

React はじめの一步

はじめに

目標

- Reactの概念を何となく理解する
- 軽めのReactを書けるようになる
- （Nodeのサーバを建てられるようになる）

やらないこと

- CSS (`CSS-modules`)
- アーキテクチャ (`Flux`, `Redux`)

事前準備

使うもの

- Google Chrome, Visual Studio Code, Yarn, Node.js (v8.x)

ちなみに

- 今回, 最終的にできるファイルは[ここ](#)
- [前回](#)のスライドを読み, ツールを使いこなせる前提で進めていきます

Menu

- 復習
- Reactとは
 - 仮想DOMとは
- Reactを動かしてみる
- メールフォームを作る
 - Promiseとfetch
 - Nodeでサーバ立てる
- ライブラリを使ってみる

Yarnの復習

- `yarn init` = package.jsonを作る
- `yarn` = 依存モジュールを全部一気に入れる
- `yarn add xxx` = 依存ライブラリ `xxx` を入れる
- `yarn add -D xxx` = 開発用ライブラリ `xxx` を入れる
- `yarn xxx` = npm scriptのタスク `xxx` を実行

JavaScript (ES6) の復習

- 変数宣言：再代入不可の `const`, 再代入可の `let`
- アロー関数 (≒即時関数・ラムダ式)

```
// 従来の関数
var f = function(x) {
  return x * 2
}

// アロー関数
const f = (x) => {
  return x * 2
}
const f = (x) => x * 2 // 1行なら中カッコ{}は省略
const f = x => x * 2  // 引数1つならカッコ()不要
```

- クラス構文
(内部的には関数に変換されるらしい)

```
class Person {  
  constructor(name) {  
    this.name = name  
  }  
  hello() {  
    console.log(`My name is ${this.name}.`)  
  }  
}  
  
const p = new Person('es6')  
p.hello()           //=> "My name is es6."
```

- `import` / `export`

- **person.js**

```
export default class Person { }
```

- **index.js**

```
`import Person from './person'
```

importすることでindex.jsでもPersonクラスが使える

React.jsとは

- FacebookのOSSで、UIのコンポーネント（構成部品）を作るためのライブラリ

特徴

- 仮想DOM（Virtual DOM）が速い
- JS内にHTMLを書くようなJSX記法（なくても可）
- Reactの記法でiOSやAndroidのネイティブアプリが書ける[React Native](#)もある

参考：[Reactを使うとなぜjQueryが要らなくなるのか](#)

仮想DOMとは

- 生のDOM（HTMLインスタンス）に1対1対応するJSのオブジェクトのこと
- その差分によって**必要最低限のDOM操作**で状態遷移を実現
- データが更新されると**自動**で差分レンダリング
- 有名なフレームワークだと `React` や `Vue` が採用

何が違うか

従来

- データに変更があったら、生のDOM要素から対応する部分を探し、中身を書き換える
→ 変更箇所を探すコードを書くのは大変

仮想DOM

- ライブラリが差分を検出し、自動更新してくれる
→ 何もしなくてもViewに流し込まれるから楽
- ただのオブジェクトの比較なので、処理が軽い

例えば？

画面のリスト `[a, b]` を 新しいリスト `[a, c]` に更新

```
<li>a</li>          <li>a</li>  
<li>b</li>    --- 更新 ---> <li>c</li>
```

- 従来
 - 現在の `` タグを全て取得し、一つ一つを新しいデータと比較して書き換える
 - 生のDOMを使って比較するので処理が重い
- 仮想DOM
 - 全自動 🍷

JSX記法

JSの言語拡張, JS内にHTMLタグが書けるイメージ

```
class App extends Component {  
  render() {  
    return (  
      <div>Hello React</div>  
    )  
  }  
}  
  
render(<App />, document.querySelector('main'))
```

拡張子は `.jsx` (`.js`でもよさそう)

`class` はJSで予約語のため, `className` と書く

準備

- VSC設定

```
"editor.formatOnSave": true,  
"files.trimTrailingWhitespace": true,
```

- Chrome拡張

- [React Dev Tools](#)

- 拡張からファイルへのアクセスを許可しとく

- プロジェクトフォルダを作り, `yarn init -y` する

依存モジュール

```
yarn add react react-dom
```

- `react` : React本体
- `react-dom` : オブジェクトをDOMと結びつける

開発モジュール

```
yarn add -D webpack babel-core babel-loader  
babel-preset-es2015 babel-preset-react  
babel-plugin-transform-decorators-legacy
```

- `webpack` : JS合体君・バンドラ
- `babel-core` : JSを変換する（**バベる**）君
- `babel-loader` : webpack上でバベれるようにする君
 - `babel-preset-es2015` : ES6 → ES5
 - `babel-preset-react` : JSX → 普通のJS
 - `babel-plugin-transform-decorators-legacy` :
@（デコレータ）を使えるようにする君

ファイルを用意

- `src/`
 - `index.jsx`
- `dist/`
 - `index.html`
 - `style.css`
- `webpack.config.js` : webpackの設定
- `jsconfig.json` :
VSCが@に対してエラー表示しないように設定

jsconfig.json

```
{
  "compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "module": "es6",
    "target": "ES6"
  },
  "exclude": [
    "node_modules"
  ]
}
```

プロジェクトルートに置いておく
今後一切触らない

index.html

```
<!DOCTYPE html>
<html lang='ja'>
  <head>
    <title>Mail Form</title>
    <meta charset='UTF-8'>
    <link rel='stylesheet' href='style.css'>
  </head>
  <body>
    <main></main>
    <script src='index.js'></script>
  </body>
</html>
```

style.cssは空で大丈夫

webpack.config.js

- `src/index.jsx`を`dist`に吐き出す
- JSでimportする時に拡張子`js`, `jsx`を省略する

```
const webpack = require('webpack')
const path = require('path')

module.exports = {
  entry: {
    index: path.join(__dirname, './src/index.jsx')
  },
  output: {
    path: path.join(__dirname, 'dist/'),
    filename: '[name].js'
  },
}
```

```
module: {
  rules: [{
    test: /\.js|jsx?$/,
    exclude: /node_modules/,
    loader: 'babel-loader',
    options: {
      presets: ['react', 'es2015'],
      plugins: ['transform-decorators-legacy']
    }
  }]
},
resolve: {
  extensions: ['.js', '.jsx']
}
}
```

- jsxのファイルに書かれたreactやらes6やらデコレータを普通のJSに変換する

index.jsx

```
import React, { Component } from 'react'
import ReactDOM, { render } from 'react-dom'

render(<h1>React</h1>, document.querySelector('main'))
```

index.html内のmainタグに

h1タグをマウント（レンダリング）する

Reactを動かしてみる

npm scriptはこんなん

```
"scripts": {  
  "build": "webpack",  
  "watch": "npm run build -- -w"  
}
```

- `yarn build`: webpackでJSを1度だけビルド
- `yarn watch`: 変更を検知してビルドし直し続ける

`yarn watch`して `dist/index.html`を開いてみよう
h1タグがレンダリングされているはず

コンポーネントを作る

- `src/components/App.jsx`を作る

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'

export default class App extends Component {
  constructor(props) {
    super(props)
  }
  render() {    // 実際のAppタグの中身
    return (
      <div>App</div>
    )
  }
}
```


index.jsx

```
import React, { Component } from 'react'
import ReactDOM, { render } from 'react-dom'
import App from './components/App'

render(<App />, document.querySelector('main'))
```

yarn watch して, ブラウザをリロードしたら
main タグの中に App タグがマウントされる

イベントをハンドリングする

関数を定義し, onClickでイベントハンドラを登録

```
// App.jsx
submit() {
  alert(`submit!!`)
}
render() {
  return (
    <div>
      <button
        onClick={this.submit.bind(this)}>submit</button>
    </div>
  )
}
```

autobind-decoratorを使う

- イベントの度に `.bind(this)` を書かなくてもOKに
- `yarn add autobind-decorator`

```
import autobind from 'autobind-decorator' // importして
```

```
@autobind // ってつけると  
submit() {  
  /* 処理 */  
}
```

```
// .bind(this)がなくなってスッキリ  
<button onClick={this.submit}>submit</button>
```

Props

親コンポーネントから渡されたプロパティ（不変）

- 例：Todoのリストを表示するアプリ

App → (list) → TodoList → (text) → Todo

- Propsの渡し方

```
<Todo text='買い物' />
```

- Stateを持たないコンポーネントは関数でも書ける

```
const Todo = props => <div>{props.text}</div>
```

State

そのコンポーネントが持っている状態（可変）

- 例：フォームなどで入力されたテキストの保持

1. `<input type='text' />`

2. inputの値はStateのデータを表示する

`value={this.state.name}`

3. 入力時，自コンポーネント内のStateに保存

`onChange={this.editName}`

Props も State も React Developer Tools で見れる

Stateを使ってみる

```
constructor(props) {  
  super(props)  
  this.state = {           // stateを宣言 & 初期値を設定  
    name: ''  
  }  
}  
@autobind  
editName(e) { // イベントe -> テキストをstateのnameに保存  
  this.setState({ name: e.currentTarget.value })  
}
```

```
<input type='text'  
  onChange={this.editName} value={this.state.name} />
```

コンポーネント内のデータと見た目が結びついた

メールフォームを作る

- これまでを踏まえて, Stateに `name` と `text` を持ち, 見た目と結びついたコンポーネントを書いてみる

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'

export default class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      name: '',
      text: ''
    }
  }
}
```

```
@autobind
editName(e) {
  this.setState({ name: e.currentTarget.value })
}
@autobind
editText(e) {
  this.setState({ text: e.currentTarget.value })
}
@autobind
submit() {
  alert(
    `氏名: ${this.state.name} \n内容: ${this.state.text}`
  )
}
```

続<


```
render() {  
  return (  
    <div>  
      <p>  
        <label>name</label>  
        <input type='text' onChange={this.editName}  
          value={this.state.name} />  
      </p>  
      <p>  
        <label>text</label>  
        <textarea onChange={this.editText}  
          value={this.state.text}></textarea>  
      </p>  
      <button onClick={this.submit}>submit</button>  
    </div>  
  )  
}
```

cssも適当に作ったのでコピる

1. <https://github.com/pvcresin/reactMailFormDemo/blob/master/dist/style.css> にアクセス
2. Rawボタンを押すと, コードがコピりやすくなる
3. `dist`の直下の`style.css`にコピる

基盤完成

- これでinputから値を取得してalertに出すまで完成
- あとはNodeでサーバを立てて、送信して結果取得したら終わり

サーバ起動準備

- `yarn add express body-parser`
 - `express`:
Nodeのサーバ立てるのに便利なフレームワーク
 - `body-parser`: JSON扱えるようにする君
- `npm script`に起動コマンド追加
 - `"start": "node server.js"`
 - Nodeの使い方
 - `node server.js`で`server.js`を実行

- `server.js` をプロジェクトルートに作成

```
const express = require('express')
const bodyParser = require('body-parser')

express()
  .use(express.static('dist'))
  .use(bodyParser.urlencoded({ extended: true }))
  .use(bodyParser.json())
  .post('/sendMessage', (req, res) => {
    console.log(req.body)
    res.json({
      server_receive_time: new Date().toString(),
      name: req.body.name,
      text: req.body.text,
    })
  })
  .listen(3000, () => {
    console.log('http://localhost:3000')
  })
```

`server.js` がやってること

- `dist` フォルダをServe (html, cssとか)
- `http://localhost:3000` でサーバを起動
- `/sendMessage` にデータを `POST` すると値を返す
 - `server_receive_time`, `name`, `text` が入ったJSON

サーバを起動

1. `yarn watch` とは別のターミナルで `yarn start`
 - VSCは統合ターミナルの右に **+** ボタンがある
 - `yarn watch`: jsxをjsに変換し続ける
 - `yarn start`: `dist` フォルダをserveし続ける
2. <http://localhost:3000> にアクセス

あとはクライアントからデータを送信するだけ

Promiseとは

- 非同期処理に起こりがちなコードのCallback地獄から救い出す君
- モダンブラウザなら大体対応している（はず）
- Nodeだけど練習にいい記事
[今更だけどPromise入門](#)

何が良いか

- 非同期処理を行う関数A, B, Cがあるとする
 - BにはAの結果が必要, CにはBの結果が必要

```
A(function(a) {           // Aの結果が帰ってきた時のCallback
  B(a, function(b){        // Bの..Callback
    C(b, function(c){      // Cの...Callback
      done(c)              // 最終的な結果
    })
  })
})
```

どんどんネストが深くなっていく...

Promise使うと

```
A()  
  .then(B)  
  .then(C)  
  .then(done)
```

- メソッドチェーンできる（脱Callback地獄）

`Promise.all()`:

複数の処理が全て終わったら呼ばれる

`Promise.race()`:

複数の処理のうち1つが終わったら呼ばれる

fetchとは

GETとかPOSTする時にPromise型で処理できる関数

- 使い方

```
fetch(url, {  
  method: 'POST',  
  body: /* 送信データ */  
}).then(response => {  
  return response.json() // 結果をJSON形式に変換  
}).then(json => {  
  /* 結果のJSONを使った処理 */  
})
```

[お疲れさまXMLHttpRequest、こんにちはfetch](#)

通信準備

```
this.state = {  
  name: '',  
  text: '',  
  result: {}  
}
```

- stateに `result` を定義
- 初期値は空のオブジェクト `{}` を設定

fetchを使ってみる

```
@autobind
submit() {
  const url = './sendMessage'
  fetch(url, {
    headers: { 'Content-Type': 'application/json' },
    method: 'POST',
    body: JSON.stringify({
      name: this.state.name,
      text: this.state.text
    })
  })
  .then(response => response.json())
  .then(json => {
    this.setState({ result: json })
  })
}
```

通信完成

1. submitボタンを押す
2. `POST`
3. サーバからレスポンス
4. JSONに変換
5. stateの`result`に保存

`Dev-Tool`のReactタブで確認！

できた~~~~！！

ライブラリを使ってみる

- Reactの良いところはコンポーネントが世界中の開発者によってたくさん公開されているところ
- うまく使って工数を削減していきたい
- 下手に古いライブラリとか使うと動かないかも！
 - 最終更新がいつか
 - ★が多いか をチェック

npmのライブラリはnpmjs.comで検索できる

react-json-viewを使ってみる

```
yarn add react-json-view
```

```
import ReactJson from 'react-json-view'
```

```
<ReactJson src={this.state.result} />
```

を `button` タグの後に入れてみる

- Propsとして `src` で渡したJSONの構造が表示される

おわり！

お疲れ様でした

- 短い間に詰め込んだのでかなり雑な部分があったと思いますが、「何となく」理解することは今後の学習において重要だと思います
- 各概念や細かいAPIなどの使い方については今回出てきたキーワードを元に検索してみてください