Shadow CLJS User's Guide

Thomas Heller · Tony Kay 著

Version 2022.5.13, Hiroki Noguchi 訳

目次

Chapter 1. はじめに

shadow-cljs は、シンプルさと使いやすさに重点を置いて、ClojureScript プロジェクトをコンパイルするために必要なすべてを提供します。提供されているビルドターゲットは、手動による設定のほとんどを抽象化し、ビルドに必要なものだけを設定することができます。各ターゲットは、それぞれの環境に最適なデフォルト値を提供し、開発時やリリースビルド時に最適化された体験をえることができます。

1.1. ハイレベルな概要

shadow-cljs は、以下の 2 つの部分から構成されています。

- shadow-cljs : 実際の仕事をすべて行う Clojure Library
- shadow-cljs : ビルド機能の大半をコマンドラインから直接実行するための便利なインターフェースを提供する npm パッケージ

必要であれば、Clojure Library である shadow-cljs を、他の Clojure/JVM ビルドツール（例えば leiningen や Clojure CLIツール）に簡単に統合することができます。

ClojureScript の開発に合わせて、より最適化された開発環境を提供するので、npm パッケージを使用することをお勧めします。

1.2. 基本的なワークフロー

shadow-cljs を使用する際には、設定ファイル shadow-cljs.edn で 1 つ以上のビルドを定義します。それぞれのビルドは、ターゲットとなる環境（例えば、ブラウザ、node.js アプリケーション、Chrome 拡張機能等）に最適化された構成プリセットを表す :target プロパティをもちます。

各ビルドは、コンパイルのトリガーとなったコマンドに応じて、開発用またはリリース用の出力を生成します。標準的なビルドコマンドは、compile 、watch 、 release です。

1.2.1. development モード

開発ビルドを一度だけコンパイルすることも、 watch プロセスを実行してソースファイルを監視して自動的に再コンパイルすることもできます（必要に応じてコードをライブロードすることもできます）。

すべての開発ビルドは、高速なフィードバックサイクルや、実行中のコードと直接やりとりできる REPL などの機能により、開発者の体験を最適化します。

開発用のビルドは、非常に大きなサイズになる可能性があり、:target に応じてコンパイルされたマシンでしか動作しない可能性があるため、決して一般に配布してはいけません。

1.2.2. release モード

release ビルドを作成すると、開発モード関連のコードがすべて取り除かれ、最後に Closure Compiler にコードを通します。これにより、JavaScript 用の最適化コンパイラが、コードの全体的なサイズを大幅に削減します。

1.3. 重要な概念

あなたが `shadow-cljs` を使用する際には、いくつかの重要な概念を理解しておく必要があります。これらの概念は、すべてがどのように組み合わされるのか、またツールがコードとどのように連動するのかを理解するために不可欠です。

1.3.1. Classpath

`shadow-cljs` は、ファイルを扱う際に Java Virtual Machine (JVM) とその classpath を使用します。

これは、多くの classpath のエントリからなる仮想ファイルシステムです。各エントリは次のいずれかです。

- 設定の `:source-paths` エントリで管理される、ローカルファイルシステムのディレクトリです。
- あるいは Clojure(Script)や JVM のライブラリを表す `.jar` ファイルです。これらは多くのファイルを含む圧縮アーカイブです(基本的には単なる `.zip` ファイルです)。これらは `:dependencies` によって追加されます。

Clojure(Script) では、すべてが名前空間化されていて、それぞれの名前がファイルに解決することが期待されています。 `(ns demo.app)` という名前空間があれば、コンパイラはクラスパス上に `demo/app.cljs` (または `.cljc`)があることを期待します。クラスパスは、それが見つかるまで順番に検索されます。以下のように設定したとします。

```
:source-paths ["src/main" "src/test"]
```

この場合、コンパイラは、以下のように探索を行います。

- `src/main/demo/app.cljs` を探す
- 次に `src/test/demo/app.cljs` を探す
- ソースパスでファイルが見つからない場合、JVM はクラスパス上の `.jar` ファイルを探す
- 上記のいずれかのライブラリのルートに `demo/app.cljs` が見つかると、そのファイルが使用される

IMPORTANT	あるファイル名がクラスパス上に複数回存在する場合、最初のものだけが使用されます。 JVM と Clojure(Script)上のすべてのものは、このような衝突を避けるために名前空間が設けられています。各パッケージが一意の名前を持たなければならない `npm` と非常によく似ています。

そのため、名前の選択には細心の注意を払い、すべてのものに適切な名前空間をつけることをお勧めします。

`(ns component.foo)` よりも `(ns your-company.components.foo)` を常に使うことは反復的に見えるかもしれませんが、後で多くの頭痛の種をなくすことができます。

これは `npm` とは異なり、パッケージ名自体がパッケージ内部で使用されることはなく、相対パスのみが使用されます。

1.3.2. server モード

`shadow-cljs` は `server` モードで起動することができます。これは `watch` のような長時間稼働するタスクに必要です。 `watch` は、サーバインスタンスがまだ起動していなければ、暗黙のうちに起動します。 `server` は、ビルドが接続する Websocket のエンドポイントを提供します。また、nREPL、Socket REPL、開発用 HTTP サーバの他のすべてのエンドポイントも同様に提供します。

`shadow-cljs` CLI を使用する場合、すべてのコマンドは新しい JVM を起動する代わりに、実行中のサーバインスタンスの JVM を再利用します。起動時間がかなり遅くなることがありますが、動作が大幅に速くなります。

しかし一度サーバが稼働すれば、`:dependencies` に変更があったときに再起動するだけで、あとはすべて REPL で行うことができます。

1.3.3. REPL

REPL は、すべての Clojure(Script) 開発の中心であり、すべての CLI コマンドは REPL から直接使用することもできます。コマンドラインの方が馴染みがあるように見えても、REPL を使いこなすことには、絶対に価値があります。

1.4. 本書について

1.4.1. 進行中

これは現在進行中の作業です。エラーを発見した場合は、それを修正するための Pull Request 、または問題の詳細を記載した issue を提出してください。

1.4.2. 貢献

この本のソースは https://github.com/shadow-cljs/shadow-cljs.github.io にあります。

1.4.3. 使用した規約

この本にはたくさんの例があります。これらの中で使われているほとんどのものは、その文脈から明らかなはずです。 しかし、誤解を防ぐためには、作者の意図を知ることが大切です。

コマンドラインの例を示すときには、BASH のコメント（ `#` ）を含めることがあります。また、コマンドとその出力の分離を示すために、標準的なユーザー UNIX プロンプトである `$` を含めます。

```
# コメントです。このコマンドは、ファイルを一覧表示します。
$ ls -l
shadow-cljs.edn
project.clj
```

多くの例はコンパイラの設定ファイルで、EDN マップが含まれています。必要なオプションについてすでに説明している場合は、わかりやすくするために省略することがあります。この場合、焦点ではない内容を示すために省略記号を入れるのが普通です。

Example 1. 依存関係の指定

```
{:dependencies [[lib "1.0"]]}
```

Example 2. ソースパスの追加

```
{...
 :source-paths ["src"]
 ...}
```

これにより、設定のネスト構造を理解するのに十分なコンテキストを含めることができます。

Example 3. ネストのオプション

```
{...
 :builds {:build-id {...
                     :output-dir "resources/public/js"}}}
```

コード例も同様に短くすることができます。

Chapter 2. インストール

2.1. npm によるスタンドアロン化

以下のことが必要となります。

- `node.js` (v6.0.0+, 最新のバージョンが望ましいです)
- `npm` または `yarn`
- 任意の Java SDK (バージョン 8 以上): OpenJDK もしくは Oracle

プロジェクトのディレクトリには `package.json` が必要であり、ない場合は `npm init -y` を実行して作成することができます。プロジェクトのディレクトリがない場合は、次のコマンドを実行して作成します。

```
$ npx create-cljs-project my-project
```

これで必要な基本ファイルがすべて作成されるので、以下のコマンドは省略できます。

すでに `package.json` があり `shadow-cljs` を追加する場合は、以下を実行します。

NPM

```
$ npm install --save-dev shadow-cljs
```

Yarn

```
$ yarn add --dev shadow-cljs
```

`npm install -g shadow-cljs` か `yarn global add shadow-cljs` とすることにより、後で `shadow-cljs` コマンドを直接実行できるようになります。あなたのプロジェクトには常に shadow-cljs のバージョンがインストールされている必要がありますが、グローバル環境へのインストールは任意です。

2.2. ライブラリ

スタンドアロン版は `npm` 経由で実行することが推奨されますが、 `shadow-cljs` を他の Clojure JVM ツール (例えば lein や boot 等) に組み込むこともできます。

artifact は以下のサイトで入手できます。

https://clojars.org/thheller/shadow-cljs

https://github.com/thheller/shadow-cljs

Chapter 3. 利用法

shadow-cljs は様々な方法で使用できますが、一般的なワークフローは同じです。

開発中には、ビルドを一度だけ compile するか、ソースファイルの変更を監視して自動的に再コンパイルする watch ワーカーを実行するかのオプションがあります。このワーカーはソースファイルの変更を監視し、自動的に再コンパイルします。enabled の場合、 watch はコードをホットリロードし、REPL を提供します。開発中は、迅速なフィードバックサイクルによる開発者の経験に重点を置いています。開発コードは決して公開してはいけません。

本格的に開発する場合には、release というビルドを作成し、本番環境に適した最適化されたビルドを作成します。これには Closure Compiler が使用され、利用可能な最も最適な出力を作成するために、コードに深刻な :advanced 最適化を適用します。ネイティブ JavaScript とのインターロップを多用している場合には、適切に動作させるためにいくつかのチューニングが必要になるかもしれません
が、ClojureScript(および Closure Library のコード)では完璧に動作します。

3.1. コマンドライン

グローバル環境にインストールされている場合は、 shadow-cljs コマンドを直接使用できます。

```
$ shadow-cljs help
```

ローカルにインストールした npm のみを使用したい場合は、 npx や yarn を介して起動できます。

```
# npm
$ npx shadow-cljs help

# yarn
$ yarn shadow-cljs help

# 手動で行う場合
$ ./node_modules/.bin/shadow-cljs help
```

例を短くするため、このガイドではグローバルインストールを前提としていますが、これは必須ではありません。

開発中によく使う shadow-cljs コマンドは以下の通りです。

```
# ビルドを一度コンパイルして終了
$ shadow-cljs compile app

# コンパイルと監視
$ shadow-cljs watch app
```

```
# ビルド用の REPL に接続（watch の実行中に利用可能）
$ shadow-cljs cljs-repl app

# スタンドアロンの Node repl への接続
$ shadow-cljs node-repl
```

次のように、本番使用に最適化されたリリースビルドを実行します。

```
$ shadow-cljs release app
```

`:advanced` のコンパイルが原因で、リリース上の問題に遭遇することがあります。これらのコマンドは、その原因を追跡するのに役立ちます。

Release デバッグコマンド

```
$ shadow-cljs check app
$ shadow-cljs release app --debug
```

3.1.1. サーバモード

`shadow-cljs` コマンドを使う際、起動にかなり時間がかかります。この点を改善するために、`shadow-cljs` はサーバモードで動作させることができます。

これにより専用のプロセスが開始され、他のすべてのコマンドがそれを利用して、新しい JVM/Clojure インスタンスを開始する必要がないため、より速く実行できます。

長時間実行される処理を行うコマンドは、暗黙のうちにサーバインスタンスを起動しますが（例えば `watch` の場合）、多くの場合は専用のサーバプロセスを起動することが望ましいでしょう。専用のサーバプロセスを実行することが望ましい場合もあります。

専用端末のフォアグラウンドでプロセスを実行することができます。サーバを終了するには `CTRL+C` を押してください。

```
$ shadow-cljs server
# または REPL にサーバプロセスを制御させたい場合
$ shadow-cljs clj-repl
```

一般的な `start|stop|restart` 関数で制御されたバックグランドでサーバを実行することもできます。

```
$ shadow-cljs start
$ shadow-cljs stop
$ shadow-cljs restart
```

いずれかサーバが起動していれば、他のコマンドはそのサーバを使用し、より速く動作します。

3.2. ビルドツールの統合

shadow-cljs は他の Clojure ツールと統合することができます。なぜなら、主要な配布物は Clojars を通じて入手できる .jar ファイルだからです。デフォルトでは :dependencies は shadow-cljs.edn で管理されますが、他のビルドツールを使って依存関係を管理することもできます。

CAUTION	スタンドアロンの shadow-cljs バージョンを使用することを強くお勧めします。このコマンドは、UX を最適化するために、他のツールでは行われない多くのことを行います (例えば、起動の高速化など)。また、依存関係の衝突やその他の関連するエラーに対処するための多くの頭痛の種を避けることができるでしょう。

3.2.1. Leiningen

Leiningen を使って依存関係を管理したい場合は、 shadow-cljs.edn の設定に :lein エントリを追加することで可能になります。この設定により、 shadow-cljs コマンドは JVM を起動する際に lein を使用し、shadow-cljs.edn 内の :source-paths や :dependencies は一切無視し、代わりに project.clj から設定された lein を使用します。

```
{:lein true
; :source-paths と :dependencies はこのファイルでは無視されます

; これらを project.clj で設定する
 :builds { ... }
```

専用の lein プロファイルの使用

```
{:lein {:profile "+cljs"}
 :builds {...}}
```

Sample project.clj

```
(defproject my-awesome-project
  ...
  :profiles
  {:cljs
   {:source-paths ["src/cljs"]
    :dependencies [[thheller/shadow-cljs "..."]
                   [reagent "0.8.1"]]}})
```

project.clj を使って :dependencies を管理する場合、 thheller/shadow-cljs アーティファクトを :dependencies に (直接またはプロファイルで) 手動で含める必要があります。

IMPORTANT	`shadow-cljs` の起動時やコンパイル・ビルド時に奇妙な Java Stackstraces に遭遇した場合、依存関係の衝突が考えられます。重要なのは、 `shadow-cljs` が、適切にマッチした `org.clojure/clojurescript` と `closure-compiler` のバージョンと一緒に使われていることです。必要なバージョンは clojars (右側)にリストアップされていますので、 `lein deps :tree` で確認できます。

Leiningen から直接タスクを実行する

また、 `shadow-cljs` コマンド自体を使いたくない場合は、 `lein` を使って `shadow-cljs` コマンドを直接実行することもできます。

IMPORTANT	コマンドを実行する際には、引き続き `shadow-cljs` コマンドを使用することをお勧めします。このコマンドは、実行中のサーバモードのインスタンスを最大限に活用します。これにより、 `lein` を直接使用して追加の JVM を起動するよりも、大幅に速くコマンドを実行することができます。

REPL やライブリロードは不要で、dev モードで一度コンパイルするだけです。

```
$ lein run -m shadow.cljs.devtools.cli compile build-id
```

リリースモードに最適化されたビルドを作成するには、次のようにします。

```
$ lein run -m shadow.cljs.devtools.cli release build-id
```

3.2.2. tools.deps / deps.edn

deps.edn を使うと、ビルトインのメソッドや `lein` を使用せずに、`:dependencies` や `:source-paths` を管理できます。 `shadow-cljs` コマンドは、代わりに新しい `clojure` ユーティリティを介して起動されます。

IMPORTANT	`tools.deps` は現在も頻繁に変更されています。必ず最新のバージョンを使用してください。

これを使うには、設定で `:deps true` プロパティを設定します。また、どの `deps.edn` のエイリアスを使用するかを設定することもできます。

`thheller/shadow-cljs` の artifact(成果物) を手動で `deps.edn` に追加する必要があります。

シンプルな *shadow-cljs.edn* の例

```
{:deps true
 :builds ...}
```

シンプルな `deps.edn` の例

```
{:paths [...]
 :deps {thheller/shadow-cljs {:mvn/version <latest>}}}
```

Example `shadow-cljs.edn` に *:cljs* のエイリアスをつけたもの

```
{:deps {:aliases [:cljs]}
 :builds ...}
```

Example deps.edn

```
{:paths [...]
 :deps {...}
 :aliases
 {:cljs
  {:extra-deps {thheller/shadow-cljs {:mvn/version <latest>}}}}}
```

clj で直接実行するには、次のように指定します。

```
{:paths [...]
 :deps {...}
 :aliases
 {:shadow-cljs
  {:extra-deps {thheller/shadow-cljs {:mvn/version <latest>}}
   :main-opts ["-m" "shadow.cljs.devtools.cli"]}}}
```

```
clj -A:shadow-cljs watch app
```

また、 `shadow-cljs -A:foo:bar ⋯` のように、コマンドラインで `-A` を使って追加のエイリアスを指定することもできます。

IMPORTANT	エイリアスは、新しいインスタンか/サーバを起動したときにのみ適用されます。shadow-cljs コマンドを使って稼働中のサーバに接続するときには適用されません。 `clj` で起動すると、常に新しい JVM を起動することになり、サーバモードをサポートしません。

3.2.3. Boot

著者は Boot の経験がほとんどないので、この章は貢献を必要としています。 Boot では関数からツールチェーンを構築できることを理解しています。 `shadow-cljs` は普通の JVM ライブラリなので、その中の関数を呼び出してタスクを起動することができます。

いくつかの Boot タスクは以下のリンクで入手できます。

https://github.com/jgdavey/boot-shadow-cljs

3.3. Clojure コードの実行

コマンドラインから特定の Clojure関数を呼び出すために、 `shadow-cljs` という CLI を使うことができます。これは、あるタスクの前後にコードを実行したいときに便利です。例えば、 `release` ビルドの出力をリモートサーバに `rsync` したいとします。

Example `src/my/build.clj` における *Clojure* の名前空間

```
(ns my.build
  (:require
    [shadow.cljs.devtools.api :as shadow]
    [clojure.java.shell :refer (sh)]))

(defn release []
  (shadow/release :my-build)
  (sh "rsync" "-arzt" "path/to/output-dir" "my@server.com:some/path"))
```

release 関数の実行

```
$ shadow-cljs clj-run my.build/release
# または
$ shadow-cljs run my.build/release
```

呼び出された関数には、コマンドラインから引数を渡すことができます。

通常の *Clojure fn* の *args* を使った引数の使用

```
...
(defn release [server]
  (shadow/release :my-build)
  (sh "rsync" "-arzt" "path/to/output-dir" server))
```

コマンドラインからのサーバの受け渡し

```
$ shadow-cljs clj-run my.build/release my@server.com:some/path
```

TIP tools.cli のように引数を解析したい場合は、通常の `(defn release [& args])` の構造でも動作します。

ここでは、Clojure　のフルパワーにアクセスできます。必要に応じて、この上にツール全体を構築することができます。おまけに、この方法で書いたものはすべて、Clojure REPL で直接利用できます。

IMPORTANT server　が実行されている場合、名前空間は自動的にはリロードされず、一度だけロードされます。　REPL　を使って開発を行い、通常通りファイルをリロードすることをお勧めします（例：`(require 'my.build :reload)`）。 `shadow-cljs clj-eval "(require 'my.build :reload)"` を実行して、コマンドラインから手動でリロードすることもできます。

3.3.1. clj-run による watch の呼び出し

デフォルトでは、 `clj-run` から呼び出された関数は、 `compile`, `release` やその他の Clojure 機能を実行するのに十分な、最小限の `shadow-cljs` ランタイムにしかアクセスできません。関数が完了すると、JVM は終了します。

あるビルドに対して `watch` を開始したい場合は、呼び出している関数が完全なサーバを必要とすることを宣言する必要があります。これにより、あなたが明示的に `(shadow.cljs.devtools.server/stop!)` を呼び出すか、 `CTRL+C` でプロセスを停止させるまで、そのプロセスは生き続けます。

```
(ns demo.run
  (:require [shadow.cljs.devtools.api :as shadow]))

;; これは完全なサーバインスタンスがないために失敗します
(defn foo
  [& args]
  (shadow/watch :my-build))

;; このメタデータは、watch が動作するようにサーバを起動することを保証します
(defn foo
  {:shadow/requires-server true}
  [& args]
  (shadow/watch :my-build))
```

Chapter 4. REPL

REPL は、Clojure(Script)のコードを扱う際に、非常に強力なツールです。 `shadow-cljs` は、すぐに始められるいくつかのビルトインバージョンと、標準的なビルドに統合されるバージョンを提供します。

いくつかのコードをすぐにテストしたい場合は、内蔵の REPL で十分です。自分で何かをするような、より複雑なセットアップが必要な場合は、実際のビルドを使うのが一番です。

4.1. ClojureScript REPL

デフォルトでは、 `node-repl` と `browser-repl` のどちらかを選択することができます。どちらも似た動作をしますが、違いは、一方が `node.js` が管理するプロセスで動作するのに対し、他方は実際のコードを評価するために使用されるブラウザウィンドウを開くことです。

4.1.1. Node REPL

```
$ shadow-cljs node-repl
```

これは、すでに接続されている `node` プロセスで、空の CLJS REPL を開始します。

> **IMPORTANT** Node REPL を終了すると、 `node` プロセスも kill されます。

`node-repl` により、追加設定をする必要がなく、すぐに使い始めることができます。

`node-repl` は、 `(require '[your.core :as x])` という通常の方法で、あなたのすべてのコードにアクセスします。ビルドに接続されていないので、ファイルが変更されてもコードの自動再構築は行われず、ホットリロードも提供されません。

4.1.2. Browser REPL

```
$ shadow-cljs browser-repl
```

これは、空の CLJS REPL を起動し、コードが実行される関連するブラウザのウィンドウを開きます。ブラウザ上で実行されるだけでなく、上記の `node-repl` と同じ機能を持っています。

> **IMPORTANT** ブラウザウィンドウを閉じると REPL は動作しません。

4.1.3. ビルドに特化した REPL

`node-repl` と `browser-repl` は、特定のビルド構成なしに動作します。つまり、あなたが指示したことだけを実行し、自分では何もしないということです。

特定のものをビルドしたい場合は、提供されているビルドターゲットの 1 つを使ってビルドを設定する必要があります。それらのほとんどは、ClojureScript REPL に必要なコードを自動的に注入します。それは追加の設定を必要としないはずです。ビルド CLJS REPL が動作するためには、次の 2 つが必要です。

ビルドのための実行中の `watch` です。 ターゲットの JS ランタイムを接続します。つまり、`:browser` ターゲットを使用している場合、生成された JS がロードされているブラウザを開く必要があります。node.js のビルドの場合は、 `node` プロセスの実行を意味します。

両方を入手したら、コマンドラインまたは Clojure REPL から CLJS REPL に接続できます。

CLI

```
$ shadow-cljs watch build-id
...

# 異なるターミナル
$ shadow-cljs cljs-repl build-id
shadow-cljs - connected to server
[3:1]~cljs.user=>
```

REPL

```
$ shadow-cljs clj-repl
...
[2:0]~shadow.user=> (shadow/watch :browser)
[:browser] Configuring build.
[:browser] Compiling ...
[:browser] Build completed. (341 files, 1 compiled, 0 warnings, 3,19s)
:watching
[2:0]~shadow.user=> (shadow/repl :browser)
[2:1]~cljs.user=>
```

TIP REPL を終了するには、`:repl/quit` と入力してください。これは REPL を終了するだけで、`watch` は引き続き実行されます。

TIP 複数の `watch` ワーカーを並行して実行し、任意の時間にそれらの REPL に接続/切断することができます。

No connected runtime error の発生

```
[3:1]~cljs.user=> (js/alert "foo")
There is no connected JS runtime.
```

これが表示された場合、ブラウザでアプリを開くか、 `node` プロセスを開始する必要があります。

4.2. Clojure REPL

提供されている ClojureScript REPL に加えて、Clojure REPL も提供されています。これは、`shadow-cljs` プロセスを制御し、他のすべてのビルドコマンドを実行するために使用できます。Clojure REPL から始めて、いつでも CLJS REPL にアップグレードすることができます(そして元に戻すこともできます)。

CLI からの実行

```
$ shadow-cljs clj-repl
...
shadow-cljs - REPL - see (help), :repl/quit to exit
[1:0]~shadow.user=>
```

名前空間 `shadow.cljs.devtools.api` には、CLI に対応する関数とほぼ 1 対1に対応する関数があります。デフォルトでは `shadow` という名前でエイリアスされています。

```
;; shadow-cljs watch foo
(shadow.cljs.devtools.api/watch :foo)

;; ns のエイリアスが用意されているため同等
(shadow/watch :foo)

;; shadow-cljs watch foo --verbose
(shadow/watch :foo {:verbose true})

;; shadow-cljs compile foo
(shadow/compile :foo)

;; shadow-cljs release foo
(shadow/release :foo)

;; shadow-cljs browser-repl
(shadow/browser-repl)

;; shadow-cljs node-repl
(shadow/node-repl)

;; shadow-cljs cljs-repl foo
(shadow/repl :foo)

;; CLJS REPL に入れば :repl/quit や cljs/quit を使って CLJ に戻ることができる
```

4.2.1. 組み込み

また、他の CLJ プロセスの中から完全に `shadow-cljs` を使用することも可能です。クラスパスに `thheller/shadow-cljs` がロードされていれば、問題ありません。

lein repl を使った例

```
$ lein repl
nREPL server started on port 57098 on host 127.0.0.1 - nrepl://127.0.0.1:57098
REPL-y 0.4.3, nREPL 0.6.0
Clojure 1.10.0
...

user=> (require '[shadow.cljs.devtools.server :as server])
nil
user=> (server/start!)
...
:shadow.cljs.devtools.server/started
user=> (require '[shadow.cljs.devtools.api :as shadow])
nil
user=> (shadow/compile :foo)
...
```

`(shadow.cljs.devtools.server/stop!)` を実行することで、組み込みサーバを停止することができます。これにより、実行中のすべてのビルドプロセスも停止します。

IMPORTANT	CLJS REPL に切り替えたい場合は、サーバの起動に使用したツールで追加の設定が必要になる場合があります。 `lein` はデフォルトで nREPL を使用するので、追加の nREPL `:middleware` を設定する必要があります。 `clj` を使用する場合は、nREPL を使用しないので、問題ありません。

Chapter 5. 設定

`shadow-cljs` はプロジェクトのルートディレクトリにある `shadow-cljs.edn` ファイルによって設定され、`shadow-cljs init` を実行して初期用のファイルを作成できます。このファイルには、グローバルな設定を行うためのマップと、すべてのビルドのための `:builds` エントリが含まれています。

```
{:source-paths [...]
 :dependencies [...]
 :builds {...}}
```

設定例は以下のようになります。

```
{:dependencies
 [[reagent "0.8.0-alpha2"]]

 :source-paths
 ["src"]

 :builds
 {:app {:target :browser
        :output-dir "public/js"
        :asset-path "/js"
        :modules {:main {:entries [my.app]}}}}}
```

この例のファイル構造は次のようになります。

```
.
├── package.json
├── shadow-cljs.edn
└── src
    └── my
        └── app.cljs
```

5.1. ソースのパス

`:source-paths` は、JVM のクラスパスを設定します。コンパイラはこの設定を使って、Clojure(Script)のソースファイル(例： `.cljs`)を探します。

すべてを 1 つのソースパスにまとめても問題ありませんが、ソースファイルを特定の方法でグループ化したい場合は、複数のソースパスを使用することができます。例えば、テストを別々にしたい場合などに便利です。

複数のソースパスの使用

```
{:source-paths ["src/main" "src/test"]
 ...}
```

```
.
├── package.json
├── shadow-cljs.edn
└── src
    └── main
        └── my
            └── app.cljs
    └── test
        └── my
            └── app_test.cljs
```

ソースファイルを拡張子で分けることはお勧めしません（例：`src/clj`, `src/cljs`, `src/cljc`）。なぜかCLJS のプロジェクトテンプレートではこの方法が広く使われていますが、使いにくくなるだけです。

5.2. 依存関係

5.2.1. Clojure(Script)

依存関係は、 `shadow-cljs.edn` 設定ファイルのルートにある `:dependencies` キーによって管理されます。これらは `lein` や `boot` のような他の Clojure ツールが使用するのと同じ記法で宣言されます。

各依存関係は `[ライブラリ名 "バージョン文字列"]` を使ったベクターとして書かれ、 1 つの外側のベクターに入れ子になっています。

Example :dependencies

```
{:source-paths ["src"]
 :dependencies [[reagent "0.9.1"]]
 :builds ...}
```

ソースパスは設定全体で一度しか指定されていないことに注意してください。システムは名前空間の依存関係グラフを使って、どのようなコードがビルドの最終出力に必要かを判断します。

5.2.2. JavaScript

`shadow-cljs` は、 `npm` エコシステムと完全に統合して、JavaScript の依存関係を管理します。

依存関係の管理には、 `npm` や `yarn` を使うことができますが、それぞれのドキュメントを参照してください。

どちらも、プロジェクトディレクトリ内の `package.json` ファイルで依存関係を管理します。 `npm` で入手できるほとんどすべてのパッケージには、そのインストール方法が説明されています。これらの説明は、現在では `shadow-cljs` にも適用されています。

JavaScript パッケージのインストール

```
# npm
$ npm install the-thing

# yarn
$ yarn add the-thing
```

それ以上は何も必要ありません。依存関係は `package.json` ファイルに追加され、これを使って管理されます。

> **TIP** `package.json` がまだない場合は、コマンドラインから `npm init` を実行してください。

JS 依存関係の欠如について

JavaScript の依存関係の欠如に関連するエラーに遭遇するかもしれません。ほとんどの ClojureScript ライブラリは、使用する `npm` パッケージをまだ宣言していません。なぜなら、それらは CLJSJS を使用することを期待しているからです。私たちは `npm` を直接使用したいと考えています。つまり、ライブラリが適切に `:npm-deps` を宣言するまでは、 `npm` パッケージを手動でインストールする必要があります。

```
The required JS dependency "react" is not available, it was required by ... (必要な JS の依存関係である "react "が利用できません。これは、 ...
によって必要とされていました。)
```

このメッセージは、 `npm install react` を行う必要があることを示しています。

> **TIP** `react` を利用する場合は、おそらく以下の3つのパッケージが必要です。`npm install react react-dom create-react-class`.

5.3. ユーザの設定

ほとんどの設定はプロジェクト自身で `shadow-cljs.edn` を通して行われますが、いくつかの設定はユーザーに依存しているかもしれません。 CIDER のようなツールは、追加の `cider-nrepl` 依存関係を必要とするかもしれませんが、 `shadow-cljs.edn` 経由でその依存関係を追加しても、Cursive を使用している別のチームメンバーには意味がありません。

制限された設定オプションのセットを `~/.shadow-cljs/config.edn` に追加することで、このユーザーのマシン上でビルドされたすべてのプロジェクトに適用されます。

依存関係を追加するには、通常の `:dependencies` キーを使用します。ここで追加された依存関係は、すべてのプロジェクトに適用されることに注意してください。依存関係は最小限にして、ツール関連の依存関係だけをここに置くようにしてください。ビルドに関連するものはすべて `shadow-cljs.edn` に置いておくべきで、そうしないと他のユーザーがコンパイルできない可能性があります。これらの依存関係は、`deps.edn` や `lein` を使用する際にも自動的に追加されます。

```
{:dependencies
 [[cider/cider-nrepl "0.21.1"]]}
;; このバージョンは古くなっている可能性があります。
```

`deps.edn` を使って依存関係を解決する際に、追加のエイリアスを有効にしたい場合があります。これは `:deps-aliases` で行うことができます。

```
;; プロジェクト内の shadow-cljs.edn
{:deps {:aliases [:cljs]}}

;; ~/.shadow-cljs/config.edn
{:deps-aliases [:cider]}
```

これにより、 `deps.edn` を使用しているプロジェクトでは、 `shadow-cljs` コマンドが `[:cider :cljs]` のエイリアスを使用するようになります。

これは、あなたの `~/.clojure/deps.edn` に追加の `:cider` エイリアスがある場合に便利かもしれません。

デフォルトでは、 `shadow-cljs` サーバモードでは、組み込まれた nREPL サーバが起動しますが、これは必要ないかもしれません。これを無効にするには、ユーザー設定で `:nrepl false` を設定します。

ユーザー設定で現在受け入れられている値は、:open-file-command のみです。他のオプションは現在のところ何の効果もありません。

5.4. サーバのオプション

このセクションでは、 shadow-cljs サーバインスタンスを構成するその他のオプションについて説明します。これらはオプションです。

5.4.1. nREPL

shadow-cljs server は nREPL サーバを TCP 経由で提供しています。起動メッセージから nREPL のポートを確認できます。nREPL のポート番号は target/shadow-cljs/nrepl.port に保存されます。

```
$ shadow-cljs watch app
shadow-cljs - HTTP server available at http://localhost:8600
shadow-cljs - server version: <version> running at http://localhost:9630
shadow-cljs - nREPL server started on port 64967
shadow-cljs - watching build :app
[:app] Configuring build.
[:app] Compiling ...
```

shadow-cljs.edn でポートや追加のミドルウェアを設定することができます。

```
{...
 :nrepl {:port 9000
         :middleware []} ; 名前空間修飾されたシンボルのオプションリスト
 ...}
```

~/.nrepl/nrepl.edn にあるデフォルトのグローバルコンフィグファイルや、ローカルの .nrepl.edn も起動時に読み込まれ、:middleware の設定に使用できます。

人気のミドルウェア cider-nrepl がクラスパス上にあれば（例：:dependencies に含まれている）、自動的に使用されます。追加の設定は必要ありません。これを無効にするには、:nrepl {:cider false} を設定します。

nrepl オプションで :init-ns を設定することで、接続時に起動する名前空間を設定することができます。デフォルトでは shadow.user となります。

```
{...
 :nrepl {:init-ns my.repl}
 ...}
```

nREPL サーバは、:nrepl false を設定することで無効にすることができます。

nREPL Usage

nREPL サーバに接続すると、接続は常に Clojure REPL として開始されます。CLJS REPL への切り替えは、 cljs-repl, non-nREPL version と同様に動作します。まず、与えられたビルドの watch を開始する必要があり、次に現在の nREPL セッションをそのビルドに切り替えるために、このビルドを選択する必

要があります。ビルドを選択すると、すべての評価は Clojure ではなく ClojureScript で行われます。

```
(shadow/watch :the-build)
(shadow/repl :the-build)
```

TIP Clojure に戻るには `:cljs/quit` を使います。

組み込み型 **nREPL** サーバ

独自の nREPL サーバを提供する他のツール（例：`lein`）に `shadow-cljs` を組み込んで使用する場合は、`shadow-cljs` ミドルウェアを設定する必要があります。そうしないと、CLJ と CLJS の REPL の間で切り替えることができません。

```
(defproject my-amazing-project "1.0.0"
  ...
  :repl-options
  {:init-ns shadow.user ;; または、あなたが選んだもの
   :nrepl-middleware
   [shadow.cljs.devtools.server.nrepl/middleware]}
  ...)
```

TIP CLJS REPL を使用する前に、embedded serverを手動で起動する必要があります。

5.4.2. Socket REPL

Clojure Socket REPL は、サーバモードで自動的に起動され、デフォルトでランダムなポートを使用します。

ツールはポート番号を含む `.shadow-cljs/socket-repl.port` を確認することで、起動されたポートを見つけることができます。

また、 `shadow-cljs.edn` で固定のポートを設定することもできます。

```
{...
 :socket-repl
 {:port 9000}
 ...}
```

Socket REPL は、`:socket-repl false` を設定することで無効にすることができます。

5.4.3. SSL

`shadow-cljs` の HTTP サーバは SSL をサポートしています。そのためには、一致する秘密鍵と証明書を提供する Java Keystore が必要です。

shadow-cljs.edn に *SSL* が設定されています。

```
{...
 :ssl {:keystore "ssl/keystore.jks"
       :password "shadow-cljs"}
 ...}
```

上記はデフォルトなので、これらを使用したい場合は、:ssl {} を設定するだけで問題ありません。

java keytool コマンドを使ってキーストアを作成することができます。信頼できる自己署名証明書を作成することも可能ですが、やや複雑です。

作成された Certificates.p12 (macOS) または localhost.pfx (Linux, Windows) ファイルは、 keytool ユーティリティーを使って、必要な keystore.jks にすることができます。

```
$ keytool -importkeystore -destkeystore keystore.jks -srcstoretype PKCS12 -srckeystore
localhost.pfx
```

IMPORTANT	localhost（または使用する任意のホスト）の SAN（Subject Alternative Name）を含む証明書を作成する必要があります。SAN は、Chrome が証明書を信頼して警告を表示しないようにするために必要です。エクスポート時に使用するパスワードは、キーストアに割り当てられたパスワードと一致する必要があります。

5.4.4. Primary HTTP(S)

shadow-cljs サーバは、 1 つのプライマリ HTTP サーバを起動します。このサーバは、ホットリロードや REPL クライアントで使用される UI やウェブソケットを提供するために使用されます。デフォルトでは、9630番ポートで待ち受けます。もしそのポートが使用中であれば、 1 つ増やして、開いているポートが見つかるまで再試行します。

使用するポートを示すスタートアップメッセージ

```
shadow-cljs - server running at http://0.0.0.0:9630
```

ssl が設定されている場合、サーバは代わりに https:// を介して利用できます。

TIP	:ssl を使用すると、サーバは自動的に HTTP/2 をサポートします。

代わりに独自のポートを設定したい場合は、:http の設定で行うことができます。

shadow-cljs.edn 内の :http の設定

```
{...
 :http {:port 12345
        :host "my.machine.local"}
 ...}
```

:ssl は、サーバを https:// のみに切り替えます。もし、 http:// のバージョンを維持したい場合は、別の :ssl-port を設定することができます。

```
{...
 :http {:port 12345
        :ssl-port 23456
        :host "localhost"}
 ...}
```

5.4.5. 開発の HTTP(S)

shadow-cljs では、:dev-http という設定項目で、追加の基本的な HTTP サーバを提供することができます。デフォルトでは、これらは設定されたパスからすべての静的ファイルを提供し、リソースが見つからない場合は index.html にフォールバックします (これは、ブラウザのプッシュステートを使用するアプリケーションを開発する際に、一般的に求められるものです)。

これらのサーバは shadow-cljs がサーバモードで動作しているときに自動的に開始されます。これらのサーバはどのビルドにも特定されず、それぞれにユニークな :output-dir が使用されている限り、複数のビルドのファイルを提供するために使用することができます。

重要なことです。これらは、静的ファイルをサーバする一般的なウェブサーバです。これらはライブリロードや REPL のロジックには必要ありません。どのようなウェブサーバでも使用できますが、これらは単に利便性のために提供されています。

基本的な例では、 http://localhost:8000 を介して public ディレクトリを serve します。

```
{...
 :dev-http {8000 "public"}
 :builds {...}}
```

:dev-http は port-number から config へのマップを期待しています。この config は、最も一般的なシナリオに対応したいくつかのショートカットをサポートしています。

ファイルシステムのルートからディレクトリを *serve* する

```
:dev-http {8000 "public"}
```

クラスルートから *serve* する

```
:dev-http {8000 "classpath:public"}
```

これは、クラスパス上の public/index.html から /index.html へのリクエストを見つけようとするものであり、.jar ファイルの中にファイルを含む場合があります。

複数のルートから *serve* する

```
:dev-http {8000 ["a" "b" "classpath:c"]}
```

これはまず、以下の順にクラスパス上で検索を試みます。

- `<project-root>/a/index.html`
- `<project-root>/b/index.html`
- `c/index.html`

もし何も見つからなければ、デフォルトのハンドラが呼び出されます。

長いバージョンのコンフィグでは、マップが要求され、サポートされているオプションは次のとおりです。

`:root`

(String) リクエストを処理するためのパスです。 `classpath:` で始まるパスは、ファイルシステムではなく、クラスパスからリクエストを処理します。すべてのファイルシステムのパスは、プロジェクトのルートからの相対パスです。

`:roots`

複数のルートパスが必要な場合は、`:root` の代わりに使用します。

`:ssl-port`

(文字列のベクトル) `ssl` が設定されている場合、ssl 接続にはこのポートを使用し、通常の HTTP サーバは通常のポートを使用します。 `ssl-port` が設定されておらず、`:ssl` が設定されている場合、デフォルトのポートは SSL リクエストのみをサーバします。

`:host`

オプションです。listen するホストを指定します。デフォルトは localhost です。

`:handler`

オプションです。完全に修飾されたシンボルです。与えられたリクエストに対してリソースが見つからない場合に使用される `(defn handler [req] resp)` です。 `(defn handler [req] resp)` は、与えられたリクエストに対してリソースが見つからない場合に使用されます。デフォルトでは `shadow.http.push-state/handle` となります。

以下の 2 つのオプションは、デフォルトの組み込みハンドラを使用する場合にのみ適用され、通常は変更する必要はありません。

`:push-state/headers`

(オプション) 応答する HTTP ヘッダーのマップです。デフォルトでは、 `text/html` という標準的なヘッダーを使用します。

`:push-state/index`

(オプション) サービスを提供するファイルです。デフォルトでは `index.html` が使用されます。

```
{...
 :dev-http
 {8080 {:root "public"
        :handler my.app/handler}}}
```

リバースプロキシのサポート

デフォルトでは、開発サーバはローカルでリクエストを処理しようとしますが、外部の Web サーバを使ってリクエストを処理したい場合もあります（例：API リクエスト）。これは `:proxy-url` で設定できます。

```
{...
 :dev-http
 {8000
  {:root "public"
   :proxy-url "https://some.host"}}}
```

`http://localhost:8000/api/foo` へのリクエストは、代わりに `https://some.host/api/foo` が返すコンテンツを提供します。ローカルファイルを持たないすべてのリクエストは、プロキシされたサーバによって提供されます。

接続処理を設定するオプションは以下の通りです。

`:proxy-rewrite-host-header`

: boolean で、デフォルトは true です。オリジナルの Host ヘッダーを使用するか、`:proxy-url` からのヘッダーを使うかを決めます。上記の例では `localhost` と `some.host` のどちらを使用するかを決定します。

`:proxy-reuse-x-forwarded`

: boolean で、デフォルトは false です。プロキシが自分自身を `X-Forwarded-For` リストに追加するか、新しいリストを開始するかを設定します。

`:proxy-max-connection-retries`

: int で、デフォルトは 1 です。

`:proxy-max-request-time`

: int でミリ秒を表し、デフォルトは 30000(30秒のリクエストタイムアウト)です。

5.5. JVM の設定

JVM の起動に `shadow-cljs.edn` を使用する場合、JVM に直接渡される追加のコマンドライン引数を設定することができます。

avsxza nm例えば、shadow-cljs.edn が使用する RAM の量を減らしたり増やしたりしたい場合があります。

これは、 `shadow-cljs.edn` のルートに `:jvm-opts` を設定して、文字列のベクトルを期待することで行われます。

Example：*RAM* 使用量を *1GB* に制限

```
{:source-paths [...]
 :dependencies [...]
 :jvm-opts ["-Xmx1G"]
 :builds ...}
```

JVM に渡すことができる引数はバージョンによって異なります。RAM の割り当てが少なすぎたり多すぎたりすると、パフォーマンスが低下することがあるので注意してください。通常はデフォルトの設定で十分です。

IMPORTANT	`deps.edn` や `project.clj` を使用する場合は、`:jvm-opts` を設定する必要があります。

Chapter 6. ビルドの設定

shadow-cljs.edn には、:builds セクションも必要です。ビルドは、ビルド ID をキーにしたビルドのマップでなければなりません。

ビルドマップの設定ファイル

```
{:dependencies [[some-library "1.2.1"] ...]
 :source-paths ["src"]
 :builds
 {:app   {:target     :browser
          ... browser-specific options ...}
  :tests {:target :karma
          ... karma-specific options ...}}}
```

各ビルドには、コンパイラがビルドする成果物が記述されています。ビルドターゲットは shadow-cljs の拡張可能な機能であり、コンパイラにはすでにかなりの数が付属しています。

6.1. ビルドターゲット

shadow-cljs の各ビルドでは、コードをどこで実行するかを定義する :target を定義する必要があります。browser と node.js のデフォルトのビルドインがあります。これらはすべて、:dev モードと :release モードを持つという基本的なコンセプトを共有しています。 dev モードでは、高速なコンパイル、ライブコードリロード、REPL など、通常の開発に必要な機能をすべて提供します。 release モードでは、プロダクション向けに最適化された出力が得られます。

ターゲットについては別章で説明しますが、一部をご紹介します。

:browser

Web ブラウザでの実行に適したコードを出力します。 :bootstrap::ブートストラップされた cljs 環境で実行するのに適したコードを出力します。

:browser-test

テストをスキャンして必要なファイルを決定し、ブラウザで実行するのに適したテストを出力します。

:karma

テストをスキャンして必要なファイルを決定し、karma-runner互換のテストを出力します。

:node-library

ノードライブラリとして使用するのに適したコードを出力します。

:node-script

ノードスクリプトとして使用するのに適したコードを出力します。

:npm-module

npm モジュールとして使用するのに適したコードを出力します。

ターゲットを選択すると残りのビルドオプションが変わるため、各々の章で詳しく説明します。

6.2. 開発オプション

通常、各ビルド `:target` は複数の開発サポートを提供します。それらは各 `:build` の `:devtools` キーの下にまとめられています。

6.2.1. REPL

`watch` を実行すると、REPL用のコードが自動的に注入され、通常は追加の設定は必要ありません。 REPLの動作を制御するための追加オプションも用意されています。

- `:repl-init-ns` では、REPL をどの名前空間で起動するかを設定することができます。デフォルトでは `cljs.user` となります。
- `repl-pprint` では、REPL が eval の結果を print するときに、通常の `pr-str` の代わりに `cljs.pprint` を使用します。デフォルトは false です。

```
{...
 :builds
 {:app {...
        :devtools {:repl-init-ns my.app
                   :repl-pprint true
                   ...}}}}
```

6.2.2. 事前ロード

開発者は開発モードで大半の時間を費やします。`figwheel`、`boot-reload`、`devtools` などのツールに精通し、`boot-reload` や `devtools` などのツールをご存知でしょう。これらのツールの 1 つ以上を自分のビルドに使用するはずです。

事前ロードは、生成された Javascript の先頭に特定の名前空間を強制的に導入するために使用され、一般的にアプリケーションが実際にロードされて実行される前にツール等を注入するために使用されます。事前ロードのオプションは `shadow-cljs.edn` の `:devtools` / `:preloads` セクション内、または特定のモジュールの `:preloads` キー内にある名前空間の単純なリストです。

```
{...
 :builds
 {:app {...
        :devtools {:preloads [fulcro.inspect.preload]
                   ...}}}}
```

次の設定では、事前ロードを開発中のメインモジュール内でのみ行い、ウェブワーカーを停止します。

```
{...
 :builds
 {:app {...
        :modules {:main {...
                         :preloads
                         [com.fulcrologic.fulcro.inspect.preload
                          com.fulcrologic.fulcro.inspect.dom-picker-preload]
                         :depends-on #{:shared}}
                  :shared {:entries []}
                  :web-worker {...
                    :depends-on #{:shared}
                    :web-worker true}}}}}
```

`:preloads` は開発用ビルドにのみ適用され、リリース用ビルドには適用されません。

NOTE

バージョン 2.0.130 以降の shadow-cljs は、 `watch` と `compile` のクラスパス上に `cljs-devtools` がある場合、自動的にその事前ロードに `cljs-devtools` を追加します。必要なことは、`dependencies` リストに `binaryage/cljs-devtools` ではなく `binaryage/devtools` があることを確認するだけです。もし特定のターゲットに `cljs-devtools` を入れない場合は、ターゲットの `:devtools` セクションに `:console-support false` を追加することで、これを抑制することができます。

6.2.3. ホットコードリロード

React と ClojureScript のエコシステムを組み合わせることで、このようなことが大変便利になります。`shadow-cljs` のシステムには、外部ツールに頼ることなく、ホットコードリロードを行うために必要なものがすべて含まれています。

ホットコードリロードを行うには、以下のようにするだけです。

```
shadow-cljs watch build-id
```

6.2.4. ライフサイクルフック

ホットコードリロードで更新されたコードが入る直前と直後に関数を実行するようにコンパイラを設定することができます。これは古いコードの上で閉じてしまうような処理を停止/開始するのに便利です。

これらの設定は、ビルド設定の `:devtools` セクションを介して、またはメタデータタグを介してコード内で直接行うことができます。

メタデータ

通常の CLJS の `defn` 変数に特定のメタデータを設定することで、ライブリロード時にこれらの関数が特定のタイミングで呼び出されるべきであることをコンパイラに知らせることができます。

メタデータを利用した *hook* の設定

```
(ns my.app)

(defn ^:dev/before-load stop []
  (js/console.log "stop"))

(defn ^:dev/after-load start []
  (js/console.log "start"))
```

これは、新しいコードを読み込む前に `my.app/stop` を呼び出し、新しいコードがすべて読み込まれたときに `my.app/start` を呼び出します。このように複数の関数をタグ付けすることができ、それらは名前空間の依存関係の順に呼び出されます。

リロード処理の前に完了すべき非同期処理を行う場合、これらの非同期版もあります。

async フックの例

```
(ns my.app)

(defn ^:dev/before-load-async stop [done]
  (js/console.log "stop")
  (js/setTimeout
    (fn []
      (js/console.log "stop complete")
      (done)))

(defn ^:dev/after-load-async start [done]
  (js/console.log "start")
  (js/setTimeout
    (fn []
      (js/console.log "start complete")
      (done)))
```

IMPORTANT	各関数には各々の作業が完了したときに呼び出されるべきコールバック関数が 1 つあり、このコールバック関数が呼び出されないとリロード処理は進みません。

名前空間にメタデータをタグ付けすれば、再コンパイルされてもリロードされないようにできます。

```
(ns ^:dev/once my.thing)
(js/console.warn "will only execute once")
```

名前空間は常にリロードするようにタグ付けすることもできます。

```
(ns ^:dev/always my.thing)
(js/console.warn "will execute on every code change")
```

設定

メタデータに加えて、ライフサイクルフックを `shadow-cljs.edn` で設定することができます。

`:before-load`

再コンパイルされたファイルを更新する直前に実行される関数のシンボル（名前空間付き）です。　関数のシンボル（名前空間付き）です。 この関数は本質的に同期的でなければなりません。

`:before-load-async`

リフレッシュする直前に実行する関数の(名前空間がついた)シンボルです。この関数は非同期処理を行うことができますが、処理が完了したことを示すために`(done)` を必ず呼び出します。

`:after-load`

ホットコードのリロードが完了した後に実行する関数のシンボル(名前空間付き)です。

`:after-load-async`

ホットコードのリロードが完了した後に実行される関数 `(fn [done])` のシンボル(名前空間を含む)です。この関数は非同期処理を行うことができますが、完了したことを示すために、 `(done)` を必ず呼び出さなければなりません。

`:autoload`

コードをホットロードするかどうかを制御する真偽値です。コールバックが設定されると暗黙的に `true` に設定されます。デフォルトは `:browser` ターゲットに対して常に有効で、無効にするには `false` を設定します。

`:deignore-warnings`

警告を含むコードをリロードするかどうかを制御する真偽値です。デフォルトは `false` です。

ライフサイクルフックの例

```
{...
 :builds
 {:app {...
        :devtools {:before-load  my.app/stop
                   :after-load   my.app/start
                   ...}}}}
```

IMPORTANT	フックは `cljs.user` 名前空間では宣言できません。フックは、それを含む名前空間が実際にビルドに含まれている場合にのみ使用されます。追加の名前空間を使用する場合は、必ず `:preloads` でインクルードしてください。

TIP	`:after-load` や `:before-load` が設定されていない場合、コンパイラは `:browser` ターゲットのコード内においてホットリロードのみを試みます。もしホットリロードを行いたいが、コールバックが不要な場合は、代わりに `:autoload true` を設定してください。

6.3. ビルドフック

カスタムコードをコンパイルパイプラインの特定の段階で実行したい場合があります。`:build-hooks` では、どの関数を呼び出すかを宣言することができ、その関数はその時点でのビルド状態に完全にアクセスすることができます。これは非常に強力で、様々なツールのオプションが可能になります。

ビルドごとに `:build-hooks` キーで設定されます。

Example :build-hooks

```
{...
 :builds
 {:app {:target ...
        :build-hooks
        [(my.util/hook 1 2 3)]
        ...}}}}
```

Example hook code

```
(ns my.util)

(defn hook
  {:shadow.build/stage :flush}
  [build-state & args]
  (prn [:hello-world args])
  build-state)
```

この例では、ビルドが `:flush` stage を完了した(つまりディスクに書き込まれた)後に `(my.util/hook build-state 1 2 3)` を呼び出します。この例では `[:hello-world (1 2 3)]` と表示されますが、実際のフックではもっと便利なことをしてください。

フックは通常の Clojure 関数にいくつかのメタデータを追加したものです。

`shadow.build/stage :flush}` メタデータは、このフックを `:flush` でのみ呼び出すようにコンパイラに通知します。

フックが複数のステージの後に呼び出されるべきであれば、代わりに `{:shadow.build/stages #{:configure :flush}}` を設定することができます。そうしないとフックは何もしないので、少なくともひとつの設定済みステージが必要です。

全てのビルドフックは、`:target` の作業が終わった後に呼び出されます。これらのフックは、最初の引数として、現在の全てのビルドデータを含む clojure マップの `build-state` を受け取り、 必ず この `build-state` を修正して、または修正しないで返します。複数のステージを使用する場合、後のステージが見ることができる追加データを `build-state` に追加することができます。誤ってビルド全体を壊してしまわないように、名前付きのキーのみを使用することを強くお勧めします。

build-state には、フックに有用ないくつかの重要なエントリがあります。

- :shadow.build/build-id - 現在のビルドの ID です。(例： :app)
- :shadow.build/mode - :dev または :release です。
- :shadow.build/stage - 現在のステージです。
- :shadow.build/config - ビルドの設定。フック用の設定データは、ビルドコンフィグに直接格納するか、フック自体の引数として渡すことができます。

IMPORTANT	watch を実行していると、すべてのフックがビルドのたびに繰り返し呼び出されます。ビルドのパフォーマンスに大きな影響を与える可能性がありますので、あまり多くの作業を行わないようにしてください。

6.3.1. コンパイル・ステージ

:build-hooks が使用できるステージは以下の通りです。

- :configure - 初期の :target 特定の設定です
- :compile-prepare - コンパイルが行われる前に呼び出されます
- :compile-finish - すべてのコンパイルが終了した後に呼び出されます
- optimize-prepare - Closure コンパイラの最適化フェーズを実行する前に呼び出されます (:release のみ)
- :optime-finish - Closure が終了した後に呼び出されます (:release のみ)
- :flush - すべてがディスクにフラッシュされた後に呼び出されます

watch を実行していると、 :configure は一度しか呼ばれません。再コンパイルのたびに、他の項目が再コンパイルのたびに（順に）呼び出されます。 build-state はビルドコンフィグが変更されるまで再利用され、その時点で破棄されて新しいものが作成されることになる。

6.4. コンパイラ・キャッシュ

`shadow-cljs` はデフォルトで全てのコンパイル結果をキャッシュします。このキャッシュは、個々のソースファイルに関連する何かが変更されるたびに無効になります（例：コンパイラの設定変更、依存関係の変更など）。これにより、インクリメンタルなコンパイルはスクラッチから始めるよりも はるかに 早くなるので、開発者の経験が大幅に改善されます。

しかし、キャッシュを無効にすることは、副作用のあるマクロ（ファイルの読み込み、コンパイラの状態外での保存など）を多く使用している場合、常に確実に実行できるとは限りません。そのような場合には、キャッシュを完全に無効にする必要があります。

副作用のあるマクロが含まれていることがわかっている名前空間は、キャッシュからブロックすることができます。その名前空間自体はキャッシュされず、それを必要とする名前空間もキャッシュされません。clara-rules ライブラリには副作用のあるマクロが含まれており、デフォルトでブロックされます。どの名前空間をグローバルにブロックするかは、`:cache-blockers` 設定で指定できます。この設定には、名前空間のシンボルのセットが必要です。

clara.rules のキャッシュブロックの例（これはデフォルトで行われます）

```
{...
 :cache-blockers #{clara.rules}
 :builds {...}}
```

さらに、`:build-options :cache-level` エントリーを使って、どの程度のキャッシングが行われるかをより広範囲に渡ってコントロールすることができます。サポートされているオプションは次のとおりです。

`:all` デフォルトでは、すべての CLJS ファイルがキャッシュされます。

`:jars` ライブラリ内の `.jar` 形式のソースファイルのみをキャッシュします。

`:off` CLJS のコンパイル結果を一切キャッシュしません（圧倒的に遅いオプションです）。

キャッシュを使わずにコンパイルする

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :build-options
   {:cache-level :off}}}}
```

キャッシュファイルはビルドごとのディレクトリに保存され、ビルド間でキャッシュは共有されません。id が `:app` のビルドには `:dev` のキャッシュがディレクトリに格納されます。

cljs/core.cljs のキャッシュ場所

```
target/shadow-cljs/builds/app/dev/ana/cljs/core.cljs.cache.transit.json
```

デフォルトの `:cache-root` は `target/shadow-cljs` であり、すべてのキャッシュファイルの書き込み先を制御します。これはグローバルにのみ設定可能で、ビルドごとに設定することはできません。

```
{:source-paths [...]
 :dependencies [...]
 :cache-root ".shadow-cljs"
 :builds ...}

;; キャッシュは、.shadow-cljs/builds/app/dev/ana/cljs/core.cljs.cache.transit.json
になります。
```

また、`:cache-root` は常にプロジェクトディレクトリからの相対パスで指定します。絶対パスを指定することもできます (例: `/tmp/shadow-cljs`)。

6.5. Closure の定義

Closure Library と Compiler では、基本的にコンパイル時の定数である変数を定義できます。これを用いてビルドの特定の機能を設定することができます。 Closure コンパイラは `:advanced` 最適化を実行する際にこれらを定数として扱うため、Dead-Code-Elimination パスが完全にサポートされており、`release` ビルドに含めるべきではないコードの特定の部分を削除するために使用することができます。

次のようにコードの中で定義することができます。

```
(ns your.app)

(goog-define VERBOSE false)

(when VERBOSE
  (println "Hello World"))
```

これにより、 `your.app/VERBOSE` 変数がデフォルトで `false` と定義されます。これにより、`:advanced` のコンパイル時に `println` が削除されます。これを `:closure-defines` オプションで `true` に変更すると、`println` が有効になります。これは、開発時のみ、または常に行うことができます。

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :modules {:app {:entries [your.app]}}
   ;; 開発時のみ有効
   :dev {:closure-defines {your.app/VERBOSE true}}
   ;; 常に有効にする
   :closure-defines {your.app/VERBOSE true}
   ;; リリース時の有効化も可能
   :release {:closure-defines {your.app/VERBOSE true}}
   }}
```

TIP

一般的には、disabled バリアントをデフォルトで使用する方が安全です。なぜなら、release ビルドに含まれるべきでないものが含まれる可能性が低くなるからです。また、`:closure-defines` 変数の設定を忘れると、ほとんどの場合、使用されるコードが増えるのではなく、減ることになります。

Closure Library の Closure 定義

- `goog.DEBUG` です。Closure Library では、多くの開発機能でこれを使用しています。 `shadow-cljs` は、 `release` のビルドでは、自動的にこれを `false` に設定します。
- `goog.LOCALE` : `goog.i18n.DateTimeFormat` のような、ある種のローカリゼーション機能を設定するために使用されます。これは標準的なロケール文字列を受け入れ、デフォルトでは `en` となります。ほとんどすべてのロケールがサポートされています。 こちら と こちら を参照してください。

6.6. コンパイラのオプション

CLJS コンパイラは、コードの生成方法に影響を与えるいくつかのオプションをサポートしています。ほとんどの場合、 shadow-cljs は各 :target に対して良いデフォルトを選んでくれますが、時折それらのいくつかを変更したいと思うかもしれません。

これらはすべて、ビルド設定の :compiler-options キーにまとめられています。

```
{:dependencies [...]
 :builds
 {:app
  {:target :browser
   ...
   :compiler-options {:fn-invoke-direct true}}}}
```

標準的な ClojureScript Compiler Optionsのほとんどは、デフォルトで有効になっているか、適用されていません。そのため、実際に効果があるものはほとんどありません。また、多くのオプションは特定の :target タイプに固有のもので、普遍的に適用されるわけではありません。例えば、:compiler-options {:output-wrapper true} は :target :browser にのみ関係します。

現在サポートされているオプションは以下の通りです。

:optimizations	:advanced, :simple, :whitespace のいずれかをサポートしていますが、デフォルトは :advanced です。 none は開発時のデフォルトで、手動で設定することはできません。none を指定した release は動作しません。
:infer-externs	:all, :auto, true または false。デフォルトは true です。
:static-fns	真偽値。デフォルトでは true です。 :fn-invoke-direct : 真偽値。デフォルトは false です。
:ide-asserts	真偽値。デフォルトは、開発版では false 、release ビルドでは true です。
:pretty-print :pseudo-names	真偽値。デフォルトで false です。shadow-cljs release app --debug を使用すると、設定に手を加えることなく、一時的に両方を有効にすることができます。これは release のビルドで問題が発生したときにとても便利です。
:source-map	真偽値。開発中のデフォルトは true 、release では false です。
:source-map-include-sources-content	真偽値。デフォルトは true で、ソースマップが .map ファイルに直接ソースを含めるかどうかを決定します。

`:source-map-detail-level`	`:all` または `:symbols`。`:symbols` は全体のサイズを少し小さくしますが、精度も少し低くなります。
`:externs`	パスのベクトル。デフォルトでは `[]` です。
`:checked-arrays`	真偽値。デフォルトは `false` です。
`:anon-fn-naming-policy`	真偽値
`:warnings`	`{warning-type true|false}` のマップとして指定します。`:warnings {:undeclared-var false}` と指定すると、特定の警告をオフにすることができます。
その他	`:rename-prefix`、`:rename-prefix-namespace`

サポートされていないか適用されていないオプション、また全く影響がない選択肢は以下の通りです。

`:verbose`	`shadow-cljs compile app --verbose` を実行することで制御されますが、ビルド設定にはありません。
`:stable-names`	常に有効で、無効にすることはできません。
`:cache-analysis`	常に有効で、無効にすることはできません。
`:hashbang`	`:node-script` ターゲットはサポートしますが、他のターゲットではサポートされません
`:compiler-stats`	詳細な情報を得るには、代わりに `--verbose` を使用してください。
`:optimize-constants`	`release` のビルドでは常に行われ、無効にはできません。
`:parallel-build`	常に有効です。
`:package-json-resolution`	`:js-options :resolve` を参照してください。
その他	`:foreign-libs`、`:libs`、`:install-deps`、`:source-map-path`、`:source-asset-path`、`:source-map-timestamp`、`:recompile-dependents`、`:preamble`、`:aot-cache`、`:watch-fn`、`:process-shim`

6.6.1. エラーとしての警告

CI(継続的インテグレーション)の環境などでは、ビルドを続行するのではなく、警告を表示してビルドを失敗させたい場合があります。`warnings-as-errors` コンパイラオプションを用いて処理方法をカスタマイズすることができます。

すべての警告をエラーとして扱うための設定

```
{...
 :builds
 {:app
  {...
   :compiler-options {:warnings-as-errors true}}}}
```

特定の警告のみを表示する

```
{...
 :builds
 {:app
  {...
   :compiler-options {:warnings-as-errors #{:undeclared-var}}}}
```

特定の名前空間にのみエラーを投げる

```
{...
 :builds
 {:app
  {...
   :compiler-options {:warnings-as-errors {:ignore #{some.ns some.library.*}
                                           :warnings-types #{:undeclared-var}}}
```

`ignore` には、名前空間を参照するシンボルのセットを指定します。直接マッチするか、または `.*` のワイルドカードを使用することができます。`:warning-types` は上記と同じ機能を持っています。これを指定しないと、無視された名前空間以外のすべての警告がスローされます。

6.7. 出力言語オプション

デフォルトで生成される JS の出力は ES5と互換性があり、すべての新しい機能はポリフィルを使用して互換性のあるコードに変換されます。これが現状で最も安全なデフォルトであり、現在使用されている大半のブラウザ（IE10+を含む）をサポートしています。よりモダンな環境にのみ関心があり、置換せずに元のコードを維持したい場合は、他の出力オプションを選択することができます（例：`node`、Chrome Extensions、...）。

IMPORTANT	これは主にnpmからインポートされた JS コードやclasspathからの `.js` ファイルに影響することに注意してください。 CLJS は現在、ES5 の出力のみを生成し、より高いオプションを設定しても影響を受けません。

`:compiler-options` の `:output-feature-set` で設定できますが、古い `:language-out` オプションは `:output-feature-set` に置き換わるので使用しないでください。以下のオプションがサポートされます。

- `:es3`
- `:es5` - `class`, `const`, `let`, `⋯`
- `:es6` - `class`, `const`, `let`, `⋯`
- `:es7` - 指数演算子 `**`
- `:es8` - `async/await`, `generators`, スプレッド付きオブジェクトリテラル, `⋯`
- `es-next` - Closure コンパイラが現在サポートしているすべての機能です。

Example

```
{...
 :builds
 {:script
  {:target :node-script
   :main foo.bar/main
   ...
   :compiler-options {:output-feature-set :es7}}}}
```

6.8. Conditional Reading

CAUTION	Conditional Reading は shadow-cljs でしか動作しません。この機能は ClojureScript プロジェクトで公式に rejected されました。 CLJS でもコンパイルは可能ですが、公式のブランチでのみ動作します(例: :cljs)。いつかは support になるかもしれませんが、今のところはそうではありません。

shadow-cljs では .cljc ファイルに追加のリーダ機能を設定できます。デフォルトでは、リーダの条件式を使用して :clj、:cljs、:cljr 用の個別のコードを生成することしかできません。

多くの CLJS のビルドでは :target に基づいて生成するコードを選択することも望ましいことです。

例えば、:browser をターゲットにしたときにのみ動作する npm パッケージを :node-script のビルドでも使用したい ns があるかもしれません。これは React アプリでサーバサイドレンダリングを試みるときによく起こるかもしれません。codemirror はそのようなパッケージの一つです。

```
(ns my.awesome.component
  (:require
    ["react" :as react]
    ["codemirror" :as CodeMirror]))

;; React :ref に CodeMirror のインスタンスを作成する場合
(defn init-cm [dom-node]
  (let [cm (CodeMirror/fromTextArea dom-node #js {...})]
    ...))
```

この名前空間は、両方のビルド(:node-script と :browser)で正常にコンパイルされますが、:node-script を実行しようとすると、 codemirror パッケージが DOM にアクセスしようとするため、失敗します。 react-dom/server は refs を使用しないので、 init-cm 関数が呼び出されることはありません。

:closure-defines を使って条件付きで init-cm fn をコンパイルすることはできますが、余分な :require を取り除くために使うことはできません。リーダの条件式を使えば、これが簡単にできます。

```
(ns my.awesome.component
 (:require
   ["react" :as react]
   ;; 注：ここでの順序は重要であり、適用可能な最初のブランチのみが使用されます。
   ;; もし :cljs が最初に使用されたとしても、:server build に引き継がれます。
   #?@(:node [[]]
       :cljs [["codemirror" :as CodeMirror]])))

#?(:node ;; Node プラットフォームのオーバーライド
   (defn init-cm [dom-node]
    :no-op)
   :cljs ;; デフォルトの処理系
   (defn init-cm [dom-node]
     ... actual impl ...))
```

`:reader-features` 設定例

```
{...
 :builds
 ;; アプリのビルドが正常に設定されており、調整は不要
 {:app
  {:target :browser
   ...}
  ;; サーバには :node reader の機能が追加される
  ;; デフォルトの :cljs の代わりに使用される
  :server
  {:target :node-script
   :compiler-options
   {:reader-features #{:node}}}}}
```

これにより、`:server` のビルドには `codemirror` の必要性がなくなり、 `init-cm` の機能も削除されます。以下のようになります。

```
(ns my.awesome.component
  (:require
    ["react" :as react]))

;; 実際にどこにも呼び出されなければ、以下はデッドコードとして削除されます。

(defn init-cm [dom-node] :no-op)
```

IMPORTANT この機能は `.cljc` 形式でのみ可能であり、`.cljs` 形式では失敗します。

6.9. CLI からのオーバーライド

`shadow-cljs.edn` の設定に静的に追加できない値や、環境によって変わる可能性のある値を使って、コマンドラインからビルド構成を少しずつ調整したい場合があります。

追加の設定データを `--config-merge {:some "data"}` コマンドラインオプションで渡すことができ、ビルド時の設定にマージされます。 CLI から追加されたデータは、 `shadow-cljs.edn` ファイルからのデータよりも優先されます。

Example `shadow-cljs.edn` の設定

```
{...
 :builds
 {:app
  {:target :browser
   :output-dir "public/js"
   ...}}}
```

CLI からの `:output-dir` をオーバーライドする

```
$ shadow-cljs release app --config-merge '{:output-dir "somewhere/else"}'
```

CLI からの `:closure-defines` をオーバーライドする

```
$ shadow-cljs release app --config-merge '{:closure-defines {your.app/DEBUG true}}'
```

`--config-merge` は 1 つの EDN マップを想定していますが、複数回使用することができ、左から右に向かってマージされます。追加されたデータは build-hooks でも確認できます。

IMPORTANT	複数のビルド ID を指定した場合、データは指定したすべてのビルドにマージされます。`shadow-cljs release frontend backend --config-merge '{:hello "world"}'` を指定すると、両方に適用されます。

6.10. 環境変数の使用

環境変数を使って `shadow-cljs.edn` の設定値を設定することは可能ですが、代わりに `--config-merge` の使用を検討すべきです。どうしても環境変数を使わなければならない場合は、 `#shadow/env "FOO"` というリーダタグを使って設定することができます。また、より短い `#env` も使用できます。

Example `shadow-cljs.edn` の設定

```
{...
 :builds
 {:app
  {:target :browser
   :output-dir "public/js"
   :closure-defines {your.app/URL #shadow/env "APP_URL"}
   ...}}}
```

また、 `#shadow/env` を使用できるサポートされたフォームもいくつかあります。

```
#shadow/env "APP_URL"
#shadow/env ["APP_URL"]
;; デフォルトの値で、env 変数が設定されていない場合に使用される
#shadow/env ["APP_URL" "default-value"]
#shadow/env ["APP_URL" :default "default-value"]
;; PORT env をデフォルトで整数に変換する
#shadow/env ["PORT" :as :int :default 8080]
```

サポートされている `:as` の強制は `:int`, `:bool`, `:keyword`, `:symbol` です。与えられた `:default` の値は変換されず、すでに正しい型になっていることが期待されます。

IMPORTANT

`shadow-cljs` プロセスが開始されたときに使用された環境変数が使用されます。サーバプロセスが使用されている場合、その環境変数は他のコマンドで設定されたものよりも優先して使用されます。これは主に開発中に関係することですが、混乱を招くかもしれません。また、 `--config-merge` にはこのような制限はありません。

6.11. ビルドとターゲットのデフォルト

すべてのビルド、または特定のタイプのすべてのターゲットに使用されるデフォルトの設定を使用することができます。

設定のマージ順序は以下の通りです。

1. `:build-defaults`
2. `:target-defaults`
3. 実際のビルドコンフィグ
4. 追加の設定によるオーバーライド

Example `shadow-cljs.edn` の設定

```
{...
 :build-defaults
 {:closure-defines
   {your.app/VERBOSE true}}

 :target-defaults
 {:browser
   {:js-options
     {:resolve {"react" {:target :global
                         :global "React"}}}}}

 :builds
 {:app
  {:target :browser
   ...}}}
```

この例では、`:app` ターゲットは、`:build-defaults` と `:browser` の `:target-defaults` の両方を継承します。

IMPORTANT	マージ順で後の方の設定は、前の設定項目を上書きすることはできても、削除することはできません。いったんデフォルトが設定されると、それを削除するにはオーバーライドするしかありません。

Chapter 7. ブラウザを対象とする

`:browser` ターゲットは、ブラウザ環境での実行を想定した出力を行います。開発時には、ライブコードリロード、REPL 、CSS リロードをサポートします。 `release` の出力は、Closure Compiler によって `:advanced` の最適化でミニマイズされます。

基本的なブラウザの設定は以下のようになります。

```
{:dependencies [...]
 :source-paths [...]

 :builds
 {:app {:target :browser
        :output-dir "public/assets/app/js"
        :asset-path "/assets/app/js"
        :modules {:main {:entries [my.app]}}}}}
```

7.1. 出力設定

ブラウザのターゲットは多くのファイルを出力し、それらすべてのためのディレクトリが必要となります。これらのアセットを何らかのサーバで提供する必要があり、Javascript コードの読み込みのためにサーバ内でのアセットへのパスを知る必要があります。以下のオプションを指定する必要があります。

`:output-dir`

コンパイラのすべての出力に使用するディレクトリです。

`:asset-path`

Web サーバのルートから `:output-dir` にあるリソースへの相対パスです。

WARNING	各ビルドにはそれぞれ :output-dir が必要で、複数のビルドを同じディレクトリに置くことはできません。 また、このディレクトリは、そのビルドが独占的に所有するものでなければなりません。そこには他のファイルがあってはいけません。 `shadow-cljs` は何も削除しませんが、そのままにしておいた方が安全です。コンパイルでは、ソースマップ、オリジナルソース、生成ソースなど、開発中のメインエントリーポイントの javascript ファイル以外にも多くのファイルが作成されます。

`:asset-path` は、生成された javascript の中のモジュール読み込みコードのパスに追加されるプレフィックスです。これにより、javascript モジュールを Web サーバーのルートの特定のサブディレクトリに出力することができます。動的な読み込み（開発時のホットコードリロードや運用時のコード分割）では、ファイルを正しく配置するためにこれが必要です。

このように、生成したファイルをディレクトリやアセットパスに配置することで、他のアセット（画像やCSSなど）が同じサーバー上で不用意に衝突することなく共存できるようになります。

例えば、`/x` というURIを求められたときに、Webサーバーが `public/x` というフォルダを提供し、モジュールの `output-dir` が `public/assets/app/js` である場合、アセットパスは `/assets/app/js` となります。アセットパスの絶対指定は必須ではありませんが、強く推奨します。

7.2. モジュール

モジュールは、コンパイルされたソースがどのように束ねられ、最終的な `.js` がどのように生成されるかを設定します。各モジュールは、エントリーネームスペースのリストを宣言し、そこから依存関係グラフを構築します。複数のモジュールを使用する場合は、最大量のコードがグラフの外縁に移動するようにコードが分割されます。その目的は、ブラウザが最初に読み込むコードの量を最小限にし、残りのコードをオンデマンドで読み込むことです。

TIP	最初は :module を気にしすぎないこと。最初は 1 つにしておいて、後で分割するようにしましょう。

コンフィグの `:modules` セクションは、常にモジュール ID をキーとしたマップです。モジュールIDは、Javascript のファイル名を生成する際にも使用されます。 モジュール `:main` は `:output-dir` に `main.js` を生成します。

1つのモジュールで利用できるオプションは以下の通りです。

`:entries`	このモジュールの出力コードの依存関係グラフのルートノードとして機能する名前空間です。
`:init-fn`	モジュールが最初にロードされたときに呼び出されるべき関数を指し示す完全修飾シンボルです。
`:depends-on`	このモジュールが必要とするものをすべて備えるために、ロードされなければならない他のモジュールの名前です。
`:prepend`	JS の出力の前に付加される文字列コンテンツです。コメントや著作権表示などに便利です。
`:append`	JS の出力に追加される文字列コンテンツです。コメントや著作権表示などに便利です。
`:prepend-js`	Closure オプティマイザーで実行される、有効な javascript をモジュールの出力に追加する文字列です。
`:append-js`	Closure オプティマイザーで実行される有効な javascript をモジュールの出力に追加する文字列です。

以下の例では、最小のモジュール構成を示しています:

Example :browser config

```
{...
 :builds
 {:app {:target :browser
        :output-dir "public/js"
        ...
        :modules {:main {:entries [my.app]}}}}}
```

Example :init-fn によるブラウザ設定

```
{...
 :builds
 {:app {:target :browser
        :output-dir "public/js"
        ...
        :modules {:main {:init-fn my.app/init}}}}}
```

`shadow-cljs` は、`:entries` にあるコードエントリポイントのルートセットから依存関係グラフをたどり、実際にコンパイルして出力に含めるために必要なものをすべて見つけます。必要とされない名前空間は含まれません。

上記の設定により、`public/js/main.js` というファイルが作成されます。開発段階では、多くのファイルが格納された `public/js/cljs-runtime` ディレクトリが追加されます。このディレクトリは `release` のビルドでは必要ありません。

7.3. CodeSplitting

複数のモジュールを宣言すると、モジュール同士がどのように関連しているか、また後でどのようにロードするかをコンパイラが把握できるように、ほんの少しだけ静的な設定を追加する必要があります。

`entries` に加えて、どのモジュールがどのモジュールに依存しているか（ `:depend-on` で）を宣言する必要があります。これをどのように構成するかはニーズ次第で、残念ながら万能のソリューションはありません。

例えば、従来の Web サイトでは、さまざまなページがあります。

- `www.acme.com` - ホームページを表示します。
- `www.acme.com/login` - ログインフォームを提供します。
- `www.acme.com/protected` - ユーザーがログインしないと利用できない保護されたセクション

このような場合には、すべてのページで共有される共通のモジュールを 1 つ用意するのが良いでしょう。そして、各ページごとに 1 つのモジュールを用意します。

Example 複数の `:modules` をもつ設定

```
{...
 :output-dir "public/js"
 :modules
 {:shared
  {:entries [my.app.common]}
  :home
  {:entries [my.app.home]
   :depends-on #{:shared}}
  :login
  {:entries [my.app.login]
   :depends-on #{:shared}}
  :protected
  {:entries [my.app.protected]
   :depends-on #{:shared}}
```

TIP `:shared` モジュールの `:entries` を空にすることで、どの名前空間が他のモジュールと共有されているかをコンパイラに把握させることができます。

生成されたファイル構造

```
.
└── public
    └── js
        ├── shared.js
        ├── home.js
        ├── login.js
        └── protected.js
```

ホームページの HTML には、各ページに必ず shared.js を記述し、他のページはユーザーがどのページにいるかに応じて条件付きで記述することになります。

ログインページの *HTML*

```
<script src="/js/shared.js"></script>
<script src="/js/login.js"></script>
```

IMPORTANT	`.js` ファイルは正しい順序でインクルードする必要があります。これには `manifest.edn` が役立ちます。

7.3.1. 動的にコードを読み込む

最近、シングルページアプリ（SPA）が人気を集めていますが、その仕組みは、どの JS を組み込むかをサーバが決めるのではなく、クライアントが自分で決めるという点で似ています。

shadow-cljs に組み込まれた **Loader Support** の使用

コンパイラは、 `shadow.loader` ユーティリティー名前空間の使用に必要なデータの生成をサポートしています。これは、実行時にオンデマンドでモジュールをロードするためのシンプルなインターフェイスを公開しています。

ビルド設定に `:module-loader true` を追加するだけでいいのです。ローダーは常にデフォルトのモジュール（他のすべてが依存するモジュール）に注入されます。

実行時には `shadow.loader` 名前空間を使ってモジュールをロードすることができます。また、ページ内で `<script>` タグを使用することで、モジュールをイーガーリーにロードすることもできます。

```
{...
 :builds
   {:app
     {:target :browser
      ...
      :module-loader true
      :modules {:main  {:entries [my.app]}
                :extra {:entries [my.app.extra]
                        :depends-on #{:main}}}}}}
```

メインのエントリーポイントに以下のようなものがあったとします。

```
(ns my.app
  (:require [shadow.loader :as loader]))

(defn fn-to-call-on-load []
  (js/console.log "extra loaded"))

(defn fn-to-call-on-error []
  (js/console.log "extra load failed"))
```

そうすると、コードの読み込みに以下のような表現が使えるようになります。

モジュールの読み込み

```
;; load は goog.async.Deferred を返し、promise のように使うことができます
(-> (loader/load "extra")
    (.then fn-to-call-on-load fn-to-call-on-error))
```

多数のモジュールの読み込み

```
;; JS配列でなければならず、goog.async.Deferred も返します。
(loader/load-many #js ["foo" "bar"])
```

コールバックを含める場合

```
(loader/with-module "extra" fn-to-call-on-load)
```

モジュールがロードされているかどうかは、`(loaded? "module-name")` で確認できます。

ローダーのコスト

ローダーの使用は非常に軽量です。ローダーにはいくつかの依存関係がありますが、他に使用することはないでしょう。実際には、`:module-loader true` を使用すると、デフォルトのモジュールに約8KB の gzip が追加されます。これは、すでに使用している `goog.net` や `goog.events` の量や、リリースビルドでどの程度の最適化を行っているかによって変わってきます。

標準的な **ClojureScript API** の使用

生成されたコードは、標準的な ClojureScript の `cljs.loader` API を使用することができます。手順については、ClojureScript ウェブサイトの documentation を参照してください。

標準API を使用することの利点は、自分のコードが他の人とうまく調和することです。これはライブラリの作者にとっては特に重要なことでしょう。不利な点は、標準配布の動的なモジュールローディング API は、現在のところ `shadow-cljs` のサポートに比べてやや使いにくいことです。

7.4. 出力ラッパー

リリースビルドのみ：Closure Compiler `:advanced` で生成されたコードは、多くのグローバル変数を作成し、ページ内で実行されている他の JS と競合する可能性があります。生成された変数を分離するために、コードを無名関数でラップし、そのスコープ内でのみ変数が適用されるようにすることができます。

`:modules` がひとつしかない `:browser` の `release` ビルドは、デフォルトでは `(function(){<the-code>}).call(this);` でラップされます。そのため、グローバル変数は作成されません。

複数の `:module` を使用している場合(別名 コードスプリッティング) 、各モジュールは依存しているモジュールが作成した変数にアクセスできなければならないため、このオプションはデフォルトでは有効になっていません。

Closure コンパイラは、`:rename-prefix-namespace` という複数の `:modules` と組み合わせた出力ラッパーの使用を有効にする追加オプションをサポートしています。これにより、コンパイラはビルドで使用されるすべてのグローバル変数を、単一のグローバル変数にスコープします。デフォルトでは、`:output-wrapper` が `true` に設定されている場合、これは `:rename-prefix-namespace "$APP"` に設定されます。

```
{...
 :builds
 {:target :browser
  ...
  :compiler-options
  {:output-wrapper true
   :rename-prefix-namespace "MY_APP"}}}
```

これは、 `MY_APP` というグローバル変数を作成するだけです。すべてのグローバル変数の前には `MY_APP.` がつくので (たとえば、 `a` だけではなく `MY_APP.a`)、コードサイズは大幅に増加します。これを短くすることが重要です。ブラウザの圧縮（例：`gzip`）は、余分なコードのオーバーヘッドを減らすのに役立ちますが、ビルド内のグローバル変数の量に応じて、これでもまだ顕著な増加が見られます。

IMPORTANT	実際には作成された変数を直接使えません。作成された変数は実際には使い物になりませんが、多くのプロパティを含んでいます。エクスポートされた変数（`^:export`等）はすべてグローバルスコープにエクスポートされ、この設定の影響を受けません。この設定はグローバル変数の作成量を制限するためだけのものであり、直接使用しないでください。

7.5. Web Worker

`modules` の設定は、Web Worker として使用されるファイルを生成するためにも使用できます。 `web-worker true` を設定することで、任意のモジュールを Web Worker として宣言することができます。生成されたファイルには、いくつかの追加ブートストラップコードが含まれます。 `:modules` の働きにより、ワーカーのみが使用するコードは、ワーカーの最終ファイルにのみ含まれることになります。各ワーカーは専用の CLJS 名前空間を持つべきです。

Web Worker スクリプトの生成の一例

```
{...
 :builds
 {:app
  {:target :browser
   :output-dir "public/js"
   :asset-path "/js"
   ...
   :modules
   {:shared
    {:entries []}
    :main
    {:init-fn my.app/init
     :depends-on #{:shared}}
    :worker
    {:init-fn my.app.worker/init
     :depends-on #{:shared}
     :web-worker true}}
   }}}
```

上記の設定を行うと、Web Worker を起動するための `worker.js` が生成されます。 Worker.js は `:shared` モジュールのすべてのコードを利用できます (ただし `:main` は利用できません)。 `my.app.worker` 名前空間にあるコードは、ワーカーの中でのみ実行されます。ワーカーの生成は、開発モードとリリースモードの両方で行われます。

なお、`:shared` モジュールで空の `:entries []` を指定すると、`:main` モジュールと `:worker` モジュールの間で共有されるすべてのコードを収集するようになります。

Example エコー・ワーカー

```
(ns my.app.worker)

(defn init []
  (js/self.addEventListener "message"
    (fn [^js e]
      (js/postMessage (.. e -data)))))
```

Sample ワーカーの使用

```
(ns my.app)

(defn init []
  (let [worker (js/Worker. "/js/worker.js")]
    (.. worker (addEventListener "message" (fn [e] (js/console.log e))))
    (.. worker (postMessage "hello world"))))
```

IMPORTANT	現在、`:shared` モジュールがあるので、HTML で適切にロードする必要があります。単に `main.js` をロードしただけでは、エラーが発生します。

HTML shared.js と main.js の読み込み

```
<script src="/js/shared.js"></script>
<script src="/js/main.js"></script>
```

7.6. キャッシュ可能な出力

Web環境では、余分なリクエストを避けるために、 `.js` ファイルを非常に長い時間キャッシュすることが望ましいです。リリースされたバージョンごとに、 `.js` ファイルにユニークな名前をつけるのが一般的です。これにより、ファイルへのアクセスに使用される URL が変更されるため、永久にキャッシュしても安全です。

7.6.1. リリースバージョン

各リリースに固有のファイル名を作成するには、`:release-version` という設定を使用します。一般的には、コマンドラインから --config-merge でこの設定を渡します。

```
shadow-cljs release app --config-merge '{:release-version "v1"}'
```

Example :modules config

```
{...
 :builds
   {:app
     {:target :browser
      ...
      :output-dir "public/js"
      :asset-path "/js"
      :modules {:main  {:entries [my.app]}
                :extra {:entries [my.app.extra]
                        :depends-on #{:main}}}}}}
```

これにより、 `main.v1.js` と `extra.v1.js` のファイルが、通常の `main.js` と `extra.js` ではなく、`public/js` に作成されます。

手動のバージョンを使うこともできますし、ビルド時に `git` sha のような自動化されたものを使うこともできます。ただ、ユーザーに何かを出荷したときには、それが何であれ、キャッシュを使って、古いファイルの新しいバージョンを要求しないようにしてください。

7.6.2. フィンガープリント・ハッシュを使ったファイル名

ビルド設定に `:module-hash-names true` を追加すると、生成される各出力モジュールファイルに MD5 署名を自動的に作成することができます。つまり、`:main` モジュールは、デフォルトの `main.js` ではなく、`main.<md5hash>.js` を生成することになります。

`:module-hash-names true` は、 32 個の完全な md5ハッシュを含みますが、より短いバージョンを好む場合は、代わりに1〜32の数字を指定できます。 1-32の間の数字を指定できます(例: `:module-hash-names 8`)。ハッシュを短くすると、コンフリクトが発生する可能性が高くなることに注意してください。 競合が発生する可能性が高くなることに注意してください。完全なハッシュを使うことをお勧めします。

Example :module-hash-names config

```
{...
 :builds
   {:app
     {:target :browser
      ...
      :output-dir "public/js"
      :asset-path "/js"
      :module-hash-names true
      :modules {:main  {:entries [my.app]}
                :extra {:entries [my.app.extra]
                        :depends-on #{:main}}}}}}
```

`main.js` を生成するのではなく、`:output-dir` に `main.<hash>.js` を生成するようになりました。

ファイル名はリリースごとに変更される可能性があるため、それらを HTML に含めるのは少し複雑になります。 HTML に含めるのは少し複雑です。Output Manifestはその手助けとなります。

7.7. 出力マニフェスト

shadow-cljs は設定された :output-dir に manifest.edn ファイルを生成します。このファイルには、モジュール設定の説明と、追加の :output-name プロパティが含まれています。

オリジナルのモジュール名を実際のファイル名にマッピングします（:module-hash-names 機能を使用する際に重要です)。

ハッシュ化されたファイル名を使用した場合の *manifest.edn* の出力例

```
[{:module-id :common,
  :name :common,
  :output-name "common.15D142F7841E2838B46283EA558634EE.js",
  :entries [...],
  :depends-on #{},
  :sources [...]}
 {:module-id :page-a,
  :name :page-a,
  :output-name "page-a.D8844E305644135CBD5CBCF7E359168A.js",
  :entries [...],
  :depends-on #{:common},
  :sources [...]}
 ...]
```

マニフェストには、すべての :module が依存関係のある順に並べられています。これを使って、:module-id を実際に生成されたファイル名にマッピングすることができます。

開発用のビルドでもこのファイルが生成されますので、新しいビルドが完了したときに修正のためにチェックすることができます。開発中は :module-hash-names が適用されないので、通常のファイル名が表示されます。

生成されるマニフェストファイルの名前は、:build-options :manifest-name エントリで設定できます。デフォルトでは manifest.edn となります。ファイル名の最後に .json を設定すると、EDN ではなく JSON が出力されます。ファイルは構成された :output-dir からの相対パスになります。

Example manifest.json の設定

```
{...
 :builds
   {:app
     {:target :browser
      ...
      :build-options {:manifest-name "manifest.json"}
      :modules {:main  {:entries [my.app]}
                :extra {:entries [my.app.extra]
                        :depends-on #{:main}}}}}}
```

7.8. 開発サポート

`:browser` の設定の `:devtools` セクションでは、ビルドやCSSのリロードのために、オプションで dev-time HTTP サーバーを設定するために、いくつかの追加オプションをサポートしています。

7.8.1. ヘッドアップディスプレイ（HUD）

`:browser` ターゲットは、HUD を使って、ビルドが開始されたときにローディング・インジケータを表示するようになりました。また、警告やエラーが発生した場合にも表示されます。

`devtools` セクションで `:hud false` を設定することで、完全に無効にすることができます。

また、`:hud #{:errors :warnings}` という設定で気になる機能を指定して、特定の機能を切り替えることもできます。これにより、エラーや警告は表示されますが、進捗状況は表示されません。利用可能なオプションは `:errors :warnings :progress` です。含まれるオプションのみが有効になり、それ以外は無効になります。

ファイルを開く

警告にはソースの場所へのリンクが含まれており、クリックするとそのファイルをエディタで開くことができます。このためには、ちょっとした設定が必要です。

この設定は、プロジェクトのための `shadow-cljs.edn` 設定の中で行うか、ホームディレクトリの `~/.shadow-cljs/config.edn` でグローバルに行うことができます。

:open-file-command の設定

```
{:open-file-command
 ["idea" :pwd "--line" :line :file]}
```

`open-file-command` では、非常にシンプルな DSL を表すベクターを想定しています。文字列はそのままで、キーワードはそれぞれの値で置き換えられます。 `clojure.core/format` スタイルのパターンを使用して、複数のパラメータを組み合わせる必要がある場合には、ネストしたベクターを使用することができます。

上記の例では、以下のように実行されます。

```
$ idea /path/to/project-root --line 3 /path/to/project-root/src/main/demo/foo.cljs
```

emacsclient の例

```
{:open-file-command
 ["emacsclient" "-n" ["+%s:%s" :line :column] :file]}
```

```
$ emacsclient -n +3:1 /path/to/project-root/src/main/demo/foo.cljs
```

利用可能な置換変数は以下の通りです。

`:pwd`
: プロセスの作業ディレクトリ（別名：プロジェクトルート）

`:file`
: 絶対ファイルパス

`:line`
: 警告・エラーの行番号

`:column`
: コラム番号

`:wsl-file`
: 変換された WSL ファイルのパス。 WSL の Bash で `shadow-cljs` を実行するときに便利です。 `mnt/c/Users/someone/code/project/src/main/demo/foo.cljs` のパスを `C:Users...` に変換します。

`:wsl-pwd`
: 変換された `:pwd`

7.8.2. CSS リローディング

Example HTML スニペット

```
<link rel="stylesheet" href="/css/main.css"/>
```

Example Hiccup 俺たちは野蛮人じゃないから

```
[:link {:rel "stylesheet" :href "/css/main.css"}]
```

内蔵されている *dev HTTP* サーバの利用

```
:dev-http {8000 "public"}
```

これにより、 `public/css/main.css` が変更されると、ブラウザは `/css/main.css` を再読み込みします。

現在、 `shadow-cljs` は CSS を直接コンパイルすることをサポートしていませんが、通常のツールは動作します。 を別途実行する必要があります。ただ、出力が正しい場所に生成されることを確認してください。

組み込みのHTTPサーバーを使用しない場合は、代わりに `:watch-dir` を指定することができ、コンテンツの配信に使用されるドキュメントルートへのパスを指定します。

Example :watch-dir の設定

```
{...
    {:builds
      {:app {...
             :devtools {:watch-dir "public"}}}}
```

HTTP サーバが仮想ディレクトリからファイルを提供していて、ファイルシステムのパスが HTML で使われているパスと完全に一致しない場合、プレフィックスとして使われる `:watch-path` を設定することで、パスを調整することができます。

Example `public/css/main.css` は `/foo/css/main.css` 下でサーブされます。

```
{...
 {:builds
  {:app
   {...
    :devtools {:watch-dir "public"
               :watch-path "/foo"}}}}
```

7.8.3. プロキシサポート

デフォルトでは、devtools クライアントは、設定された HTTP server (通常は `localhost`) 経由で `shadow-cljs` プロセスへの接続を試みます。リバースプロキシを使用して HTML を配信している場合、接続できない場合があります。 `:devtools-url` を設定することにより、使用する URL を設定することができます。

```
{...
 :builds
 {:app {...
        :devtools {:before-load  my.app/stop
                   :after-load   my.app/start
                   :devtools-url "https://some.host/shadow-cljs"
                   ...}}}}
```

`shadow-cljs` は、リクエストの際に `:devtools-url` をベースとして使用します。これは最終的な URL ではありませんので、設定したパス (例: `/shadow-cljs/*`) で始まるすべてのリクエストが、 `shadow-cljs` が実行されているホストに転送されるようにする必要があります。

プロキシへの Incoming リクエストは、

```
https://some.host/shadow-cljs/ws/foo/bar?asdf
```

以下に転送する必要があります。

```
http://localhost:9630/foo/bar?asdf
```

クライアントは、ファイルを読み込むため、通常の XHR リクエストだけでなく WebSocket リクエストも行います。プロキシが WebSocket を適切にアップグレードするようにしてください。

IMPORTANT	リクエストは、ビルド自体で設定したものではなく、メインの HTTP server に転送されなければなりません。

Chapter 8. React Native をターゲットにする

`:target :react-native` は、デフォルトの `react-native` ツール (例: `metro`) に統合することを意図したコードを生成します。これらのツールをラップした `expo` のようなツールは自動的に動作し、追加の設定は必要ありません。

他のターゲット (`:source-paths` など) と同じ基本的な main configuration が必要になります。ビルド専用の設定は非常に少なく、(`:target` 自体の他に) 少なくとも2つのオプションが必要です。

`:init-fn` (必須) アプリの init関数の名前空間修飾されたシンボルです。この関数は起動時に一度だけ呼び出され、おそらく何かをレンダリングするはずです。

`:output-dir` (必須) 出力ファイルを書き込むためのディレクトリです。

:react-native の設定例

```
{:source-paths [...]
 :dependencies [...]
 ...
 :builds
 {:app
  {:target :react-native
   :init-fn demo.app/init
   :output-dir "app"}}}
```

これをコンパイルすると、 `react-native` ツールのエントリーポイントとして使用される `app/index.js` ファイルが生成されます。開発中、`:output-dir` にはさらに多くのファイルが格納されますが、生成された `app/index.js` を直接参照するだけにしてください。 `release` のビルドでは、最適化された `app/index.js` が生成されるだけで、追加のファイルは必要ありません。

8.1. React Native

`react-native` には、ネイティブコードやライブラリを使用できる純粋な `react-native` と expo でラップされた `react-native` の 2 つの利用法があります。上記の手順は全て、純粋な `react-native` で `shadow-cljs` を使い始めるのに十分なものです。次の例をご覧ください。

- https://github.com/thheller/reagent-react-native

8.2. Expo

expo を使うと、 react-native の作業がとても簡単になります。 2 つのサンプルが用意されています。

- https://github.com/thheller/fulcro-expo
- https://github.com/thheller/reagent-expo

どちらの例も expo init を使用して生成されています。設定で調整した唯一の変更点は、生成された app.json に適切な entryPoint を追加したことです。

```
{
  "expo": {
    "name": "hello-world",
    "slug": "reagent-expo",
    ...
    "entryPoint":"./app/index.js",
    ...
  }
}
```

expo では、起動時に React Component が登録される必要があります。これは手動で行うこともできますし、Component の作成を行い、代わりに React Element インスタンスがレンダリングを開始することを直接期待する shadow.expo/render-root 関数を使用することもできます。

Reagent の例

```
(defn start
  {:dev/after-load true}
  []
  (expo/render-root (r/as-element [root])))

(defn init []
  (start))
```

init は起動時に一度だけ呼ばれます。この例では特別な設定をする必要がないので、単に start を直接呼び出します。start は、コードの変更がリロードされた後に watch が実行されるたびに、繰り返し呼び出されます。

reagent.core/as-element 関数は、reagent hiccup のマークアップから必要な React Element を生成するために使用することができます。

Chapter 9. Node.js を対象とする

独立したスクリプトとして使用することを目的としたコードや、ライブラリとして使用することを目的としたコードの生成をサポートしています。設定ファイルに必要な基本的な設定については、common configuration の項を参照してください。

9.1. Node.js のスクリプト

`target :node-script` は、`Node.js` を使用して実行できるシングルファイルのスタンドアローン出力を生成します。 コードは単なる ClojureScript であり、エントリーポイントの定義も簡単です。

```
(ns demo.script)

(defn main [& cli-args]
  (prn "hello world"))
```

9.1.1. ビルドオプション

他のターゲット（ `:source-paths` など）と同じ基本的な main configuration が必要になりますが、いくつかのノード固有のビルドターゲットオプションが必要になります。

`:main`	(必須) スクリプトのエントリーポイントとなる関数の名前空間修飾されたシンボルです。
`:output-to`	(必須) 生成されるスクリプトのパスとファイル名です。
`:output-dir`	(オプション) 開発モードでサポートするファイルのパスです。デフォルトではキャッシュディレクトリになります。

Node スクリプトのビルド例

```
{:source-paths [...]
 ...
 :builds
 {:script
  {:target :node-script
   :main demo.script/main
   :output-to "out/demo-script/script.js"}}}
```

これをコンパイルすると、スタンドアロンの `out/demo-script/script.js` ファイルが作成され、 `node script.js <command line args>` で呼び出されるようになります。実行すると、起動時に `(demo.script/main <command line args>)` 関数が呼び出されます。これは `:output-to` で指定されたファイルのみを生成します。その他のサポートファイル(開発モード用など)は、一時的なサポートディレクトリに書き込まれます。

9.1.2. ホットコードリロード

サーバーやその他の長時間実行されるプロセスとして動作するスクリプトを書くことがあります。ホットコードリロードはそのような際に非常に便利で、設定も簡単です。start/stop　コールバック関数の追加してフックを使用するようにビルドを設定します。以下、node の http サーバーの例を紹介します。

ホットコードリロードのための *start/stop* フックを備えた *Node* のサンプル

```
(ns demo.script
  (:require ["http" :as http]))

(defn request-handler [req res]
  (.end res "foo"))

; サーバーを停止/起動できるようにするための、サーバーに接続するための場所
(defonce server-ref
  (volatile! nil))

(defn main [& args]
  (js/console.log "starting server")
  (let [server (http/createServer #(request-handler %1 %2))]

    (.listen server 3000
      (fn [err]
        (if err
          (js/console.error "server start failed")
          (js/console.info "http server running"))
        ))
    (vreset! server-ref server)))

(defn start
  "開始するためのフック。ホットコードのリロード用のフックとしても使用される"
  []
  (js/console.warn "start called")
  (main))

(defn stop

"ホットコードリロードフックで、ホットコードリロードが機能するようにリソースをシャットダウンする"
  [done]
  (js/console.warn "stop called")
  (when-some [srv @server-ref]
    (.close srv
      (fn [err]
        (js/console.log "stop completed" err)
        (done)))))

(js/console.log "__filename" js/__filename)
```

関連する設定は（ `shadow-cljs.edn` ）です。

ホットコードリロードのためのフックの追加

```
{...
 :builds
   { :script {... as before

              ; リロードフックの追加
              :devtools {:before-load-async demo.script/stop
                         :after-load demo.script/start}}}}
```

WARNING	多くのライブラリは状態を隠したり、ホットコードのリロードがうまく機能しないような動作をします。コンパイラはこれらのライブラリが何をしているのかを知らないので、これを改善することはできません。 ホットコードリロードは、使用されているアーティファクトをきれいに stop および restart できる状況でのみうまく機能します。

9.2. Node.js ライブラリ

`:target :node-library` は、標準的なノードライブラリとして（ `require` を介して）使用できるコードを出力します。これは、コンパイルされたJavascript の成果物として再利用するためにコードを公開するのに便利です。

他のモードと同様に、main configuration optionsが適用され、設定する必要があります。 ターゲット固有のオプションは以下の通りです。

- `:target`: `:node-library` を使用します。
- `:output-to` : (必須) 生成されるライブラリのパスとファイル名です。
- `:output-dir` : (オプション) 開発モードでサポートするファイルのパスです。デフォルトではキャッシュディレクトリになります。

ホットコードリロードのストーリーは、the script targetと似ていますが、すべてのコードを簡単にコントロールできないため、うまく機能しないかもしれません。 ロードされるすべてのコードを簡単にコントロールすることができないからです。

実際にエクスポートされるコードを制御するには、以下のオプションがあります。

- `:exports` - キーワードから完全修飾シンボルへのマップ
- `:exports-var` - 完全修飾形式のシンボル
- `:exports-fn` - 完全修飾されたシンボル

9.2.1. 単一で静的な デフォルト・エクスポート

`:exports-var` は、その var の下で宣言されているものをそのまま返します。 これは `defn` や通常の `def` を指しています。

:exports-var を使用してコンフィグを構築する

```
{...
 :builds {:lib {:output-to "lib.js"
                :exports-var demo.ns/f
                ...}}}
```

Example CLJS

```
(ns demo.ns)

(defn f [...] ...)
;; または
(def f #js {:foo ...})
```

```
$ node
> var f = require('./lib.js');
f(); // 実際の demo.ns/f 関数
```

事実上モジュールを生成しています。 `exports = demo.ns.f;`

9.2.2. 複数の静的な名前付きエクスポート

複数のエクスポートを行うビルド構成

```
{...
 :builds {:lib {:exports {:g       demo.ns/f
                          :h       other.ns/thing
                          :ns/ok?  another.ns/ok?}
                ...}}}
```

このキーワードは、エクスポートされたオブジェクトのエントリの名前として使用されます。このキーワード名には、一切の処理が行われません。(しかし、名前空間は削除されます）。したがって、上記の例では、cljs の `f` を `g` にマッピングします。

```
$ node
> var lib = require('./lib.js');
lib.g(); // call demo-ns/f
lib["ok?"](); // another-ns/ok? を呼び出す
```

全く同じことをするには、`:exports-var` で `def` を指定します。

```
(def exports #js {:g f
                  ...})
```

9.2.3. 動的エクスポート

さらに完全修飾シンボルとして `:exports-fn` を指定することもできます。これは、JS オブジェクト(または関数)を返す引数のない関数を指す必要があります。この関数は `node` が戻り値をキャッシュするため、一度だけ呼ばれます。

```
(ns demo.ns
  (:require [demo.other :as other]))

(defn generate-exports []
  #js {:hello hello
       :foo other/foo})
```

```
{...
 :builds {:lib {:exports-fn demo.ns/generate-exports
                ...}}}
```

NOTE	exports コンフィグはエクスポートされたシンボルを自動的に追跡し、最適化ステージに渡します。つまり、`:exports` に記載されているものは、Google Closure の最適化によって名前が変更されることはありません。

9.2.4. 完全な例

以下の例では、通常の Node `require` メカニズムで使われることを想定して `lib.js` ファイルを作成します。

```
(ns demo.lib)

(defn hello []
  (prn "hello")
  "hello")
```

ビルド構成は以下のようになります。

```
{...
 :builds {:library {:target    :node-library
                    :output-to "out/demo-library/lib.js"
                    :exports   {:hello demo.lib/hello}}}}
```

ランタイムの使用感は期待通りです。

```
$ cd out/demo-library
$ node
> var x = require('./lib');
undefined
> x.hello()
hello
'hello'
```

`node-script` と同様に `:output-to` で指定されたファイルのみが作成されます。

`exports` マップは CLJS の変数をエクスポートされるべき名前にマッピングします。

NOTE	開発モードでは、ノードスクリプトと同じ同じ設定になります（追加の依存関係があります）。

Chapter 10. JS エコシステムへの組み込み - :npm-module ターゲット

CLJSを既存のJSプロジェクトに統合することを目的としたターゲットがある場合、少しの設定で webpack、browserify、babel、create-react-app 等の JS ツールに統合できます。

`:output-dir` はデフォルトでは `node_modules/shadow-cljs` であり、`:entries` はコンパイルされる名前空間のシンボルを含むベクトルです。

Example `shadow-cljs.edn` の設定

```
{...
 :builds
 {:code
  {:target :npm-module
   :entries [demo.foo]}}}
```

デフォルトの `:output-dir` である `"node_modules/shadow-cljs"` を使用すると、JS で `require("shadow-cljs/demo.foo")` を使用して、宣言された名前空間にアクセスすることができます。 `node_modules` にないものを使用する場合は、相対パスを使ってインクルードする必要があります。 `:output-dir "out"` では、プロジェクトルートからの `require("./out/demo.foo")` となります。

npm でコードを配布する予定であれば、代わりに`:node-library` targetを使用した方が、エクスポートや最適化をより細かく制御できるので良いでしょう。

10.1. 最適化の作業

`gnome-library` ターゲットとは異なり、module ターゲットはエクスポートするシンボルをどのように呼びたいかわからないので、そのままエクスポートします。高度なコンパイルを使用している場合は、すべてのシンボルに minified munged の名前が付けられます。

保存したいシンボルに `:export` メタデータを追加するだけで、簡単に解決できます。

```
(ns demo.foo)
(def ^:export foo 5.662)
(defn ^:export bar [] ...)
```

これは、ClojureScriptで理解される標準的なアノテーションで、Google Closure がアーティファクトの名前を変更するのを防ぎます。JS コードは最適化後もそれらにアクセスすることができます。 `^:export` のヒントがなければ、Closure コンパイラがそれらを削除したり、名前を変更したりするでしょう。

```
var ns = require("shadow-cljs/demo.foo");

ns.foo;
ns.bar();
```

Chapter 11. テスト

shadow-cljs は、テストの構築を少しでも簡単にするために、いくつかのユーティリティターゲットを提供しています。

すべてのテストターゲットはテストランナーを生成し、設定可能な :ns-regexp にマッチするすべての名前空間を自動的に追加します。デフォルトのテストランナーは cljs.test 用に作られていますが、他のテストフレームワークを使いたい場合は、カスタムランナーを作ることができます。

デフォルトの :ns-regexp は "-test$" なので、最初のテストは以下のようになります。

File: src/test/demo/app_test.cljs.

```
(ns demo.app-test
  (:require [cljs.test :refer (deftest is)]))

(deftest a-failing-test
  (is (= 1 2)))
```

Clojure の世界では、テストファイルをそれぞれのソースパスに置いておくのが一般的なので、上記の例では、 shadow-cljs.edn の設定で :source-paths ["src/main" "src/test"] を設定していると仮定しています。通常のアプリのコードは src/main に入り、テストは src/test に入ります。ただし、これはオプションで、すべてを src に置いて、:source-paths ["src"] を使うだけでも全く問題ありません。

11.1. Node.js におけるテスト

このターゲットは、指定された正規表現にマッチするすべてのテスト名前空間を含むテストランナーを作成します。

関連する設定オプションは以下の通りです。

:target :node-test

:output-to テストの実行に使用される最終的な出力ファイルです。

:ns-regexp (オプション) プロジェクトファイルの名前空間にマッチする正規表現です。これはファイルをスキャンするだけで、ジャーはスキャンしません。デフォルトでは "-test$" となります。 :autorun: (オプション) (boolean, optional) ビルドが完了した際に、node を介してテストを実行します。これは主に watch と組み合わせて使用することを想定しています。実行中の JVM を強制的に終了させる必要があるため、 node プロセスの終了コードは返されません。

:main デフォルトは shadow.test.node/main で、これは cljs.test を使ってテストを実行します。

すべての *-spec 名前空間にマッチするテストコンフィグ

```
{...
 :builds
 {:test
  {:target    :node-test
   :output-to "out/node-tests.js"
   :ns-regexp "-spec$"
   :autorun   true}}}
```

node-test ターゲットは、テストファイルを生成するだけです。これを node 経由で実行することができます。

```
$ shadow-cljs compile test
# または
$ shadow-cljs release test

# 手動でテストを実行する場合、:autorun で自動的に実行されます。
$ node out/node-tests.js

# コンパイルとテストの結合
$ shadow-cljs compile test && node out/node-tests.js
```

成功すると node プロセスの終了コードは 0 に設定され、失敗すると 1 に設定されます。:autorun を使用している場合は、node プロセスの終了コードは返されません。

11.2. ブラウザにおけるテスト

このターゲットは、(ファイル名のパターンマッチに基づいて) テストを含む名前空間を集めて、テストランナーを起動するためのものです。

このターゲットには cljs.test テストを自動的にスキャンして実行するビルトインのランナーが含まれています。

関連する設定オプションは以下の通りです。

:target :browser-test

:test-dir ファイルを出力するフォルダです。以下を参照してください。

:ns-regexp (オプション) プロジェクトファイルの名前空間にマッチする正規表現です。これはファイルをスキャンするだけで、jar ファイルはスキャンしません。 ジャーのスキャンは行いません。デフォルトは"-test$"です。 :runner-ns: (オプション) start、stop、init関数を含むことができる名前空間です。デフォルトは shadow.test.browser です。

通常の :devtools オプションはサポートされていますので、通常はファイルを提供するための http サーバを作成します。 一般的には、次のような設定が必要です。

```
{...
 :builds {:test     {:target    :browser-test
                     :test-dir  "resources/public/js/test"
                     :ns-regexp "-spec$"
                     :runner-ns tests.client-test-main
                     :devtools  {:http-port          8021
                                 :http-root          "resources/public/js/test"}}
```

test ディレクトリには、index.html と js フォルダがあることを覚えておいてください。

カスタムの :runner-ns を指定する場合は、以下のようになります。

```
(ns tests.client-test-main)

(defn start []
  ... run the tests...)

(defn stop [done]
  ; テストは非同期にすることができます。
  ; ランナーが実際に終了したことがわかるように、
  ; done を呼ばなければなりません。
  (done))

(defn ^:export init []
  (start))
```

これには init, start, stop というメソッドがあります。 init は起動時に一度だけ呼び出され、 stop はコードがリロードされる前に呼び出され、 start はすべてのコードがリロードされた後に呼び出されます。

TIP :runner-ns はオプションで、デフォルトを使用するには省略してください。

11.2.1. :test-dir に生成された出力

出力には、 test-dir フォルダにある 2 つの主要なアーティファクトが含まれます。

- index.html - index.html ファイルがまだ存在していない場合に限ります。デフォルトでは、生成されたファイルはテストをロードして、:runner-ns で init を実行します。上書きされないカスタムバージョンを編集または追加することができます。
- js/test.js - Javascript のテストです。テストは常にこの名前になります。モジュールのエントリーは自動生成されます。

11.3. 継続的インテグレーションのためにテストを Karma にターゲットする

CLJS のテストをある種の CI サーバー上でブラウザに対して実行したい場合、コマンドラインからテストを実行してステータスコードを返すことができる必要があります。

Karma はよく知られていてサポートされているテストランナーで、これを実行することができます。

また、`shadow-cljs` にはターゲットが含まれており、テストの周りに適切なラッパーを追加することができるので、それによりテストが動作するようになります。

11.3.1. Karma のインストール

詳しい説明は website をご覧ください。通常、`package.json` には以下のようなものが必要になります。

```
{
  "name": "CITests",
  "version": "1.0.0",
  "description": "Testing",
  ...
  "devDependencies": {
    "karma": "^2.0.0",
    "karma-chrome-launcher": "^2.2.0",
    "karma-cljs-test": "^0.1.0",
    ...
  },
  "author": "",
  "license": "MIT"
}
```

つまり、Karma 、ブラウザランチャー、そして cljs-test の統合が必要なのです。

11.3.2. Build

ビルドのオプションは次のとおりです。

`:target` `:karma`

`:output-to` `:karma` : js ファイルのパス/ファイル名です。

`:ns-regexp` (オプション) テストの名前空間にマッチさせる正規表現、デフォルトは "-test$" です。

つまり、次のようになります。

```
{...
 :builds
 {:ci
  {:target :karma
   :output-to  "target/ci.js"
   :ns-regexp  "-spec$"}}}
```

また、 karma.conf.js も必要です。

```
module.exports = function (config) {
    config.set({
        browsers: ['ChromeHeadless'],
        // 出力ファイルが存在するディレクトリ
        basePath: 'target',
        // ファイル自体
        files: ['ci.js'],
        frameworks: ['cljs-test'],
        plugins: ['karma-cljs-test', 'karma-chrome-launcher'],
        colors: true,
        logLevel: config.LOG_INFO,
        client: {
            args: ["shadow.test.karma.init"],
            singleRun: true
        }
    })
};
```

すると、以下のようにテストを実行することができます（ツールのグローバル実行ファイルがインストールされていることが前提です）。

```
$ shadow-cljs compile ci
$ karma start --single-run
12 01 2018 01:19:24.222:INFO [karma]: Karma v2.0.0 server started at
http://0.0.0.0:9876/
12 01 2018 01:19:24.224:INFO [launcher]: Launching browser ChromeHeadless with
unlimited concurrency
12 01 2018 01:19:24.231:INFO [launcher]: Starting browser ChromeHeadless
12 01 2018 01:19:24.478:INFO [HeadlessChrome 0.0.0 (Mac OS X 10.12.6)]: Connected on
socket TcfrjxVKmx7xN6enAAAA with id 85554456
LOG: 'Testing boo.sample-spec'
HeadlessChrome 0.0.0 (Mac OS X 10.12.6): Executed 1 of 1 SUCCESS (0.007 secs / 0.002
secs)
```

Chapter 12. JavaScript との統合

12.1. npm

npm は、JavaScript のデファクトスタンダードのパッケージマネージャとなっています。ほとんどすべての JS ライブラリはそこで見つけることができ、shadow-cljs はそれらのパッケージにアクセスするためのシームレスな統合を提供します。

12.1.1. npm パッケージの使用

大半の npm パッケージには実際のコードの使用法に関する説明も含まれています。古い CommonJS のスタイルでは `require` の呼び出しがあるだけです。

```
var react = require("react");
```

```
(ns my.app
  (:require ["react" :as react]))
```

require を呼び出すときに使われた "string "パラメータは全てそのまま `:require` に送られます。 `as` のエイリアスは任意です。これにより他の CLJS の名前空間と同じようにコードを使うことができます。

```
(react/createElement "div" nil "hello world")
```

`shadow-cljs` では。常に `ns` 形式と、あなたが提供した `:as` のエイリアスを使用してください。 また、`:refer` や `:rename` を使用することもできます。これは、`:foreign-libs`/CLJSJS が行う、名前空間にものをインクルードしても、コードの中ではグローバルな `js/Thing` を使用するというものとは異なります。

パッケージの中には、1 つの関数をエクスポートするものもあります。

`(:require ["thing" :as thing])` と書くと `(thing)` で直接呼び出すことができます。

最近では、いくつかのパッケージが ES6 の `import` 文を例に挙げています。これらもほぼ 1 対1で翻訳されていますが、デフォルトのエクスポートに関するわずかな違いがあります。

以下の表は、比較対照に使用することができます。

IMPORTANT	この表は、消費されるコードが実際の ES6+コードとしてパッケージされている場合にのみ適用されます。コードが CommonJS としてパッケージされている場合は、`:default` は適用されないかもしれません。詳細は以下のセクションを参照してください。

Table 1. ES6 Import から CLJS Require へ

ES6 Import	CLJS Require
`import defaultExport from "module-name";`	`(:require ["module-name" :default defaultExport])`
`import * as name from "module-name";`	`(:require ["module-name" :as name])`
`import { export } from "module-name";`	`(:require ["module-name" :refer (export)])`
`import { export as alias } from "module-name";`	`(:require ["module-name" :rename {export alias}])`
`import { export1 , export2 } from "module-name";`	`(:require ["module-name" :refer (export1 export2)])`
`import { export1 , export2 as alias2 , [···] } from "module-name";`	`(:require ["module-name" :refer (export1) :rename {export2 alias2}])`
`import defaultExport, { export [ , [···] ] } from "module-name";`	`(:require ["module-name" :refer (export) :default defaultExport])`
`import defaultExport, * as name from "module-name";`	`(:require ["module-name" :as name :default defaultExport])`
`import "module-name";`	`(:require ["module-name"])`

以前は、実際には必要のない多くのコードを含むバンドルコードを使用していたことに注意してください。しかし、今回はより良い状況になりました。 ライブラリの中には、必要な部分だけをパッケージ化したものもあり、最終的なビルドに含まれるコードの量を大幅に減らすことができます。

```
// react-virtualized
から名前付きでエクスポートすることで、任意のコンポーネントをインポートすることができます。
import { Column, Table } from 'react-virtualized'

// いくつかの react-virtualized コンポーネントしか使用していない場合、
// またはアプリケーションのバンドルサイズが大きくなることを懸念している場合、
// 必要なコンポーネントだけを直接インポートすることができます。
import AutoSizer from 'react-virtualized/dist/commonjs/AutoSizer'
import List from 'react-virtualized/dist/commonjs/List'
```

```
(ns my-ns
  ;; 全て
  (:require ["react-virtualized" :refer (Column Table)])
  ;; もしくは一つずつ
  (:require ["react-virtualized/dist/commonjs/AutoSizer" :default virtual-auto-sizer]
            ["react-virtualized/dist/commonjs/List" :default virtual-list]))
```

:default Exports について

現在、`:default` オプションは `shadow-cljs` でのみ利用可能です。

https://dev.clojure.org/jira/browse/CLJS-2376 で標準化されることを期待します。

標準的な CLJS との互換性を保ちたい場合は、いつでも `:as alias` を使用して、 `alias/default` を呼び出すことができます。

デフォルトのエクスポートは ECMAScript モジュールで新たに追加されたもので、CommonJS のコードには存在しません。時々、 `import Foo from "something"` という例を見かけることがありますが、実際には CommonJS のコードです。このような場合、 `(:require ["something" :default Foo])` は動作せず、`(:require ["something" :as Foo])` を代わりに使用する必要があります。

もし、`:require` が正しく機能していないようであれば、REPL で見てみることをお勧めします。

```
$ shadow-cljs browser-repl (or node-repl)
...
[1:1]~cljs.user=> (require '["react-tooltip" :as x])
nil
[1:1]~cljs.user=> x
#object[e]
[1:1]~cljs.user=> (goog/typeOf x)
"function"
[1:1]~cljs.user=> (js/console.dir x)
nil
```

任意の JS オブジェクトを表示することは必ずしも便利ではないので（上の例のように）、代わりに `(js/console.dir x)` を使用すると、ブラウザのコンソールでより便利な表現を得ることができます。

また、 `goog/typeOf` が役に立つ場合もあります。上の例では `"function"` を示しているので、`:default` を使ってもうまくいきません。`:default` は基本的に `x/default` のシンタックスシュガーに過ぎないからです。

12.1.2. パッケージプロバイダ

`shadow-cljs` は、 `npm` パッケージをビルドに含めるためのいくつかの異なる方法をサポートしています。これらの方法は `:js-options :js-provider` 設定で設定できます。各 `:target` は通常、ビルドに適したものを設定しますが、ほとんどの場合、この設定を変更する必要はありません。

現在、サポートされている JS プロバイダーは 3 つです。

`:require`

JS の `require("thing")` 関数呼び出しに直接マッピングします。実行時にネイティブで `require` を解決することができるため、すべての `node.js` ターゲットのデフォルトとなります。インクルードされた JS は何も処理されません。

`:shadow`

JS を `node_modules` 経由で解決し、参照されている各ファイルの最小バージョンをビルドに含めます。これは `:browser` ターゲットのデフォルトです。`node_modules` のソースは `:advanced` のコンパイルを通

りません。

`:closure`

`shadow` と同様に解決されますが、Closure Compiler CommonJS/ES6 rewrite facilities を通じて、含まれる全てのファイルを処理しようとします。これらのファイルは `:advanced` のコンパイルでも処理されます。

`:external`

これは、他の JS ビルドツールでも処理できるようになっており、実際に JS の依存関係を提供します。発行されたインデックス・ファイルには、CLJS の出力が JS の依存関係にアクセスできるようにするための、ちょっとしたグルーコードが含まれています。外部インデックスファイルの出力は、CLJS 出力の前に読み込まれなければなりません。

`:shadow` vs `:closure`

理想的には、主要な JS プロバイダとして `:closure` を使用したいところです。しかし実際には、`npm` 経由で入手できる多くのコードは、`:advanced` のコンパイルによる積極的な最適化とは互換性がありません。これらのコードは、まったくコンパイルできないか、実行時に特定が非常に困難な微妙なバグを露呈します。

`shadow` は、`:simple` を介してコードを処理するだけの、その場しのぎのソリューションのようなもので、適度に最適化されたコードを取得しつつ、より信頼性の高いサポートを実現しています。その出力は、`webpack` のような他のツールが生成するものと同等(あるいはそれ以上)であることが多いです。

Closure のサポートがより確実なものになるまでは、`:shadow` が `:browser` ビルドの推奨 JS プロバイダとなります。

Example `:browser` のビルドで `:closure` を使用するための設定

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :js-options {:js-provider :closure}
   }}}
```

12.1.3. CommonJS vs ESM

最近では、多くの `npm` パッケージが複数のビルドバリアント(Build Variant)をユーザに提供しています。`shadow-cljs` はデフォルトで `package.json` の `main` か `browser` キーの下にリンクされているバリアントを選択します。多くの場合 CommonJS のコードを指しています。

最近のパッケージの中には `module` エントリを提供しているものもありますが、これは通常 ECMAScript のコード (つまりモダンな JS) を指しています。CommonJS と ESM の間の相互運用は難しいので、`shadow-cljs` のデフォルトは CommonJS を使用するようになっていますが、ESM を使用することが有益な場合もあります。

これが機能するかどうかは、使用しているパッケージに大きく依存します。JS オプションの :entry-keys を使って、module エントリを優先するように shadow-cljs を設定することができます。これは package.json に含まれる文字列キーのベクトルを受け取り、順に試行されます。デフォルトでは "["browser" "main" "module"] となっています。

Example :browser のビルドで :closure を使用するための設定

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :js-options {:entry-keys ["module" "browser" "main"]} ;; まずは "module"を試す
   }}}
```

これを切り替える際には、必ず十分なテストを行い、build report の出力を比較してサイズの違いを確認してください。結果は良い意味でも悪い意味でも大きく変わる可能性があります。

12.1.4. パッケージを解決する

デフォルトでは shadow-cljs はすべての (:require ["thing" :as x]) の要求を npm の規則に従って解決します。つまり、 <project>/node_modules/thing/package.json を見て、そこからコードを追っていきます。この動作をカスタマイズするために、 shadow-cljs は :resolve 設定オプションを公開しており、これによって物事がどのように解決されるかをオーバーライドすることができます。

CDN の利用

CDN経由ですでに React がページに含まれているとします。JS/React を再び使い始めることもできますが、私たちは正当な理由でそれをやめました。その代わりに、 (:require ["react" :as react]) を使い続けることができますが、"react "がどのように解決されるかを設定することができます。

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :js-options
   {:resolve {"react" {:target :global
                       :global "React"}}}}

  :server
  {:target :node-script
   ...}}}
```

:app のビルドでは、グローバルな React インスタンスが使用され、:server のビルドでは、引き続き "react " npm パッケージが使用されます。これを動作させるためにコードを変更する必要はありません。

require をリダイレクトする

ビルドに応じて、どの `npm` パッケージが実際に使用されるかをもっとコントロールしたい場合があります。コードを変更することなく、ビルド設定から特定の require をリダイレクトすることができます。これは、そのようなパッケージを使用しているソースにアクセスできない場合や、あるビルドのためだけに変更したい場合に便利です。

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :js-options
   {:resolve {"react" {:target :npm
                       :require "preact-compat"}}}
```

また、ファイルを使って依存関係を上書きすることもできます。パスはプロジェクトルートからの相対パスです。

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :js-options
   {:resolve {"react" {:target :file
                       :file   "src/main/override-react.js"}}}
```

制限事項

`shadow-js` と `:closure` は `:resolve` を完全に制御することができ問題なく動作しますが、`:js-provider :require` はより限定的です。最初の require にのみ影響を与えることができ、それ以降は標準の `require` が制御します。つまり、パッケージが内部で `require` するものに影響を与えることはできません。したがって、`require` を直接使用するターゲット（例：`:node-script`）と一緒に使用することはお勧めできません。

react を *preact* にリダイレクトする例

```
{...
 :builds
 {:app
  {:target :node-script
   ...
   :js-options
   {:resolve {"react" {:target :npm
                       :require "preact-compat"}}}
```

Example react-table の利用

```
(ns my.app
  (:require
    ["react-table" :as rt]))
```

すべての `"react"` require が置きかえられるため、上記の例はブラウザで問題なく動作します。内部的には `"react-table"` がもつ `"react"` require を含みます。しかし `:js-provider :require` においては、`require("react-table")` が emit され、`node` が解決法をコントロールします。つまり、私たちが設定した `"preact"` ではなく、標準の `"react"` に解決されるということです。

12.1.5. 代替モジュールのディレクトリ

デフォルトで `shadow-cljs` は JS パッケージを解決する際に `<プロジェクトディレクトリ>/node_modules` のみを見ます。これは、`:js-options` の `:js-package-dirs` オプションで設定できます。これは、グローバルまたはビルドごとに適用することができます。

相対パスは、プロジェクトのルートディレクトリを基準に解決されます。パスは左から右に向かって試行され、最初にマッチしたパッケージが使用されます。

shadow-cljs.edn のグローバル設定

```
{...
 :js-options {:js-package-dirs ["node_modules" "../node_modules"]}
 ...}
```

単一のビルドに適用される設定

```
{...
 :builds
 {:app
  {...
   :js-options {:js-package-dirs ["node_modules" "../node_modules"]}}}}
```

12.2. .js ファイルへの対応

この機能は実験的なものであるため、自分の責任で使用してください。現在は shadow-cljs でのみサポートされており、これを使おうとすると他の CLJS ツールに怒られます。この機能は CLJS のコアでは拒否されましたが、私は便利だと思いますし、さらなる議論なしに破棄されるべきではなかったと思います。CLJS には別の実装もありますが shadow-cljs ではサポートされていません。私はこの実装がある種の面で不足していると感じたので、別の解決策を選びました。両方のアプローチの長所と短所を議論するのは楽しいことです。

私たちは npm パッケージがどのように使用されるかを取り上げました。すでに JavaScript のコードを中心に進めている場合、すべてを ClojureScript で書き直したくはないでしょう。

shadow-cljs は、JavaScript と ClojureScript の間の完全な相互運用性を提供します。つまり、JS から CLJS を使うことができ、CLJS から JS 使うこともできます。

この機能を確実に動作させるためには、いくつかの規則に従わなければなりませんが、すでに実行していることもあるでしょう。

12.2.1. JS を require する

先ほど、 npm パッケージに名前でアクセスする方法を説明しましたが、クラスパス上では、 .js ファイルにフルパスまたは現在の名前空間からの相対パスでアクセスします。

クラスパスから *JS* を読み込む

```
(ns demo.app
  (:require
    ["/some-library/components/foo" :as foo]
    ["./bar" :as bar :refer (myComponent)]))
```

TIP	文字列が必要な場合、拡張子 .js が自動的に追加されますが、必要に応じて拡張子を指定することができます。ただし、現在は .js のみサポートしています。

some-library/components/foo のような絶対的な要求は、ローカルファイルシステムからファイルをロードしようとする node とは異なり、コンパイラがクラスパス上で some-library/components/foo.js を探すことを意味します。同じクラスパスのルールが適用されるので、ファイルは :source-paths にあるか、使用しているサードパーティの .jar ライブラリにあるかもしれません。

相対的な require は、まず現在の名前空間を見て、その名前からの相対パスを解決します。上の例では demo/app.cljs のなかで ./bar は demo/bar.js に解決されるので、 (:require ["/demo/bar"]) と同じになります。

IMPORTANT	ファイルは物理的に同じディレクトリにあってはいけません。ファイルの検索はクラスパス上で行われます。これは、相対的な要求が常に物理的なファイルに解決されることを期待する node とは異なります。

Example パスが分かれているファイル構造

```
.
├──── package.json
├──── shadow-cljs.edn
└──── src
    └──── main
        └──── demo
            └──── app.cljs
    └──── js
        └──── demo
            └──── bar.js
```

12.2.2. 言語サポート

IMPORTANT	クラスパスには、コンパイラが前処理をしなくても利用できる JavaScript だけが含まれていることが期待されています。 `npm` にもよく同様の規約があります。

Closure Compiler は、クラスパス上で見つかった全ての JavaScript を、その言語設定である `ECMASCRIPT_NEXT` を使って処理します。この設定が正確に何を意味するのかはよくわかっていませんが、ほとんどのブラウザではまだサポートされていないかもしれない次世代の JavaScript コードを表しています。 ES6 は非常によくサポートされており、ほとんどの ES8 の機能もサポートされています。標準的な CLJS と同様に、これは必要に応じてポリフィルを用いて ES5 にコンパイルされます。

Closure Compiler は常にアップデートされていますので、新しい機能は徐々に利用可能になっていきます。ただ、最新の最先端のプレビュー機能がすぐに使えるとは思わないでください。最近追加された `async/await` のような機能は、すでに十分に機能しています。

JS は、 `import` と `export` を使って、ES モジュール構文を使って書く必要があります。 JS ファイルは、他の JS ファイルをインクルードしたり、CLJS のコードを直接参照することができます。また、`npm` パッケージに直接アクセスすることもできますが、ひとつ注意点があります。

```
// 標準的な JS の require
import Foo, { something } from "./other.js";

// npm の require
import React from "react";

// CLJS または Closure Library JS の require
import cljs from "goog:cljs.core";

export function inc(num) {
  return cljs.inc(1);
}
```

IMPORTANT	Closure Compiler の厳密なチェックにより、CLJS や npm のコードを必要とする場合、 `import * as X from "npm";` の構文を使用することはできません。他の JS ファイルを必要とする場合には問題なく使用できます。

12.2.3. JavaScript の方言

一般的な JavaScript の方言（JSX、CoffeeScript など）には、Closure Compiler では直接解析できないものが多いため、クラスパスに置く前に前処理を行う必要があります。 babel は JavaScript の世界でよく使われているので、ここでは例として `.jsx` ファイルを処理するために `babel` を使用します。

Example shadow-cljs.edn の設定

```
{:source-paths
 ["src/main"
  "src/gen"]
 ...}
```

Example ファイル構成

```
.
├── package.json
├── shadow-cljs.edn
└── src
    └── main
        └── demo
            └── app.cljs
    └── js
        ├── .babelrc
        └── demo
            └── bar.jsx
```

IMPORTANT	`src/js` が `:source-paths` に追加されていないことに注目してください。これはクラスパスに含まれないことを意味します。

src/js/demo/bar.jsx

```
import React from "react";

function myComponent() {
  return <h1>JSX!</h1>;
}

export { myComponent };
```

babel を実行してファイルを変換し、設定された `src/gen` ディレクトリに書き込んでいます。どのディレクトリを使うかはあなた次第です。私は生成されたファイルには `src/gen` を使いたい。

```
$ babel src/js --out-dir src/gen
# 開発中は次のようにする
$ babel src/js --out-dir src/gen --watch
```

babel 自体は src/js/.babelrc を通して設定されます。公式 example for JSX を参照してください。

```
{
  "plugins": ["transform-react-jsx"]
}
```

いったん babel が src/gen/demo/bar.js を書けば、それは ClojureScript を通して使用できるようになり、ClojureScript のソースと同じようにホットロードされることもあります。

12.2.4. JS から CLJS へのアクセス

JS ソースは、名前空間を goog: というプレフィックスでインポートすることで、すべての ClojureScript（および Closure Library）に直接アクセスすることができます。このプレフィックスは、コンパイラが名前空間をデフォルトの ES6 エクスポートとして公開するように書き換えます。

```
import cljs, { keyword } from "goog:cljs.core";

// JS で {:foo "hello world"} を作る。
cljs.array_map(keyword("foo"), "hello world");
```

TIP	現在は goog: という接頭辞は ES6 形式のファイルでのみ動作します。require("goog:cljs.core") は動作しません。

12.3. cljsjs.* の移行について

> CLJSJS は Javascript のライブラリをパッケージにすることで、 ClojureScript から利用できるようにしようという試みです。

shadow-cljs は npm packages に直接アクセスできるので、再パッケージ化された CLJSJS packages に頼る必要はありません。

しかし、多くの CLJS ライブラリはまだ CLJSJS パッケージを使用しており、 shadow-cljs はそれらをもうサポートしていないので、それらは壊れてしまいます。しかし、これらの cljsjs 名前空間を模倣するのはとても簡単です。なぜなら、それらはほとんど npm パッケージから構築されているからです。それには、 cljsjs.thing を元の npm パッケージにマップして、期待されるグローバル変数を公開する shim ファイルが必要です。

React の場合は、 src/cljsjs/react.cljs のようなファイルが必要です。

```
(ns cljsjs.react
  (:require ["react" :as react]
            ["create-react-class" :as crc]))
```

```
(js/goog.object.set react "createClass" crc)
(js/goog.exportSymbol "React" react)
```

これは誰もが手動で行うのは面倒なので、私は shadow-cljsjs というライブラリを作りました。 ライブラリを作成しました。すべてのパッケージが含まれているわけではありませんが、これからも追加していきますので、ご協力をお願いします。

注：shadow-cljsjs ライブラリは、shim ファイルを提供するだけです。実際のパッケージは、自分で 自分で実際のパッケージを npm install する必要があります。

12.3.1. CLJSJS を使いませんか?

CLJSJS のパッケージは、基本的に npm からパッケージを取り出し、 .jar に入れて clojars で再公開するだけです。おまけに Externs もバンドルされています。コンパイラはこれらのファイルに対して何もせず、生成された出力の先頭に追加するだけです。

これは、 npm に直接アクセスできないときには非常に便利でしたが、すべてのパッケージが他のパッケージと簡単に結合できるわけではないので、ある種の問題があります。あるパッケージは react に依存しているかもしれませんが、 npm を通してこれを表現するのではなく、 それらは 自分自身の react をバンドルします。注意しないと、 2 つの異なる react バージョンをビルドに含めることになり、非常に紛らわしいエラーが発生したり、少なくともビルドサイズが大幅に大きくなったりする可能性があります。

また、すべての npm パッケージが CLJSJS で利用できるわけではなく、パッケージのバージョンを同期させるには手作業が必要なため、パッケージが古くなってしまうこともあります。

shadow-cljs は、コード内の競合を避けるために、CLJSJS を全くサポートしません。あるライブラリが古い cljsjs.react を使おうとする一方で、別のライブラリはより新しい (:require ["react"]) を直接使うかもしれません。そうすると、再びページ上に 2 つのバージョンの react が存在することになります。

そのため、唯一欠けているのはバンドルされた Externs です。多くの場合、externs inferenceが改善されたため、これらは必要ありません。また、これらの Externs はサードパーティのツールを使って生成されていることが多いので、いずれにしても完全には正確ではありません。

結論: npm を直接使う。:infer-externs auto を使う。

Chapter 13. プロダクションコードの生成 — 全ターゲット

開発モードでは、常に各々の名前空間に対して個別のファイルを出力するため、それらを分離してホットロードすることができます。実際のサーバにコードをデプロイする準備ができたら、Closure Compiler を実行して、それぞれの module に対して単一の最小化された結果を生成します。

デフォルトでは、リリースモードの出力ファイルは、開発モードの出力ファイルをそのまま置き換えるようになっています。 これらのファイルを HTML にインクルードする方法に違いはありません。また ブラウザターゲットでのキャッシュ特性を改善するために filename hashing を使用することができます。

最小化された出力の生成

```
$ shadow-cljs release build-id
```

13.1. リリースの構成

通常、ビルド用のリリースバージョンを作成するために、追加の設定をする必要はありません。デフォルトの設定は必要なものをすべて含んでおり、追加の設定が必要になるのは、デフォルトを上書きしたい場合だけです。

各 `:target` には、各プラットフォームに最適化された優れたデフォルトがすでに用意されているので、心配する必要はありません。

13.1.1. 最適化

最適化レベルは、設定の `:compiler-options` セクションで選択できます。

IMPORTANT	`:target` ではすでに適切なレベルに設定されているので、通常は `:optimizations` を設定する必要はありません。
IMPORTANT	`:optimizations` は `release` コマンドを使ったときにのみ適用されます。開発用のビルドは Closure Compiler によって最適化されることはありません。開発用のビルドは常に `:none` に設定されます。

```
{...
 :build
   {:build-id
     {...
      :compiler-options {:optimizations :simple}}}}
```

利用可能な最適化レベルの詳細については Closure コンパイラのドキュメント を参照してください。

13.1.2. release 固有 vs 開発環境

リリースビルドを実行する際に、ビルド内の設定値を別々にしたい場合は、ビルドセクションに `:dev` や `:release` セクションを含めることで、設定を上書きすることができます。

Example shadow-cljs.edn ビルド設定

```
{:source-paths ["src"]
 :dependencies []
 :builds
 {:app
  {:target :browser
   :output-dir "public/js"
   :asset-path "/js"
   :modules {:base {:entries [my.app.core]}}

   ;; 開発向けの設定
   :dev {:compiler-options {:devcards true}}

   ;; 本番環境向けの設定
   :release {:compiler-options {:optimizations :simple}}}}}
```

13.2. extern

Closure Compiler `:advanced` のコンパイルによってビルドを完全に最適化したいので Externs を扱う必要があります。 Externs は、`:advanced` のコンパイルの際に含まれないコードの断片を表します。 `advanced` はプログラム全体の最適化を行いますが、どうしても含まれないコードもあるため、Externs はコンパイラにそのコードを知らせます。 Externs がないと、コンパイラは含めるべきでないコードの名前を変えたり、削除したりすることがあります。

通常、すべての JS Dependencies は外部の扱い(foreign) であり、`:advanced` ではパスされないため、Externs が必要となります。

> **TIP** extern が必要なのは `:advanced` モードのみで、`:simple` モードでは必要ありません。

13.2.1. Externs の推論

Externs を扱うために、 `shadow-cljs` コンパイラは、強化された externs の推論を提供します。

Example Config

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :compiler-options {:infer-externs :auto}
   }}}}
```

`auto` では、コンパイラはコンパイル時にあなたのファイルだけを対象に追加のチェックを行います。また、ライブラリコードに含まれる externs の問題については警告しません。 `all` では、すべてのファイルに対して有効になりますが、多くの警告を受ける可能性があることに注意してください。

有効にすると、コンパイラが JS と CLJS のどちらのコードを使用しているかを判断できない場合に警告が表示されます。

Example Code

```
(defn wrap-baz [x]
  (.baz x))
```

Example Warning

```
------ WARNING #1 ----------------------------------------------------------------
 File: ~/project/src/demo/thing.cljs:23:3
--------------------------------------------------------------------------------
  21 |
  22 | (defn wrap-baz [x]
  23 |   (.baz x))
---------^----------------------------------------------------------------------
 Cannot infer target type in expression (. x baz)
--------------------------------------------------------------------------------
```

`advanced` では、コンパイラが `.baz` をより短いものにリネームしますが、Externs はこれがリネームしてはいけない外部プロパティであることをコンパイラに知らせます。

`shadow-cljs` は、ネイティブな相互運用を行うオブジェクトに typehint を追加すると、適切な externs を生成することができます。

extern の生成を助ける型ヒント

```
(defn wrap-baz [x]
  (.baz ^js x))
```

typehint の `^js` を指定すると、コンパイラが適切な externs を生成するようになり、警告が消えます。これで、このプロパティは名前を変えても大丈夫です。

複数の *Interop* コール

```
(defn wrap-baz [x]
  (.foo ^js x)
  (.baz ^js x))
```

Interop　コールを個別にアノテーションするのは面倒なので、変数のバインディング自体をアノテーションすることができます。この変数は、この変数のスコープ全体で使用されます。両方の呼び出しのextern はまだ生成されます。

x を直接アノテーションする

```
(defn wrap-baz [^js x]
  (.foo x)
  (.baz x))
```

IMPORTANT

すべてを `^js` でアノテーションしないでください。時には、CLJS や ClosureJS のオブジェクトでインターロップを行うことがあります。これらはextern を必要としません。 CLJS オブジェクトを扱うことが確実な場合は、 `^clj` のヒントを使うことをお勧めします。 間違って `^js` を使用しても世界の終わりではありませんが、変数の名前を変更できるのに変更されない場合、いくつかの最適化に影響を与える可能性があります。

ヒントは不要、*extern* は自動的に推測される

```
(js/Some.Thing.coolFunction)
```

また、`:require` バインディングの呼び出しも自動的に推論されます。

`.as` と `:refer` のバインディングにヒントは必要ありません。

```
(ns my.app
  (:require ["react" :as react :refer (createElement)]))

(react/createElement "div" nil "hello world")
(createElement "div" nil "hello world")
```

13.2.2. 手動のextern

一部のライブラリでは、extern を個別の `.js` ファイルとして提供しています。 Externs は、`:externs` コンパイラオプションを使って、ビルドに組み込むことができます。

手動の*extern* 設定

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :compiler-options {:externs ["path/to/externs.js" ...]}
   }}}
```

TIP

コンパイラは、プロジェクトルートに相対するファイルを最初に探します。ファイルが見つからない場合は、クラスパスからの読み込みも試みます。

13.2.3. 簡易なextern

Externs を手動で書くことは困難ですが、shadow-cljs にはより便利な書き方があります。shadow-cljs check <your-build> とすることで、不足している Externs を素速く追加することができます。

まず、 externs/<your-build>.txt を作成します。ビルド :app は externs/app.txt となります。このファイルの各行は、名前を変更してはいけない JS のプロパティを1つの単語で指定します。グローバル変数の前には、 global: をつけます。

Example externs/app.txt

```
# これはコメントです。
foo
bar
global:SomeGlobalVariable
```

この例では、コンパイラは something.foo(), something.bar() という名前の変更を止めます。

13.3. コード・ストリッピング

Closure コンパイラは、不要なコードを名前で削除することをサポートしています。これにより、通常のデッドコード除去では除去できない、あるいは除去したくないコードを除去することができます。これは、実際に気になるコードが削除される可能性があるため非常に危険ですが、多くの開発者専用コードを簡単に削除することができます。これは 4 つのオプションに分類されていて、そのうち ClojureScript に関連するのはほぼ :strip-type-prefixes だけですが、他にも有用なオプションがあるかもしれません。

Example cljs.pprint のすべての使用を削除

```
{...
 :builds
 {:app
  {:target :browser
   ...
   :compiler-options {:strip-type-prefixes #{"cljs.pprint"}
   }}}
```

これらのオプションは、それぞれ文字列のセットとして指定されます。ここで指定されている名前はすべて JS の名前なので、特定の CLJS の名前はマングされなければならないことに注意してください。my-lib.core は my_lib.core となります。

:strip-types

deftype/defrecord の宣言や使用を削除できるようにします。#{"my.ns.FooBar} は (defrecord FooBar []) を削除します。

:strip-type-prefixes

与えられた Prefix のいずれかで始まるものをすべて削除します。 CLJS の名前空間全体を削除することができます。

`:strip-name-prefixes`

与えられたプレフィックスで始まるすべてのプロパティを削除します。プレフィックスでプロパティを削除することができます。`#{"log"}` は `this.logX` や `(defn log-me [...])` を削除します。

`:strip-name-suffixes`

サフィックスでプロパティを削除することができます。`#{"log"}` は `this.myLog` または `(defn my-log [...])` を削除します。

> **DANGER:** これらのオプションには注意が必要です。これらのオプションはビルド全体に適用され、実際に必要なコードが削除される可能性があります。また、自分が書いたものではないライブラリのコードを誤って削除してしまう可能性もあります。このオプションを使用する前に、必ず他のオプションを検討してください。

13.4. ビルドレポート

`shadow-cljs` では、 `release` のビルドに対して、含まれているソースの詳細な内訳と、それぞれが全体のサイズにどれだけ貢献しているかを含む、詳細なレポートを生成することができます。

レポートのサンプルは こちらをご覧ください。

```
$ npx shadow-cljs run shadow.cljs.build-report <build-id> <path/to/output.html>
# 例
$ npx shadow-cljs run shadow.cljs.build-report app report.html
```

上記の例では、`:app` をビルドする際に、プロジェクトディレクトリに `report.html` を生成します。

> **TIP** 生成される `report.html` は完全に自己完結型で、必要なデータ/js/css をすべて含んでいます。他の外部ソースは必要ありません。

Chapter 14. エディタの統合

14.1. Cursive

Cursive は現在、 `shadow-cljs.edn` による依存関係の解決をサポートしていません。 `shadow-cljs pom` を実行して `pom.xml` を生成し、それを IntelliJ.NET を使ってインポートすることができます。

```
$ shadow-cljs pom
```

続いて、Cursive の File → New → Project from Existing Sources で、プロジェクトディレクトリに生成された `pom.xml` を選択します。

IMPORTANT	このためには、Build Tools→Maven プラグインを有効にする必要があります。デフォルトでは有効になっていない場合があります。

あるいは、ダミーの `project.clj` を作成したり、完全なLeiningen integrationを使用することもできます。

```
(defproject your/project "0.0.0"
  :dependencies
  [[thheller/shadow-cljs "X.Y.Z"]]

  :source-paths
  ["src"])
```

IntelliJ が提供する Terminal の中で `npx shadow-cljs server` を実行し、 `Clojure REPL → Remote` Run Configuration を使って、提供されるnREPL serverに接続することができます。Cursive Clojure REPL → Remote で "Use port from nREPL file "オプションを選択するか、お好みで固定の nREPL ポートを設定するだけです。

最初に接続したときの Cursive REPL は、常に CLJ REPL として起動することに注意してください。これを CLJS に切り替えるには、 `(shadow/repl :your-build-id)` を呼び出します。これにより、自動的に Cursive オプションも切り替わります。CLJ REPL に戻るには、`:cljs/quit` と入力します。

14.2. Emacs / CIDER

このセクションは、CIDER のバージョン 0.20.0 以上を対象に書かれています。お使いの Emacs にこのバージョンの `cider` パッケージが入っていることを確認してください。インストールの詳細については、リンク: CIDER documentation を参照してください。

14.2.1. ClojureScript REPL の起動

```
M-x cider-jack-in-cljs
```

CIDER は、ClojureScript REPL の種類を尋ねてきます。

```
Select ClojureScript REPL type:
```

ClojureScript REPL タイプを選択します。

shadow と入力します。

```
Select shadow-cljs build:
```

ビルドターゲットの名前を入力してください（例：`app`）。

Emacs は `shadow-cljs` サーバへの新しい nREPL 接続を開き、ClojureScript REPL 環境にブートストラップを行います。

```
shadow.user> To quit, type: :cljs/quit
[:selected :app]
cljs.repl>
```

これで、ClojureScript を評価したり、 `cider-find-var` で変数の定義にジャンプしたりすることができるようになります。

例えば、ブラウザにアラートを表示する場合などです。

```
cljs.repl> (js/alert "Jurassic Park!")
```

14.2.2. dir-local によるスタートアップの簡素化

プロジェクトのルートに `.dir-locals.el` ファイルを作成することで、スタートアップの流れを簡単にすることができます。

```
((nil . ((cider-default-cljs-repl . shadow)
     (cider-shadow-default-options . "<自身のビルド名>"))))
```

14.3. Proto REPL (Atom)

Proto REPL は主に Clojure の開発を目的としており、大半の機能は ClojureScript では動作しません。単純な評価のためにそれを使用することは可能です。

使えるようにするには、いくつかの設定が必要です。

(1) `:source-paths` の中に、 `user.clj` を作成します。

```
(ns user)

(defn reset [])
```

Proto REPL は接続時にこの fn を呼び出すので、このファイルは `user/reset` fn を定義していなければなりません。

もし `user/reset` が見つからなければ、`tools.namespace` が呼び出され、実行中の `shadow-cljs` サーバが破壊されます。私たちはこれを望んでいません。ここで何かできるかもしれませんが、CLJS のために何かする必要はありません。

(2) ~/.shadow-cljs/config.edn または `shadow-cljs.edn` の `:dependencies` に `[proto-repl "0.3.1"]` を追加してください。

(3) 固定のnREPL ポートを設定する

(4) `shadow-cljs server` または `shadow-cljs watch your-build` を起動します。

(5) Atom コマンド `Proto Repl: Remote Nrepl Connection` で `localhost` と設定したポートに接続します。

(6) `(shadow.cljs.devtools.api/watch :your-build)` を評価する (4 で `server` を使用した場合)

(7) `(shadow.cljs.devtools.api/nrepl-select :your-build)` を評価します。 REPL 接続は CLJS モードになり、評価するものはすべて JS で評価されることになります。 Clojure モードに戻るには、`:repl/quit` とすることができます。もし `[:no-worker :browser]` が出たら、まず `watch` を起動する必要があります。

(8) CLJS を評価する前に、クライアントを接続する必要があります（例：`:browser` App を構築する際のブラウザ）。

(9) `(js/alert "foo")` のように、いくつかの JS を評価します。接続されている JS ランタイムがありませんと表示される場合は、クライアントが正しく接続されていません。それ以外の場合は、ブラウザにアラートが表示されます。

14.4. Chlorine (Atom)

Chlorine は、Atom を Socket REPL に接続しますが、同時に名前空間のリフレッシュも試みます。まず、Chlorine パッケージのコンフィグを開き、コンフィグ `Should we use clojure.tools.namespace to refresh` が `simple` に設定されているかどうかを確認します。そうしないと、実行中の `shadow-cljs` サーバが破壊されてしまいます。

設定が正しいことを確認したら、shadow アプリを起動できます（`app` は任意のビルドに置き換えてください）。

```
$ shadow-cljs watch app
```

あとは、atom コマンドの `Chlorine: Connect Clojure Socket Repl`. これで Clojure のコードを評価する REPL が接続されます。次に、 `Chlorine: Connect Clojure Socket Repl` を実行してください。 `Connect Embeded` を実行すると、ClojureScript の REPL も接続されます。

これで、 Chlorine: Evaluate… コマンドを使って、任意の Clojure または ClojureScript REPL を評価することができます。 .clj ファイルは Clojure として、 cljc ファイルは ClojureScript として評価されます。

14.5. Calva (VS Code)

今のところ browser ターゲットでしかテストしていません。おそらく他のターゲットでも動作します。

14.5.1. 依存関係

VS Code が必要で Calvaエクステンションをインストールする必要があります。

Calva は nREPL と cider-nrepl のミドルウェアを使用しているので、この依存関係を ~/.shadow-cljs/config.edn または shadow-cljs.edn に含める必要があります。

```
[cider/cider-nrepl "0.21.0"]
```

shadow-cljs は、この依存関係を確認すると、必要な cider-nrepl ミドルウェアを注入します。

14.5.2. Calva と REPL の接続

その後、shadow 製アプリケーションを起動します。app 以外のビルドでも使用できます。

```
$ shadow-cljs watch app
```

アプリがブラウザに読み込まれ、アプリを起動したターミナルに JS runime connected と表示されると、Calva はその REPL に接続できるようになります。VS Code でプロジェクトを開くと、Calva はデフォルトで自動接続を試み、shadow-cljs.edn から読み込んだビルドのリストを表示します。正しいもの(この例では :app)を選択すると、Calva の Clojure と Clojurescript のサポートが有効になります。

アプリ起動時にすでに VS Code でプロジェクトを開いている場合は、コマンドを実行してください。

```
Calva: Connect to a Running REPL Server in the Project
```

14.5.3. 特徴

利用可能んs機能の一部をご紹介します。

Intellisense など

- hover で定義を見る
- 自動補完のヘルプを表示する
- 定義ファイルへの移動（Mac では cmd-click、Windows や Linux では ctrl-click ）。

ファイル、フォーム、セレクションの評価

- ファイルを評価する: `ctrl+alt+c enter` (ファイルを開くときに自動的に行われます。)
- インライン評価をする: `ctrl+alt+c e`.
- エディタ内で評価して置きかえる: `ctrl+alt+c r`.
- 評価結果を整形して表示する: `ctrl+alt+c p`.
- 評価のために統合ターミナルの repl にフォームを送る: `ctrl+alt+c alt+e`.

テストの実行

- 名前空間のテストを実行する: `ctrl+alt+c t`
- すべてのテストを実行する: `ctrl+alt+c shift+t` (これまでの大規模プロジェクトでは非常に不便でした)
- 以前に失敗したテストを再実行する: `ctrl+alt+c ctrl+t` (テストの失敗はエクスプローラーやエディタでマークされ、簡単にアクセスできるように Problem タブにリストアップされます)

ターミナルの **REPL**

- ターミナルレプリの名前空間を、現在開いているファイルの名前空間に切り替える: `ctrl+alt+c n`
- 現在のファイルを読み込んで、名前空間を切り替える: `ctrl+alt+c alt+n`

Cljc のファイル群

- Clojure と Clojurescript の repl を `ctrl+alt+c ctrl+alt+t` で切り替えます。または、ステータスバーの緑の `cljc/clj` ボタンをクリックすることでも可能です。これにより、どの repl がエディタをバックアップしているか、どの端末の repl にアクセスしているかの両方が決定されます（上記参照）。

14.6. Fireplace.vim (Vim/Neovim)

Fireplace.vim は nREPL クライアントとして動作することで、Clojure REPL の統合を提供する Vim/Neovim プラグインです。Shadow-CLJS と組み合わせることで、ClojureScript REPL の統合も可能になります。

このガイドでは、公式 Shadow-CLJS Quick Start ガイドで作成されたアプリを例にしているため、アプリの `shadow-cljs.edn` にあるいくつかの設定項目を参照しています。とはいえ、これらの設定項目はかなり一般的なものなので、ちょっとした修正で他のアプリにも適用できるはずです。

14.6.1. 依存関係

Fireplace.vim を、Vim/Neovim でプラグインをインストールするお好みの方法でインストールします。

nREPL クライアントとして Fireplace.vim は CIDER-nREPL (これは一般的なエディタに依存しない REPL 操作を提供する nREPL ミドルウェアです)に依存しているため、この依存関係を~/.shadow-cljs/config.edn または `shadow-cljs.edn` (次のサブセクションで示すように) にこの依存関係を含める必要があります。Shadow-CLJS は、この依存関係を確認すると、必要な CIDER-nREPL ミドルウェアを注入します。

14.6.2. アプリの準備

公式ガイドの Shadow-CLJS Quick Start に沿ってサンプルアプリを作成し、shadow-cljs.edn を以下のように修正します。

```
;; shadow-cljs の設定
{:source-paths
 ["src/dev"
  "src/main"
  "src/test"]

 ;; 追加 - Fireplace.vim で必要な CIDER-nREPL ミドルウェア
 :dependencies
 [[cider/cider-nrepl "0.22.4"]]

 ;; 追加 - Fireplace.vim が接続する REPL サーバーのポート(例：3333)
 :nrepl
 {:port 3333}

 ;; 追加 - アプリを提供する開発時の HTTP サーバーのポート(例：8080)
 :dev-http
 {8080 "public"}

 :builds
 {:frontend  ; 注：これは以下の各所で参照されているビルド ID です。
  {:target :browser
   :modules {:main {:init-fn acme.frontend.app/init}}}}}
```

これが完了したら、アプリを起動します（ shadow-cljs.edn で指定されている Shadow-CLJS のビルド ID、frontend に注意してください）。

```
npx shadow-cljs watch frontend
```

http://localhost:8080/ 、ブラウザでアプリを開きます。この手順を行わないと、Vim/Neovim 内から REPL サーバに接続しようとすると Fireplace.vim から次のようなエラーメッセージが表示されます。

```
No application has connected to the REPL server.
Make sure your JS environment has loaded your compiled ClojureScript code.
```

どのアプリケーションも REPL サーバに接続していません。 JS環境がコンパイルした ClojureScript コードを読み込んでいることを確認してください。

14.6.3. Fireplace.vim と REPL サーバの接続

Vim/Neovim で ClojureScript のソースファイルを開き、以下のコマンドを実行して Fireplace.vim を REPL サーバに接続します（REPL サーバのポートは、 shadow-cljs.edn で指定されている 3333 であることに注意してください）。

```
:Connect 3333
=>
Connected to nrepl://localhost:3333/
Scope connection to: ~/code/clojurescript/acme-app (ENTER)
```

これにより、（ClojureScript ではなく）Clojure の REPL セッションが作成されます。次のコマンドを実行して、セッションに ClojureScript のサポートを追加します（`shadow-CLJS.edn` で指定されている Shadow-CLJS のビルド ID、`frontend` に注意してください)。

```
:CljEval (shadow/repl :frontend)
=>
To quit, type: :cljs/quit
[:selected :frontend]
Press ENTER or type command to continue
```

これで、REPL サーバに対して Fireplace.vim コマンドが実行できるようになります。実行可能なコマンドの全リストについては Fireplace.vim のドキュメントを参照してください。

Chapter 15. トラブルシューティング

15.1. 起動時のエラー

時々 shadow-cljs が正しく起動しないことがあります。このエラーはしばしば非常に分かりづらく、特定するのが困難です。最も一般的な原因は、いくつかの重要な依存関係が衝突していることです。単に :dependencies の管理に shadow-cljs.edn を使用している場合には、これらの種類のエラーから守るためにいくつかの特別なチェックを行いますが、 deps.edn や project.clj を使用している場合には、これらの保護を行うことができないため、これらのツールを使用している場合には、これらのエラーがより頻繁に発生します。

一般的に、注意すべき重要な依存関係は以下の通りです。

各 shadow-cljs バージョンは、ある特定のバージョンの組み合わせでしかテストされておらず、最良の互換性を得るためには、そのバージョンセットに固執することが推奨されます。異なるバージョンを使用している場合でも動作するかもしれませんが、何らかのおかしな問題が発生した場合は、まず依存するバージョンの修正を検討してください。

各バージョンに必要な依存関係は、clojars で確認できます。

- https://clojars.org/thheller/shadow-cljs

これらの問題を診断する方法はツールによって異なりますので、詳細は該当するセクションを参照してください。

一般的に、確実にしたいのであれば、選択した shadow-cljs のバージョンと一緒に、一致する依存関係のバージョンを直接宣言すればよいのですが、それは shadow-cljs をアップグレードするたびに、それらのバージョンも更新しなければならないことを意味します。不要な依存関係を正しく特定することは、より多くの作業を必要とするかもしれませんが、将来のアップグレードを容易にするでしょう。

shadow-cljs は、上に挙げたすべての依存関係について常に最新のバージョンである可能性が高いため、古い依存関係を維持する必要がある場合は、shadow-cljs のバージョンも古いものにする必要があるかもしれません。

そのため、 org.clojure/clojurescript が通常提供しているバージョンに依存していると、問題が発生する可能性があります。そのため、 org.clojure/clojurescript が通常供給するバージョンに依存している場合、問題が発生する可能性があります。

もし、あなたの生活をより楽にしたいのであれば、できれば shadow-cljs.edn を使って依存関係を管理してください。このような問題が発生する可能性は非常に低いですし、少なくともあなたに直接警告してくれます。

正しいバージョンを入手したにもかかわらず、問題が解決しない場合は Github Issueを開き、依存関係のリストを含む問題の詳細をご報告ください。

15.1.1. deps.edn / tools.deps

shadow-cljs.edn の :deps キーで依存関係を管理するために deps.edn を使用している場合、さらなる診断のために clj ツールを直接使用することが推奨されます。まず最初に、あなたが shadow-cljs.edn を通し

てどのエイリアスを適用しているかを確認する必要があります。つまり、もしあなたが `:deps {:aliases [:dev :cljs]}` を設定しているのであれば、さらにコマンドを実行する際に、これらのエイリアスを指定する必要があります。

まず最初に、 `deps.edn` で直接宣言されているすべての依存関係が、期待されるバージョンであることを確認する必要があります。推移的な依存関係によって、問題のあるバージョンが含まれていることがあります。すべての依存関係をリストアップするには、以下の方法があります。

アクティブな依存関係をすべてリストアップ

```
$ clj -A:dev:cljs -Stree
```

これにより、すべての依存関係がリストアップされます。これを追いかけるのはちょっと大変ですが、上述の依存関係のために正しいバージョンを取得しているかどうかを確認する必要があります。

詳細は tools.depsの公式ドキュメントをご参照ください。

15.1.2. project.clj / Leiningen

`project.clj` を使って依存関係を管理している場合、 `lein` を直接使って問題を診断するときには、`shadow-cljs.edn` から設定した `:lein` プロファイルを指定する必要があります。例えば、`:lein {:profiles "+cljs"}` とすると、すべてのコマンドに対して `lein with-profiles +cljs` が必要になります。

Example deps のリスト

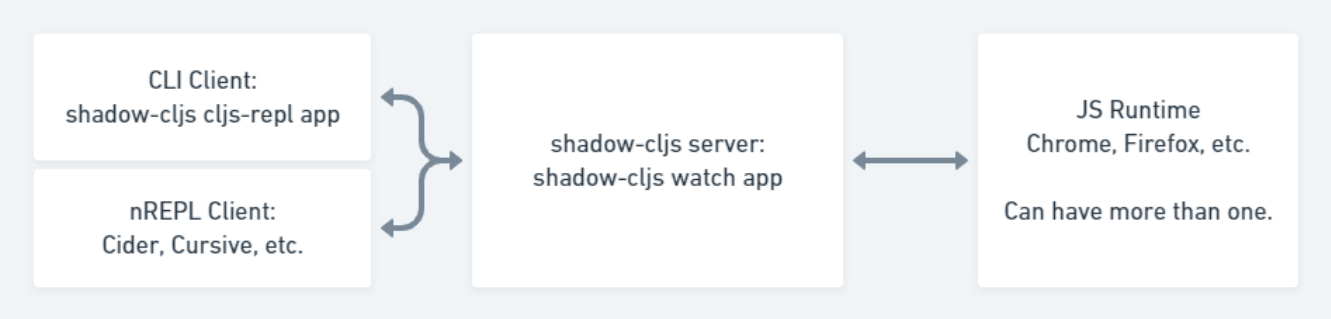
```
# no profile
$ lein deps :tree

# with profile
$ lein with-profiles +cljs deps :tree
```

これは通常、一番上に現在のコンフリクトをすべてリストアップし、一番下に依存関係のツリーで提案を提供します。提案は必ずしも完全には正確ではありませんので、誤解を招かないように、また `thheller/shadow-cljs` アーティファクトに除外項目を追加しないようにしてください。

詳しくは Leiningen のドキュメントをご参照ください。

15.2. REPL

CLJS REPL を動作させることは、時として厄介であり、すべての可動部品が非常に複雑であるため、多くの問題が発生する可能性があります。このガイドでは、人々が遭遇する最も一般的な問題と、その解決方法を取り上げたいと思います。



15.2.1. CLJS REPL の分析

Clojure の REPL は、その名が示す通りのことを行います。 1 つのフォームを読み、それを評価し、結果を print し、それを再び行うためにループします。

しかし ClojureScript では、コンパイルは JVM上で行われますが、結果は JavaScript のランタイムで評価されるので少し複雑です。普通の REPL の経験をエミュレートするためには、さらにいくつかのステップが必要になります。`shadow-cljs` では、通常の CLJS とは実装が少し異なりますが、基本的な原理は同じです。

まず、REPL クライアントが必要です。これは単に CLI (例: `shadow-cljs cljs-repl app`) または `nREPL` を介して接続されたエディタかもしれません。クライアントは常に `shadow-cljs` サーバと直接会話し、残りの部分は `shadow-cljs` が処理します。クライアント側からは、まだ通常の REPL のように見えますが、バックグラウンドではさらにいくつかのステップが行われています。

1) Read : すべては与えられた InputStream から単数の CLJS フォームを読み取ることから始まります。これは `stdin` から直接ブロック化して読み取るか、 `nREPL` の場合は文字列から読み取ることになります。文字のストリームは実際のデータストラクチャーに変えられ、 `"(+ 1 2)"` (文字列)は `(+ 1 2)` (リスト)になります。

2）Compile : そのフォームは、 `shadow-cljs` JVM 側でコンパイルされ、一連の命令に変換されます。

3）Transfer Out : これらの命令は、接続された JavaScript ランタイムに転送されます。これはブラウザであったり、 `node` プロセスであったりします。

4) 評価 : 接続されたランタイムは、受け取った命令を `eval` します。

5) Print : JS ランタイムでは、 `eval` の結果が文字列として出力されます.

6) Transfer Back : Print された結果は、 `shadow-cljs` の JVM 側に転送されます。

7) Reply : JVM 側は受信した結果を最初の呼び出し元に転送し、結果は適切な OutputStream にプリントされます（または nREPL メッセージとして送信されます）。

8）Loop : 1）から繰り返します。

15.2.2. JavaScript のランタイム

shadow-cljs の JVM 側では、関連するすべての REPL コマンドを処理する、与えられたビルドのための watch の実行が必要になります。これは専用のスレッドを使用し、開発中に起こりうるすべてのイベントを管理します（例：REPL の入力、ファイルの変更など）。

しかし、コンパイルされた JS コードは、JS ランタイム(例えば、ブラウザや node プロセス)によって読み込まれなければならず、その JS ランタイムは実行中の shadow-cljs プロセスに接続しなければなりません。ほとんどの :target 設定では、必要なコードがデフォルトで追加されており、自動的に接続されるはずです。その接続がどのように行われるかはランタイムに依存しますが、通常は WebSocket を使用して、実行中の shadow-cljs HTTP server に接続します。

接続が完了すると、REPL を使用できるようになります。 JS ランタイムをリロード（例：ブラウザのページを手動でリロード）すると、ランタイムのすべての REPL の状態が消去されますが、コンパイラ側の状態の一部は、 watch が再起動されるまで残りますのでご注意ください。

複数の JS ランタイムが watch プロセスに接続する可能性があります。 shadow-cljs はデフォルトで、最初に接続した JS ランタイムを eval の対象として選択します。ある :browser ビルドを複数のブラウザで開いた場合、最初のブラウザだけが eval コードに使用されます。また、開発中に iOS と Android で隣り合わせに :react-native アプリを開くこともできます。評価できるのは 1 つのランタイムだけで、そのランタイムが切断されると、接続された時間に基づいて次のランタイムが引き継ぐことになります。

15.2.3. JS ランタイムの欠落

Missing JS runtime

> No application has connected to the REPL server. Make sure your JS environment has loaded your compiled ClojureScript code.

(どのアプリケーションも REPL サーバに接続していません。JS環境がコンパイルした ClojureScript コードを読み込んでいることを確認してください。)

このエラーメッセージは、JS ランタイム (例: ブラウザ) が shadow-cljs サーバに接続していないことを意味します。 REPL クライアントは正常に shadow-cljs サーバに接続しましたが、上で説明したように、実際に eval するためにはまだ JS ランタイムが必要です。

通常の shadow-cljs ビルドは JS ランタイムを管理していないので、実行する責任があります。

:target :browser

:target :browser ビルドの場合、 watch プロセスは、設定された :output-dir (デフォルトは public/js) に与えられたコードをコンパイルします。生成された .js は、ブラウザで読み込む必要があります。ロードされると、ブラウザコンソールには WebSocket connected というメッセージが表示されます。何らかのカスタム HTTP サーバを使用している場合や、過剰なファイアウォールが接続をブロックしている場合は、いくつかの追加設定を行う必要があるかもしれません（例：:devtools-url経由）。目標は、primary HTTP serverに接続できるようにすることです。

:target :node-script, :node-library

これらのターゲットは、 `.js` ファイルを作成し、 `node` プロセスで実行することを意図しています。しかし、様々なオプションがあるため、それらを自分で実行する必要があります。例えば、`:node-script` を `node the-script.js` 経由で実行すると、起動時に `shadow-cljs` サーバへの接続を試みます。起動時には `WebSocket connected` というメッセージが表示されるはずです。出力はコンパイルされたマシン上でのみ動作するように設計されていますので、 `watch` の出力を他のマシンにコピーしないでください。

:target :react-native

生成された `<:output-dir>/index.js` ファイルを `react-native` アプリに追加し、実際のデバイスやエミュレータで読み込む必要があります。また、起動時には `shadow-cljs` サーバへの接続を試みます。ログの出力は `react-native log-android|log-ios` で確認することができ、アプリが実行されると `WebSocket connected` というメッセージが表示されるはずです。もし起動時に WebSocket 関連のエラーが表示される場合は、代わりに shadow-cljs プロセスへの接続に失敗した可能性があります。これは、IP 検出が誤った IP を選択した場合に発生する可能性があります。どの IP が使用されたかは `shadow-cljs watch app --verbose` で確認できますし、 `shadow-cljs watch app --config-merge '{:local-ip "1.2.3.4"}'` で上書きすることもできます。

Chapter 16. ライブラリの publish

ClojureScript のライブラリは、Clojure と同様に `maven` リポジトリに公開されます。最も一般的には Clojars に公開されますが、他のすべての標準的な maven リポジトリでも動作します。

しかし、ClojureScript ライブラリは、JAR(基本的には単なる ZIP 圧縮ファイル)で公開されたアンコンパイルされたソースファイルであるため、maven に公開できる一般的なツールであれば動作します。(例: `mvn`, `gradle`, `lein`, など). 公開するために余分なコンパイルやその他のステップは必要ありません。ClojureScript コンパイラや、そのための shadow-cljs は全く関係ありません。

16.1. Leiningen

ライブラリを公開するには様々なオプションがありますが、現在は Leiningen をお勧めしています。セットアップは非常に簡単で、設定は全く必要ありません。

IMPORTANT	これは、ライブラリ自体の開発時に Leiningen を使用しなければならないということではありません。公開時には Leiningen を使用し、それ以外は通常通り `shadow-cljs` を使用することが推奨されています。実際の `:dependencies` の定義をコピーする必要があるのは、公開した後だけです。しかし、開発に関連する依存関係を排除することを忘れないでください。

すでに、すべての主要なソースが `src/main` に配置されている推奨のプロジェクト構造を使用している場合、非常にシンプルな `project.clj` で公開することができます。

```
(defproject your.cool/library "1.0.0"
  :description "Does cool stuff"
  :url "https://the.inter.net/wherever"

  ;; これはオプションで、必要なものを追加したり、削除したりすることができます。
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}

  :dependencies
  ;; Clojure では常に provided を使用します。(Script)
  [[org.clojure/clojurescript "1.10.520" :scope "provided"]
   [some.other/library "1.0.0"]

  :source-paths
  ["src/main"])
```

これにより、必要な `pom.xml` が生成され、 `src/main` からのすべてのソースが、公開された `.jar` ファイルに格納されます。あとは `lein deploy clojars` を実行するだけで公開されます。この作業を初めて行う場合は、まず適切な認証を設定する必要があります。その設定方法については、公式の Leiningen および Clojars のドキュメントを参照してください。

16.1.1. JAR 署名の無効化

Leiningen はライブラリを公開する前に GPG で署名することをデフォルトとしています。これは良いデフォルトですが、設定が面倒であったり、実際に署名を検証する人があまりいないことを考えると、`project.clj` にシンプルな `:repositories` 設定を追加することで、このステップを無効にすることができます。

```
(defproject your.cool/library "1.0.0"
  ...
  :repositories
  {"clojars" {:url "https://clojars.org/repo"
              :sign-releases false}}
  ...)
```

16.1.2. JAR をクリーンに保つ

ライブラリのテストやその他の開発関連のコードを書く場合は、ライブラリと一緒に公開しないように、必ず `src/dev` や `src/test` に置いてください。

また、 `resources/*` に出力することも避けてください。Leiningen や他のツールが `.jar` にそれらのファイルを含める可能性があり、下流のユーザーに問題を引き起こす可能性があります。 `.jar` には実際のソースファイルのみを入れ、コンパイルされたコードは一切入れないようにしてください。

IMPORTANT	lein jar を実行し、 `jar -tvf target/library-1.0.0.jar` で最終的に含まれるファイルを検査することで、すべてがクリーンであることを確認できますし、そうすべきです。

16.2. JS の依存関係を宣言する

現在、 `shadow-cljs` のみが `npm` とのクリーンな自動インターロップストーリーを持っていることに注意してください。これは、他のツールを使用しているあなたのライブラリのユーザーにとって問題となるかもしれません。CLJSJS のフォールバックを提供したり、 `webpack` 関連のワークフローのための特別なドキュメントを公開したりすることを検討するとよいでしょう。

プロジェクト（例：`src/main/deps.cljs`）の中に、`:npm-deps` 付きの `deps.cljs` を含めることで、 `npm` の依存関係を直接宣言することができます。

Example src/main/deps.cljs

```
{:npm-deps {"the-thing" "1.0.0"}}
```

ここでは、追加の `:foreign-libs` 定義を提供することもできます。これらは `shadow-cljs` には影響しませんが、他のツールには役立つかもしれません。

詳しくは https://clojurescript.org/reference/packaging-foreign-deps をご覧ください。

Chapter 17. 上手くいかない場合はどうすればいいか

JS の世界は今でも急速に進化しており、誰もが同じ方法でコードを書いたり配布したりしているわけではないので、`shadow-cljs` が自動的に回避できないものもあります。

これらは通常、カスタムの `:resolve` 設定で解決できますが、バグや見落としもあるかもしれません。

この章の説明で解決できない場合は、次のような場所で質問してみてください。 `#shadow-cljs` Slack チャンネルで聞いてみてください。

Chapter 18. Hacking

18.1. ライブラリのパッチ適用

`shadow-cljs` コンパイラは、あなたのソースパス上にあるものが最初にコンパイルされるようにし、JARからのファイルを優先します。つまり、ライブラリからソースファイルをコピーしてパッチを当て、それを自分のソースディレクトリにインクルードすることができるのです。

これは、そのプロジェクトをクローンして、そのセットアップやビルドなどを理解しなくても、修正をテストするための便利な方法です(`shadow-cljs` 自体もテストできます!)。 プロジェクトをクローンして、そのセットアップやビルドなどを理解することなく、修正をテストする便利な方法です。