フロントエンド

2025-08-18

id:mizdra

はじめに

- 「Webフロントエンド」と聞いて、何を思い浮かべますか?
- 「Webフロントエンド」の開発をする上で、必要な知識は何だと思いますか?

必要な知識の一例

- Webページ/ブラウザ/サーバの関係性・役割
- HTML/CSS/JavaScript (言語)
- React, Next.js (View ライブラリ/フレームワーク)
- XSS, CSP (セキュリティ)
- Web Vitals (パフォーマンス)
- アクセシビリティ
- bundler, linter, formatter, test runner (開発ツール)

フロントエンドへのよくある印象 ①

- 「覚えることが多い」
  - そうかもだが、フロントエンド以外の領域でも同じだと思う
- アクセシビリティ/言語ツールは、この領域特有かもしれないが...
  - 言語/ライブラリ/パフォーマンスの知識は、他の領域でも必要
- あんまり恐れる必要ない

フロントエンドへのよくある印象 ②

- 「技術の流れが早い」
  - そうかも
- フロントエンドはユーザに近くて、コーディング人口が多い領域
  - そのため、頻繁に新技術が出てくる [1]
- 向き合い方を変えるべき
  - 流行を追うのも良いけど...その技術が登場した背景を考えよう
  - ライブラリの使い方を覚えるのも良いけど...長く通用する知識も身につけよう

---

[1] 諸説あります。

この講義のゴール

- フロントエンド開発で必須の知識を押さえる
- それぞれの技術が登場した背景を知る
- 長く通用する知識を身につける

講義が終わった後も、Webフロントエンドを学べるような手助けになれば良いと思っています。

JavaScript について(10min)

JavaScript について

- 皆さんが実際にコードを読む際に知っておいて欲しい JavaScript 上の概念について紹介します

変数宣言

```
// 変数と宣言
let a = "a" // let は上書き可能
const b = "b" // 上書き不可能

a = "A" // OK
b = "B" // Cannot assign to "b" because it is a constant
```

- 変数宣言は const をできるだけ使うと良い
  - 変数の値が変わることを考慮しなくて済む

プリミティブ型 / オブジェクト

```
const id1 = "1234" // string
const id2 = '1234' // string
const name = null // null
const age = 2022 // number
const isAdmin = false // boolean

// object
const user = {
  id: "1234",
  username: null,
  age: 2022,
  isAdmin, // `isAdmin: isAdmin` の省略形
}
user.age // 2022
// 未定義プロパティだと undefined が返る
user.abc // undefined
```

関数 / 配列

```
// 関数
function add(a, b) {
  return a + b
}
add(1, 2) // 3

// 配列
const array1 = [1, 2]
```

```
// 配列操作
array1[0] // 1
for (elm of array1) {
  console.log(elm)
}
array1.forEach((elm) => {
  console.log(elm)
})
array1.map((elm) => elm * 2) // [2, 4]
```

Arrow Function

- 簡潔に関数を書くための構文

```
const add = (a, b) => {
    return a + b
}

// 1行だけなら return などを省略可
const add = (a, b) => a + b

// 引数が 1 つのときは引数を囲う `()` も省略可
const hello = name => `Hello, ${name}`

// 返り値がオブジェクトのときは `()` で囲う必要がある
const getProps = () => ({ a: "foo", b: "bar" })
```

typeof 演算子

- 値の実行時の型を返す演算子

```
const str = "hello world"
console.log(typeof str) // 'string'
console.log(typeof 10) // 'number'
console.log(typeof { name: "jonh", age: 20 }) // 'object'
console.log(typeof undefined) // 'undefined'
console.log(typeof ["a"]) // 'object'
console.log(typeof null) // 'object'
```

in 演算子

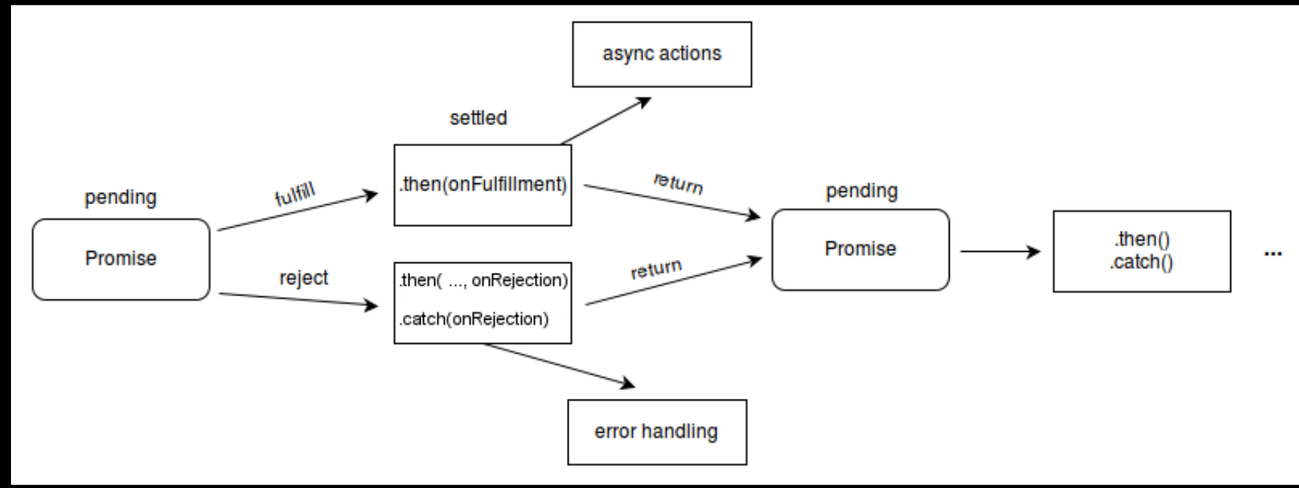
値に特定のプロパティが有ることを判定する演算子。ブラウザが特定の API に対応しているかどうかを識別することにも使われる。

```
const user = { name: "hatena", age: 20 }
if ("name" in user) {
  console.log("user has name property")
}

if ("fetch" in window) {
  console.log("This browser supports fetch API")
}
```

Promiseについて

- 非同期処理を抽象化したオブジェクト[2]
- 3つの状態を表現できる
  - Pending: 初期状態。成功も失敗もしていない。
  - Fulfilled: 非同期処理が成功した
  - Rejected: 非同期処理が失敗した
  - まず Pending になって、その後 Fulfilled or Rejected になる



[2] Promise は非同期処理を扱うためのオブジェクトで、JavaScript の非同期処理の基本的な仕組みです。詳細は付録「Promise について」を参照してください。

Promise の生成方法

- `new Promise((resolve, reject) => {...})` で Promise を生成
- `resolve` を呼ぶと `Fulfilled` になり、`reject` を呼ぶと `Rejected` になる

```
function sleep(ms) {
  return new Promise((resolve, reject) => {
    if (typeof ms !== "number") {
      reject(new Error("ms must be a number"))
      return
    }
    setTimeout(() => {
      resolve(ms)
    }, ms)
  })
}
```

コールバックの登録

- 非同期処理が完了した時に呼ばれるコールバック関数を登録できる
  - then メソッド: Fulfilled になった時に呼ばれるコールバックを登録
  - catch メソッド: Rejected になった時に呼ばれるコールバックを登録

```
sleep(1000)
  .then((ms) => console.log(`sleep: ${ms}ms`))
  .catch((e) => console.error(e))
```

fetch を使った例

- fetch は HTTP リクエストを送信する API
  - リクエスト送信は非同期処理なので、Promise を返すようになってる

```
const promise = fetch("https://api.example.com/user/1") // ①
  .then((res) => /* ② */ res.json())
  .then((json) => /* ③ */ console.log(json.user))
  .catch((e) => /* ④ */ console.error(e))
console.log(promise) // ⑤
```

- ① => ⑤ => ② => ③ => ④ の順に実行されることに注意
  - コールバック関数は非同期処理が完了後に実行される (遅延される)

async/await

- 非同期処理を簡潔に書くための構文
  - then や catch を使わずに、同期的なコードっぽく書ける

async/await

```
const getUser = async (id) => {
  try {
    const res = await fetch(`https://api.example.com/user/${id}`)
    const json = await res.json()
    const user = json.user
    console.log(`${user.name}のidは${user.id}`)
    return user
  } catch (e) {
    console.error(e)
    throw new Error("error reason: *********")
  }
}
```

エラーハンドリングには try {} catch (e) {} を使用します。

async/await

- async function 自体も暗黙的に Promise を返す
- 関数の返り値に対して、then を呼び出せる

```
getUser(1)
  .then(console.log) // {id: 1, name: 'hatena'}
  .catch(console.error)
```

ECMAScript Modules (ES Modules)

- プログラムを**モジュール**という単位に分割する機能
  - 1 ファイル == 1 モジュール
  - スコープはモジュールごと
  - 関数や変数などを import/export できる
- 歴史的経緯で CommonJS[3]という形式が Node.js などを中心に長く採用されてきたが、ECMAScript[4]で標準化された ES Modules が今では Node.js、ブラウザなどでも広くサポートされている[5]

---

[3] `module.exports` という特殊なオブジェクトを経由してエクスポートし、`require` 関数を利用してインポートする方式。https://nodejs.org/docs/latest/api/modules.html

[4] ECMAScript は JavaScript の仕様における名称。ECMAScript 仕様については付録「JavaScript やフロントエンド周辺技術の標準(化)について」で解説しています。

[5] 実際にはサポートしていないブラウザやパフォーマンスのために Babel を使って変換した後に、モジュールを解釈して 1 つ〜複数のファイルをまとめて配信するために webpack などを利用することが多い。詳細は付録「JavaScript をウェブアプリケーションで提供する際に使用するツールチェインについて」を参照してください。

named export, named import

- 変数宣言や関数宣言に export を付加すると named export できる

```
// lib.js
export const logLevel = {
  WARN: "warn",
  ERROR: "error",
};
export function log(message, level) {/* ... */}

// main.js
import { logLevel, log } from './lib.js'
```

default export

```
// lib.js
export default function (message, level) {/* ... */}

// main.js
import awesome from "./lib"
```

- `export default` というキーワードでも export できる
  - 名前を付けずにエクスポートできる
  - `export default` できるのは、1つのモジュールにつき１つだけ
- `import` 時に任意の名前を設定できる

import/export の細かい挙動

- as でリネーム出来る

```
import { logType as LOGTYPE } from "./namedModule"
```

- * as で export されているもの全てをオブジェクトにまとめる[6]

```
import * as Logger from "./namedModule"
Logger.hello() // 'hello'
```

[6] 必要なものだけを取り込むことで受けられる恩恵(webpack による TreeShakingなど)も多いので、基本的には * as は避ける方のがオススメ。

for...of

- 反復可能(iterable)なオブジェクトの要素を順番に取り出せる
  - 配列、文字列、NodeList、Map、Setなど

```
const iterable = [10, 20, 30]

for (const value of iterable) {
  console.log(value)
}
```

Nullish coalescing operator ??

- `a ?? b` のようにして使う
  - 左辺が `undefined` or `null` の時に右辺の値を返す
  - それ以外なら左辺の値を返す
- デフォルト値に fallback させるのに便利

```
function greet(name) {
  return `Hello, ${name ?? "mizdra"}!`
}
```

Optional chaining ?.

- プロパティアクセス (a.b) の亜種で、a?.b のように書く
- a が null または undefined のときは undefined を返す
  - それ以外のときは通常通りプロパティアクセスを行う

```
const userId1 = session.user?.id;
const userId2 = session.user ? session.user.id : undefined;
```

- 関数呼び出しとも組み合わせられる

```
const result = someObject?.someMethod?.(arg1, arg2);
```

Spread Syntax

- `...` を使うと配列やオブジェクトを展開出来る

```
const sum = (a, b, c, d) => a + b + c + d
const nums = [1, 2]
const copied = [...nums] // 中身を複製した配列を作れる

const moreNums = [...copied, 5] // [1, 2, 5]
sum(...nums, 10) // 13

const obj = { a: 10, b: "foo" }
const obj2 = { b: "bar", c: true }
// 2つ以上のobjectをmergeする。キーが重複している場合は後に書いた方で上書きされる

const merged = { ...obj, ...obj2 } // {a: 10, b: 'bar', c: true}
const getUserConfig = (received) => ({
  force: false,
  ...received, // デフォルト値を渡された値があれば上書きする

})
```

Rest Parameters

- 関数の引数も ... で受け取ることで可変な長さの引数を受け取れる
  - Spread Syntax と違って、１つだけ且つ引数の末尾でしか使えない

```
//
const sum = (num1, num2, ...nums) =>
  num1 + num2 + nums.reduce((a, b) => a + b, 0)
const numbers = [1, 2, 3]
sum(3, ...numbers, 6) // 15
```

TypeScript について(10min)

TypeScript について

- JavaScript に静的型付けを導入した言語
  - JavaScript + 型注釈
  - コンパイラ (tsc) で型チェックを行う
- 現代では生の JavaScript 書くより、TypeScript で書くことが多い

```
function hello(name: string): string {
  return `Hello, ${name}!`
}
const result = hello(1)
//                   ^
// Type Error: Argument of type '1' is not assignable to parameter of type 'string'.
```

なぜ TypeScript が必要か？

- 型エラーを未然に防ぐため
  - 実行した時ではなく、コードを書いてる時に気付けるように
- コードを変更しやすくするため
  - Rename/補完
- コードを読みやすくするため
  - 型がドキュメント代わりに
  - コードジャンプ

`tsc`: TypeScript compiler

- TypeScript 言語のためのコンパイラ
- 主な機能
  - 型チェックをする
  - TypeScript で書かれたコードを JavaScript に変換する
- 変換と言っても、型アノテーション等の削除くらい

型宣言部分を読めるようになろう

TypeScript の代表的な表現などを紹介していきます。

変数宣言時の型アノテーション

```
// JavaScriptの場合
const a = 'hello';
// TypeScriptの場合は変数名と=の間に:を置いて型アノテーションを書く
const a: 【ここに型アノテーション】 = 'hello';
// 例えば
const a: string = 'hello';
```

これくらいだったら推論されるので、普通は省略されます。

代表的な表現 ①

- プリミティブ型: string, number, boolean, null, undefined
  - 'hello', 12, true, false のような特定の値(リテラル型と呼びます)も使える

```
const a: number = 10
const b: boolean = false
const c: string = 11 // Type Error
const d: "hello" = "hello"
const n: null = null
```

- 配列

```
const arr: string[] = ["hello", "world"]
const arr2: Array<number> = [1, 2, 3, 5, 8]
const arr3: [string, number] = ["year", 2021] // タプル(Tuple)型とも
```

代表的な表現 ②

- オブジェクト
  - JavaScript のオブジェクト同様に書いて、値を書くところに型を書く
  - キー名に ? を付けるとオプショナルなキーになって、省略可能になる。[7]

```
const person: {
  name: string
  age: number
  address?: string
} = {
  name: "john",
  age: 21,
}
```

[7] `string | undefined` のような記述と同じと紹介されることもある。値が入っているかどうかで JavaScript として実行した際の振る舞いが変わることがある(`Object.keys()` など)ので、厳密には同じではないことに注意。特に TypeScript 4.4 で導入された `exactOptionalPropertyTypes` を有効にすると、tsc での型解析時の振る舞いも変わります。

type

type を使うと型にエイリアスを付けられる

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

type Team = Person[]
```

Union Type (合併型)

複数の型のいずれかを満たす

```
type Color = "red" | "green" | "blue" | "yellow" | "purple"

const c: Color = "red"
const d: Color = "black" // Type Error
```

Narrowing

緩い型をいくつかの型に絞り込んでから、絞り込まれたそれぞれに対して処理したいことがある。

```
function padLeft(padding: number | string, input: string): string {
  if (padding が number なら) {
    return " ".repeat(padding) + input;
  } else if (padding が string なら) {
    return padding + input;
  }
  throw new Error("unreachable");
}
```

一部の JavaScript の演算子を使うと、型の絞り込み (Narrowing) ができる。

typeof 演算子

- JavaScript にある演算子だが、TypeScript で使うと TypeScript の型の絞り込みもされる

```
function padLeft(padding: number | string, input: string): string {
  if (typeof padding === "number") {
    // このブロック内では `padding` は `number` 型
  } else if (typeof padding === "string") {
    // このブロック内では `padding` は `string` 型
  }
}
```

in 演算子

- これも JavaScript にある演算子
- TypeScript では、特定のプロパティを持つ型へと絞り込める

```
type Fish = { name: string, swim: () => void }
type Bird = { name: string, fly: () => void }
const move = (x: Fish | Bird) => {
  if ("swim" in x) {
    // Fish 型に絞り込まれる
    return x.swim()
  }
  // ここでは Bird 型に絞り込まれる
  return x.fly()
}
```

Tagged Union Types

- Union Type の個々の型に、`kind` のようなプロパティを持たせるテクニックがある
  - `x.kind === ...` で型の絞り込みができる
- `in` による絞り込みより堅牢な書き方で、おすすめ

```
type Fish = {
  kind: "fish"
  // ...
}
type Bird = {
  kind: "bird"
  // ...
}
const move = (x: Fish | Bird) => {
  if (x.kind === "fish") {
    return x.swim()
  }
  return x.fly()
}
```

as を用いた型アサーション(Type Assertion)

- TypeScript によって推論された型を上書きしたいときに使う
  - 型キャストではない(ランタイム上での振る舞いがなんら変わることはない) [8]
- 多くの場合は害になるので、本当に必要な場合だけ利用する
  - 例えば、古い JavaScript のコードを移植するなど

[8] 例えば `"str" as number` のように書いた時に、実行時に文字列から数値に型変換されたりする訳ではない。TypeScript の型システム上での型が `number` に変わるだけ。

as を用いた型アサーション(Type Assertion)

```
type Foo = {
  bar: number
  piyo: string
}
const foo1: Foo = { bar: 1, piyo: "2" } // OK
const foo2: Foo = {} // NG
const foo3: Foo =  {} as Foo // OK
```

const アサーション

as const とすることで変数代入時などに変更不可能としてアサーションしてくれる。

```
const a = [1, 2, 3] // aの型はnumber[]となる
const b = [1, 2, 3] as const // bの型はreadonly [1, 2, 3]となる
// readonly な配列には push や pop などの変更を加えるメソッドが存在しない
a.push(4) // OK
b.push(4) // NG


type Pallet = {
  color: Color
}
const setPallet = (p: Pallet) => {
  /* do something */
}

const pallet = {
  color: "red",
} // ここに as const を付けないと{ color: string }と推論されてエラーになる
setPallet(pallet)
```

不定な型を扱う方法

TypeScript 上で不定な型を扱うための方法を紹介します。

any

- どんな値でも入れられる型
- any 型に対する操作はtscで型エラーにならない

```
let anything: any = {}
// anyには何でも代入できる
anything = () => {}
anything = null
anything.split(",") // anyの場合はメソッドもなんでも参照できる
```

- Rust の unsafe のようなもの
  - 自由に書けるが、コンパイラは何も警告しない
- as 同様に避けられる場合は避ける

unknown

- any 同様にどんな値でも入れられる
- any と違い、unknown はプロパティアクセスが型エラーになる

```
const val: unknown = { name: "foo" };
val.name // Type Error: Property 'name' does not exist on type 'unknown'.
```

- unknown ≒ {} | null | undefined
- 型を絞り込んでからアクセスする必要がある

```
if (typeof val === "object" && 'name' in val) {
  console.log(val.name) // OK
}
```

関数

既にサンプルでは何度も出てきているけど、引数や返り値の型の書き方のパターンたち紹介しておきます。

```
const f = (x: string): number => {
  return x.length
}
// 特にreturnをしない場合は返り値にvoidを指定する
const a: () => void = () => {
  console.log("a")
}
// オプショナルな引数はkeyに?を付ける
// 推論されるもので良いなら返り値の型は省略可
const b = (n?: number) => `${n}`
// Rest Parametersを受け取る場合はこういう感じ
const c = (...texts: string[]) => {
  return texts.join("|")
}
```

型引数(Generics)

- 関数の返り値の型に関する制約を外から与えて、関数内部で利用できる。
- よくよく型定義などを見ると定義されていて利用可能だけど、気付いていないということもよくある…
  - TypeScript で querySelector メソッドを使うときに型引数を指定する - Hatena Developer Blog

```
const getJSON = <T>(url: string): Promise<T> => {
  // res.jsonはanyとならずに型引数で渡されたものと解釈される
  return fetch(url).then<T>((res) => res.json())
}
// ここでusersはUser[]になる
const users = await getJSON<User[]>("/api/users")
// ここでblogsはBlog[]になる
const blogs = await getJSON<Blog[]>("/api/blogs")
```

続: 型引数

- extends を使うと指定した型/インターフェースを満たすように指定できる
- 型引数をそのまま関数の引数の型に使うことで静的解析時に呼び出し元の引数の型から返り値を推論してくれる

```
const echo = <T extends string>(text: T): T => {
  return text
}

const a = echo("foo") // a の型は 'foo'
const str: string = "foo"
const b = echo(str) // b の型は 'string'
```

TypeScript の書き方で困ったら?

- 公式ドキュメントの Handbook を読もう
  - https://www.typescriptlang.org/docs/handbook/intro.html
- Playground で試し書きしよう
  - https://www.typescriptlang.org/play
- 難しい型の書き方は Type Challenge に結構載ってる
  - https://github.com/type-challenges/type-challenges

React について(20min)

React とは

- ユーザインターフェースを構築するための View ライブラリ
- UI を関数で定義する
  - 「仮想 DOM」と呼ばれるオブジェクトを返す
  - React がその仮想 DOM を元に、実際の DOM を更新する
- JSX という HTML-like な拡張構文を使う

```tsx
import React, { useState } from 'react';
export const TodoApp = () => {
  const [todos, setTodos] = useState<string[]>([]);
  const [newTodo, setNewTodo] = useState<string>('');

  const addTodo = () => {
    setTodos([...todos, { id: nanoid(), text: newTodo }]);
    setNewTodo('');
  };

  return (
    <div>
      <h1>やることリスト</h1>
        <ul>
          {todos.map((todo) => (
            <li key={todo.id}>
              {todo.text}
            </li>
          ))}
        </ul>
      <form onSubmit={(e) => { e.preventDefault(); addTodo(); }}>
        <input type="text" value={newTodo} onChange={(e) => setNewTodo(e.target.value)} />
        <button onClick={addTodo}>追加する</button>
      </form>
    </div>
  );
}
```

仮想(Virtual) DOM

- React の内部で持っている、実際の DOM[9]の対になる構造体
- 状態の変更を検知すると...
  - 変更前後の仮想 DOM の差分を計算し、その差分だけを実際の DOM に反映

https://ja.react.dev/learn/preserving-and-resetting-state

[9] Document Object Model の略。HTML や XML 文書の構造を操作するためのプログラミングインターフェイスのこと。

React の何が嬉しい?

React <=> 生の JavaScript で書いた時を比較するとよくわかる。

```tsx
import React, { useState } from 'react';
export const TodoApp = () => {
  const [todos, setTodos] = useState<string[]>([]);
  const [newTodo, setNewTodo] = useState<string>('');

  const addTodo = () => {
    setTodos([...todos, { id: nanoid(), text: newTodo }]);
    setNewTodo('');
  };

  return (
    <div>
      <h1>やることリスト</h1>
        <ul>
          {todos.map((todo) => (
            <li key={todo.id}>
              {todo.text}
            </li>
          ))}
        </ul>
      <form onSubmit={(e) => { e.preventDefault(); addTodo(); }}>
        <input type="text" value={newTodo} onChange={(e) => setNewTodo(e.target.value)} />
        <button onClick={addTodo}>追加する</button>
      </form>
    </div>
  );
}
```

```html
<div>
  <h1>やることリスト</h1>
  <ul id="todo_list"></ul>
  <form onSubmit="return false">
    <input type="text" id="newTodo" />
    <button onclick="addTodo()">追加する</button>
  </form>
</div>
<script>
  function removeTodo(elm) {
    elm.target.remove();
  }
  function addTodo() {
    const newTodo = document.getElementById("newTodo");
    if (!newTodo.value) return;
    const todoList = document.getElementById("todo_list");
    const newTodoElement = document.createElement("li");
    const newTodoText = document.createTextNode(newTodo.value);
    newTodoElement.appendChild(newTodoText);
    newTodoElement.addEventListener("click", removeTodo);
    todoList.appendChild(newTodoElement);
    newTodo.value = "";
  }
</script>
```

React の何が嬉しい?

- DOM をどう更新するかを意識しなくて済む
  - 完成形の仮想 DOM を返せば、React がいい感じに更新してくれる
- DOM の状態更新を簡潔に書ける
  - `id=...` を付けて、 `getElementById` で要素を取ってきて...が不要に
  - `value={newTodo}` と書くだけで OK
- マークアップとロジックを近くに置ける
  - 関連するものが近くにあることで、認知負荷が下がる (コロケーション[10])
  - 1つの関数にまとまってるので、テストもしやすい

[10] 関連するリソース同士を近くに置いておくことで、様々な負荷を軽減するという考え方。https://www.mizdra.net/entry/2022/12/11/203940 を参照。

React の書き方・読み方

JSX

- JavaScript に HTML っぽい記法を追加した拡張構文
- `.jsx`/`.tsx` という拡張子の中で書ける

```
<h1 className="hello">My name is Clementine!</h1>
```

- HTML の属性名ではなく、キャメルケースの命名規則を使用 [11]
- `class` は `className` と記述される
  - これは `class` が JavaScript において予約語であるため [12]

---

[11] `aria-*` や `data-*` 属性は例外です。

[12] https://github.com/facebook/react/issues/13525#issuecomment-671892643

関数コンポーネントとクラスコンポーネント

React ではコンポーネントの書き方が 2 種類あります。

```
<HelloMessage name="hatena" />
```

関数コンポーネント

```
const HelloMessage = ({ name }) => {
  return <div>Hello {name}</div>
}
```

クラスコンポーネント

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>
  }
}
```

関数コンポーネントとクラスコンポーネント

- 基本的にはどちらも同じことができる
- 関数コンポーネントのほうがシンプルで、書きやすい
  - 公式ドキュメントでも関数コンポーネントが推奨されてる
  - 関数コンポーネントを使おう
- ただし、一部 API がクラスコンポーネントでしか使えない
  - Error Boundary 関連の API など
  - そういう時だけ、クラスコンポーネントを使うと良い

Function Component と TypeScript

- 色々な書き方がある
  - arrow function or `function` 宣言
  - type alias で Props を定義する or inline で書く [13]
  - `React.FC<Props>` を使う or 使わない(型推論に任せる) [14]
- どう書くかは好みで良いと思う

```
type Props = { name: string }
const Welcome: React.FC<Props> = ({ name }) => {
  return <h1>Welcome {name}</h1>
}
function Welcome({ name }: { name: string }) {
  return <h1>Welcome {name}</h1>
}
```

---

[13] `interface Props { name: string }` と書くパターンもある。微妙に挙動は違うが、Props の定義にはどちらを使ってもほぼ同じ。

[14] `React.FC<Props>` を使うと、React Component として不正な`undefined`を返すことをコンパイル時に防止できます

Props と State

- React Component には値を持つ方法が大きく 2 つある
- 関数の引数として受け取る**Props**と内部状態を保持する**State**

Props を渡す/受け取る

- 受け取る側は関数の第１引数でオブジェクトとして受け取る

```
type Props = { name: string }
const Welcome: React.FC<Props> = ({ name }) => {
  return <h1>Welcome {name}</h1>
}
```

- 渡す側(親側)は JSX の属性値の記法で渡す

```
<Welcome name="John" />
// <h1>Welcome John</h1>
```

Hooks

https://ja.react.dev/learn/state-a-components-memory#meet-your-first-hook によると...

フックを使うことで、さまざまな React の機能に「接続 (hook into)」して使用することができます。

Hook を使うことで、色々なことができるようになります。

例: useState

- コンポーネントに状態を持たせるための Hook

```
const Counter = () => {
  const [count, setCount] = useState(0)
  const increaseCount = ()
    => setCount((prevCount) => prevCount + 1)

  return (
    <div>
      カウント: {count}
      <button onClick={increaseCount}>カウント</button>
    </div>
  )
}
```

Hooks の掟

フックのルール – React

- 名前はuseから始める
- トップレベルで呼ぶ [15]
  - `if`の中などで呼ばない
- early return する前に必ず呼ぶ

これらは`eslint-plugin-react-hooks`で検出してくれるように出来る

[15] ただし React 19 で導入された use は、例外的に条件分岐の中で呼び出せます

代表的な組み込み Hooks の紹介

useState

- `useState(initial)`で初期値を渡す
- 返り値はタプル
  - 1 つ目が現在の値で、2 つ目が setter
- setter を呼ぶと内部状態が更新されたことが React に通知される
  - 仮想 DOM の再生成[16]、比較、レンダリングの更新が行われる
- 推論できないものを型指定したい場合は型引数を用いることが出来る

```
const [count, setCount] = useState(0)
const [color, setColor] = useState<Color>("red")
```

[16] React によって関数コンポーネント自体が再実行され、その返り値が新しい仮想 DOM となります

useStateの setter について

- 新しい値を渡す

```
const [color, setColor] = useState<Color>("red")
const change2Blue = () => setColor("blue")
```

- 直前の値を利用して新しい値を決定する

```
const [count, setCount] = useState(0)
const increaseCount = () => setCount((prevCount) => prevCount + 1)
```

useEffect

- 外部システムに接続し、同期させるための Hook
- 例えば
  - API からデータを取得する
  - 生の DOM API を使う
  - アニメーションさせる
- React の外のシステムと接続したい時に使う

setInterval でタイマーと同期する

```
const Timer = () => {
  const [duration, setDuration] = useState(1000)
  useEffect(() => {
    setInterval(() => {
      console.log("tick")
    }, duration)
  }, [])
  return (
    <div>
      <input type="number" value={duration} onChange={(e) => setDuration(+e.target.value)} />
      <div>間隔: {duration}</div>
    </div>
  )
}
```

これで Component のマウント[17] 時に setInterval が呼ばれ、指定した間隔で tick が出力されます。

[17] Component に対応した DOM が挿入(初回描画)されることをマウント(mount)、その DOM が削除されることをアンマウント(unmount)と呼びます

useEffectと依存配列

- デフォルトでは、エフェクトはレンダー時に毎回実行される
  - しかし、それが望ましくない場合も
- `useEffect` の第 2 引数 (**依存配列**) で、不必要な実行を防げる
- `useEffect(任意の処理関数, [])`
  - マウント時にだけ副作用を実行
- `useEffect(任意の処理関数, [val1, val2])`
  - `val1` や `val2` のいずれかが変更されたときにエフェクトを実行

カウンターのカウントが変わるたびに、サーバーにメトリクスを送る例

```
const Counter = () => {
  const [name, setName] = useState("インターンに参加した回数")
  const [count, setCount] = useState(0)
  useEffect(() => {
    fetch(`/api/user-metrics?count=${count}`)
  }, [count])
  return (
    <>
      <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      <button onClick={() => setCount(c => c + 1)}>
        Increment
      </button>
      <div>{count}</div>
    </>
  )
}
```

name が変わっても、メトリクスは送られない。

依存配列は自分で選ぶものではない

- 基本的には、エフェクトから参照されてる値を全て依存配列に入れる[18] [19]
- ESLint + `eslint-plugin-react-hooks` を使うと、入れ忘れてる値を警告してくれる
  - この警告に従うのがセオリー

```
useEffect(() => {
  fetch(`/api/user-metrics?count=${count}`)
}, [])
// ^^ React Hook useEffect has a missing dependency: 'count'.
//    Either include it or remove the dependency array.
```

ESLint + `eslint-plugin-react-hooks` は後からの導入が面倒なので、初手で**必ず**導入しましょう。

---

[18] `ref` やコンポーネントの外で定義された関数などは不変なので、依存配列に入れなくて良いという例外的なルールがあったりします。

[19] 全ての参照値を依存配列に入れるといっても、「この値が変わる度に実行されると困る」ケースもあります。その場合はエフェクトを分割するなどがセオリーとされてます。

useEffect とクリーンアップ

- 実はエフェクトから関数を返せる
  - **クリーンアップ関数** と呼ばれる
  - 次のエフェクトが実行される前[20]に呼ばれる
- これを利用すると副作用のクリーンアップが出来る

前回の `setInterval` の例をクリーンアップを追加すると以下のようになる

```
useEffect(() => {
  const id = setInterval(() => {
    console.log("tick")
  }, duration);

  return () => {
    // clearInterval でタイマーを停止する関数
    clearInterval(id)
  }
}, [duration])
```

[20] 空配列を指定している場合などはアンマウント時にも実行される

独自フック(Custom Hook)

- 内部で他の Hook を呼び出す関数で、 use から名前が始まるもののこと
- 複数の Hooks を組み合わせたり、Component の振る舞いを共通化して、1 つの関数に切り出すときなどに利用する

```
const useUserStatus = ({ userId }) => {
  const [status, setStatus] = useState(null)
  useEffect(() => {
    const handler = (user) => {
      setStatus(user.status)
    }
    Api.subscribe(userId, handler)
    return () => Api.unsubscribe(userId, handler)
  })
  return status
}
function SomeComponent({ userId }) {
  const status = useUserStatus({ userId })
  return <div>{status}</div>
}
```

独自フックについての Tips

- 独自フックに切り出すことで、その部分をテストできる
- 独自フックの中では `useMemo`、`useEffect`、`useCallback` を積極的に使う
  - パフォーマンス最適化のためになる
  - 参考: useCallback はとにかく使え！　特にカスタムフックでは - uhyo/blog

標準化について (5min)

標準化について

ブラウザで動く言語や API などはそれぞれ以下のような仕様で標準化されてます。

標準化団体	策定している仕様
WHATWG	HTML Living Standard[^49] DOM Living Standard Fetch Living Standard
W3C	CSS Specifications WCAG
TC39	ECMAScript Internationalization API
IETF	RFC xxx

こうした仕様を知ることは重要

- API の正しい振る舞いを定義しているのは仕様
- 世の中に公開されている情報が数多くあるが...
  - その情報の正しさを最終的に保証するための情報源が仕様
- 例えば JavaScript や CSS などの振る舞いがブラウザ間で異なる場合...
  - その時に正しさを保証してくれるのは仕様である
  - 実際、開発をしてると仕様とブラウザの差異に遭遇することも [21]

[21] ウェブブラウザにバグ報告をするときにやること - ぱすたけ日記

ECMAScript について

- JavaScript の標準仕様は ECMAScript と呼ばれる
  - Ecma International[22]の中の Technical Committee 39（TC39）という技術委員会により策定
- 現在の仕様はhttps://github.com/tc39/ecma262
  - 常に最新版が公開され続けてる
  - こういう形式を Living Standard と呼ぶ
- 毎年 6 月頃に ECMAScript 2021 のようなタグが打たれる
  - バージョン番号が付与したものも公開される

[22] Ecma International は C#などの標準化作業も行っている

ECMAScript の Stage

- ECMAScript には自由に提案(proposal)を出せる
  - GitHub 上で proposal が公開される
  - issue、PR、TC39 のミーティングなどでの議論を経て、Stage が上がる [23]
  - Stage 4 になると仕様に取り込まれる

[23] 責任者であるチャンピオンのステージを進めたいという意思も必要

Stage の詳細

https://jsprimer.net/basic/ecmascript/ より引用・一部改変。

ステージ	ステージの概要
0	アイデアの段階
1	機能提案の段階
2	機能の仕様書ドラフトを作成した状態
2.7	仕様がある程度固まってて、実装前にプロトタイプなどを作って実験する段階
3	仕様としては完成しており、ブラウザの実装やフィードバックを求める段階
4	仕様策定が完了し、2 つ以上の実装[24][25]が存在している状態

[24] V8, SpiderMonkey, Hermes といった JavaScript エンジンなどに実装されてることが要求されてる。

[25] 主に Stage3 以前時点での実装などはその後の仕様変更などの可能性もあるので、ブラウザの開発者向けフラグを有効にしているときだけ使えたり、prefix を付けた名前で API が提供されることもある。それらも実装として許容されるなどの緩さはある。

ECMAScript の仕様と実装

- Stage 4 になる前に、ブラウザなどに実装される
  - 実装上の困難さ、使用上や仕様の問題などを確認するため
  - その過程で仕様にフィードバックができたり、有用性を示せる
- Stage 4 になる前に、polyfill や Babel によるトランスパイルがサポートされることも
  - ブラウザに実装するより前に、ユーザに試してもらえる

ECMAScript は全てが Open

- 誰でもプロポーザルを出せるし、読めるし、議論できる
- 興味のある提案があったら覗いたり、使ってみよう
  - フィードバックすれば、JavaScript をより良くできる [26]

[26] ECMAScript のプロポーザルは多くの場合、その提案が「どのようなモチベーションがあるのか」、「どのような問題を解決するのか」、「どのようなユースケースがあるのか」などが記されているので、その有用性などを示すことも貢献に繋がります。

JavaScript API を策定する仕様

実は、JavaScript の全てが ECMAScript で策定されてる訳では無いです。

仕様	策定している内容
ECMAScript (TC39)	JavaScript の構文や基本的な API
DOM Living Standard (WHATWG)	`document.querySelector` などの DOM API
Fetch Living Standard (WHATWG)	`fetch` API
Internationalization API (TC39)	`Intl.DateTimeFormat` などの国際化 API

ブラウザベンダと WHATWG 仕様

- Google は HTML や DOM に関わる API[27]の提案を数多く行ってる
  - Chrome には取り込まれるが、Apple や Moziila の反対により、標準化されないことも
  - Chrome の仕様トラッカー: Chrome Platform Status
    - Apple や Mozilla それぞれから立場を表明するウェブサイトがある
  - Mozilla Specification Positions
  - Standards Positions | Webkit
- W3C 内の Web Incubator Community Groupで様々な提案が作成・議論されている

[27] https://scrapbox.io/pastak-pub/面白 WebAPI100 連発 で色々紹介しています

ビルドツールについて (5min)

ビルドツールについて

- TypeScript がそのまま実行されることは少ない
- 最適化などの様々な都合により、大抵はコードを変換してから実行する
  - その変換に使われるツールを「ビルドツール」と呼ぶ

代表的なビルドツール ①

- .ts => .js へ変換するツール
  - .ts は JavaScript ランタイムで直接実行できないので、変換が必要
  - tsc, swc, esbuild など [28]
- 古い ECMAScript バージョンへ変換 (downlevel) するツール
  - 古いブラウザなどをサポートするために必要
  - tsc, babel, swc, esbuild など

[28] ブラウザ組み込みのページ遷移に代わって、JavaScript でページ遷移を行うこと。これにより、ページを完全に読み込み直すことなく、シームレスな遷移が可能になる。

代表的なビルドツール ②

- JavaScript ファイルを結合するツール (bundler)
  - ページアクセス直後のネットワークリクエストの数を減らすために必要
  - webpack, rollup など
  - CSS や画像ファイルなども bundle できる
- JavaScript ファイルを圧縮するツール (minifier)
  - ネットワーク転送量を減らすために必要
  - terser など

代表的なビルドツール ③

- 統合的なビルドツール
  - 上記の機能をまとめて提供するツール
  - `next build`/`next dev`, Vite など
  - 内部的には swc や terser などを使ってる
  - 基本的には、これを使うと良い

統合的なビルドツールに備わってる機能

実は他にも色々な機能が備わってます。

- Watch ビルド
  - ファイルの変更を監視して、変更があったら自動でリビルドする
- 開発サーバー
  - `localhost:3000` などで開発中のアプリケーションを配信してくれる
- Hot Module Replacement (HMR)
  - リビルド結果をブラウザに開いているページに、リロードなしで反映する[29]

[29] ファイルの構成に基づいてルーティングすること。Next.js であれば `app/user/page.tsx` を作成すると `/user` にルーティングできるようになる仕組みのこと。

Web フレームワークについて

- Web フレームワークと呼ばれるものもある
  - Next.js, Remix, Nuxt.js, Astro, ...
- フロントエンド開発をすぐに始められるよう、色々組み込まれてる

Web フレームワークが組み込んでるもの

- ビルドツールとその推奨設定
  - いい感じの設定が組み込まれてて、ほぼ設定不要で使える
- ルーティング
  - ページ遷移時にソフトナビゲーション[28] したり、File-based routing[29] をサポートしたり
- サーバーサイドレンダリング (SSR)[30]
  - Node.js サーバー上でコンポーネントをレンダーしてから HTML を返す技術
  - SEO や初回表示の高速化に寄与する
- テストランナーの提供
  - すぐにユニットテストやコンポーネントテストが書ける

---

[28] ブラウザ組み込みのページ遷移に代わって、JavaScript でページ遷移を行うこと。これにより、ページを完全に読み込み直すことなく、シームレスな遷移が可能になる。

[29] ファイルの構成に基づいてルーティングすること。Next.js であれば `app/user/page.tsx` を作成すると `/user` にルーティングできるようになる仕組みのこと。

[30] 詳しくは https://speakerdeck.com/mizdra/react-server-components-noyi-wen-wojie-kiming-kasu?slide=15 を参照。

どれを使えば良いか

作りたいものの要件に応じて適切なものを選びましょう。

- React 使って SSR もしたい
  - Next.js を使う
- React 使うけど SSR は不要で、SEO も気にしない
  - Vite + React を使う
- Node.js 向けライブラリを作りたい
  - `tsc` だけで十分
  - `.js` をネットワーク経由で取得しないので、bundler/minifier は不要

それぞれのツールの役割や目的を知っていれば、自ずと分かるはずです。

心構え的な話 (3m)

心構え的な話

- フロントエンドは、直接ユーザが触れる部分
  - ユーザからの評価に直結する
- 良い UI を実装しよう
  - ユーザビリティやパフォーマンス改善をちゃんとやる
  - ユーザのことを考える

心構え的な話

アクセシビリティに気を使いましょう

- 具体的な配慮点
  - キーボードで操作できたり
  - 機械翻訳できたり
  - 文字サイズを自由に変えたり
- 障害者の方のため、だけではない
  - 健常者も文字サイズ変えたいことはある
  - 皆のためにもなる

心構え的な話 (時間があれば)

色々な職種の方と協力しよう。

- デザイナーと協力する
  - デザイナーさんの力だけでは実現が難しいものを、エンジニアがサポートしたり
    - アニメーションの PoC 作ってみるとか
- プランナーと協力する
  - UI の実装をしているからこそ、ユーザビリティの改善点が見つかるはず
  - エンジニア視点で新機能の提案してみるとか

協力して、より良いものを作っていきましょう。

お疲れさまでした

- JavaScript/TypeScript/React についてざっと紹介しました
  - これだけで完璧に理解した！とならないと思いますが...
  - 開発に入るため・学ぶための足がかりになったはず
- JavaScript やフロントエンドの世界は更に広がっています
  - ブラウザを介して、ここまで多くのユーザの目に触れる分野は中々ありません
  - 是非フロントエンドの世界を楽しんでください

付録

本編に入り切らなかった踏み込んだ補足や解説について書いています。興味があれば読んでください的なコーナーです。

|| と ?? の違い

- ?? は比較的新しい構文で、昔のエンジンで動かなかった
  - 古い JavaScript コードでは、代わりに || が使われがち
- || は左辺が Falsy (偽とみなせるもの) なら右辺の値を返す
  - Falsy な値の例: false/null/undefined/NaN/0/'' (空文字)

```
const price1 = 0 || 100; // 100
const price2 = 0 ?? 100; // 0
```

- 挙動が難しいので、?? を使うのがオススメ

Arrow Function に置き換え出来ないケース

- `function` を使ってコンストラクタ関数にしている(`new` している)
- `arguments` を参照している
- `this` を参照している
  - このケースについて解説

Arrow Function と `this`

- Arrow Function と `function xxx(){...}` で `this` の扱いが異なる
- このことによって単純な置き換えが不可な場合がある

this のスコープの違いについて

- function だと呼び出し元のオブジェクトが this になる

```
const person = {
  name: "chris",
  say: function () {
    setTimeout(function () {
      console.log(`I'm ${this.name}`)
    }, 100)
  },
}
person.say() // I'm
```

- この場合は window.setTimeout(window は省略できる) からの呼び出しなので、 window になる

this のスコープの違いについて

- Arrow Function だとスコープが外と同じになる

```
const person = {
  name: "chris",
  say: function () {
    setTimeout(() => {
      console.log(`I'm ${this.name}`)
    }, 100)
  },
}
person.say() // I'm chris
```

useState の state を更新する際の注意

新しい値を渡すときの落とし穴

```
const [count, setCount] = useState(0)
const increase = () => setCount(count + 1)
```

としてしまうと、

```
const incrementDouble = () => {
  increment()
  increment()
}
```

のようなものを作ったときに、incrementDouble を呼んでも 1 しか増えません。

関数内の変数スコープと useState

例えば、以下のようなコードがあった時:

```
const Component = () => {
  const [count, setCount] = useState(0)
  const increment = () => setCount(count + 1)
  const incrementDouble = () => {
    increment()
    increment()
  }
}
```

関数内の変数スコープと useState

これと同じ意味になる:

```
const Component = () => {
  const [count, setCount] = useState(0)
  const incrementDouble = () => {
    setCount(count + 1) // count = 0 の時、`setCount(1)` になる
    setCount(count + 1) // count = 0 の時、`setCount(1)` になる
  }
}
```

setter に関数を渡すと良い

`setCount((prevCount) => prevCount + 1)` とすると、期待通りになる

```
const Component = () => {
  const [count, setCount] = useState(0)
  const increment = () => setCount((prevCount) => prevCount + 1)
  const incrementDouble = () => {
    increment() // count = 0 の時、`setCount(0 + 1)` になる
    increment() // count = 1 の時、`setCount(1 + 1)` になる
  }
}
```

更新後の state の値が更新前の値に依存している場合は、関数を渡す形式を使いましょう。

Hooks の依存配列の変更検知について

- 依存配列の値が変わったかは `Object.is` で検証される
- 同じ内容のオブジェクトでも、参照が異なると変わったと認識される

```
console.log(Object.is("foo", "foo")) // true
console.log(Object.is({ prop: "foo" }, { prop: "foo" })) // false

const objA = { prop: "foo" }
const objB = objA
console.log(Object.is(objA, objB)) // true
```

依存配列にオブジェクトを入れるケースについて

```
function Component() {
  const config = { theme: "sports" }
  useEffect(() => {
    loadConfig(config).then(() => {})
  }, [config])
}
```

のような場合にはレンダリングの度に、config が再生成される。よって、異なる値として認識されてしまい、毎回エフェクトが実行されてしまう。[38]

[38] このような例の場合は依存に [config.theme] という風に値を書いてしまっても良いが、依存するオブジェクトについて知っている必要があるので難しい。

シンプルな回避策

- 一番簡単な回避策は Component の外で初期化すること

```
const config = { theme: "sports" }

function Component() {
  useEffect(() => {
    loadConfig(config).then(() => {})
  }, [config])
}
```

- 一方で、Props を利用してオブジェクトを生成してる場合は採用できない

useMemo を使った回避策

- useMemo は React に組み込まれている Hooks で、値のメモ化[39]ができる
- これで過度なエフェクトの再実行を防げる

```
function Component({ theme }) {
  const config = useMemo(() => ({ theme }), [theme])
  useEffect(() => {
    loadConfig(config).then(() => {})
  }, [config])
}
```

[39] メモ化はパフォーマンス改善のために計算結果をキャッシュしたりすること

useCallback で関数をメモ化する

- 実は関数も実態はオブジェクト
  - コンポーネント内で関数宣言すると、毎回再生成されてしまう
- 値同様に関数をメモ化したい場合は useCallback を利用する

```
const handler = useCallback((val) => alert(val), [])
useEffect(() => {
  Api.notification.subscribe(handler)
}, [handler])
```

React Component 内で DOM にアクセスする

- React に組み込まれている、参照を保持できる ref オブジェクトを作成する useRef を使う
  - ref オブジェクトは current プロパティに現在の値を持っている

```
const textInput = useRef(null)
const submit = (e) => {
  e.preventDefault()
  if (textInput.current.value.length < 100)
    return alert("101文字以上が必要です")
  createPost({ body: textInput.current.value })
}
return (
  <form onSubmit={submit}>
    <input type="text" ref={textInput} />
  </form>
)
```

DOM へのアクセスを避ける方が良い

- DOM に直接アクセスすると、React の制御外のところで値が取得されたり変更されることに
- ライブラリの都合などで本当に必要なときのみにしておくと良い

```
const [text, setText] = useState("")
const submit = (e) => {
  e.preventDefault()
  if (text < 100) return alert("101文字以上が必要です")
  createPost({ body: text })
}
return (
  <form onSubmit={submit}>
    <input type="text" onChange={(e) => setText(e.target.value)} />
  </form>
)
```