

# Harmoware-VIS のチュートリアル

現代のJavaScriptをおさらいし、Reactライブラリの利用から  
時空間情報可視化ライブラリである Harmoware-VIS の利用、  
開発に必要な知識を身に付けることができます

# 目次

1. モダン JavaScript 開発の現状
2. モダン JavaScript 開発の環境構築
3. Reactを使ったフロントエンド開発
4. Reduxを使ったアプリケーション構築
5. deck.glの紹介
6. Harmoware-VISとは
7. Harmoware-VISを使ってみる

# 1. モダン JavaScript開発の現状

# 簡単なJavaScriptの歴史・変異

モダンなJavaScriptを習得する上でJavaScriptが辿ってきた進化を押さえておく事は非常に重要です。

なぜ今のようなエコシステムになっているのかをJavaScriptの歴史から学んでいきます。

# JavaScriptの誕生

- 1995年 Netscape社から「LiveScript」という名前で登場（後に当時勢いのあったJavaにのり、JavaScriptと改名）
- 1996年 MicroSoft社がJavascriptを搭載しようとしたが、ライセンスの問題でJavaScriptに似た「JScript」を開発
- 標準化などはされておらず、各ベンダーがおのれのに独自実装を行なっていた

# 標準化へ

- 互換性がない問題を解決するため、1997年Netscape社が標準化機関ECMAに依頼
- ECMA-262初版ができる（ECMAScript）
- 順調に見えた標準化だが、ECMA-262 第4版で各ベンダーの意見相違により破棄
- しばらく ECMA-262 第3.1版 で進化が止まる

# この時代のJavaScript（2000年代前半）

- 速度も遅くセキュリティも甘かった為、ブラクラやウィルスが多く出た
- またECMAScriptを元にしたAction Scriptが入ったFlashの登場により、JavaScriptの評価はいまいちな状況に

# 救世主登場

2005年 GoogleがGoogle Mapを発表し世間を驚かせる



# リバイバル

- Google Mapの登場からJavaScriptが見直される流れとなる
- 2010年、jQueryの登場。簡単に実装でき多くのライブラリによりwebサイトではなくてはならないものに
- 2009年、ECMA-262 第5版（ES5）が発表
- ただ進んだように見えた標準化だが、この後6年間標準化が進まない。

# 新たな考え方の登場 - CommonJS

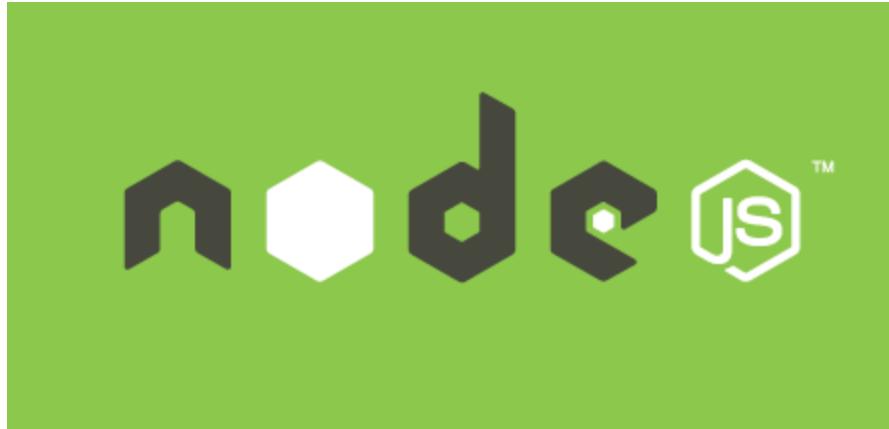
- ブラウザだけではなく、サーバーサイドで動させたい
- モジュールを使いたい (JavaScript最大の問題点)

このような要望から生まれてきたのが「CommonJS」というサーバーサイドで動く新しい仕様が誕生しました。

```
// ECMAScript5 (ES5) での実装
<script src="./lib.js"></script>
<script src="./lib2.js"></script>
<script src="./main.js"></script>
```

```
// これがやりたい
var lib = require '外部モジュール';
var lib2 = require '外部モジュール2';
```

# Node.js の誕生



CommonJSの仕様ができ、その実装版であるNode.jsが2009年に誕生し流行ります。多くのライブラリやコマンドラインツールが開発されました。

\*注意: 現状Node.jsはCommonJSの仕様にはしたがっておらず独自に実装を行なっている状況です

# npm (Node Package Manager)

Node.jsの誕生により、モジュール（パッケージ）を扱うことができるようになりました。またこれにより、パッケージを管理するツールの必要性がでてきました。

そのツールが「npm」です。

```
$ npm install jquery --save
```

# ECMAScript2015（ES6）の登場



Node.jsの登場を受け、ECMAScriptにもようやくモジュールシステムの仕様が入った「**ECMAScript2015（ES6）**」が2015年に誕生しました。他にも多くの変更行われ、このES6からが現在におけるECMAScriptのモダンなJavaScript環境といつていいくらいでしょ。

# ECMAScript2015 (ES6) の登場

またこのES6から、不定期だったバージョンアップではなく、毎年新しい仕様がリリースされるようになりました。

- ECMAScript2015 (ES6)
- ECMAScript2016 (ES7)
- ECMAScript2017 (ES8)

# 簡単にECMAScript2015の紹介

# let・constキーワードによる変数宣言

let -> 「再宣言」が不可

```
var hoge = 'a'; // OK
var hoge = 'b'; // OK

let hoge2 = 'a'; // OK
let hoge2 = 'b'; // NG
```

const -> 定数

```
let hoge3 = 'a'; // OK
hoge3 = 'b'; // OK

const hoge4 = 'c'; // OK
hoge4 = 'd'; // NG
```

## let,constのスコープ

let, constはifブロックなどのブロック内でスコープが閉じるようになります。

```
if (true) {  
  var a = 'hello';  
}  
  
console.log(a); // hello  
  
if (true) {  
  let b = 'hello';  
  const c = 'hello';  
}  
  
console.log(b); // ReferenceError  
console.log(c); // ReferenceError
```

# export・import モジュール機構

## 名前付きエクスポート・インポート

```
const hoge1 = 'aaa';

const hoge2 = () => {};

export { hoge1, hoge2 }; // 宣言済みの変数をエクスポート

export const hoge3 = 'bbb'; // そのまま変数をエクスポート
```

```
import * from './lib.js'; // ワイルドカードで全てインポート

import {hoge1, hoge2} from './lib.js'; // 指定してインポート

import {hoge1 as fuga1, hoge2 as fuga2} from './lib.js'; // エイリアス
```

# export・import モジュール機構

デフォルトエクスポート（インポートする際に、指定がない場合、defaultでエクスポートされたものがインポートされます）

```
export default () => {};
```

```
export default class {};
```

```
import hoge from './lib.js';
```

```
import Hoge as HogeClass './lib.js';
```

# class構文

```
class Hoge extends Fuga {  
  
    // コンストラクター  
    constructor(x, y) {  
        super(x, y); // 親コンストラクタ  
  
        this.x = x; // プロパティ  
        this.y = y; // プロパティ  
    }  
  
    method1() {  
  
    }  
  
    method2() {  
  
    }  
}  
  
const obj = new Hoge(1, 2);
```

# アロー関数

ES5

```
var hoge = function (arg1) {
    return arg1 + 1;
}
```

ES6

```
const hoge = (arg1) => {
    return arg1 + 1;
};

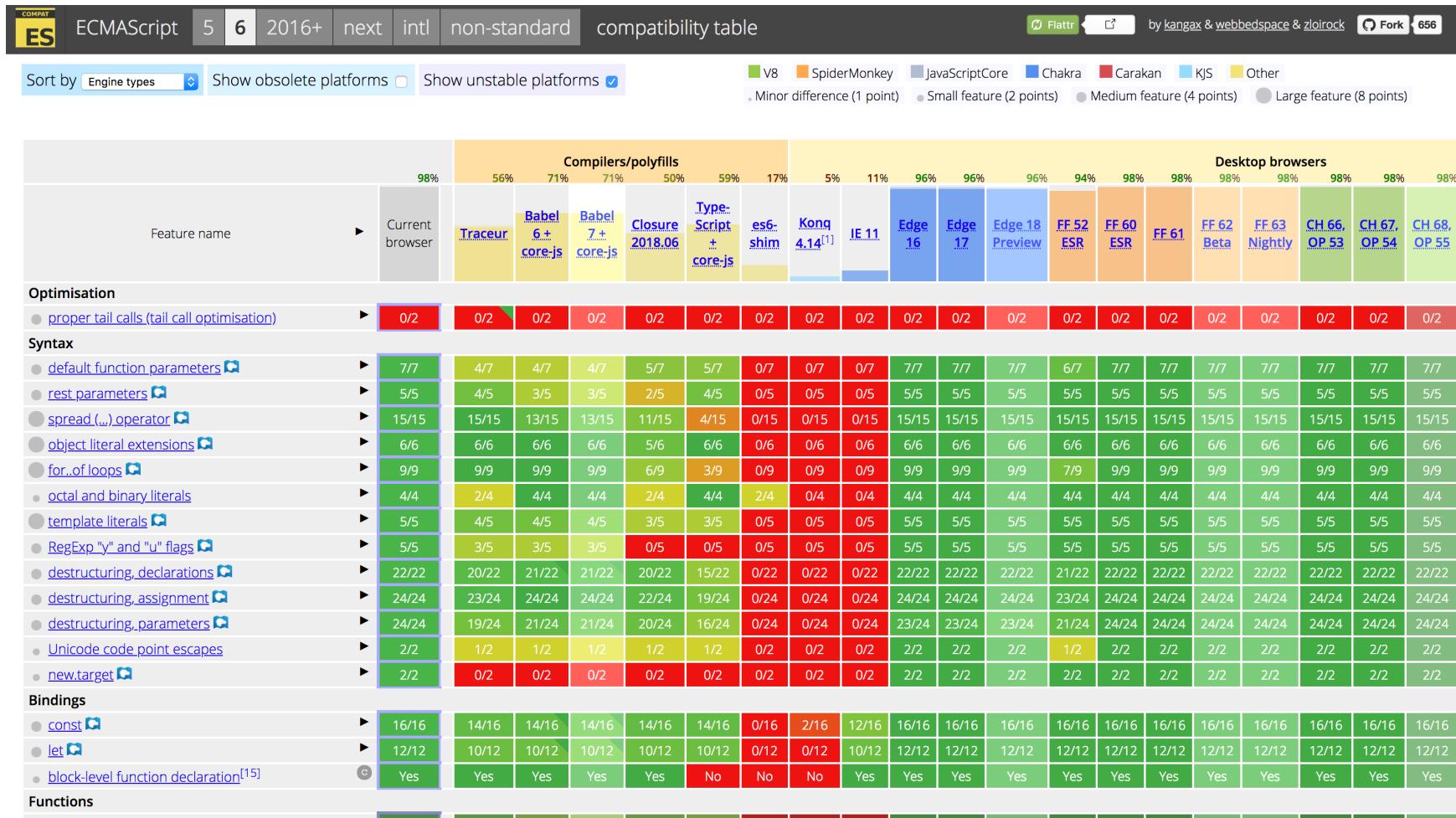
const hoge = arg1 => arg1 + 1; // 省略バージョン
```

# ブラウザという特殊な実行環境

ブラウザで動くJavaScriptの実行環境を実装するのは各ブラウザ（IE, Chrome, FireFox, Safariなど）毎のベンダー（Microsoft, Google, Mozilla, Appleなど）になります。また新しい仕様を動かすためには、サーバーサイドで動くアプリケーションと違い、各クライアント（一般ユーザ）がブラウザのバージョンアップを行う必要があります。

# 各ブラウザでの実装状況

<https://kangax.github.io/compat-table/es6/>



# ブラウザという特殊な実行環境による影響

- 仕様においつかず実装が遅れる（モダン環境のChromeでさえ、正式にモジュールが実装されたのは2017年）
- ベンダー間での実装方法の違い
- 古い環境でも動作させなければならない
- ビジネス上、新しい仕様はあるのに、実践では使えない

# トランスペイロールの登場 - Babel

このような特殊な環境でも最新仕様で開発するために生まれてきたのが「Babel」です。



トランスペイロールとは、あるプログラミング言語で書かれたコードを、別のプログラミング言語に変換することをいいます。BabelはES6で書かれたコードをES5に変換して出力します。

これにより、新しい仕様（ES6）で開発しても、各ベンダーのES6実装状況を気にせずに開発できるようになりました。

# Polyfill

前のスライドで「Babel」を導入すれば、ES6実装状況を気にせずに開発できるようになりました。とありましたが実は正しくありません。

あくまで「Babel」はES6の文法をES5に変換するためのもので、ES6で追加された新しいメソッドは、ブラウザで実装されていなければ動作しません。

そこで、新しいメソッドを使いたい場合は、ES5で書かれた同じ機能を提供するライブラリを導入する必要があります。

- es6-shim
- core-js

\*注意: あくまでES5で実現できる機能のみ（全てのES6メソッドに対応はしていない）

# バンドルツール

「Babel」「Polyfill」により、ES6での記述ができるようになったのですが、解決できない問題があります。それがモジュール機構の実現です。「トランスペイラ」「Polyfill」とも最終的には「ES5」の機能で実現しているのですが、モジュール機構はES5の文法では再現不可能なためです。

そこで登場するのが、バンドルツールと呼ばれるツールです。

- Browserify
- webpack
- Rollup

仕組みは省きますが、簡単にいうとモジュールによって別れた複数ファイルを1つのファイルにバンドルし、依存関係・実行順序などをうまく解決するツールです。

# AltJS - TypeScript

ここまでエコシステムを組み合わせれば、  
ECMAScript2015（ES6）以上のモダンな環境で動作させることは  
できます。しかしECMAScript自体、まだまだプログラム言語全体  
からみると成長途中の言語です。

そこでもう少し高度な文法で動作せたいという要望があり、  
MicroSoftが開発している「TypeScript」というECMAScriptのスー  
パーセットとして動作するJavaScriptに変わる言語も出てきました。  
こちらもトランスペイラツールを使いJavaScriptとしてブラウ  
ザ上で動作します。

- 静的型付
- 名前空間
- ジェネリック
- インターフェイス...

## Flow

TypeScriptと似たような静的型づけの手段としてFlowTypeを導入するという手段もあります。TypeScriptよりは実行速度が遅くなるが、導入や取り外しが楽なので、小規模な開発の場合だとこちらを採用するということもあるようです。

# TypeScript / Flow

例えば下記のような型チェックが行えます。

```
function concat(a: string, b: string) {
  return a + b;
}

concat(1, 2); //error
concat('A', 'B'); //works
```

# JSまとめ - JavaScriptが辿ってきた歴史

- もともとはブラウザで動く簡易な言語だった
- 普及するにつれ、他言語と同じように扱いたいという要望がでてくる
- JavaScriptは一言では言えない。（ECMAScript, CommonJS, Node.js, AltJS, TypeScript）
- 以上のような背景 & クライアントのバージョンアップ問題もあり、様々なツール（エコシステム）を利用して古い環境でも動くようなシステムとして進化してきた

# JSまとめ - モダン開発の環境・フロー

ブラウザという特殊な環境のため、現状モダンな開発をするためには、様々なツールを組み合わせて開発を行う必要がある

以下、ECMAScript2015 (ES6) 以上で開発する場合

- ECMAScript2015 (ES6) 以上でのコーディング
- BabelによりES6以上のコードをES5の文法に変換
- 開発するものに合わせ、Polyfillを導入
- これら必要なパッケージはnpmでパッケージ管理
- WebpackやBrowserifyによりバンドルしてモジュールを解決
- 生成されたJavaScriptをブラウザで読み込む

# JSまとめ

モダンな環境で開発するためには、様々なツールと連携しないといけなく、またそのツールや仕様なども毎年どんどん進化しています。日々情報をキャッチしモダンな環境を手に入れましょう。

環境を作るだけでハードルの高いモダンJavaScript環境ですが、昔に比べたら圧倒的に開発しやすくなったことは確かです。一つ一つのツールにどのような歴史・意味があるかを考えると複雑な環境も理解する役に立つかと思います。

## 2. モダン JavaScript 開発の環境構築

# Node.jsをインストール

モダンなJavaScript開発の環境構築にはNode.jsのインストールが必要不可欠です。まずはNode.jsを使えるようにしましょう。

## Node.jsをインストール

下記のサイトを利用することでインストールマネージャーを使って簡単にNode.jsの環境が整います。

<https://nodejs.org/en/download/>

## Node.jsのバージョン管理ツール

プロジェクトに応じて使用するNode.jsのバージョンが異なるケースもあります。またチーム間で使用するNode.jsのバージョンを合わせておかないと、Node.jsの実行結果が異なるケースも出てきてしまっています。そう行った時にはNode.jsのバージョン管理ツールを使うと良いでしょう。

## Node.jsのバージョン管理ツール

Macでは `nodebrew` や `ndenv`、Windowsでは `nodist` というツールを使ってNode.jsのバージョンコントロールができます。

以下では、Mac, Windows それぞれについて、バージョン管理ツールを解説します。

# Node.jsのバージョン管理ツール（Mac編）

今回は `ndenv` というバージョン管理ツールをインストールしてみましょう。

<https://github.com/riywo/ndenv>

Macで `ndenv` をインストールするには下記のコマンドをターミナル上に入力します。

```
$ git clone https://github.com/riywo/ndenv ~/.ndenv
$ echo 'export PATH="$HOME/.ndenv/bin:$PATH"' >> ~/.bash_profile
$ echo 'eval "$(ndenv init -)"' >> ~/.bash_profile
$ exec $SHELL -l
```

# Node.jsのバージョン管理ツール（Mac編）

`ndenv` インストール後は、下記のように `ndenv` コマンドが利用できるようになり、Node.jsの各バージョンのインストールや、プロジェクト内でのバージョン指定ができるようになります。

## Node.jsの特定バージョンインストール

```
$ ndenv install v9.11.2
```

## Node.jsのバージョン指定

```
$ ndenv local v9.11.2
```

## 現在のNodeのバージョンを確認

```
$ node -v  
v9.11.2
```

# Node.jsのバージョン管理ツール（Mac編）

```
$ nndenv local v9.11.2
```

また上記のコマンドを入力すると、`.node-version` というファイルが自動で生成され、このファイルがあることによって、自動で `node` のバージョンも `9.2.0` になります。チームでプロダクトを開発する時に `git` に `.node-version` を含めておけば、意識することなく、同じNode.jsのバージョンでプロダクト開発することができます。

## Node.jsのバージョン管理ツール（Mac編）

また、現在インストールされているNode.jsのバージョンを下記のコマンドで一覧表示できます。

```
$ nndenv versions  
v6.10.1  
v6.10.2  
v6.9.4  
v7.5.0  
* v9.11.2
```

\* が表示されているバージョンがデフォルトのNode.jsのバージョンとなります。

## Node.jsのバージョン管理ツール（Windows編）

Macでいう `ndenv` とよく似たNode.jsバージョン管理ツールとして、Windowsでは `nodist` というツールが使われます。 `nodist` は Macと同じ `.node-version` ファイルを参照するので、Macで開発していてもWindowsで開発していてもNode.jsのバージョンを合わせることができます。

# Node.jsのバージョン管理ツール（Windows編）

Windowsの場合、インストールは非常に楽で下記のリンクよりインストーラーを実行することで自動的に環境変数が設定され、`nodist` ヘパスが通ります。

<https://github.com/marcelklehr/nodist/releases>

インストールできたかどうか、下記のコマンドで確かめてみましょう。

```
C:\ nodist -v  
0.8.8
```

# Node.jsのバージョン管理ツール（Windows編）

Node.jsの特定バージョンインストール

```
C:\ nodist + 9.11.2
```

Node.jsのバージョン指定

```
C:\ nodist 9.11.2
```

```
C:\ node -v  
v9.11.2
```

# Node.jsのバージョン管理ツール（Windows編）

Node.jsのローカル環境でのバージョン指定  
(多くの場合は、こちらを指定したほうが良い)

```
C:\nodist env 9.11.2  
9.11.2
```

また、現在インストールされているNode.jsのバージョンを下記のコマンドで一覧表示できます。

```
C:\nodist list  
(x64)  
7.2.1  
9.5.0  
9.11.2 (global: 9.11.2)
```

## Node.jsのパッケージ管理ツール npm について

次は、Node.jsのパッケージ管理ツール `npm` の使い方をご紹介します。

`npm` は元々は `Node.js` 用のパッケージ管理ツールだったのですが、現在ではフロントエンド用のライブラリを管理するためのツールとしても使用されます。

最近のJavaScriptプロジェクトでは、`npm init` コマンドより `package.json` というファイルを作成するのが一般的です。

# Node.jsのパッケージ管理ツール npm について

```
$ npm init
```

実行するとプロジェクト名やバージョン、概要などの入力を求められますが、決まってなければ `enter` を入力することでスキップできます。

最終的には以下のようなJSONファイルが生成されます。

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

# Node.jsのパッケージ管理ツール npm について

この `package.json` は `npm` でインストールしたライブラリの情報が記述されます。

試しに下記のコマンドをターミナル上で入力してみましょう

```
$ npm install jquery --save
```

# Node.jsのパッケージ管理ツール npm について

すると、`package.json` にインストールした、`jquery` に関する情報が追記されました。この機能のおかげで使用しているライブラリのバージョンを自分や他の人が瞬時に把握することができます。

```
{  
  "name": "test",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "jquery": "^3.3.1"  
  }  
}
```

# Node.jsのパッケージ管理ツール npm について

また、`package.json` によく使うコマンドを登録することができます。

`package.json` の `scripts` という項目にコマンドを登録していきます。 `npm init` した段階では、デフォルトで、`test` というコマンドが書かれており、これは `npm run test` とターミナルに入力することで、実行できます。

このような `package.json` によるコマンドライン追加を `npm scripts` と呼びます。

## Node.jsのパッケージ管理ツール npm について

試しに、`http-server` というモジュールをインストールして簡易 Web サーバーを立ち上げてみましょう。

はじめに以下のコマンドを入力します。

```
$ npm install http-server --save-dev
```

# Node.jsのパッケージ管理ツール npm について

次に、`npm scripts` を追加します。`package.json`を編集し `serve` というコマンドを追加しましょう。`npm scripts` にコマンドを記述することにより、`./node_modules/.bin` の中に自動でパスが通るため、環境変数の設定などは不要です。

```
{  
  "name": "test",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1",  
    "serve": "http-server"  
  },  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "http-server": "^0.11.1"  
  }  
}
```

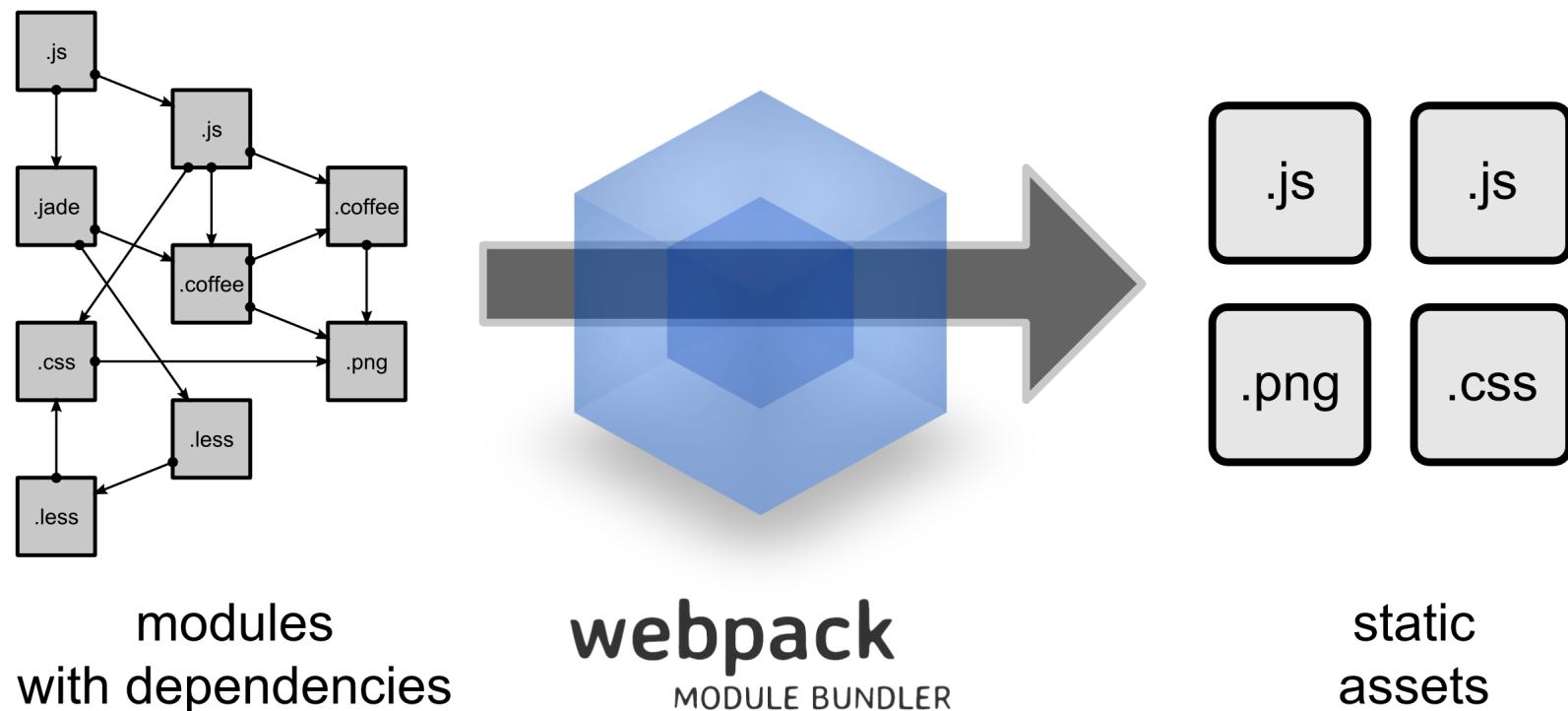
## Node.jsのパッケージ管理ツール npm について

`npm run serve` を実行すると下記のURLにて現在のディレクトリがドキュメントルートになった、簡易Webサーバーが立ち上がりります。

<http://localhost:8080/>

# ES2015以降のJavaScript開発を行うためのWebpackとBabelについて

ここでは、モダン JavaScript 開発に欠かせない Webpack と Babel というツールの導入方法を紹介します。



# Webpackとは

複数のJavaScriptファイルの依存関係を解決して、1ファイルにまとめてくれるツール。JavaScriptだけではなく、sassなどをコンパイルしてモジュールとして取り込むことができます。

# Webpackの導入

## プロジェクトの開始

次にWebpackの導入です。

```
mkdir my_project  
cd ./my_project  
npm init
```

# Webpackの導入

## webpackのインストール

次にいよいよWebpackをプロジェクトに追加します。以下のコマンドを入力することで `node_modules` というディレクトリーが自動で作成され、その中に、`webpack` とその依存モジュールがインストールされます。

```
npm i webpack --save-dev
```

# Webpackの導入

webpack-cliのインストール

またWebpack4からは別途、 `webpack-cli` というモジュールが必要になりました。これは `webpack` モジュールをコマンドラインから利用するために必要なモジュールです。

```
npm i webpack-cli --save-dev
```

# Webpackの導入

## ビルドコマンドの実行

`webpack` 及び、`webpack-cli` をインストールしたことにより、プロジェクト内で `webpack` を利用できるようになりました。

`package.json` の `scripts` の中に以下のコードを追加してみましょう。そうすることで、`npm run build` とターミナル上で実行することで `webpack` が動作します。

```
"scripts": {  
  "build": "webpack"  
}
```

# Webpackの導入

## ビルドコマンドの実行

デフォルトでは、`./src/index.js` がエントリーポイントになっているので、`src` ディレクトリーを作成して、その中にindex.jsを設置します。

例えば、以下のようなJavaScriptを記述します。

```
console.log('hello world');
```

# Webpackの導入

ビルドコマンドの実行

そして、`npm run build` を実行すると `./dist/main.js` が作成されているのがわかると思います。

# Webpackの導入

ビルドコマンドの実行

エントリーポイントを指定したい場合

```
"scripts": {  
  "build": "webpack ./src/index.js --output ./dest/bundle.js"  
}
```

# Webpackの導入

developmentモードと、buildモード

```
"dev": "webpack --mode=development",
"build": "webpack --mode=production"
```

# Webpackの導入

developmentでのビルドを実行

実行スピードが早いが、minifyや最適化されていません。素早く実行結果を確認したいときに有効です。

```
npm run dev
```

# webpack-dev-serverの利用

webpack-dev-serverのインストール

```
npm i webpack-dev-server --save-dev
```

```
"start": "webpack-dev-server --mode=development --open",
"build": "webpack --mode=production"
```

```
npm run start
```

# webpack-dev-serverの利用

実行すると自動でブラウザが立ち上がり、JavaScript を保存した段階でブラウザが自動で再リロードされ、変更が反映されていることが確認できると思います。

# Babelの使用

- babel-core
- babel-loader
- babel-preset-env

が必要

# Babelの使用

先ほどのパッケージ群をインストールします。

```
npm i babel-core babel-loader babel-preset-env --save-dev
```

# Babelの使用

## .babelrcの設定

`babel-preset-env` を使うことで、ターゲットブラウザのバージョンを指定して必要なBabelの `preset` を読み込んでくれます。

```
{  
  "presets": [  
    "env"  
  ]  
}
```

# Babelの使用

webpack.config.jsの作成

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      }
    ]
  }
};
```

下記のような JavaScript を書いてみましょう。 npm run build を実行すると、 dist/main.js にアロー関数が解決されたJavaScriptが生成されています。

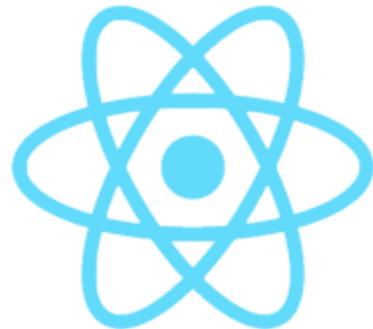
## index.js

```
const sayHello = (person) => {
  console.log(`hello ${person}`);
}
```

### 3.Reactを使ったフロントエンド開発

# Reactとは

- 簡単にいえば、テンプレートエンジン
- 仮想DOMを利用したHTMLの差分更新が特徴



React

# Reactとは

他の似たようなUI系ライブラリとして以下があります。

- Vue.js ( <https://vuejs.org/> )
- Riot.js ( <https://riot.js.org/> )
- Angular.js ( <https://angularjs.org/> )
- Polymer ( <https://www.polymer-project.org/> )



# ではなぜReactなのか？

- あくまでViewの差分更新のみの機能に特化している
- ES2015以降の知識とJSXの知識さえあれば実装できる。

# Reactを実行してみよう

```
git clone https://github.com/steelydylan/react-sample.git  
npm install  
npm run start
```

# Reactを実行してみよう

## 出力結果

増加ボタンを押すことで、数字が1増加し、減少ボタンを押すことで1減少します。

カウント数: 0

増加 減少

# Reactを実行してみよう

カウント数を0にするボタンをSampleコンポーネントに追加してみましょう。

1. renderメソッド内にボタンを追加

```
<button onClick={this.reset.bind(this)}>リセット</button>
```

# Reactを実行してみよう

## 2. resetメソッドを定義

```
...
reset() {
  this.setState({
    count: 0
  });
}
...
```

# Reactを利用する際に便利なJSX記法

JSXはReactを完結に記述するためにFacebook社によって開発された記法です。

HTMLに似た書き方ですが、`class` を `className` と書いたり、若干、HTMLとは記述が異なります。

# Reactを利用する際に便利なJSX記法

JSXの記法例

<https://github.com/steelydylan/react-sample/blob/master/src/index.js>

# Reactを利用する際に便利なJSX記法

属性に代入する値は文字列の場合はそのまま " " で囲うことができるが、数字や変数などは {} で囲う必要があります。 style はオブジェクトで記述する点にご注意ください。

```
const Modal = <div className="modal" tabIndex={-1} style={{display:'none'}}>
  <div className="modal-inner">
    <div className="modal-header">
    </div>
    <div className="modal-body">
    </div>
  </div>
</div>
```

# Reactのライフサイクル

また、Reactのコンポーネントにはライフサイクルというものがあり、ライフサイクルを覚えておくと、コンポーネントの生成時や削除時に処理を挟むことができます。

# Reactのライフサイクル

`componentWillMount()`

コンポーネントがhtml上に出力される寸前の処理

# Reactのライフサイクル

`componentDidMount()`

コンポーネントがhtml上に出力された直後の処理

# Reactのライフサイクル

## `componentWillReceiveProps(props)`

コンポーネントが親コンポーネントから値を受け取る直前の処理。引数には受け取る予定の `props` の値が入っている。

# Reactのライフサイクル

`componentWillUnmount()`

コンポーネントがhtml上に出力されなくなった時の処理

# Reactの導入

パッケージをインストール

```
npm install react react-dom --save
```

# Reactの導入

JSXをサポートするための文法プリセットをインストール

```
npm i babel-preset-react --save-dev
```

.babelrcの設定

```
{  
  "presets": ["env", "react"]  
}
```

## **create-react-appを利用**

create-react-appとは内部にwebpackを内包した、react環境を素早く構築するためのコマンドラインツール。

npmでインストール可能。

webpack初心者でも導入が簡単

# create-react-appの実行

```
npm install create-react-app -g  
create-react-app パッケージ名  
yarn start
```

# 付随するReact周辺ツールの知識

フロントエンド開発者はReact以外にも様々なツールを知っておくことで開発の幅が広がります。

<https://github.com/adam-golab/react-developer-roadmap>

- Redux（アプリケーションの状態を管理するためのツール）
- jest（Reactに特化したJavaScriptのユニットテストツール）
- GraphQL（REST APIに代わる新たなAPI サーバーへの問い合わせのためのデータクエリ言語）

などなど

## 4. Reduxを使ったアプリケーション構築

# Reduxとは

アプリケーションの状態（`state`）を環境に左右されず一貫したルールのもと管理するための仕組みです。



# Reduxとは

Reduxを理解するのに一番重要になってくるのが `store` という概念です。

`store` はアプリケーションの状態である `state` を保持し、そのステートは `Action` というチケットを通して、`Reducer` という関数が実行され、その結果が新しい `state` として再び `store` にセットされます。



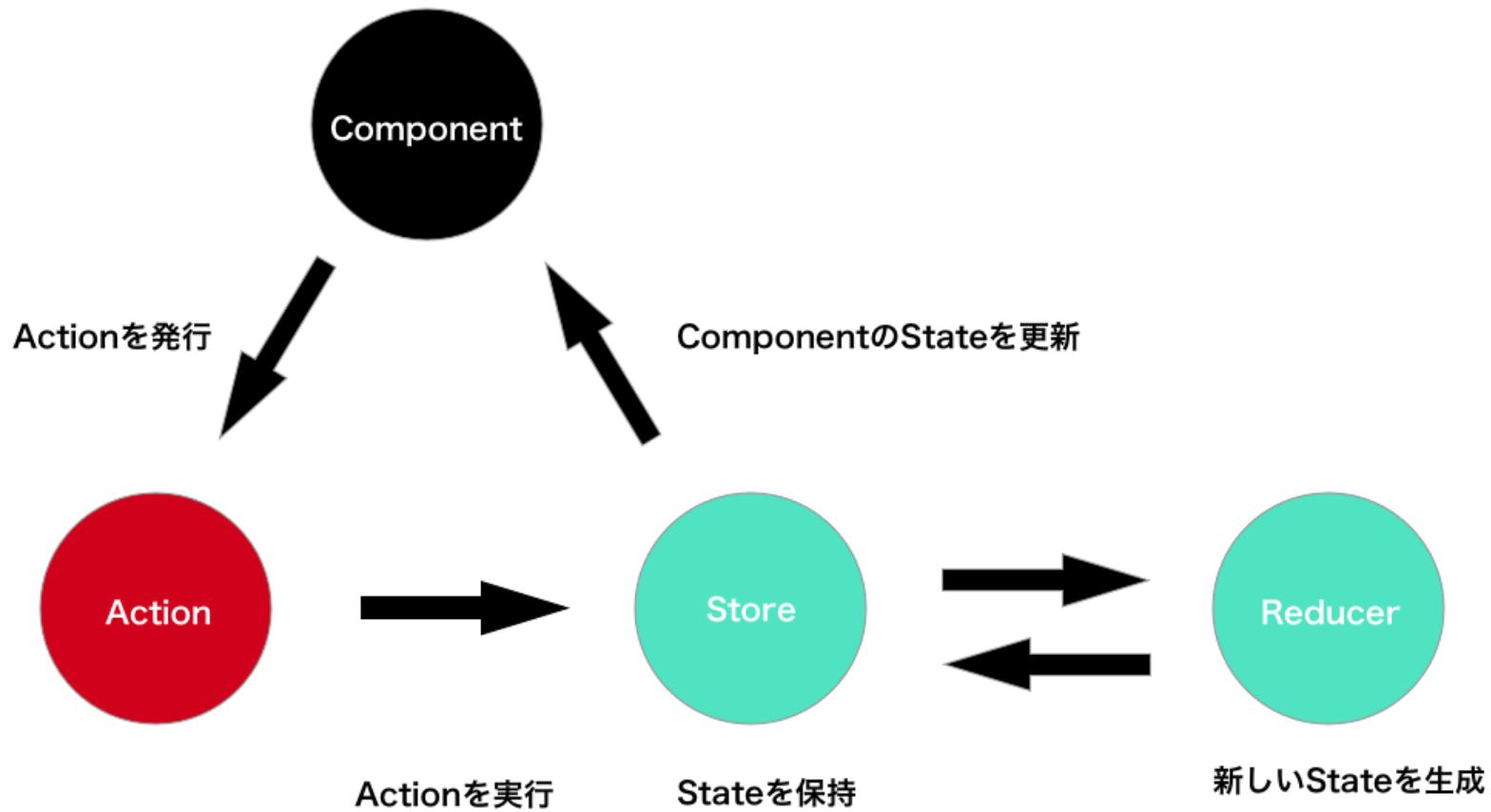
# Redux導入方法

```
npm install redux --save
```

# ReactとReduxを連携する

ReduxのState管理の仕組みとReactのコンポーネントを紐づけるために `react-redux` というツールを使います。 `react-redux` を使うことによって、 `Store` の状態を `Component` に紐づけたり、 `Component` から `Action` を実行することができます。

# ReactとReduxを連携する



# ReactアプリケーションにReduxを導入するメリット

例えば、コンポーネントの中に複数のコンポーネントが含まれているようなピラミッド型の構造が必要になってくるようなアプリケーションだと、データのやり取りに以下ののようなcallbackが複数回必要になってきて冗長になってしまうケースがあります。

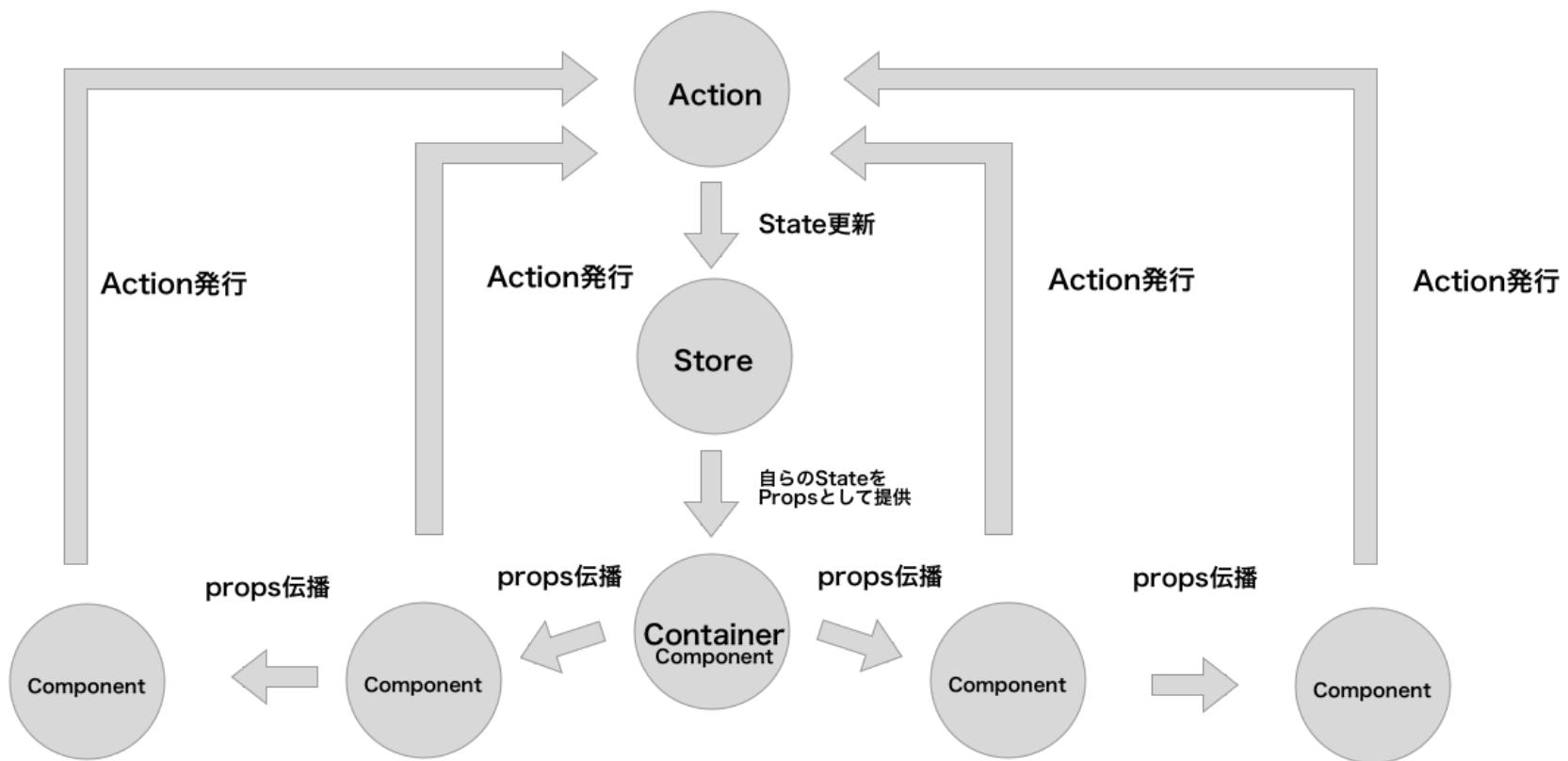
```
<ParentComponent hoge={this.state.hoge}>
  <ChildComponent onChange={(changedValue) => {
    this.setState({
      hoge: changedValue
    });
  }} />
</ParentComponent>
```

# ReactアプリケーションにReduxを導入するメリット

Reduxを使うことにより、アプリケーションに必要な state の更新を Action を介してどのコンポーネントからでも実行できるので、無駄なデータのやり取りのためのcallbackが不要になります。

# ReactアプリケーションにReduxを導入するメリット

以下が、孫コンポーネントから親コンポーネントを更新するイメージ図です。



# ReactアプリケーションにReduxを導入するメリット

全てのReactアプリケーションにReduxを導入した方がいいというわけではありません。むしろプロジェクトによってはReduxの導入で煩雑化してしまうこともあります。ですので、`State` がローカルのコンポーネントだけで完結していいものなのか、全体で管理した方がいい `State` なのか見極めが大事になってきます。

# ReactとReduxを連携する

## インストール方法

```
npm install react-redux --save
```

# ReactとReduxを連携する

下記のコードでは `mapStateToProps` `mapDispatchToProps` を `connect` して、 `Reducer` の `state` や `action` をAppコンポーネントのpropsで参照可能にしています。

<https://gist.github.com/steelydylan/6654d72c6c95f5b9a1fbeaa209e6d280>

# ReactとReduxを連携する

連携後は、`connect`されたコンポーネントから、  
`this.props.state`、`this.props.actions.action()`のような形で呼び出すことができます。

# React Redux 連携サンプル

<https://github.com/steelydylan/react-redux-sample>

```
git clone https://github.com/steelydylan/react-redux-sample.git .
npm install
npm run start
```

# React Redux 連携サンプル

## 実行画面

カウント数: 0

増加 減少

# React Redux 連携サンプル

現在、増加ボタンと減少ボタンがあり、ボタンをクリックすると、値が $\pm 1$ 増減します。さらに、RESETというアクションを追加して、値を0に戻すボタンを追加してみましょう。

# React Redux 連携サンプル

1. constants/ActionTypes.js で RESET を定数として追加

```
export const RESET = 'RESET';
```

## 2. actions/index.js で reset をアクションとして追加

```
export const reset = () => ({ type: types.RESET });
```

### 3. reducers/index.js で RESET に対する処理を記述

```
...
case types.RESET:
    return Object.assign({}, state, { count: 0 });
...
```

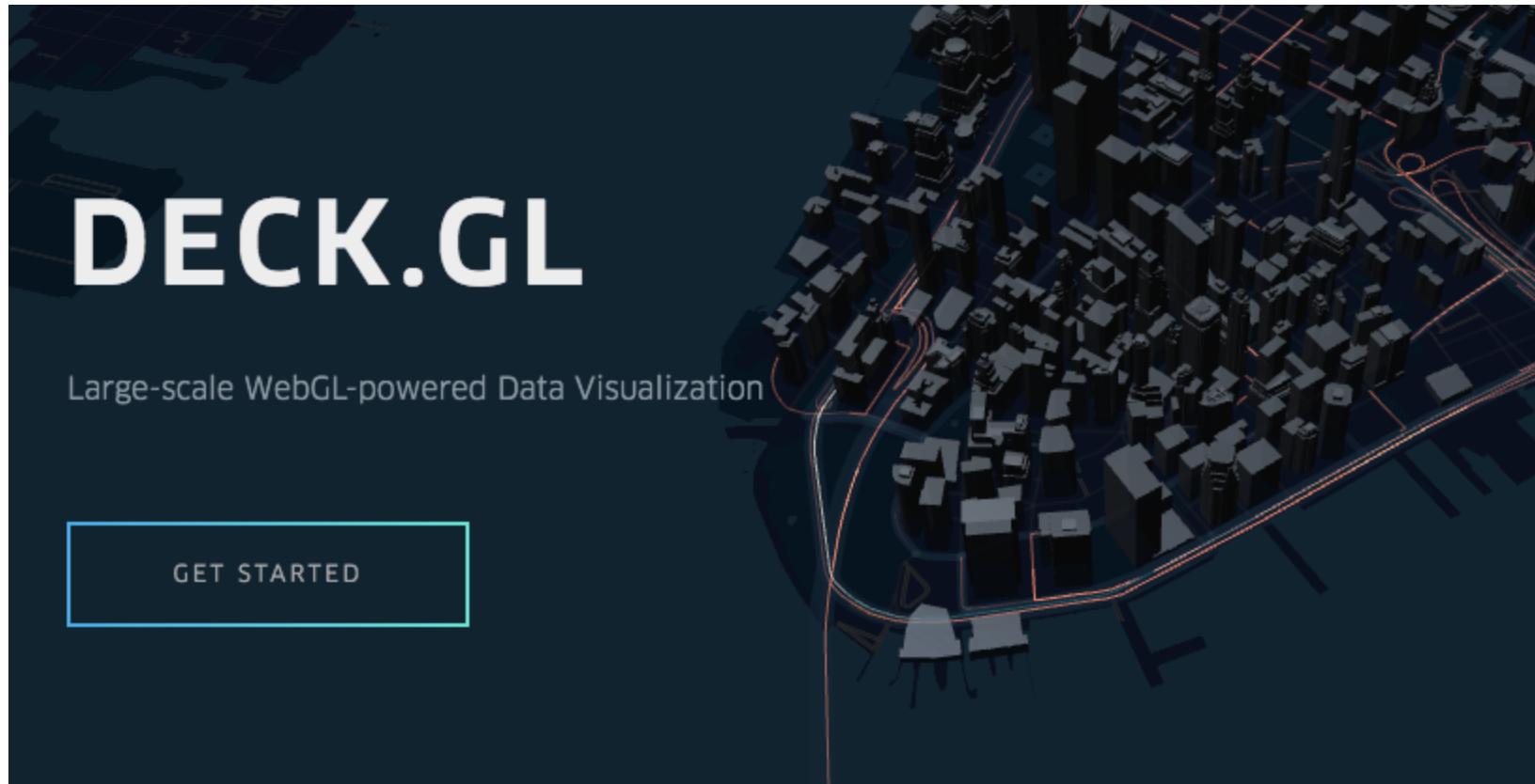
- components/sample.js でアクションを実行するためのボタンを追加

```
<button onClick={() => { this.props.reset(); }}>  
  リセット  
</button>
```

## 5. deck.glの紹介

# deck.glの紹介

WebGLベースの地理情報視覚化用のコンポーネント集  
 Reactとの連携が可能。



# deck.gl 導入方法

以下の3つのライブラリーが必要

- react-map-gl
- luma.gl
- deck.gl

```
npm install deck.gl luma.gl react-map-gl --save
```

# react-map-gl

OpenStreetMapのライブラリであるmap-glをReactベースで使える  
ようにしたライブラリ。

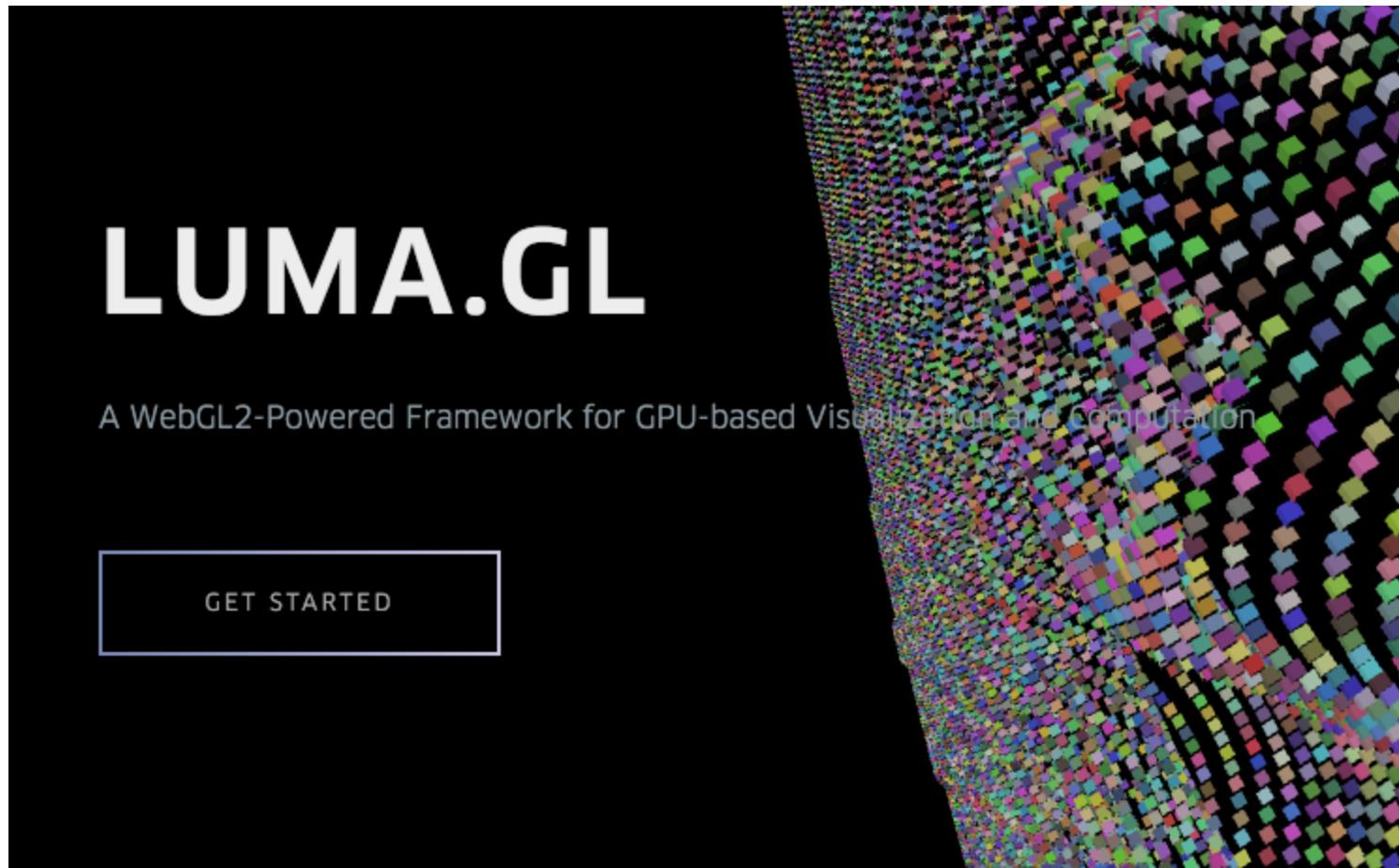
Uber社が開発



# luma.gl

Uber社が開発した、 WebGL用データビジュアライゼーションライブラリー。

これを利用して、 deck.glが動作します。



# 6. React + deck.gl

## 記述例

```
import DeckGL, {ArcLayer} from 'deck.gl';

const flights = new ArcLayer({
  data: [] // Some flight points,
  ...
});

<MapGL
  {...viewport} mapStyle={mapStyle} perspectiveEnabled
  onChangeViewport={onChangeViewport}
  mapboxApiAccessToken={mapboxApiAccessToken}
>
  <DeckGL width={1920} height={1080} layers={[flights]} />
</MapGL>
```

## 6. Harmoware-VISとは

# Hamoware-VIS とは

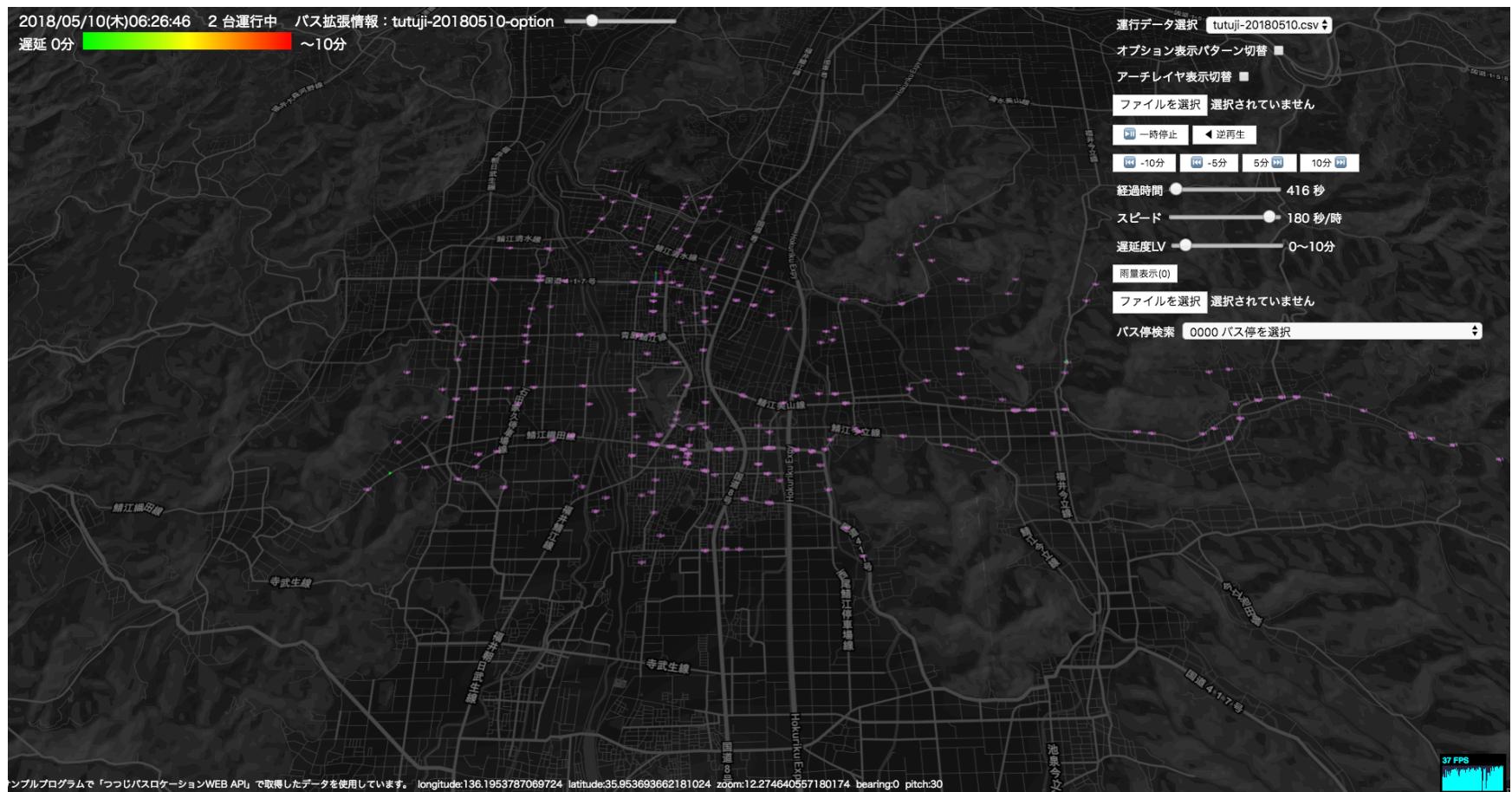
React Reduxを使って、deck.gl上での移動体の情報可視化や管理などを行うためのライブラリで、主に名古屋大学河口研究室が中心になって開発しています。

先ほど紹介した以下のライブラリに依存しています。

- react-map-gl
- [luma.gl](#)
- [deck.gl](#)

# Hamoware-VIS とは

## Hamoware-VIS の実行画面



## Harmoware-VISを使うメリット

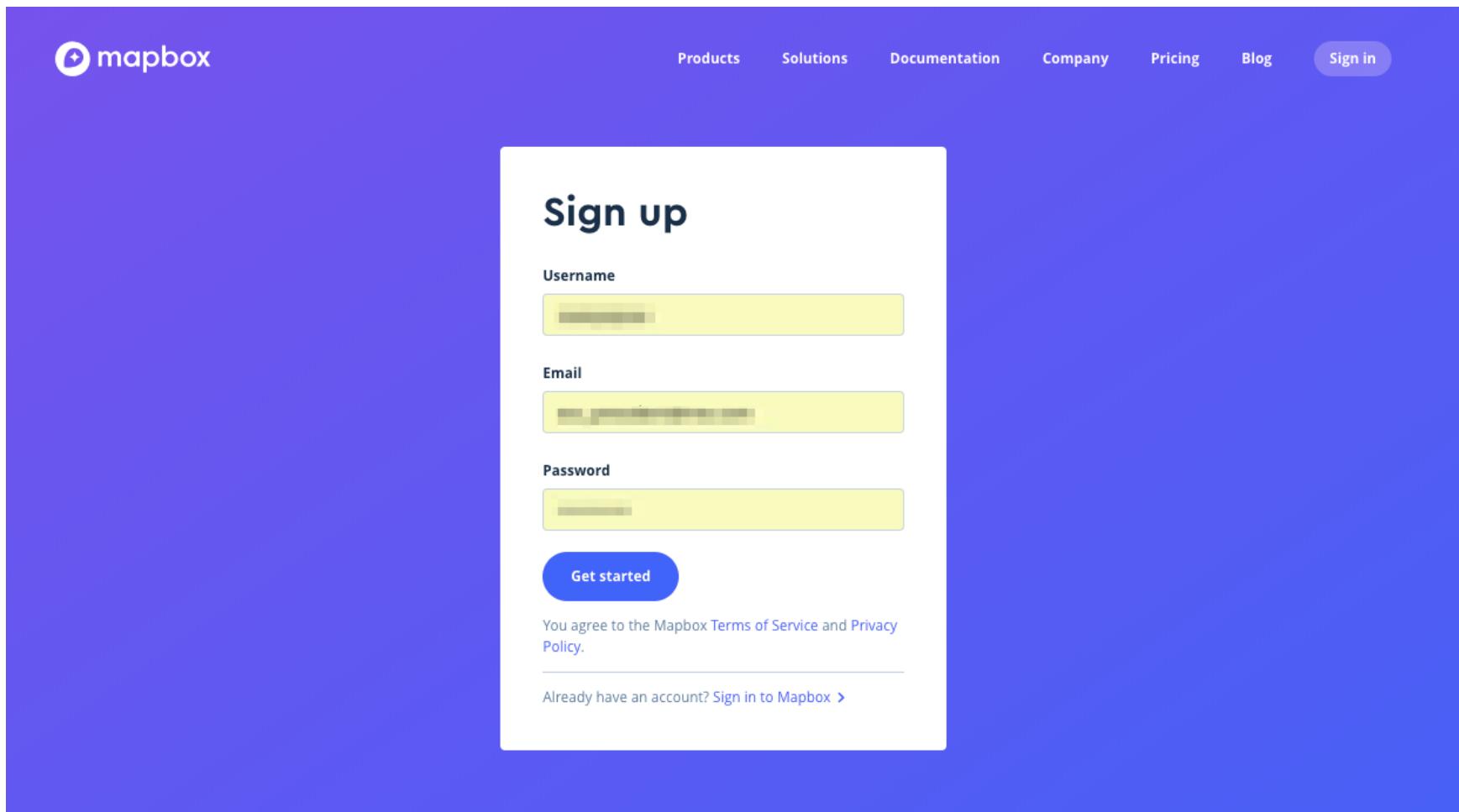
1. 地理情報の時間更新の自動処理
2. 時間コントロール系のコンポーネント
3. 移動体や停留所を表示するためのコンポーネント
4. その他今後の研究開発の進展に応じた拡張

## 7. Harmoware-VISを使ってみる

# Harmoware-VISを使ってみる

まずは、`mapbox` に登録してアクセストークンを取得する必要があります。

<https://www.mapbox.com/signup/?plan=paygo-1>



# Harmoware-VISを使ってみる

登録後は下記のURLより アクセストークン を取得できます。

<https://www.mapbox.com/install/>

## Add Mapbox to your app or website

### Install the Maps SDK

Install beautiful interactive maps in your app or website.



iOS



Android

JS



Unity

### Get your access token

Just looking for your [access token](#)?

Here it is!



### Design a map style

Design a [map style](#) with your own data and visual appearance.

[Design a map style →](#)

# Harmoware-VISを使ってみる

取得したアクセストークンは環境変数として登録しておきます。

```
set MAPBOX_ACCESS_TOKEN=XXXXXXXXXX
```

# Harmoware-VISを使ってみる

まずは `harmoware-demo` というディレクトリーを作成して、そこに `package.json` を作成します。

```
mkdir harmoware-demo  
cd harmoware-demo  
npm init
```

# package.jsonの編集

以下のようなpackage.jsonを作成しましょう。

<https://github.com/steelydylan/harmoware-demo/blob/master/package.json>

Harmoware-VISはまだ npm に公開されていませんので、 package の場所が github となることに注意してください。

## webpack.config.jsを作成

<https://github.com/steelydylan/harmoware-demo/blob/master/webpack.config.js>

依存関係を解決するために `mapbox-gl` のエイリアスだけ作っておくことに注意してください。

## .babelrcの作成

<https://github.com/steelydylan/harmoware-demo/blob/master/.babelrc>

`babel` の設定は `react` と `env` があれば充分です。さらに、  
`transform-object-rest-spread` があればオブジェクトの受け渡しの  
際に便利かもしれません。

## src/index.jsの作成

<https://github.com/steelydylan/harmoware-demo/blob/master/src/index.js>

Harmoware-VISでは react , redux を使用して、 action および state を管理しています。

## src/containers/app.jsの作成

<https://github.com/steelydylan/harmoware-demo/blob/master/src/containers/app.js>

`connectToHarmowareVis` 関数を使ってコンポーネントに Harmoware-VIS の `state` および `action` を `container` コンポーネントに紐付けます。

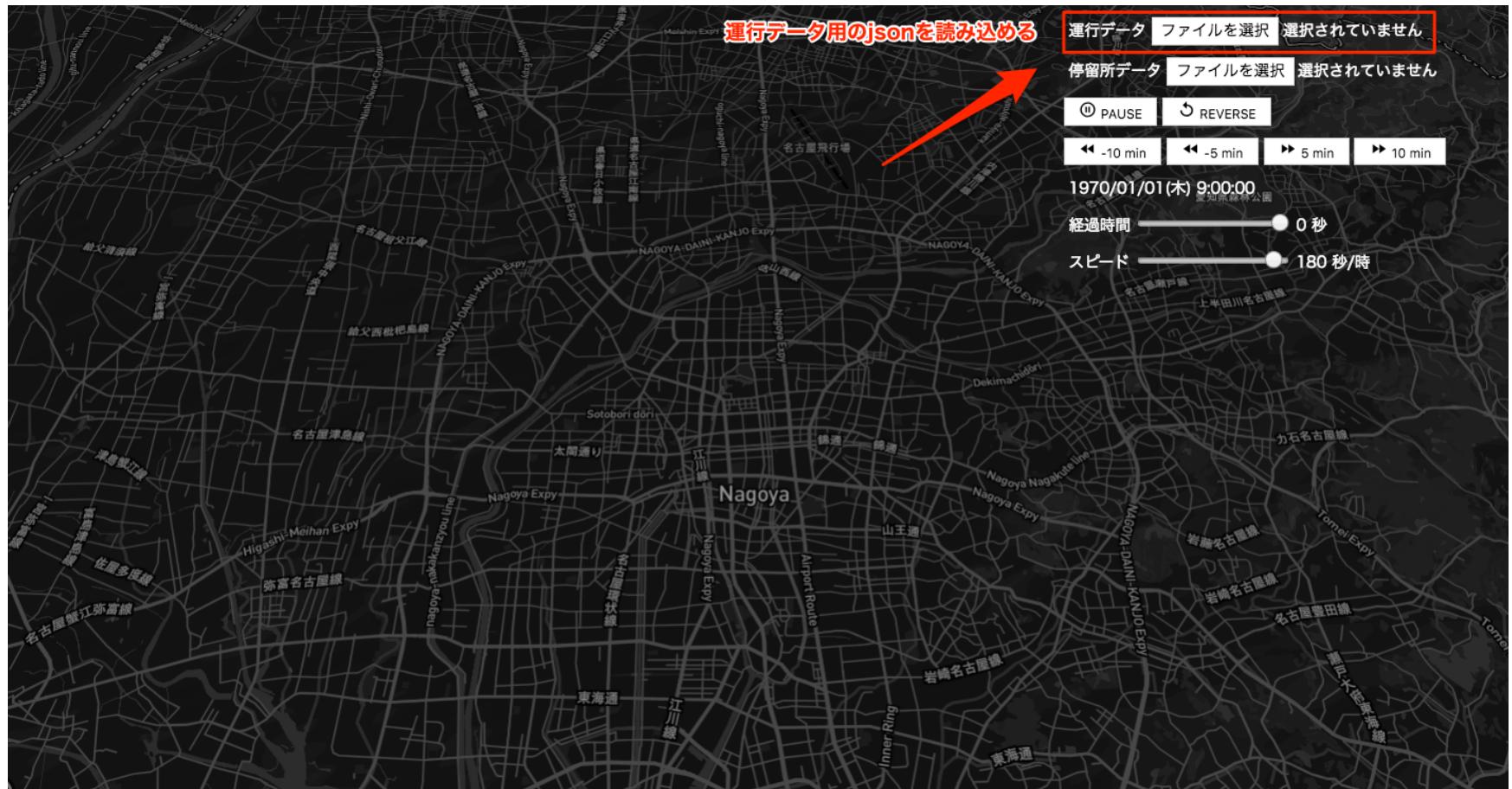
# Harmoware-VISの実行

```
npm install  
npm run start
```

# Harmoware-VISの実行



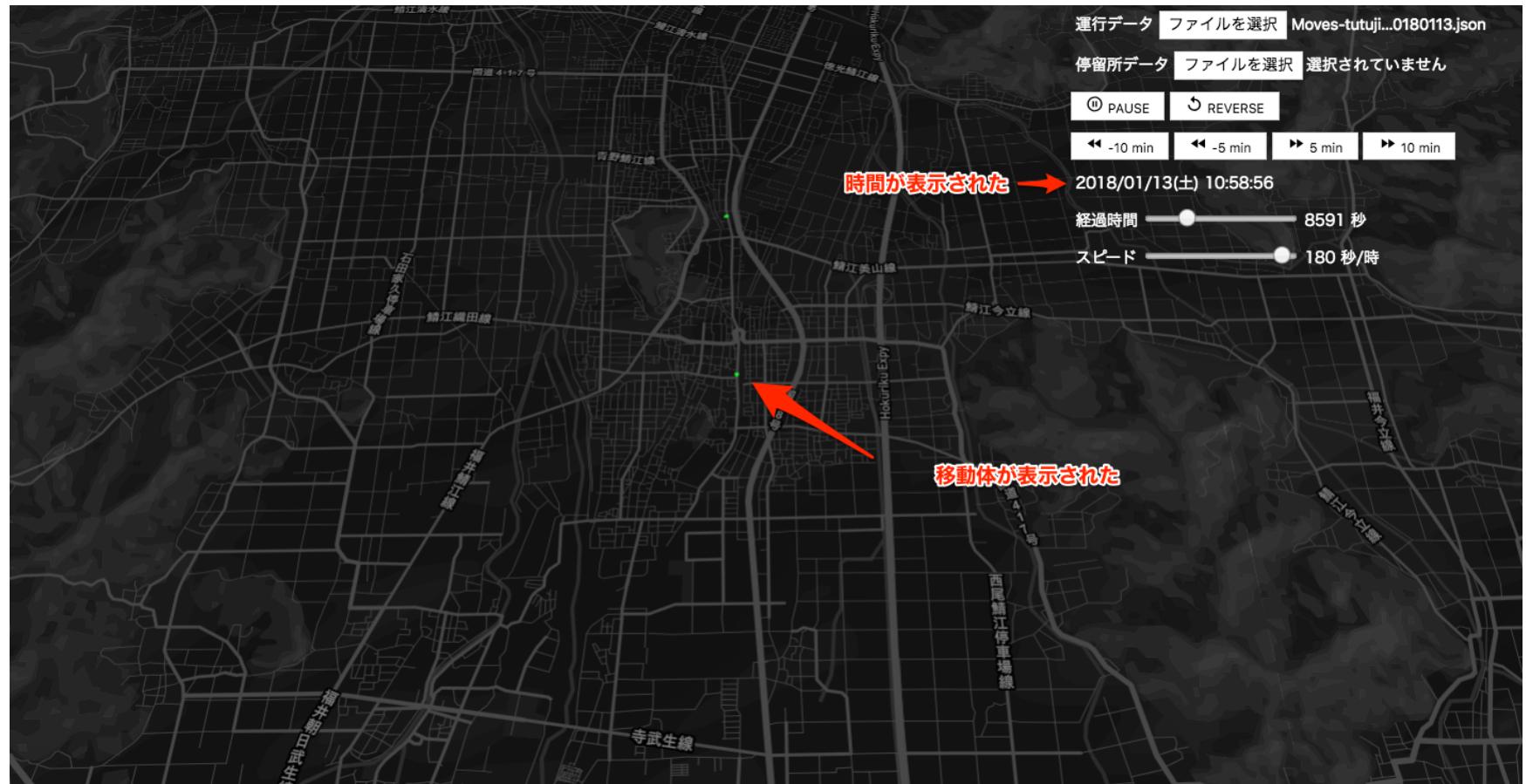
# Harmoware-VISの実行



運行データのjsonを読み込むことができます。  
以下のjsonファイルを読み込んでみましょう。

<https://raw.githubusercontent.com/steelydylan/harmoware-demo/master/json/Moves-tutuji-20180113.json>

# Harmoware-VISの実行



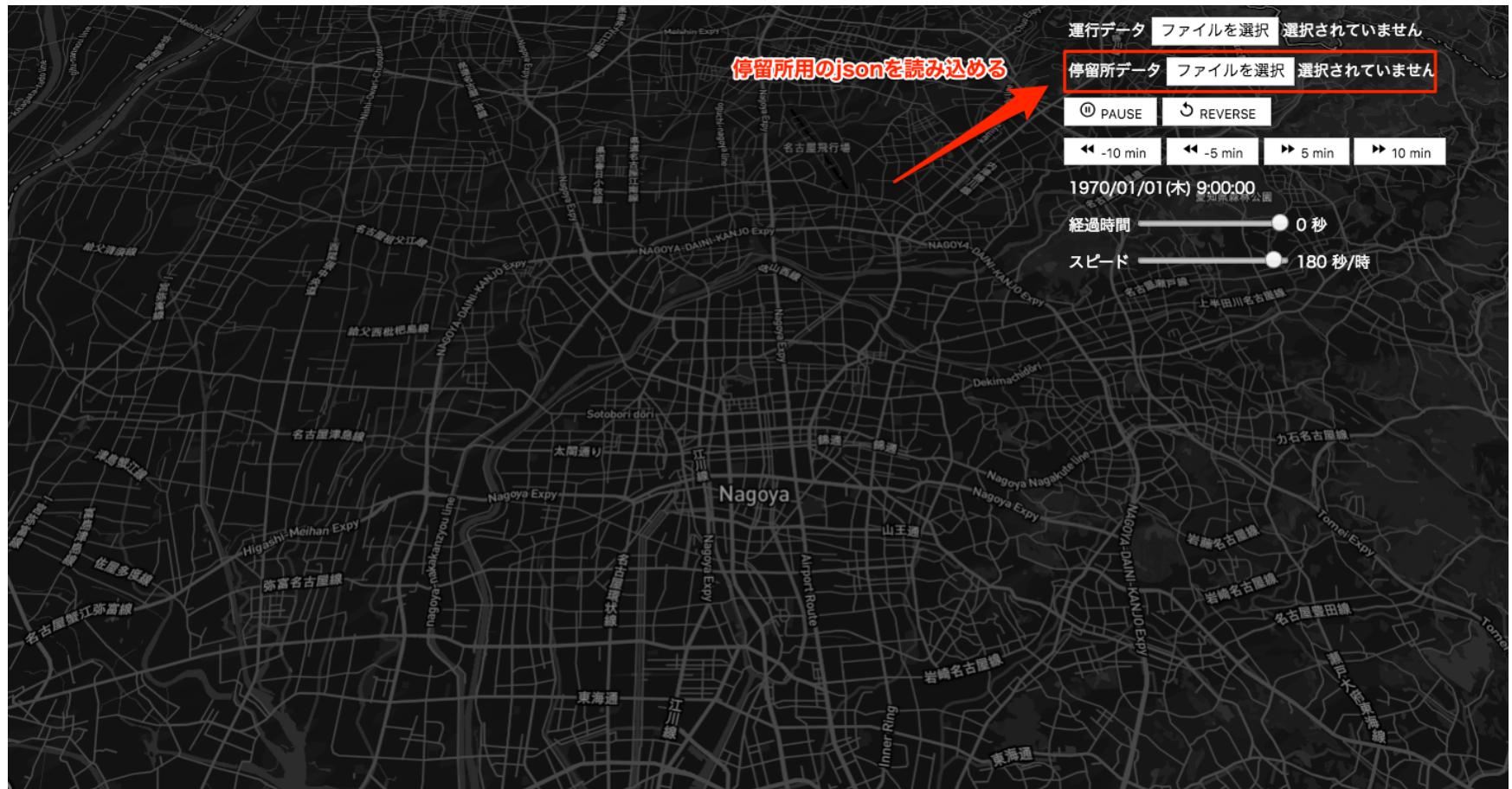
マップ状に移動体が表示され、どの時間にどの位置に移動体がいるのかが可視化されました。

# Harmoware-VISの実行

運行データのJSON形式

<https://github.com/Harmoware/Harmoware-VIS/blob/master/README.jp.md#運行シミュレーションデータファイルのjsonフォーマット>

# Harmoware-VISの実行

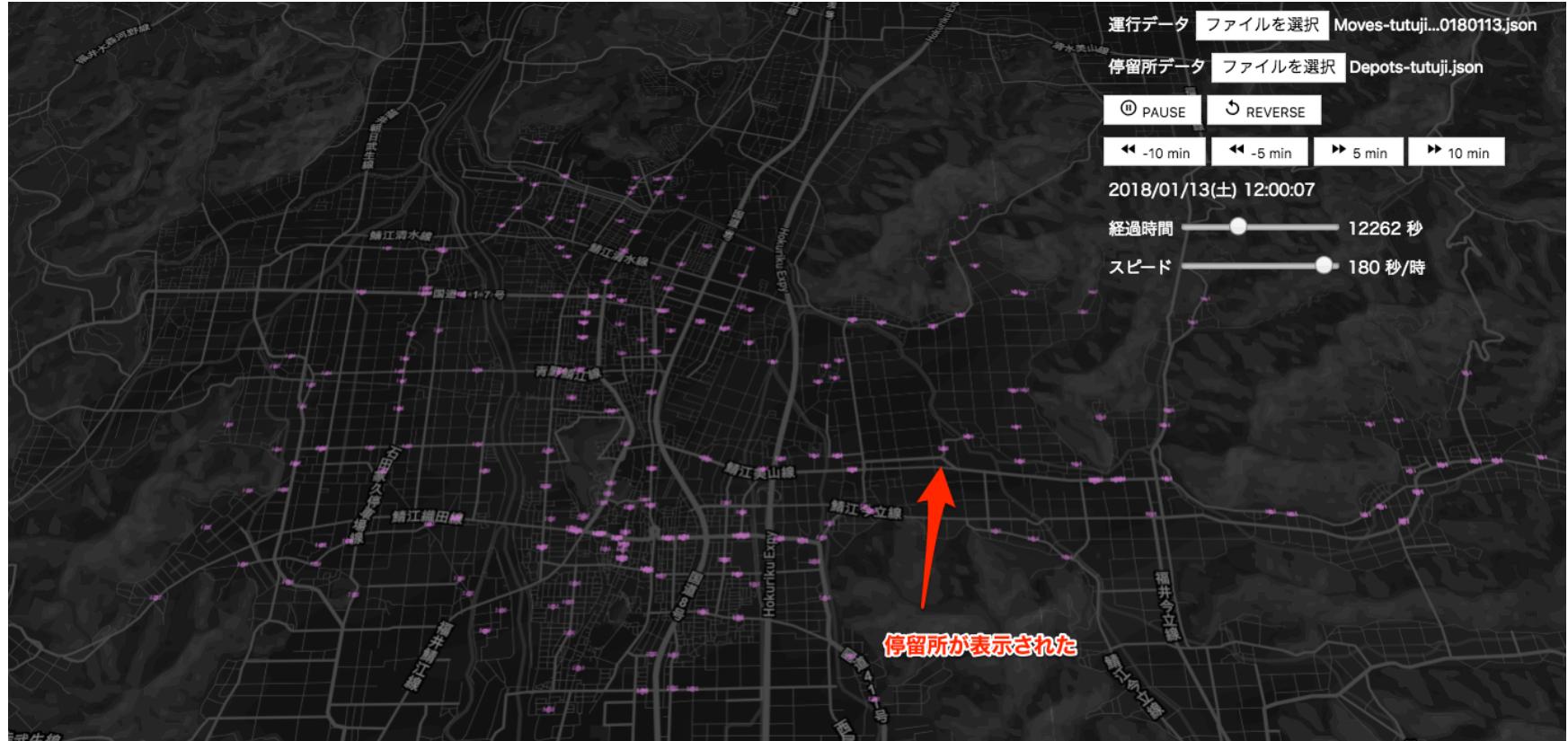


停留所データのjsonを読み込むことができます。

以下のjsonファイルを読み込んでみましょう。

<https://raw.githubusercontent.com/steelydylan/harmoware-demo/master/json/Depots-tutuji.json>

# Harmoware-VISの実行



移動体の他に、その移動体が停止する停留所（バス停など）が表示されました。

# Harmoware-VISの実行

停留所データのJSON形式

<https://github.com/Harmoware/Harmoware-VIS/blob/master/README.jp.md#停留所情報データのjsonフォーマット>

# Harmoware-VISの実行

## mouseover 時に移動体の情報を表示する

また、MovesLayer, DepotsLayerは onHover の callback より  
mouseover 時にhoverされたオブジェクトの情報を取得することができます。

```
new MovesLayer({ routePaths, movesbase,  
  movedData, clickedObject, actions,  
  onHover: (el) => {console.log(el)}  
});
```

## Harmoware-VIS レイヤー一覧

また、Harmoware-VISではいくつかのLayerをデフォルトで用意しています。

<https://github.com/Harmoware/Harmoware-VIS#harmoware-vis-layers>

各レイヤーの `import`

```
import {
    MovesLayer,
    MovesNonmapLayer,
    DepotsLayer,
    FixedPointLayer,
    LineMapLayer
} from 'Harmoware-VIS';
```

# MovesLayer

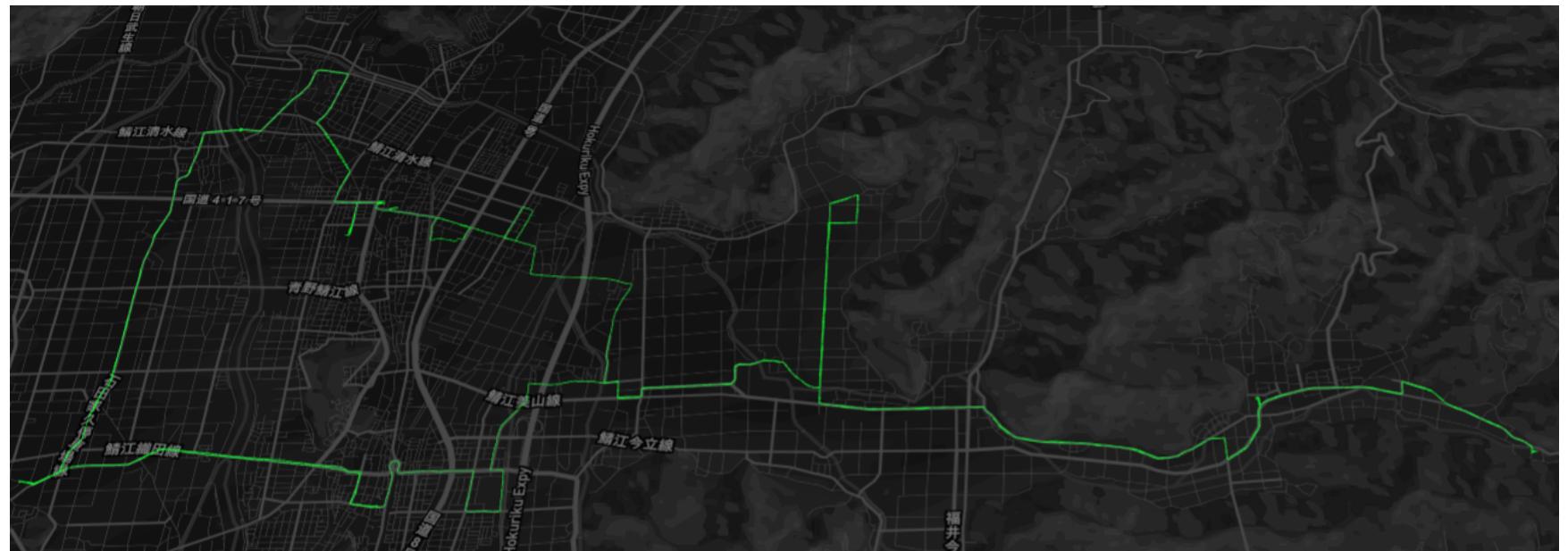
車両などの移動体をmapboxより取得したマップ上に表示します。  
下の図の緑の円が `MovesLayer` で表示されているオブジェクトです。

```
import { Harmovislayers, MovesLayer } from 'harmoware-vis';
...
<Harmovislayers layers={[new MovesLayer({
  routePath,
  movesbase,
  movedData,
  clickedObject,
  actions
})]}/>
```



## MovesLayer

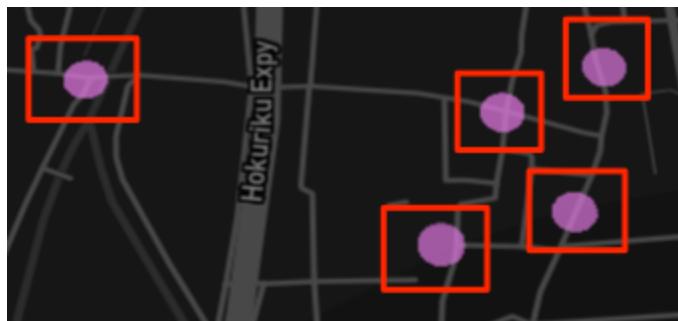
クリックすることで、その移動体の辿る経路が緑のライン上で表示されます。



## DepotsLayer

停留所や駅などをmapboxより取得したマップ上に表示します。下の図のピンクの円が `DepotsLayer` で表示されているオブジェクトです。

```
const {depotsData} = this.props;  
  
<HarmoVisLayers  
  layers={[  
    new DepotsLayer( { depotsData } )  
  ]}  
/>
```



## Harmoware-VIS コントロール用のコンポーネント一覧

Harmoware-VISでは自身のState更新用のコンポーネントをいくつか提供しています。

<https://github.com/Harmoware/Harmoware-VIS#harmoware-vis-control-component>

各コンポーネントの `import`

```
import { MovesInput, DepotsInput,
  LinemapInput, AddMinutesButton,
  PlayButton, PauseButton, ForwardButton,
  ReverseButton, ElapsedTimeRange, SpeedRange,
  SimulationDateTime
} from 'Harmoware-VIS';
```

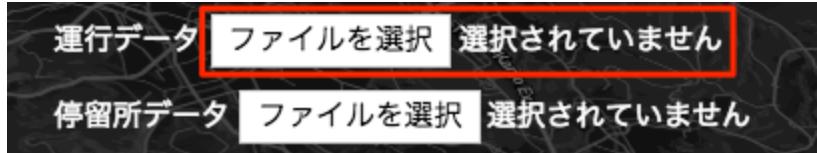
## MovesInput

移動体データを地図上に可視化するためのファイルを読み込むコンポーネント。

「運行シミュレーションデータ」を設定したファイルを選択するダイアログを表示し、読み込んだデータより Harmoware-VIS の `bounds`、`timeBegin`、`timeLength`、`movesbase` を設定します。

ここからjsonを読み込むことにより、`MovesLayer` が表示されます。

```
<MovesInput actions={actions} />
```



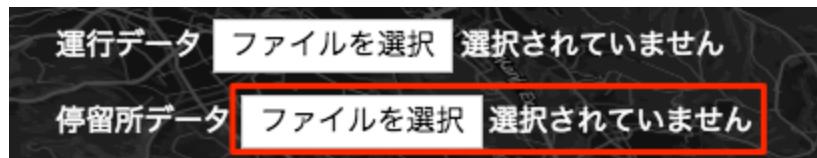
## DepotsInput

停留所データを地図上に可視化するためのファイルを読み込むコンポーネント

「停留所情報データ」を設定したファイルを選択するダイアログを表示し、読み込んだデータより Harmoware-VIS の depotsBase に設定します。

```
<DepotsInput actions={actions} />
```

ここからjsonを読み込むことにより、 DepotsLayer が表示されます。



## Harmoware-VISの State

以下が、Harmoware-VISが持っている state です。

action を介して、state を更新することができます。

<https://github.com/Harmoware/Harmoware-VIS#harmoware-vis-reducer>

## Harmoware-VISのaction

Harmoware-VISでは以下のactionをデフォルトで用意しています。

<https://github.com/Harmoware/Harmoware-VIS#harmoware-vis-actions>

## containerからstateの取得

例) Harmoware-VISの現在の時間を取得する

```
const { settim } = this.props;
```

# Harmoware-VISのプロジェクト構成

基本的に、Harmoware-VISのプロジェクト構成は型定義のための `types` ディレクトリーや、レイヤーのための `layer` ディレクトリー、ユーティリティのための `library` ディレクトリーが存在する点を除いて、`redux` のスタンダードな構成に従っています。

The screenshot shows the GitHub repository page for Harmoware / Harmoware-VIS. The repository has 3 issues, 0 pull requests, 0 projects, and a wiki. It has 9 stars, 4 forks, and was created 4 hours ago. The commit history for the master branch is displayed, showing the following commits:

Commit	Message	Time
steelydylan refactor getcontainerprop method	webpackを最新に更新、babelの設定などの見直し	11 days ago
actions	srcコンポーネントのstyleをclass指定可能に変更	6 days ago
components	XbandmeshLayer関連コードをharmoware-VISからbus3dサンプルに移動	20 days ago
constants	webpackを最新に更新、babelの設定などの見直し	11 days ago
containers	nonmapmoveで描画したrouteが消えない件の対策	7 days ago
layers	refactor getcontainerprop method	4 hours ago
library	XbandmeshLayer関連コードをharmoware-VISからbus3dサンプルに移動	20 days ago
reducers	eslintエラー対応	7 days ago
types	export default reducer	10 hours ago
index.js		

# Harmoware-VISのプログラム構成

Harmoware-VIS 本体は拡張されて使用されることが前提にあるので、前述の `actions` や `reducers`、`components`、`constants` などの設定は全て Harmoware-VIS のエントリーポイントで `export` しています。

<https://github.com/Harmoware/Harmoware-VIS/blob/master/src/index.js>

従って、Harmoware-VISをライブラリとして利用する際は、以下のように `actions` や `reducers` に関わらず、すべからく同一の方法で `import` 可能です。

```
import {Actions, MovesInput, DepotsInput,
LinemapInput, PlayButton, settings, Container, MovesLayer,
DepotsLayer, connectToHarmowareVis, getCombinedReducer, reducer}
from 'harmoware-vis';
```

# Harmoware-VIS の拡張

Harmoware-VISでは `redux` を使用しており、移動体や停車場の情報管理のための `store` をデフォルトで用意しています。

Harmoware-VISでは、この `store` を拡張するための機能が用意されています。 `store` を拡張することで、移動体や停留所以外の情報も管理できるようになります。今回は下のデモのようにバンクーバーの地域ごとの時価をグラフ表示できるようにしてみましょう。

<http://deck.gl/showcases/gallery/geojson-layer>

以下のjsonデータを読み込み、表示できるようにします。

<https://raw.githubusercontent.com/uber-common/deck.gl-data/master/examples/geojson/vancouver-blocks.json>

# Harmoware-VIS の拡張

1. src/reducer/geo.jsの作成
2. action及びconstantsの作成
3. baseのreducerと結合
4. 結合したreducerの使用
5. geo情報を読み込むためのコンポーネントを設置
6. containerとstoreの接続

## 1. src/reducer/geo.jsの作成

まずは、jsonデータをstateとして保持するためのreducerを作成します。

今回は geo.js としています。

<https://github.com/steelydylan/harmoware-demo2/blob/master/src/reducers/geo.js>

## 2. action及びconstantsの作成

先ほど作成した geo.js を活用するためにそれ用の action と constants を定義します。

src/constants/index.js

```
export const SETGEO = 'SETGEO';
```

src/actions/index.js

```
import * as types from '../constants';

export const setGeo = (geo) => ( {
  type: types.SETGEO, geo
});
```

### 3. baseのreducerと結合

`getCombinedReducer` という `harmoware-vis` で用意されたAPIを使用して自作reducerとHarmoware-VISのreducerを結合できます。

```
import geo from './geo';
import { getCombinedReducer } from 'harmoware-vis';

export default getCombinedReducer({ geo });
```

## 4. 結合したreducerの使用

先ほど合体したreducerをimportして store を作成します。  
react-reduxモジュールから import した Provider コンポーネント  
で container (App) を囲います。

```
import App from './containers';
import { Provider } from 'react-redux';
import reducer from './reducers';
const store = createStore(reducer);

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('app')
);
```

## 5. geo情報を読み込むためのコンポーネントを設置

次に実際にファイルを読み込み `action` を実行するためのコンポーネントを作成します。

この時に、`actions.setGeo()` だけではなく、読み込まれた位置に `viewport` を移動するために、`actions.setViewport()` も実行すると親切です。

<https://github.com/steelydylan/harmoware-demo2/blob/master/src/components/geo-layer-input.js>

## 6. containerとstoreの接続

先ほどの `geo-layer-input.js` を含んだ `container` コンポーネントを作成し、`connectToHarmowareVis` メソッドで、`container` と `Harmoware-VIS` を結びつけます。

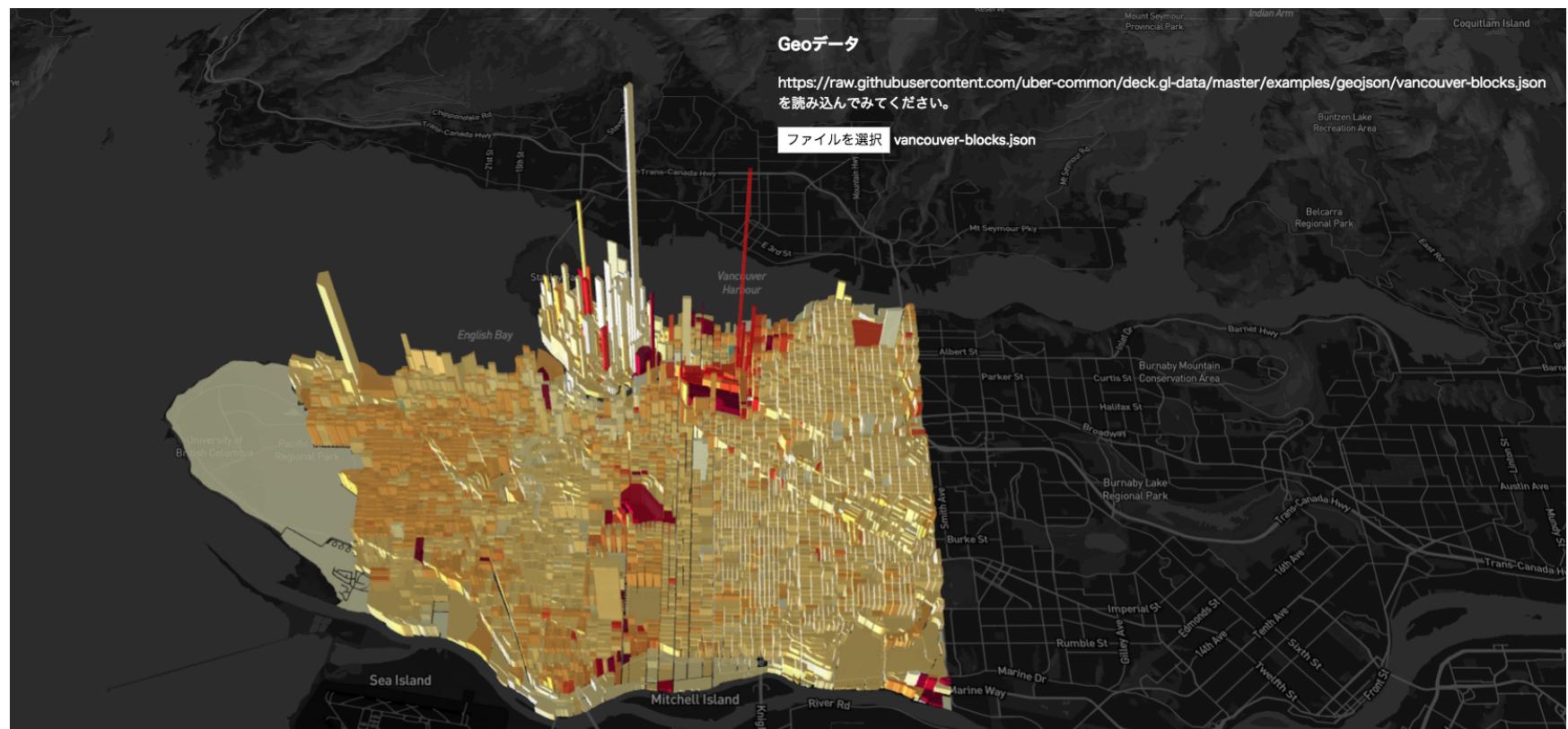
<https://github.com/steelydylan/harmoware-demo2/blob/master/src/containers/app.js>

以下のように json を読み込むとバンクーバーの時価情報が表示されていれば成功です。

このように、Harmoware-VISを拡張するには redux の知識が必要になってきます。

実際のソースコードはこちらにおいてあります。

<https://github.com/steelydylan/harmoware-demo2/>



以上で、JavaScriptの入門から Harmoware-VIS の使い方、カスタマイズ方法までのチュートリアルは終了です。Harmoware-VIS のさらに詳しいカスタマイズ方法は以下の3つのサンプルを見ていただくと参考になるかと思います。

- <https://github.com/Harmoware/Harmoware-VIS/tree/master/examples/bus3d>
- <https://github.com/Harmoware/Harmoware-VIS/tree/master/examples/visualize-sample-nonmap>
- <https://github.com/Harmoware/Harmoware-VIS/tree/master/examples/visualize-sample>