Roppongi.js#3

# TypeScript in Wantedly

29.May.2018 - Kento Moriwaki

# TypeScript導入しました🎉🎉🎉

# 背景

1. Wantedly内の会社ページのリニューアルした
2. SSRを導入する必要があり、全体の構成を見直した
3. もともと型がほしい声は多かったので、いいタイミングだと思い導入した

# 入れてよかった 

# 1. 安心してコードを変更できる

- Storeの構造を変えたりしたとき、変更すべき箇所がすぐにわかる

# 2. 補完が効く

- タイピング数も減るし、Typoも減る

# 3. エラーが減る

- undefinedエラーがほぼ無くなる

# こうやって導入しました

# 1. `tsc`でビルドしない

# ビルド周りはやはりめんどくさい

- webpackにloaderを追加して、それをSSR用のビルドも変更して、jestで.tsをビルドする方法を調べて。。設定周りはやはりめんどくさい。

# 結局babelは必要

- polyfill入れたり、react-hot-loaderのプラグインや、styled-componentsの最適化したり、TypeScriptのビルド結果をbabelにかける必要がどうしても出てくる。

実は、babelだけでTypeScriptのビルドができる

# @babel/preset-typescript

- babelのtransformで、TypeScriptのシンタックスを削除してくれる素晴らしいやつ
- https://github.com/babel/babel/tree/master/packages/babel-preset-typescript

# やることは .babelrc に一行加えるだけ

- presets: ["@babel/preset-typescript"],
- webpackの変更はいらない、jestもそのまま動く

# 注意点

## babel 7(beta)から使える

- 半年以上betaの開発が続いている
- 7.0.0-beta.49まで進んでいるけど

## 完全な互換性は持たない

- 基本的には、TypeScriptのコードから型アノテーションを削除しているだけ
- `namespace`,`const enum`などの機能が使えない
  - workaroundはちゃんとある
- `<number>`などのキャストでできない
  - JSXとの区別ができない？`as number`を使えばいい

## TypeScriptは型チェックのみ

# `tsc --noEmit`で型チェックのみ行う

- linterなどと同じレベルの扱い
  - 型エラーがあるとCIがちゃんと落ちるようにする
- 開発中は型が間違っていても、ビルドはちゃんとできる

# 2. typescript-fsa

# Reduxのaction, reducer周りどう書いたらいいのかベストプラクティスがわからない

- 調べたらいろんな書き方が出てくる
- Redux v4で型定義がよくなるらしい(当時)
  - つまり今はよくないのかな？
  - (v4はすでにリリース済みですがどう変わったのかは知らない)
- レールが欲しい！

# そんなときに出会ったのが、TypeScript FSAでした

- https://github.com/aikoven/typescript-fsa
- Action Creator library for TypeScript. FSA-compliant.
- すごく薄いUtility
- familyがいくつかある
  - typescript-fsa-reducers
  - typescript-fsa-redux-thunk
  - typescript-fsa-redux-saga

# redux-actionsと似ている

```
import actionCreatorFactory from 'typescript-fsa';

const actionCreator = actionCreatorFactory();

// Specify payload shape as generic type argument.
const somethingHappened = actionCreator<{foo: string}>('SOMETHING_HAPPENED');

// Get action creator type.
console.log(somethingHappened.type);  // SOMETHING_HAPPENED

// Create action.
const action = somethingHappened({foo: 'bar'});
console.log(action);  // {type: 'SOMETHING_HAPPENED', payload: {foo: 'bar'}}
```

# 非同期なActionの定義

開始、終了、失敗のアクションを同時に作ってくれる

```
import actionCreatorFactory from 'typescript-fsa';

const actionCreator = actionCreatorFactory();

const doSomething =
  actionCreator.async<{foo: string},   // parameter type
                      {bar: number},   // success type
                      {code: number}   // error type
                     >('DO_SOMETHING');

console.log(doSomething.started({foo: 'lol'}));
// {type: 'DO_SOMETHING_STARTED', payload: {foo: 'lol'}}

console.log(doSomething.done({
  params: {foo: 'lol'},
  result: {bar: 42},
}));

console.log(doSomething.failed({
  params: {foo: 'lol'},
  error: {code: 42},
}));
```

# typescript-fsa-reducers

reducer関数に、.caseというメソッドが生え、switch文の代わりにそれを使う

第一引数にfsaアクションを渡し、第二引数にその場合のreducer処理をかく。この引数がきちんと型が決まったものが渡ってくる

```
import { reducerWithInitialState } from "typescript-fsa-reducers";
import { somethingHappened, doSomething } from "./actions";

const reducer = reducerWithInitialState(INITIAL_STATE)
  .case(somethingHappend, (state, { foo }) => ({ ...state, foo }))
  .case(doSomething.started, (state, payload) => ...)
  .case(doSomething.done, (state, payload) => ...)
  .case(doSomething.failed, (state, err) => ...)
```

# typescript-fsa-redux-thunk

```
import { bindThunkAction } from "typescript-fsa-redux-thunk";

const loadCompanyPosts = actionCreator.async<RootState, string, CompanyPostsResponse, Error>(
  "LOAD_COMPANY_POSTS"
);
const loadCompanyPostsWorker = bindThunkAction(loadCompanyPosts, async (id, dispatch) => {
  const req = requestToLoadCompanyPosts(companyIdOrName);
  const res = await dispatch(req);
  if (isFailure(res)) {
    throw res.payload.body;
  }
  const json = res.payload.body;
  return json;
});
```

bindThunkActionで、非同期なActionを囲って処理を実装する

返り値が結果のpayloadになる、dispatchの結果をisFailure, isSuccessで評価する

# request-creator(internal)

```
export const requestToLoadSimilarCompanies = requestCreator<number, Response, Error>(companyId => {
  return {
    path: `/api/v2/companies/${companyId}/similar_companies`,
    query: { ... }
    },
  };
});

// In action.
const res = await dispatch(requestToLoadSimilarcompanies);
if (isFailure(res)) {
  throw res.error;
}
```

- fsa形式でAPIリクエストを定義できる

# 3. Componentの型定義

# コンポーネントごとに毎回型定義を書くのはめんどくさい

- ほぼ同じようなpropsを受けている場合は、使いまわしたい
- 例えばCompanyがあったら、どのフィールドが必要かはコンポーネントによるが、どういう型なのかは共通のはず

# WtdModels

ドメイン内で共通なEntityを定義していく

APIは、Railsのactive-model-serializersに依存しているので、各Serializerに対応した型を定義する

```
declare namespace WtdModels {
  export type PostCategory = "feed" | "post_article";
  export type PostState = "draft" | "published";

  export interface Post {
    id: number;
    category_cd: PostCategory;
    state: PostState;
    title: string | null;
    created_at: Date | null;
    published_at: Date | null;
    ...
  }
}
```

# WtdModels.Partial

ドメイン内で共通なEntityをもとに、そのComponentで必要なフィールドをWtdModels.Partialを使って宣言する。

{Component}Modelsという名前にその型をおいて、コンポーネントと一緒にexportするルール

```
type Company = WtdModels.Partial<
  WtdModels.Company,
  "name" | "url" | "mission_statement",
  {
    avatar: WtdModels.HasOne<Image>;
  }
>;

export interface CompanyCardModels {
  Company: Company;
}

const CompanyCard: React.SFC<{ company: Company }> = props => {
  ...
}
```

# WtdModels.Union

複数のComponentを組み合わせたComponentでは、それぞれの{Component}Modelsの定義を`WtdModels.Union`で結合して型定義する。

中身は、基本的に`&`しているだけ。

```
type Company = WtdModels.Union<
  CompanyCardModels.Company,
  CompanyHeaderModels.Company
>;

const CompanyPage: React.SFC<{ company: Company }> = props => {
  return <>
    <CompanyHeader company={company} />
    <CompanyCard company={company} />
  </>
}
```

# まとめ

# まとめ

1. TypeScriptは生産性高い
2. Babel 7を使えば設定も楽
3. typescript-fsaがおすすめ
4. 型定義が楽にかけるように工夫していこう

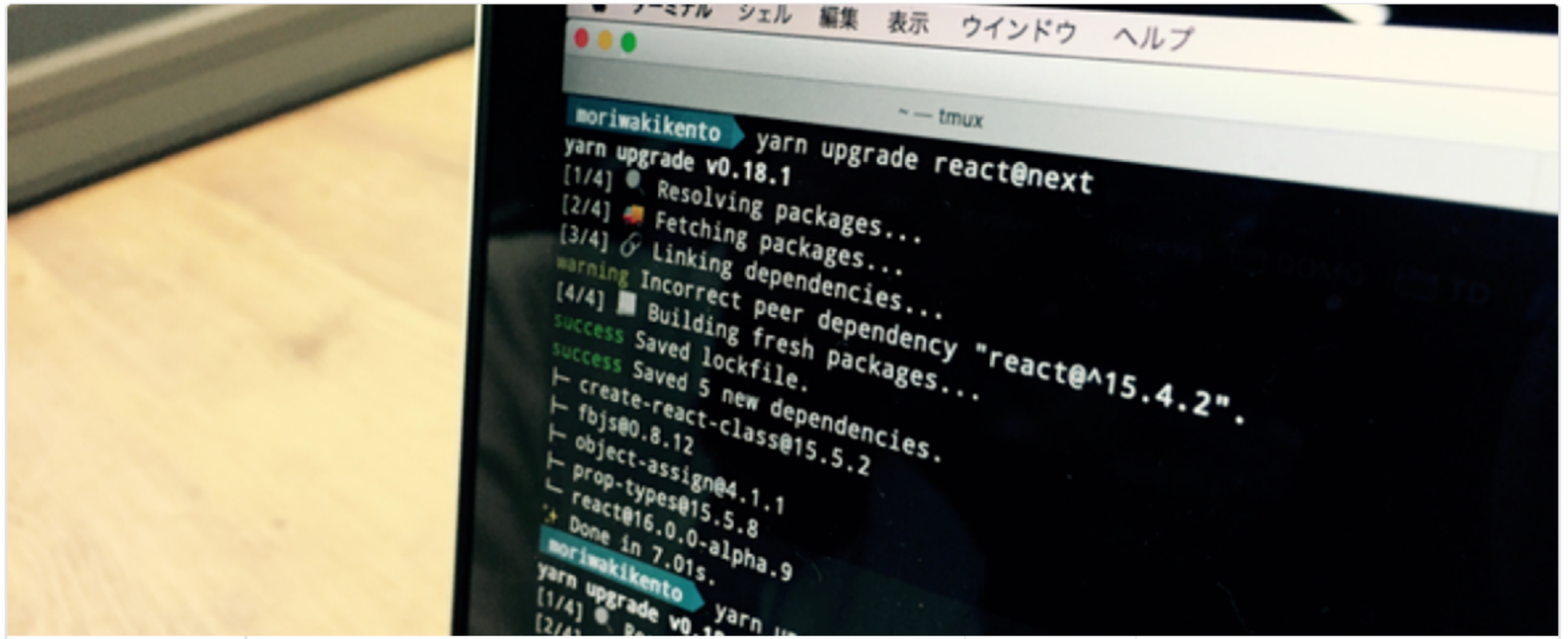
# 最後に

Web エンジニア

## モダンな環境でReactを書きたいエンジニアWanted！！

Wantedly, Inc.

埋め込む　いいね！ 34　ツイート　B! Bookmark 0




2016/06/29　社員との共通のつながり (561)　+ 556　2556　応援する 113

we are hiring!
https://www.wantedly.com/projects/59809