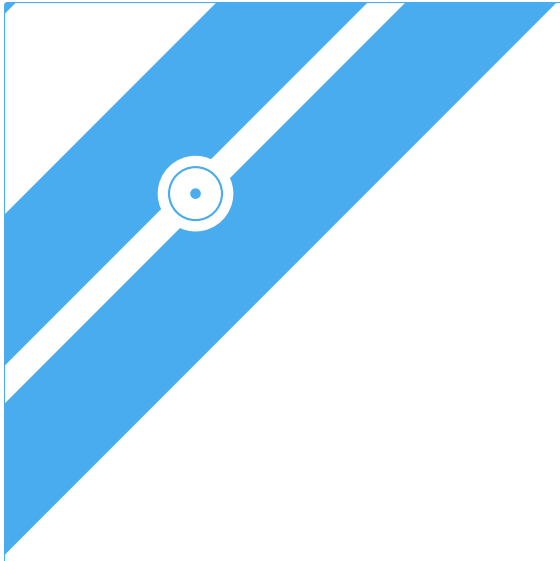

進捗状況

React Hook

B4 小林紹子

October 16, 2025

目次

useCallback

- useCallback は関数をメモ化するためのフック.
- useCallback の第一引数は関数で, 第二引数は依存配列.
- 関数の再描画が行われる際に,useCallback() は依存配列の中の値を比較.
- 依存配列の値がそれぞれ同じ場合は,useCallback() はメモ化している関数を返す.
- 異なる場合は, 現在の第一引数の関数をメモに保存する.$\longrightarrow$ 新しい関数を返す.

memo 関数と useCallback の関係

- memo 関数
  - props が同じであれば再描画をスキップする.
- useCallback
  - 関数の参照をメモ化.
  - 依存配列が変わらない限り, 同じ関数オブジェクトを再利用.
  - 再レンダーされても「新しい関数が作られる」ことはない.

⟶useCallback と memo 関数を同時に使うことで再描画を防げる.

```
24  //DoubleButtonはメモ化した関数コンポーネントでボタンを表示する
25  const DoubleButton = React.memo((props: ButtonProps) => {
26    const { onClick } = props;
27    console.log('DoubleButtonが再描画されました');
28
29    return <button onClick={onClick}>DoubleButton</button>;
30  });
```

```
42    //useCallbackを使って関数をメモ化する
43    const double = useCallback(() => {
44      setCount((c) => c * 2);
45
46      //第二引数は空行列なのでuseCallbackは常に同じ関数を返す
47    }, []);
48
49    return (
50      <div>
51        <p>Count : {count}</p>
52        <DecrementButton onClick={decrement} />
53        <IncrementButton onClick={Increment} />
54        <DoubleButton onClick={double} />
```

Next.js と React I

- Next.js： React をベースに開発されたフレームワーク.
- React： Javascript におけるライブラリの一つ.

ライブラリ：
開発時に使われるコードの集まり. 作業を簡略化するための関数や定義・クラスの集まり.
フレームワーク:
テンプレートのようなもの. ログイン機能や決算機能など必要な機能がデフォルトで揃っている.
⇒Next.js は React ライブラリのフレームワーク

Next.js と React II

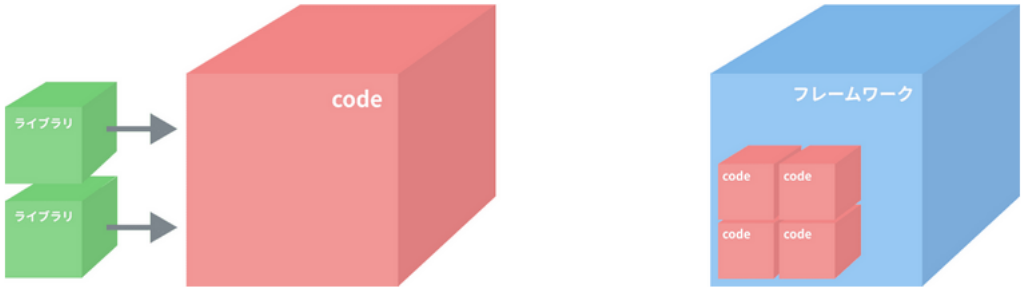

Figure: React と Next.js

Next.js の特徴 I

- RCS(React Server Components)
  - React は従来 SSR(Server Side Rendering) を行ってきた.
  - SSR : 全てのコンポーネントをサーバーで HTML に変換する.
  - RCS では、それをサーバーで実行する部分とクライアントで実行する部分を分けることができる.
  - これにより, 無駄なリロードが減る.

Next.js の特徴 II

- App Router
  - ディレクトリ構造がそのまま URL の構造になる.
  - ルーティングは/app 配下のディレクトリベースで行う.
  - page.tsx でルートの画面,router.tsx で API エンドポイントを書くなどファイル規約がある.

```
src
  └ app
     ├ page.js          -> /
     ├ about
        └ page.js       -> /about
     └ users
        ├ page.js       -> /users
        ├ list
           └ page.js    -> /users/list
        └ [id]
           └ page.js    -> /users/1, /users/2,,,(動的ルート)
```

環境構築

- npm をインストール（sudo apt npm）version：v22.15.0
- Next.js をインストール（npx create-next-app@latest –ts next-sample）version：v22.15.0
- ここで Turbopack が開発サーバークラッシュしてるとエラーがでた.
- →nodemodule と package.json を削除
- npm i で再インストールするとうまくいった.
- npm run dev で初期画面がでてきた.（http://localhost:3000）

画面描画

- React Flow：インタラクティブなダイアグラムを簡単に実装できる.
- npm install reactflow
- 作れはしたが、もともとチャートフロウなどを書く用に作られたものだったので上手な図形は作れず.

コード1

```
"use client"
import React, { useEffect, useState } from "react"
import ReactFlow, { Background } from "reactflow"
import "reactflow/dist/style.css"

export default function GraphPage() {
  const [nodes, setNodes] = useState<any[]>([])
  const [edges, setEdges] = useState<any[]>([])
  const [selectedNode, setSelectedNode] = useState<string | null>(null)
  const [loading, setLoading] = useState(true)

```

コード II

```
12  useEffect(() => {
13    fetch("/api/graph")
14      .then((res) => res.json())
15      .then((data) => {
16        setNodes(data.nodes)
17        setEdges(data.edges)
18        setLoading(false)
19      })
20  }, [])
```

○×問題

- まずはバックエンド・フロントエンドに問題・回答を書いた.
- JSON で答えをサーバからもらって判断する.
- 将来的にはデータベースから問題を取得するようにする.

フロントエンド (page.tsx) I

```tsx
"use client";

import { useState } from "react";

type Problem = {
  id: number;
  text: string;
  image: string;
};

```

フロントエンド (page.tsx) II

```tsx
export default function ProblemsPage() {
  // 現在の問題
  const [problem] = useState<Problem>({
    id: 2,
    text: "このグラフは木である。○か×か？",
    image: "/graph2.png",
  });

  const [result, setResult] = useState<string | null>(null);
```

フロントエンド (page.tsx) III

```
21  const handleAnswer = async (answer: boolean) => {
22    try {
23      const res = await fetch("/api/check_answer", {
24        method: "POST",
25        headers: { "Content-Type": "application/json" },
26        body: JSON.stringify({
27          id: problem.id,
28          answer: answer,
29        }),
30      });
31
```

フロントエンド (page.tsx) IV

```
32        const data = await res.json() as { correct?: boolean; error?: string };
33
34        if (data.error) {
35          setResult("エラー: " + data.error);
36          return;
37        }
38
39        setResult(data.correct ? "正解！🎉" : "不正解❌");
40      } catch (err) {
41        console.error(err);
42        setResult("通信エラーが発生しました");
```

バックエンド (route.tsx) I

```tsx
export async function POST(req: Request): Promise<Response> {
  const { id, answer }: { id: number; answer: boolean } = await req.json();

  // 問題ごとの正解データ（本来はDBなどに置く）
  const correctAnswers: Record<number, boolean> = {
    1: true,  // 問題1は○
    2: false, // 問題2は×
    3: true,  // 問題3は○
  };
```

バックエンド (route.tsx) II

```
11  const correct = correctAnswers[id];
12
13  if (correct === undefined) {
14    return new Response(JSON.stringify({ error: "無効な問題ID" }), {
15      status: 400,
16      headers: { "Content-Type": "application/json" },
17    });
18  }
19
```

バックエンド (route.tsx) III

```
19
20    const isCorrect = answer === correct;
21
22    return new Response(JSON.stringify({ correct: isCorrect }), {
23      headers: { "Content-Type": "application/json" },
24    });
25  }
26
```