フロントエンド開発に関して

7110

0. JS の歴史
1. TypeScript を使おう
2. Zod を使おう
3. Vue を用いた Container/Presentational パターン
4. JavaScript と TypeScript の共存
5. Vue で JavaScript と TypeScript の共存

JS の歴史

～ JS の歴史は Web とブラウザの歴史～

その ①: JS の誕生
1995 年: Netscape 社が Web ページに動きを加えることを目的として開発され、Netscape Navigator というブラウザに搭載された。
※当時 Java が人気だったためあやかって JavaScript と名付けた。全くの別言語であることに注意。
※メロンとメロンパンとかよく言われる

その ②: ブラウザ戦争と標準化
1996 年: Microsoft が Internet Explorer（IE ※ちょっと前に終了したよね）というブラウザに JavaScript 互換の JScript を搭載した。
それぞれで開発が進むごとに互換性の問題が発生した。（※ブラウザ戦争）
1997 年: これを解決するために「標準化団体 ECMA」が各ブラウザが準拠すべき標準として「ECMAScript（ES）」を策定した。
当時の JavaScript は簡単な動作を可能にする程度の物だった。
また、モジュール機能が無かったので、`<script>` タグを複数個書く必要があった。
その結果グローバルな名前の衝突や実行順序の問題が出ていた。

その ③: Mozilla の設立
1998 年: 買収を機に Netscape 社がオープンソース路線として Mozilla を設立。
2004 年: オープンソースブラウザの Firefox をリリースし、Microsoft（IE）の独占から Web を守った。

その ④: GoogleMap の誕生と Web2.0
2005 年: Google が Google Maps をリリース。JavaScript が今までの簡単な動作だけの言語から Web アプリケーション全体を構築する言語になった。
AJAX（エージャックス）と呼ばれる非同期通信の技術でページ遷移の無いインタラクションを実現したことから画期的に手法が広まった。
Web2.0 の始まり

その ⑤: Chrome の誕生と Node.js の誕生
2008 年: Google が「V8 JavaScript エンジン」を搭載した Chrome をリリース。
ブラウザ上で JavaScript を JIT コンパイル（.NET と Java も実は JIT コンパイル言語）することで、高速な動作を可能にした。
2009 年: Ryan Dahl（ライアン・ダール）というエンジニアが、V8 エンジンを利用して、サーバーサイドで JavaScript を動作させる環境として Node.js を開発。
Node.js は JavaScript をサーバーサイドで動作させるために「CommonJS」という独自仕様を用いた。
※CommonJS はモジュール管理に `require()` と `module.exports` を用いていた。

その ⑥: モジュール標準化と JavaScript ツール
2015 年: ECMA が ES6 で `import` と `export` を使ったモジュール管理の仕様を標準化
ブラウザと Node.js は ECMA の後追いなので、対応が遅れる。
ブラウザが対応に遅れているので、開発者は ES6 で開発したものを ES5 に変換（トランスパイル）したり、JS モジュールの依存関係を解決しまとめる（バンドル）しながら開発しなければいけなかった。
バンドラー（Webpack 等）やトランスパイラー（Babel 等）が流行した。

その ⑦: ESM 対応の広がり
2017 年頃から主要ブラウザに ESM のモジュール管理サポートが広がる `<script type="module">` ← これ。
とはいえ、モジュールの取得効率は依然として課題があるので、バンドラー文化は残っている。
dist の中見ると 1 枚の min.js にまとめられているよね。

2020 年頃から CommonJS でも ESM サポートが広がる。
※CommonJS の対応が遅れたことにより、現代の Web 開発には課題が残っている。

その ⑧: SPA フレームワークの台頭と残された課題
現代において、React/Vue 等を用いた SPA 開発は主流になった。
これらのアプリケーションはブラウザ上で動くが、ビルドや開発はブラウザ上ではなく、Node.js 上で行われる。
しかし、Node.js は CommonJS ベースなので、ブラウザ主流の ESM との間に互換性問題が残っている。
※2020 年頃はマジで面倒くさかったが、現在はビルドツールが発達したことで割と解消されてそうな気がする。

その ⑨: TypeScript の誕生
2015 年: Microsoft が TypeScript をリリース。
JavaScript は動的型付け言語なので、保守性が低い（※実装者依存になってしまう）。
TypeScript の登場で、開発時に静的解析ができ、堅牢なコードを作ることが可能になった。

TypeScript を始めよう

TypeScript はあくまでも JavaScript の構文に静的型解析を追加されただけであり、全くの別言語というわけではない。
実行自体は JavaScript エンジンで行われるので、TypeScript は JavaScript にトランスパイルされる必要がある。
そのために `tsc` 等の TypeScript トランスパイラーを使い、JavaScript コードへビルドする必要がある。
なので、JavaScript でできないことは TypeScript でもできない。

TypeScript の基本構文は「変数: 型」という感じ

```
const name: string = '7110'

function greet(name: string) {
  console.log(`hello ${name}`)
}
```

オブジェクトもいける

```
type User = {
  name: string
  age: number
}

const user: User = {
  name: 'naito',
  age: 27,
}

function greet(user: User) {
  console.log(`hello ${user.name}`)
}
```

型は定義なしでも行けるが、定義した方が良い

```
const user: {
  name: string
  age: number
} = {
  name: 'naito',
  age: 27,
}

function greet(user: { name: string; age: number }) {
  console.log(`hello ${user.name}`)
}
```

TypeScript の静的型解析によって、開発時に補完が効き、明らかに動かないコードを書きづらくなる

```
type RoleType = 'admin' | 'general' | 'guest'

function doSomething(role: RoleType) {
  switch (role) {
    case 'admin':
  }
}
```

※ `|` で型をつなぐと ○○ または △△ の型となる(Union 型)
上の例で `RoleType` は `admin` または `general` または `guest` の文字列が入る型ということになる。

`as const` キーワードで、ハードコードにも型がつく

```
const prefectures = ['hokkaido', 'tokyo', 'tokyo', 'kanagawa'] as const

// ×: tokyoはprefecturesの1要素目ではないため開発時にエラーが出る
if (prefectures[0] === 'tokyo') {
}

// ○: こっちは値通りなのでOK
if (prefectures[0] === 'hokkaido') {
}
```

TypeScript の型は脆い

TypeScript では `as ○○` とすると ◯◯ 型として使えてしまう。(※型アサーション)

これをやるとその型として TypeScript が扱えてしまうので、静的型解析の安全性を受けられなくなってしまう。(C#でも同じだけど...)

linter の設定ではじくのがおすすめ

```
type User = {
  name: string
  age: number
}

const user: User = {} as User
```

TypeScript は「Nominal Typing（公称型）」ではなく「Structural Typing（構造型）」なので 同じ構造の型であれば、**異なる型名でも代入できてしまう**

```ts
type UserType = {
  name: string
  age: number
}

type PersonType = {
  name: string
  age: number
}

function greetUser(user: UserType) {
  console.log(`hello ${user.name}`)
}

const user: UserType = { name: 'naito', age: 27 }
const person: PersonType = { name: 'shigehito', age: 27 }

greetUser(user) // ✅ OK
greetUser(person) // ⚠️ 通る（構造が同じだから）
```

Zod を使おう

Zodとは、JavaScript のスキーマベースのバリデーションライブラリ

```
// スキーマを定義
const userSchema = z.object({
  name: z.string(),
  age: z.number().int().nonnegative(),
})

// スキーマから型を生成
type UserType = z.infer<typeof userSchema>

// 通らないので例外が発生
userSchema.parse({ name: '7110', age: -1 })

// OK
userSchema.parse({ name: '7110', age: 100 })
```

Zod には BrandedTypes という機能がある。

```
const userNameSchema = z.string().brand<'UserNameBrand'>()
const userAgeSchema = z.number().int().nonnegative().brand<'UserAgeBrand'>()
const userSchema = z
  .object({
    name: userNameSchema,
    age: userAgeSchema,
  })
  .brand<'UserBrand'>()

type UserNameType = z.infer<typeof userNameSchema>
type UserAgeType = z.infer<typeof userAgeSchema>
type UserType = z.infer<typeof userSchema>
```

Brand を使うと「{ __brand: "ブランド名" }」が型に追加される

```
type UserNameType = string & { __brand: 'UserNameBrand' }
type UserAgeType = string & { __brand: 'UserAgeBrand' }
type UserType = string & { __brand: 'UserBrand' }
```

つまり、TypeScript の型に Nominal Typing の要素をつけてくれる。

```
function greet(userName: UserNameType) {
  /*  略 */
}

const itemName: string = 'ネジ'

// 型がただのstringなので静的解析に引っかかる
greet(itemName)
```

ZodBrandedTypes は状態オブジェクトの定義にも使える
例えば「納品」=> 「検品」 => 「出荷」の場合
※JavaScript へトランスパイル時に型情報は取り除かれる。
brand プロパティ がついたまま実行されることはないので、開発時だけの便利ツールとして利用できる。

```ts
const deliverableItemSchema = z.object().brand<'DeliverableItemBrand'>()
const inspectedItemSchema = z.object().brand<'InspectedItemBrand'>()
type DeliverableItemType = z.infer<typeof deliverableItemSchema>
type InspectedItemType = z.infer<typeof inspectedItemSchema>

// 受入は納品物だけ
function recieve(item: DeliverableItemType) {}

// 検品できるのは納品物だけ
// 戻り値を検品物の型にする
function inspect(item: DeliverableItemType): InspectedItemType {
  //検品処理
}

// 出荷できるのは検品した品目だけ
function ship(item: InspectedItemType) {}
```

Vue を用いた Container/Presentational パターン

Container/Presentational パターンとは 1 コンポーネントを 2 種類に分ける設計手法

1. Container
   状態・振る舞いに関心を持つ
   ※style は書いてはいけない
2. Presentational
   UI の描画に関心を持つ
   ※script の中に状態やビジネスロジックを持ってはいけない
   ※状態更新は全て emit する

```
<script setup lang="ts">
  import { getUserList } from '@/core/repositories'
  import { UserListPagePresenter } from '.'

  const userList = ref<UserType[]>([])
  const selectedUser = ref<UserType|null>(null)

  function onSelect(user: UserType) {
    selectedUser.value = user
  }

  onMounted(async() => {
    const userList.value = await getUserList()
  })
</script>
<template>
  <UserListPagePresenter
    :user-list="userList"
    @click="onSelect" />
</template>
```

container.vue

```
<script setup lang="ts">
  definePropsとに
    userList: Readonly<UserType[]>
  }>()

  const emits = defineEmits<{
    click: [userId: UserType['userId']]
  }>()
</script>
<template>
  <div :class="$style['user-list__container']">
    <h2>Users</h2>
    <div
      v-for="user in userList"
      :key="user.userId"
      :class="$style['user-card']">
      <UserCard
        :user="user"
        @click="emits('click', $event)" />
    </div>
  </div>
</template>
<style lang="css" module>
  /* □ */
</style>
```

presenter.vue

メリット

- 状態が 1 つに集約されるので、ビジネスロジックも分散しない。
  状態とビジネスロジックを親に集めていない場合は、子コンポーネント内で computed x watch 地獄が始まる
- UI の変更に状態等を管理しているファイルが影響を受けにくい。
- 責務が分散されるので、テストコードが書きやすい
  - container => ロジックのテスト
  - presenter => 描画のテスト
    ↑ これだけで単体テストが完了する

デメリット

- props が増える
  - domain ごとの composables を作ってさらに責任を分離することもできるが、カオスになるので、ルール策定が必要
  - UI の描画に関わる ref が増えて、それが props と emit を増やしがち
    - フォームを内蔵した Modal とかね...
    - composables でカバーできるし、していい気がするが、正しいか判断がついていない...
      ※人事評価では決めきれず、props 地獄へ突入した
- ファイル数が増える
- 教育コストが増える

個人的に、さらに分けて運用している

1. ページ専用の composables
   状態・ビジネスロジックに関心を持つ
2. Container
   ビジネスを表現するページ(Presentational)のレンダリングに関心を持つ
   基本的には composables と Presentational のパイプだが、取得できない場合に NotFound を表示したり、権限エラーの場合に Forbidden を表示するのは Container の責任
   ※style は書いてはいけない
3. Presentational
   UI の描画に関心を持つ
   ※script の中に状態やビジネスロジックを持ってはいけない
   ※状態更新は全て emit する

```
export default function useUserListPage() {
  const userList = ref<UserType>([])
  const selectedUser = ref<UertType | null>(null)

  const { authorized } = useAuth('user:list')

  async function onLoad() {
    userList.value = await getUserList()
  }

  function onSelect(user: UserType) {
    selectedUser.value = user
  }

  onMounted(() => {
    onLoad()
  })

  return {
    authorized: readonly(authorized),
    userList: readonly(userList),
    selectedUser: readonly(selectedUser),
  }
}
```

```
<script setup lang="ts">
  import { UserListPagePresenter, useUserListPage } from '.'
  const { authorized, userList, onSelect } = useUserListPage()
</script>
<template>
  <ForbiddenPage v-if="!authorized" />
  <UserListPagePresenter
    v-else
    :user-list="userList"
    @click="onSelect" />
</template>
```

```
<script setup lang="ts">
  definePropsとに
    userList: Readonly<UserType[]>
  }>()

  const emits = defineEmits<{
    click: [userId: UserType['userId']]
  }>()
</script>
<template>
  <div :class="$style['user-list__container']">
    <h2>Users</h2>
    <div
      v-for="user in userList"
      :key="user.userId"
      :class="$style['user-card']">
      <UserCard
        :user="user"
        @click="emits('click', $event)" />
    </div>
  </div>
</template>
<style lang="css" module>
  /* □ */
</style>
```

この構成のメリット

- 曖昧だった Container と Presentartional の責務がさらに分離
  - 今までは Forbidden の描画が Presentational でやるべきでは感があった...
- ビジネスロジックと常体のテストを vue から切り離すことができたのでロジックのテストが vue のライフサイクルに縛られなくなった。
  - ※onMounted とかあるので、ちょっと工夫は必要
- readonly する際に新たに 変数 を定義する必要が無くなったのも地味に嬉しい

JavaScript と TypeScript の共存

tsconfig.ts の設定で JavaScript と TypeScript を共存させることが可能

```
{
   /* 略 */
   "compilerOptions": {
      /* 略 */
      "allowJs": true, // JSを許可する
      "checkJs": false, // JSでの型チェックをしない
      "strict": false, // 厳格な型チェックをしない
      "isolatedModules": false // 解析できないモジュールを警告しない
   },
   "include": [
      /*   略 */
      "src/**/*.js" // jsをincludeする
   ]
}
```

この環境、以下を install することで遊べます

- node >= 20
- pnpm@10.11.0

```
pnpm install
pnpm build
```

おしまい！！