ高レベルの抽象化を提供します。

■Mult 値を多くの他のチャンネルにブロードキャストする必要があるチャンネルがある場合は、mult を使用して、指定されたチャンネルを複数作成できます。モジュールができると tap を用いてチャンネルに接続し、untap してチャンネルを切り離します。また、mults は untap-all を使用して、タップしたすべてのチャンネルを一度に削除することもできます。

mult のソースチャンネルに入力されたすべての値は、タップされたすべてのチャンネルにブロードキャストされて、次のアイテムがブロードキャストされる前にすべてのチャンネルがそれを受け入れる必要があります。遅い taker が mult の値をブロックするのを防ぐためには、タップしたチャンネルのバッファリングを慎重に使用する必要があります。

クローズタップしたチャンネルは自動的にミュートから解除されます。まだタップがない状態でソースチャンネルに値を設定すると、その値はドロップされます。

```
(require '[cljs.core.async :refer [chan put! <! close! timeout mult tap]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Source channel and mult
(def in (chan))
(def m-in (mult in))

;; Sink channels
(def a-ch (chan))
(def another-ch (chan))

;; Taker for  a-ch
(go-loop [value (<! a-ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (recur (<! a-ch)))))

;; Taker for  another-ch , which sleeps for 3 seconds between takes
(go-loop [value (<! another-ch)]
  (if (nil? value)
    (println [:b] "I'm done!")
    (do
      (println [:b] "Got" value)
      (println [:b] "Resting 3 seconds")
      (<! (timeout 3000))
      (recur (<! another-ch)))))

;; Tap the two channels to the mult
(tap m-in a-ch)
(tap m-in another-ch)

;; See how the values are delivered to  a-ch  and  another-ch
(put! in 1)
(put! in 2)

;; [:a] Got 1
;; [:b] Got 1
;; [:b] Resting for 3 seconds
;; [:a] Got 2
;; [:b] Got 2
;; [:b] Resting for 3 seconds
```

■Pub-sub `mult` について学んだ後で、`mult` の上に `pub-sub` の抽象化を実装する方法を想像してみてください。`core.async` はすでにこの機能を実装しています。

source チャンネルから mult を作成する代わりに、`pub` にチャンネル、またメッセージのトピックを抽出するために使用する関数を指定して publicztion を作成します。

`sub` を持つ publicztion を subscribe して、subscribe したい publicztion、関心のあるトピック、そのトピックを持つメッセージを配置するチャンネルを指定できます。1 つのチャンネルを複数のトピックに subscribe できます。

unsub には、そのようなチャンネルをトピックから unsubscribe するための publication、トピック、チャンネルを指定できます。unsub-all には、publication とトピックを指定して、指定したトピックからすべてのチャンネルを unsubscribe できます。

```
(require '[cljs.core.async :refer [chan put! <! close! pub sub]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Source channel and publication
(def in (chan))
(def publication (pub in :action))

;; Sink channels
(def a-ch (chan))
(def another-ch (chan))

;; Channel with  :increment  action
(sub publication :increment a-ch)

(go-loop [value (<! a-ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Increment:" (inc (:value value)))
      (recur (<! a-ch)))))

;; Channel with  :double  action
(sub publication :double another-ch)

(go-loop [value (<! another-ch)]
  (if (nil? value)
    (println [:b] "I'm done!")
    (do
      (println [:b] "Double:" (* 2 (:value value)))
      (recur (<! another-ch)))))

;; See how values are delivered to  a-ch  and  another-ch  depending on their action
(put! in {:action :increment :value 98})
(put! in {:action :double :value 21})

;; [:a] Increment: 99
;; [:b] Double: 42
```

■Mixer core.async のセクションで学んだように、複数のチャンネルを 1 つに結合する merge 関数を使用できます。複数のチャンネルを merge する場合、入力チャンネルに入力されたすべての値はマージされたチャンネルになります。ただし、入力チャンネルに入力された値が出力チャンネルに出力されるようにするには、より細かく制御する必要がある場合があります。この場合、Mixer が便利です。

core.async では、複数の入力チャンネルを 1 つの出力チャンネルに結合するために使用できる Mixer の抽象化が提供されます。Mixer の面白いところは、特定の input チャンネルだけをミュート、一時停止、再生できることです。

`mix` で出力チャンネルを指定して Mixer を作成できます。Mixer ができたら、`admix` を使って入力チャンネルを mix に追加したり、`unmix` を使って削除したり、`unmix-all` を使ってすべての入力チャンネルを削除したりできます。

入力チャンネルの状態を制御するために、Mixer とチャンネルからその状態へのマップを与える toggle 関数を使用します。マップは Mix の現在の状態とマージされるため、`toggle` を使用してチャンネルを Mix に追加できます。チャンネルの状態はマップで、boolean にマップされる `:mute :pause :sole` のキーを持つことができます

チャンネルの mute、pause、sole の意味を見てみましょう。

- ミュートされた入力チャンネルは、値を取得している間は出力チャンネルに転送されないことを意味します。したがって、チャンネルがミュートされている間は、そのチャンネルに入力されているすべての値が破棄されます。

- 一時停止された input チャンネルは、そこから値を取得しないことを意味します。つまり、チャンネルに入力された値は出力チャンネルに転送されず、破棄されません。

- 1 つまたは複数のチャンネルを sole にすると、出力チャンネルは sole に設定されたチャンネルの値だけを受け取ります。デフォルトでは、sole されていないチャンネルはミュートされますが、sole されていないチャンネルをミュートするか一時停止するかは sole モードで決定できます。

多くの情報があったので、理解を深めるための例を見てみましょう。まず、アウトチャンネルの Mixer を設定し、ミックスに 3 つの入力チャンネルを追加します。その後、出力チャンネルで受信したすべての値を出力して、入力チャンネルへの制御を見てみましょう。

```
(require '[cljs.core.async :refer [chan put! <! close! mix admix
                                   unmix toggle solo-mode]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Output channel and mixer
(def out (chan))
(def mixer (mix out))

;; Input channels
(def in-1 (chan))
(def in-2 (chan))
(def in-3 (chan))

(admix mixer in-1)
(admix mixer in-2)
(admix mixer in-3)

;; Let's listen to the  out  channel and print what we get from it
(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done")
    (do
      (println [:a] "Got" value)
      (recur (<! out)))))
```

デフォルトでは、入力チャンネルに設定されたすべての値が出力チャンネルに設定されます。

```
(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 2
;; [:a] Got 3
```

n-2 チャンネルを一時停止し、すべての input チャンネルに値を入力して、in-2 を再開します。

```
(toggle mixer {in-2 {:pause true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 3

(toggle mixer {in-2 {:pause false}})

;; [:a] Got 2
```

上記の例からわかるように、一時停止されたチャンネルに入力された値は破棄されません。入力チャンネルに入力された値を破棄するには、ミュートする必要があります。次に例を示します。

```
(toggle mixer {in-2 {:mute true}})
;; => true

(do
  (put! in-1 1)
  ;; out will never get this value since it's discarded
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 3

(toggle mixer {in-2 {:mute false}})
```

in-2 チャンネルに値 2 を設定しました。その時点でチャンネルがミュートされていたため、この値が破棄されて out に入ることはありません。チャンネルが Mixer の中にある 3 番目の状態 sole を見てみましょう。

前述したように、ミキサーのチャンネルを sole にすると、デフォルトで残りのチャンネルがミュートされます。

```
(toggle mixer {in-1 {:solo true}
               in-2 {:solo true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  ;; out will never get this value since it's discarded
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 2

(toggle mixer {in-1 {:solo false}
               in-2 {:solo false}})
```

ただし、sole のチャンネルがある間は、sole でないチャンネルのモードを設定できます。デフォルトの非 sole モードを、デフォルトの mute ではなく pause に設定します。

```
(solo-mode mixer :pause)
;; => true
(toggle mixer {in-1 {:solo true}
               in-2 {:solo true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 2

(toggle mixer {in-1 {:solo false}
               in-2 {:solo false}})

;; [:a] Got 3
```

明解 ClojureScript

2020 年 2 月 29 日

著　者　Andrey Antukh and Alejandro Gomez
翻　訳　Hiroki Noguchi