TypeScriptユーザーが 1週間Flowで開発してみた

TSKaigi Mashup #5 any

セイ

自己紹介

- セイ
- 25卒 FE / TypeScript・Next.js
- 普段は TypeScript ユーザーです
- 今回は好奇心で Flow を触りました

React のソース、
読んだことありますか？

実際の React のコード

ReactFiberHooks.js

```
function mountState<S>(
  initialState: (() => S) | S,
): [S, Dispatch<BasicStateAction<S>>] {
  const hook = mountStateImpl(initialState);
  const queue = hook.queue;
  // ...
}
```

拡張子は `.js` なのに、ジェネリクスや型注釈がある。これ何？

facebook/react · ReactFiberHooks.js

調べたら **Flow**。

存在は知ってたけど、
あまり主流じゃないっぽい。

なんで React は Flow？

- Flow も React も **Meta 製**
- React コア自体が **Flow で書かれている**

気になったので、
実際に Flow で開発してみた。

Flow って今どうなってる？

Flow の現在地（超ざっくり）

- JS に静的型チェックを足すツール（別言語ではない）
- Meta がメンテ継続中（2026年に Rust 移植など）
- 外の世界では TypeScript 一強、という前提で話します

今日は「Flow を使え」ではなく、触ってみた感想の話です。
批判したいわけではなく、TS ユーザーとしての体験談です

Flow のコードはどう動く？

書いたコード

src/sum.js

```
// sum.js
/* @flow */

function sum(a: number, b: number): number {
  return a + b;
}

console.log(sum(1, 2));
// console.log(sum(1, '2')); // 型エラーになる
```

Babel で変換したコード

dist/sum.js

```
// sum.js
function sum(a, b) {
  return a + b;
}

console.log(sum(1, 2));
```

```
pnpm exec babel src --out-dir dist
```

Flow は型チェック、実行時は型注釈を取り除いた JavaScript

何を作ったか

flow-sample

illionillion/flow-sample

デモを開く

- Todo 追加 / 完了 / 削除 / インライン編集
- `localStorage` 永続化

ビルドの流れ:

1. **Flow** で型チェック
2. **Babel** で型を剥がす
3. **esbuild** でバンドル

感想① 型の具体例

TypeScript でよくやることを、
Flow でもやってみた。

① opaque type ≈ branded type

TSKaigi で話題になってた branded type、
Flow だと **opaque** でできた。

TodoId

```
export opaque type TodoId: string = string;

export const createTodoId = (): TodoId => uuidv4();
```

↓ `Todo` の `id` に使う

Todo

```
export type Todo = {
  readonly id: TodoId,
  readonly name: string,
  readonly checked: boolean,
};
```

ファイル外から生の `string` を代入できず、 `id` は必ず `TodoId` になる

データ作成時にも使う

src/js/types.js

```
export const createTodo = (name: string): Todo => ({
  id: createTodoId(),
  name,
  checked: false,
});
```

`createTodoId()` 経由でのみ `TodoId` を作る

② Maybe (`?T`)

src/js/types.js

```
/**
 * Maybe type `?Todo` = `Todo | null | void`.
 * Not optional chaining (`?.`) — that is a runtime operator.
 */
export const findTodo = (
  todos: ReadonlyArray<Todo>,
  id: TodoId,
): ?Todo => todos.find((todo) => todo.id === id);
```

- `?Todo` = `Todo | null | void`
- **Maybe は「値」をオプショナルにできる**

TS の `?:` とは別モノ。
でも TS みたいに**キーにも使える**

TS の `?:` は `undefined` だけ、Flow の `?T` は `null` も含む

`?Todo`（値側）/ `draft?: string`（キー側）

③ Result + discriminated union

src/js/result.js

```
export type Result<out T, out E = string> =
  | { readonly ok: true, readonly value: T }
  | { readonly ok: false, readonly error: E };

export const ok = <T, E = string>(value: T): Result<T, E> => ({
  ok: true,
  value,
});

export const err = <T, E = string>(error: E): Result<T, E> => ({
  ok: false,
  error,
});
```

`ok` で成功 / 失敗を分けて、narrowing できる

境界で Result を返す

localStorage の JSON → 失敗しうるので Result で返す

src/js/storage.js

```
export const deserializeTodos = (
  raw: string,
): Result<Array<Todo>, string> => {
  let parsed: unknown;
  try {
    parsed = JSON.parse(raw);
  } catch {
    return err('invalid JSON');
  }
  return parseTodos(parsed);
};
```

呼び出し側でも使う

src/js/index.js

```
const initial = loadTodos();
let todos: Array<Todo> = initial.ok
  ? initial.value
  : [];
```

`initial.ok` で narrowing → `value` かフォールバック

EditState も discriminated union

src/js/types.js

```
export type EditState =
  | { type: 'idle' }
  | { type: 'editing', id: TodoId, draft: string };
```

`type` で narrowing できる

Type Challenges 風も Flow でできた

型が同じかを判定する `Equal` も書けた。

type-challenges/utils.js

```
type Equal<X, Y> =
  [X] extends [Y]
    ? ([Y] extends [X] ? true : false)
    : false;

type Expect<T extends true> = T;

// flow check で通る / 落ちる
type _ok = Expect<Equal<{ a: string }, { a: string }>>;
// type _ng = Expect<Equal<{ a: string }, { a: number }>>;
```

双方向の `extends` で「同じ型か」を見る。
illionillion/flow-sample · Equal

記法は TypeScript に寄ってきてる

現代 Flow でも使えるもの:

- `keyof` / `Omit` / `Partial` / `unknown` / `readonly`

旧記法は非推奨・削除が進んでいる:

- `$Keys` / `$Diff` / `mixed` / `+T` など

感想② エディタ・運用・ビルド

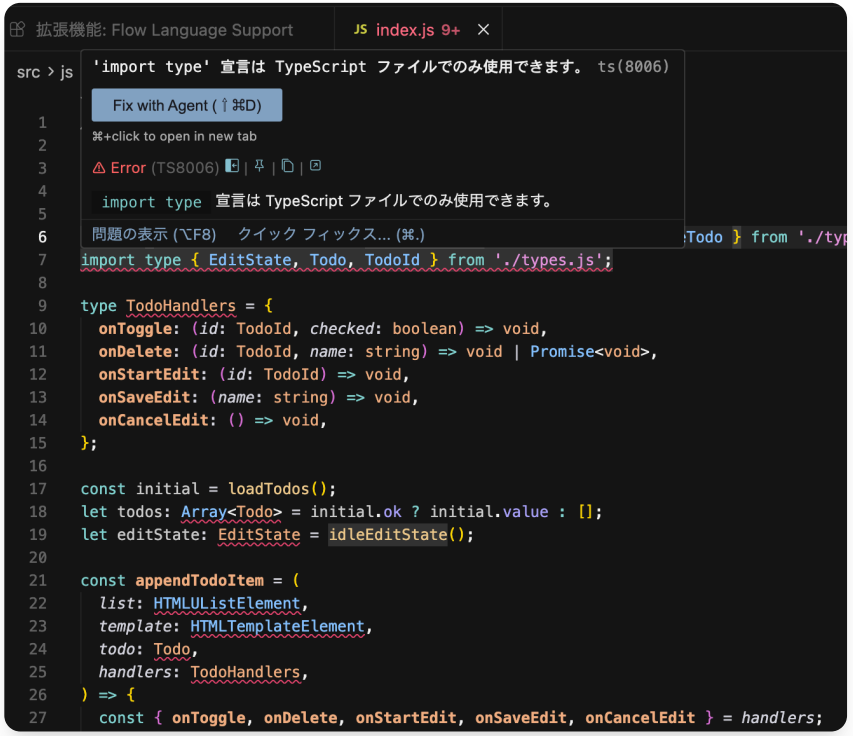
エディタまわり

1. `@flow` しても、そのままではエディタが対応してない
   - → Flow Language Support を入れた
2. TS の language server が誤判定してエラーを出す
   - → このリポだけ JS の validate を切った

.vscode/settings.json

```
{
  "javascript.validate.enable": false
}
```

初めて「vscode の設定をリポにコミットしてもいいんだ」ってなった



pre-commit / CI

- typecheck は `flow` でできる
- TypeScript と同じように
  - pre-commit
  - CI でチェック運用できる

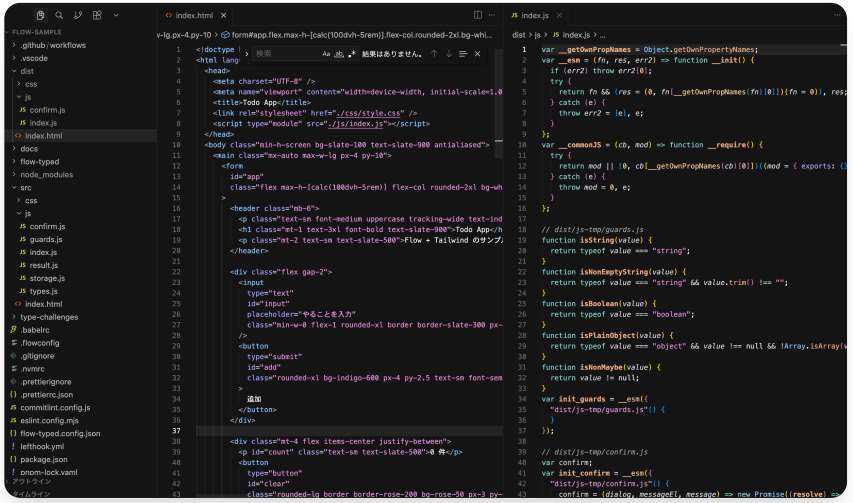
`flow check` を CI に載せるだけ

ビルドを自前で組んだ

```
Flow で型チェック
  → Babel で型を剥がす
  → esbuild でバンドル
  → HTML から JS / CSS を読む
```

型を剥がすだけじゃない。
Tailwind やライブラリも自分でビルドして、
HTML から読む構成まで組んだ。

仕組みは知ってたけど、初めて自前でやった



まとめ

まとめ

- Flow は今も Meta がメンテしている。アプリ開発の主流は TypeScript
- opaque / Maybe / Result など、触ると引き出しは増える
- エディタ周りは少し手間。check / CI の運用感は近い
- アプリ開発の第一選択は、今も **TypeScript** でよい

今日の話は Flow を貶める意図ではなく、隣を覗いた感想です

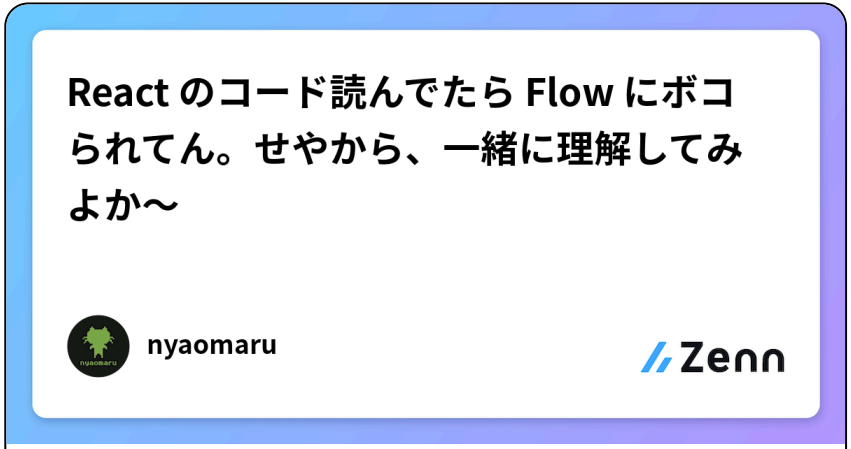
Thanks

github.com/illionillion/flow-sample

#tskaigi_msup

ご質問・懇親会で話しましょう

これ読むとわかりやすかったです



React のコード読んでたら Flow にボコられてん。せやから、一緒に理解してみよか〜

zenn.dev · nyaomaru