React with TypeScript 勉強会

基礎から応用まで

目次

1. イントロダクション

諸注意

- 何か質問があれば、遠慮なくどうぞ
- 詰め込みすぎて、理解しづらい部分があるかもしれません
- 発表者も詳しくない部分があるかもしれません

Reactとは

- Facebookが開発したUIライブラリ
- 宣言的なアプローチでUIを構築
- 仮想DOM (Virtual DOM) を使用して効率的に更新

TypeScriptとは

- JavaScriptのスーパーセット
- 静的型付けを提供
- トランスパイル時にJavaScriptに変換

ReactはJavaScriptのライブラリだが、TypeScriptと組み合わせて使うことが一般的。

なぜReactとTypeScriptを組み合わせるのか

1. 型安全性
    - PropsやStateの型チェック
    - コンパイル時のエラー検出
2. 開発者体験の向上
    - インテリセンスとコード補完
    - リファクタリングの容易さ
3. ドキュメンテーション
    - コードが自己文書化
    - 型定義が仕様書の役割を果たす

2. 環境構築 (nvm + Vite + TypeScript)

Node.js, npmとは

- Node.js: JavaScriptランタイム
- npm: Node.jsのパッケージマネージャ
  - Pythonでいうところのpipや Rubyでいうところのgemに相当
  - Reactなどのライブラリやツールをインストールするために使用

nvmとは

- nvm (Node Version Manager): 複数のNode.jsバージョンを管理するためのツール
  - PythonのpyenvやRubyのrbenvに相当
- プロジェクトごとに異なるNode.jsバージョンを簡単に切り替え可能

nvmのインストール

ここを参照

1. nvmのインストール
    - macOS/Linux:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

    - Windows:
        - nvm-windowsを使用
2. nvmの設定を反映
    - macOS/Linux:

```
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
```

Node.jsのインストール

3. Node.jsの最新版インストール

```
nvm install node
nvm use node
```

- 個別にバージョンを指定する場合は、`nvm install <version>`を使用

4. インストール確認

```
node -v
npm -v
```

プロジェクトの作成

- 通常は、Reactプロジェクトを作成する際に `create-react-app` を使用する
- 今回はViteを使用してReactプロジェクトを作成する

Viteとは

- 高速な開発サーバーと最適化されたビルドを提供
- ES Modulesを活用した効率的な開発環境

作成と実行を行うコード

```
npm create vite@latest my-react-ts-app -- --template react-ts
cd my-react-ts-app
npm install
npm run dev
```

プロジェクト構造

```
my-react-ts-app/
├── node_modules/
├── public/
├── src/
│   ├── App.css
│   ├── App.tsx
│   ├── index.css
│   ├── main.tsx
│   └── vite-env.d.ts
├── .gitignore
├── index.html
├── package.json
├── tsconfig.json
├── tsconfig.node.json
└── vite.config.ts
```

重要なファイル

- `tsconfig.json` : TypeScriptの設定ファイル
- `vite.config.ts` : Viteの設定ファイル
- `src/vite-env.d.ts` : Vite固有の型定義
- `src/main.tsx` : アプリケーションのエントリーポイント
- `src/App.tsx` : メインのAppコンポーネント

3. TypeScript・モダンJSの基礎

基本的な型

- 全ての変数や関数は型を持つ
- 型推論: TypeScriptは型を推論するため、明示的な型注釈が不要な場合がある

```
// プリミティブ型
let isDone: boolean = false;
let decimal: number = 6;
let color: string = "blue";

// 配列
let list: number[] = [1, 2, 3];
let names: Array<string> = ["John", "Jane"];

// タプル
let x: [string, number] = ["hello", 10];
```

基本的な型 (続き)

```
// enum
enum Color {Red, Green, Blue}
let c: Color = Color.Green;

// any
let notSure: any = 4;
notSure = "maybe a string instead";

// void
function warnUser(): void {
    console.log("This is a warning message");
}

// null and undefined
let u: undefined = undefined;
let n: null = null;
```

インターフェースと型エイリアス

```typescript
// インターフェース
interface User {
  name: string;
  age: number;
  email?: string; // オプショナルプロパティ
}

const user: User = {
  name: "John",
  age: 30
};

// 型エイリアス
type Point = {
  x: number;
  y: number;
};

const point: Point = { x: 10, y: 20 };
```

ジェネリクス

- 型パラメータを使用して、異なる型の値を扱う関数やクラスを作成できる

```
function identity<T>(arg: T): T {
    return arg;
}

let output = identity<string>("myString");
```

ユニオン型とインターセクション型

```
// ユニオン型: 複数の型を許容
type StringOrNumber = string | number;

// インターセクション型: 複数の型を結合
type Employee = Person & { employeeId: number };
```

型アサーション

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;

// または(あんまり使わない)
let strLength: number = (<string>someValue).length;
```

リテラル型

```
type Direction = "North" | "South" | "East" | "West";
let dir: Direction = "North";
```

アロー関数

- アロー関数は、通常の関数と比べてシンプルで短く書ける

```
// 通常の関数
function add(a: number, b: number): number {
  return a + b;
}

// アロー関数
const add = (a: number, b: number): number => a + b;
```

分割代入

- オブジェクトや配列の要素を分割して変数に代入できる

```
// オブジェクトの分割代入
const user = { name: "John", age: 30 };
const { name, age } = user;

// 配列の分割代入
const numbers = [1, 2, 3];
const [first, second] = numbers;

console.log(first); // 1
console.log(second); // 2
```

論理積演算子と論理和演算子

- 論理積演算子(`&&`)と論理和演算子(`||`)は、JavaScriptと同様に使用可能

```
const x = 10;
const y = 20;

if (x > 0 && y > 0) {
  console.log("Both x and y are positive");
}

if (x > 0 || y > 0) {
  console.log("Either x or y is positive");
}
```

`||`, `&&` の挙動

- `||` は、左辺が `falsy` なら右辺を返す
  - デフォルト値を設定する際などに使用
- `&&` は、左辺が `falsy` なら左辺を返す
  - 条件式の結果を利用する際などに使用

```
const name = "";
const defaultName = "John";

const result = name || defaultName;

console.log(result); // John
console.log(result && `Hello, ${result}`); // Hello, John
```

4. Reactの基本概念 with TypeScript

コンポーネント

- Reactコンポーネント: UIを構築するための基本単位
- JSXと呼ばれるJavaScriptの拡張構文を使用して記述
  - HTMLライクな構文でコンポーネントを記述
- 一度定義すると、 `<h1>` や `<div>` のようなHTML要素のように再利用可能
- **関数コンポーネント**
  - シンプルで軽量
  - React Hooksでステート管理可能
- **クラスコンポーネント**
  - ステートとライフサイクルメソッドを持つ
  - 複雑なロジックに適している

関数コンポーネントの例

```tsx
import React from 'react';

// 関数コンポーネント
const Title: React.FC = () => {
  return <h1>Hello, React!</h1>;
};

// 使用例
const App: React.FC = () => {
  return (
    <div>
      <Title />
      <p>Welcome to the React workshop</p>
    </div>
  );
};
```

クラスコンポーネントの例

- クラスコンポーネントに相応しい、状態管理などを含む少し難しい例

```tsx
import React, { Component } from 'react';

interface CounterProps {
  initialCount: number;
}

interface CounterState {
  count: number;
}

class Counter extends Component<CounterProps, CounterState> {
  constructor(props: CounterProps) {
    super(props);
    this.state = {
      count: props.initialCount
    };
  }

  increment = () => {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Props

Reactコンポーネントには、プロパティ（props）を介してデータを渡すことができる

一覧画面など、同じUIだが異なるデータを表示するコンポーネントを作成できる

```
interface UserProps {
  name: string;
  age: number;
  email?: string;
}

const User: React.FC<UserProps> = ({ name, age, email }) => {
  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      {email && <p>Email: {email}</p>}
    </div>
  );
};
```

イベントハンドリング

例: ボタンクリック時のイベントハンドリング

```tsx
import React, { useState } from 'react';

const InputComponent: React.FC = () => {
  const [inputValue, setInputValue] = useState<string>('');

  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value);
  };

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    console.log('Submitted value:', inputValue);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={inputValue} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
};
```

5. React Hooksの詳細 (TypeScript版)

React Hooksとは

- React Hooks: 関数コンポーネントで状態管理や副作用を扱うための機能
- クラスコンポーネントを使わずに、よりシンプルにReactの機能を利用できる
- そのため、近年のReact開発では関数コンポーネント+Hooksが主流となっている

主なReact Hooks

1. `useState` : コンポーネントの状態を管理するためのフック
2. `useEffect` : 副作用（API叩く、JavaScript実行など画面描画後の処理）を処理するためのフック
3. `useContext` : コンテキストAPIを利用するためのフック
4. `useReducer` : 複雑な状態管理を行うためのフック
5. `useCallback` : コールバック関数をメモ化するためのフック
6. `useMemo` : 計算結果をメモ化するためのフック
7. `useRef` : DOM要素や任意の値を保持するためのフック

useState

`useState`: コンポーネントの状態を管理するためのフック

例: カウンターアプリ

```
import React, { useState } from 'react';

const Counter: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>
        Increment
      </button>
    </div>
  );
};
```

useEffect

`useEffect`：副作用を処理するためのフック

```tsx
import React, { useState, useEffect } from 'react';

const DataFetcher: React.FC = () => {
  const [data, setData] = useState<string | null>(null);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data');
      const result = await response.text();
      setData(result);
    };

    fetchData();
  }, []);

  return <div>{data ? <p>{data}</p> : <p>Loading...</p>}</div>;
};
```

useContext

```tsx
import React, { createContext, useContext, useState } from 'react';

interface ThemeContextType {
  theme: string;
  toggleTheme: () => void;
}

const ThemeContext = createContext<ThemeContextType | undefined>(undefined);

export const ThemeProvider: React.FC = ({ children }) => {
  const [theme, setTheme] = useState<string>('light');

  const toggleTheme = () => {
    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

export const useTheme = (): ThemeContextType => {
  const context = useContext(ThemeContext);
  if (context === undefined) {
    throw new Error('useTheme must be used within a ThemeProvider');
  }
  return context;
};
```

useReducer

現在の状態とアクションを引数に取り、新しい状態を返すreducer関数を使用する。初期状態とreducer関数を引数に取り、現在の状態とdispatch関数を返す。

```tsx
import React, { useReducer } from 'react';

interface State {
  count: number;
}

type Action =
  | { type: 'increment' }
  | { type: 'decrement' }
  | { type: 'reset' };

const initialState: State = { count: 0 };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return initialState;
    default:
      throw new Error();
  }
}

const Counter: React.FC = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </>
  );
};
```

useState, useContext, useReducerの使い分け

全て状態管理に関連するフックだが、それぞれの使い分けがある

useState

- **用途**: 単純な状態管理
- **例**: カウンター、フォーム入力

useContext

- **用途**: グローバルな状態管理や、コンポーネントツリー全体で共有されるデータの管理
- **例**: 認証情報、テーマ設定

useReducer

- **用途**: 複雑な状態管理、特に複数のサブ値を持つ状態や、状態遷移が複雑な場合
- **例**: フォームのバリデーション、複数のアクションを伴う状態管理

6. ハンズオン：Todoアプリの作成

詳細

- シンプルなTodoアプリを作成
- 目標: 90分程度
- できること
  - Todoの追加、完了、削除
  - エラーハンドリング

Step 1: 型定義

```ts
// src/types.ts
export interface Todo {
  id: number;
  text: string;
  completed: boolean;
}
```

Step 2: Todoリストコンポーネント I

```tsx
// src/components/TodoList.tsx
import React, { useState } from 'react';
import { Todo } from '../types';

const TodoList: React.FC = () => {
  const [todos, setTodos] = useState<Todo[]>([]);
  const [input, setInput] = useState<string>('');

  const addTodo = () => {
    if (input.trim() !== '') {
      setTodos([...todos, { id: Date.now(), text: input.trim(), completed: false }]);
      setInput('');
    }
  };

  const toggleTodo = (id: number) => {
    setTodos(todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
    ));
  };

  // ...（次のスライドに続く）
};
```

Step 2: Todoリストコンポーネント II

```tsx
// src/components/TodoList.tsx（続き）

const removeTodo = (id: number) => {
  setTodos(todos.filter(todo => todo.id !== id));
};

return (
  <div>
    <input
      type="text"
      value={input}
      onChange={(e) => setInput(e.target.value)}
      placeholder="Add a new todo"
    />
    <button onClick={addTodo}>Add</button>
    <ul>
      {todos.map(todo => (  // 後でコンポーネントに分割する
        <li key={todo.id}>
          <input
            type="checkbox"
            checked={todo.completed}
            onChange={() => toggleTodo(todo.id)}
          />
          <span style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}>
            {todo.text}
          </span>
          <button onClick={() => removeTodo(todo.id)}>Remove</button>
        </li>
      ))}
    </ul>
  </div>
);
```

Step 3: TodoItemコンポーネント

```tsx
// src/components/TodoItem.tsx
import React from 'react';
import { Todo } from '../types';

interface TodoItemProps {
  todo: Todo;
  toggleTodo: (id: number) => void;
  removeTodo: (id: number) => void;
}

const TodoItem: React.FC<TodoItemProps> = ({ todo, toggleTodo, removeTodo }) => {
  return (
    <li>
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => toggleTodo(todo.id)}
      />
      <span style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}>
        {todo.text}
      </span>
      <button onClick={() => removeTodo(todo.id)}>Remove</button>
    </li>
  );
};

export default TodoItem;
```

Step 4: TodoListコンポーネントの更新

```tsx
// src/components/TodoList.tsx
import React, { useState } from 'react';
import TodoItem from './TodoItem';  // 追加
import { Todo } from '../types';

const TodoList: React.FC = () => {
  const [todos, setTodos] = useState<Todo[]>([]);

  // ...

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new todo"
      />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map(todo => (
          <TodoItem
            key={todo.id}
            todo={todo}
            toggleTodo={toggleTodo}
            removeTodo={removeTodo}
          />
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

Step 5: アプリケーションの組み立て

```tsx
// src/App.tsx
import React from 'react';
import TodoList from './components/TodoList';

const App: React.FC = () => {
  return (
    <div>
      <h1>Todo App</h1>
      <TodoList />
    </div>
  );
};

export default App;
```

Step 6: カスタムフック - useTodos

- カスタムフックを作成して、Todoの状態管理ロジックを分離

```
// src/hooks/useTodos.ts
import { useState } from 'react';
import { Todo } from '../types';

export const useTodos = () => {
  const [todos, setTodos] = useState<Todo[]>([]);

  const addTodo = (text: string) => {
    setTodos([...todos, { id: Date.now(), text, completed: false }]);
  };

  const toggleTodo = (id: number) => {
    setTodos(todos.map(todo =>
      todo.id === id ? { ...todo, completed: !todo.completed } : todo
    ));
  };

  const removeTodo = (id: number) => {
    setTodos(todos.filter(todo => todo.id !== id));
  };

  return { todos, addTodo, toggleTodo, removeTodo };
};
```

Step 7: TodoListコンポーネントの更新（カスタムフック使用）

```tsx
// src/components/TodoList.tsx
import React, { useState } from 'react';
import TodoItem from './TodoItem';
import { useTodos } from '../hooks/useTodos';

const TodoList: React.FC = () => {
  const { todos, addTodo, toggleTodo, removeTodo } = useTodos();
  const [input, setInput] = useState<string>('');

  const handleAddTodo = () => {
    if (input.trim() !== '') {
      addTodo(input.trim());
      setInput('');
    }
  };

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new todo"
      />
      <button onClick={handleAddTodo}>Add</button>
      <ul>
        {todos.map(todo => (
          <TodoItem
            key={todo.id}
            todo={todo}
            toggleTodo={toggleTodo}
            removeTodo={removeTodo}
          />
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

Step 8: エラー処理の追加

- エラー処理を追加して、アプリケーションの堅牢性を向上させる
- エラー文の文言を表示するための `error` ステートを追加

```
// src/hooks/useTodos.ts
import { useState } from 'react';
import { Todo } from '../types';

export const useTodos = () => {
  const [todos, setTodos] = useState<Todo[]>([]);
  const [error, setError] = useState<string | null>(null);

  const addTodo = (text: string) => {
    if (text.trim() === '') {
      setError('Todo text cannot be empty');
      return;
    }
    setTodos([...todos, { id: Date.now(), text, completed: false }]);
    setError(null);
  };

  // 次ページに続く
};
```

Step 8: エラー処理の追加（続き）

```
// ...

export const useTodos = () => {
  // ...

  const toggleTodo = (id: number) => {
    const todoToToggle = todos.find(todo => todo.id === id);
    if (!todoToToggle) {
      setError(`Todo with id ${id} not found`);
      return;
    }
    setTodos(todos.map(todo =>
      todo.id === id ? { ...todo, completed: !todo.completed } : todo
    ));
    setError(null);
  };

  const removeTodo = (id: number) => {
    const todoToRemove = todos.find(todo => todo.id === id);
    if (!todoToRemove) {
      setError(`Todo with id ${id} not found`);
      return;
    }
    setTodos(todos.filter(todo => todo.id !== id));
    setError(null);
  };

  return { todos, addTodo, toggleTodo, removeTodo, error };
};
```

Step 9: エラー表示の追加

TodoListコンポーネントにエラー表示を追加

```tsx
// src/components/TodoList.tsx
// ...

const TodoList: React.FC = () => {
  const { todos, addTodo, toggleTodo, removeTodo, error } = useTodos();  // errorを追加

  // ...

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new todo"
      />
      <button onClick={handleAddTodo}>Add</button>
      {error && <p style={{ color: 'red' }}>{error}</p>}
      <ul>
        {todos.map(todo => (
          <TodoItem
            key={todo.id}
            todo={todo}
            toggleTodo={toggleTodo}
            removeTodo={removeTodo}
          />
        ))}
      </ul>
    </div>
  );
};

export default TodoList;
```

Step 10: ビルド

- Reactプロジェクトは、ビルドすることで本番環境での実行に適したコードを生成できる
- 最終的には index.html と bundle.js などの静的ファイルが生成される

```
npm run build
```

次のステップ

- ローカルストレージを使用した永続化
- フィルタリング機能の追加（完了済み、未完了のTodoの表示）
- ユニットテストの追加
- デザインの改善（CSSフレームワークの導入）

まとめ

1. 型定義：`Todo`インターフェースを作成
2. コンポーネント分割：`TodoList`と`TodoItem`
3. カスタムフック：`useTodos`で状態管理ロジックを分離
4. エラー処理：ユーザー入力と操作のバリデーション
5. TypeScriptの利点：
    - 型安全性によるバグの早期発見
    - コード補完によるDX（開発者体験）の向上
    - 自己文書化コードによる可読性の向上

7. コンポーネントのパターン

高階コンポーネント (Higher-Order Component)

- コンポーネントを受け取り、新しいコンポーネントを返す関数

```tsx
// withLoader.tsx
import React from 'react';

interface WithLoaderProps {
  loading: boolean;
}

function withLoader<P extends object>(
  WrappedComponent: React.ComponentType<P>
) {
  return class extends React.Component<P & WithLoaderProps> {
    render() {
      const { loading, ...props } = this.props;
      return loading ? (
        <div>Loading...</div>
      ) : (
        <WrappedComponent {...(props as P)} />
      );
    }
  };
}

export default withLoader;
```

HOCの使用例

```tsx
// MyComponent.tsx
import React from 'react';
import withLoader from './withLoader';

interface MyComponentProps {
  name: string;
}

const MyComponent: React.FC<MyComponentProps> = ({ name }) => {
  return <div>Hello, {name}!</div>;
};

export default withLoader(MyComponent);

// Usage
<MyComponentWithLoader loading={isLoading} name="John" />
```

レンダープロップ

- コンポーネント間でコードを共有するためのテクニック

```tsx
// MouseTracker.tsx
import React, { useState } from 'react';

interface MousePosition {
  x: number;
  y: number;
}

interface MouseTrackerProps {
  render: (mousePosition: MousePosition) => React.ReactNode;  // レンダープロップ
}

const MouseTracker: React.FC<MouseTrackerProps> = ({ render }) => {
  const [mousePosition, setMousePosition] = useState<MousePosition>({ x: 0, y: 0 });

  const handleMouseMove = (event: React.MouseEvent) => {
    setMousePosition({ x: event.clientX, y: event.clientY });
  };

  return (
    <div style={{ height: '100vh' }} onMouseMove={handleMouseMove}>
      {render(mousePosition)}
    </div>
  );
};

export default MouseTracker;
```

レンダープロップの使用例

- `children` でも同様のことができるが、レンダープロップの方が柔軟性が高い
  - 複数のコンポーネントを渡せる
  - 親から孫コンポーネントにデータを渡すことができる
    - 複数の親や子コンポーネントを使うとき、振る舞いを共通化できる

```tsx
// App.tsx
import React from 'react';
import MouseTracker from './MouseTracker';

const App: React.FC = () => {
  return (
    <MouseTracker
      render={({ x, y }) => (
        <h1>The mouse position is ({x}, {y})</h1>
      )}
    />
  );
};
```

コンポーネント合成

- 複数の小さなコンポーネントを組み合わせて、より複雑なUIを構築する手法
- 例: `TodoList` と `TodoItem` を組み合わせて、Todoアプリを構築
- コンポーネント合成の利点:
  - コンポーネントの再利用性が向上
  - コンポーネントの責務が明確になる

8. スタイリング in React

CSS Modules

CSS Modulesを使用すると、スタイルをローカルスコープに限定できる

```
/* Button.module.css */
.button {
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.button:hover {
  background-color: #0056b3;
}
```

CSS Modulesの使用例

```tsx
// Button.tsx
import React from 'react';
import styles from './Button.module.css';

interface ButtonProps {
  onClick: () => void;
  children: React.ReactNode;
}

const Button: React.FC<ButtonProps> = ({ onClick, children }) => (
  <button className={styles.button} onClick={onClick}>
    {children}
  </button>
);

export default Button;
```

Styled-components

- CSS-in-JSライブラリ
- コンポーネントとスタイルを密接に結びつけることができる

```
import styled from 'styled-components';

interface ButtonProps {
  primary?: boolean;
}

// スタイル付きの <button> タグに対応するButtonコンポーネント
const Button = styled.button<ButtonProps>`
  background: ${props => props.primary ? "palevioletred" : "white"};
  color: ${props => props.primary ? "white" : "palevioletred"};
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;
```

Styled-componentsの使用例

```
// 使用例
const Example: React.FC = () => (
  <div>
    <Button>Normal Button</Button>
    <Button primary>Primary Button</Button>
  </div>
);
```

Styled-componentsとTypeScript

Styled-componentsはTypeScriptと相性が良く、型安全なスタイリングが可能

```
import styled from 'styled-components';

interface ThemeProps {
  theme: {
    primary: string;
    secondary: string;
  }
}

const Button = styled.button<ThemeProps>`
  background-color: ${props => props.theme.primary};
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;

  &:hover {
    background-color: ${props => props.theme.secondary};
  }
`;

// ThemeProviderの使用
import { ThemeProvider } from 'styled-components';

const theme = {
  primary: '#007bff',
  secondary: '#6c757d'
};

const App: React.FC = () => (
  <ThemeProvider theme={theme}>
    <Button>Themed Button</Button>
  </ThemeProvider>
);
```

Emotion

- Styled-componentsと似た機能を持つもう一つの人気のあるCSS-in-JSライブラリ

```
/** @jsxImportSource @emotion/react */
import { css } from '@emotion/react';

const buttonStyle = css`
  background-color: hotpink;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 4px;
  cursor: pointer;

  &:hover {
    background-color: deeppink;
  }
`;
```

Emotionの使用例

```
const Button: React.FC = ({ children }) => (
  <button css={buttonStyle}>{children}</button>
);
```

Tailwind CSS

Tailwind CSSは、ユーティリティファーストのCSSフレームワークで、事前定義されたクラスを使用してスタイリングを行う

```
const Button: React.FC = ({ children }) => (
  <button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
    {children}
  </button>
);
```

Tailwind CSSとTypeScriptを組み合わせる場合

- 追加の型定義は必要ないが、クラス名の自動補完のためにVSCode拡張機能を使用すると便利

Tailwind CSSの複雑なスタイリング

Bootstrapと違い動的にスタイルシートを生成するため、複雑なスタイリングも可能

```tsx
import React from 'react';

const ComplexStylingExample: React.FC = () => {
  return (
    <div className="p-4 bg-gray-100">
      <h1 className="text-2xl font-bold mb-4">Complex Tailwind CSS Styling</h1>

      {/* List with complex styling */}
      <ul className="space-y-4">
        {[1, 2, 3, 4, 5].map((item) => (
          <li
            key={item}
            className={`
              relative
              ...  // 他のクラス名

              /* スマートフォン表示で3n+1番目の要素にホバー時のafter疑似要素のスタイル */
              sm:nth-child(3n+1):hover:after:content-['']
              sm:nth-child(3n+1):hover:after:absolute
              ...  // 他のクラス名
            `}
          >
            <h2 className="text-lg font-semibold mb-2">Item {item}</h2>
            <p className="text-gray-600">
              This is a complex styled item using Tailwind CSS.
            </p>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ComplexStylingExample;
```

スタイリング手法の比較

1. CSS Modules:
    - 👍 スコープ付きCSS、既存のCSSスキルを活用可能
    - 👎 動的スタイリングが少し面倒
2. Styled-components / Emotion:
    - 👍 コンポーネントと密接に結合、動的スタイリングが容易
    - 👎 新しい構文を学ぶ必要がある、ビルドサイズが大きくなる可能性
3. Tailwind CSS:
    - 👍 高速な開発、一貫したデザイン
    - 👎 クラス名が長くなる、カスタマイズが必要な場合は設定が複雑

9. パフォーマンス最適化

メモ化（React.memo, useMemo, useCallback）

メモ化は不要な再レンダリングを防ぐ重要なテクニック

```
import React, { useMemo, useCallback } from 'react';

interface Item {
  id: number;
  name: string;
}

interface ItemListProps {
  items: Item[];
  onItemClick: (item: Item) => void;
}

// メモ化されたItemListコンポーネント
// itemsかonItemClickが変更されない限り、再レンダリングされない
const ItemList: React.FC<ItemListProps> = React.memo(({ items, onItemClick }) => {
  console.log('ItemList rendered');

  return (
    <ul>
      {items.map(item => (
        <li key={item.id} onClick={() => onItemClick(item)}>
          {item.name}
        </li>
      ))}
    </ul>
  );
});
```

React.memo

```
const App: React.FC = () => {
  const [items, setItems] = React.useState<Item[]>([
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
  ]);
  const [count, setCount] = React.useState(0);

  // itemsの値が変更されない限り、ソート処理は再実行されない
  const sortedItems = useMemo(() => {
    console.log('Sorting items');
    return [...items].sort((a, b) => a.name.localeCompare(b.name));
  }, [items]);

  // 初回のみ関数が作成され、2回目以降は同じ関数が再利用される
  const handleItemClick = useCallback((item: Item) => {
    console.log('Item clicked:', item.name);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <ItemList items={sortedItems} onItemClick={handleItemClick} />
    </div>
  );
};
```

コード分割とLazy Loading

React.lazyとSuspenseを使用して、コンポーネントを動的にインポートし、アプリケーションの初期ロード時間を改善できる

```
import React, { Suspense } from 'react';

const HeavyComponent = React.lazy(() => import('./HeavyComponent'));

const App: React.FC = () => {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <HeavyComponent />
      </Suspense>
    </div>
  );
};
```

ビルトインのプロファイラの使用

React DevToolsには、コンポーネントの再レンダリングを分析するためのプロファイラが含まれている

1. React DevToolsをインストール
2. 開発者ツールでProfilerタブを選択
3. 記録を開始し、アプリケーションを操作
4. 結果を分析し、最適化が必要な箇所を特定

Webパフォーマンス最適化

1. コードの分割とLazy Loading
2. 画像の最適化（適切なサイズ、フォーマット、遅延読み込み）
3. キャッシュの活用
4. サーバーサイドレンダリング（SSR）やIncrementalのStatic Generation（ISG）の使用（Next.jsなど）

10. テスト

JestとReact Testing Library

JestはJavaScriptのテストフレームワークで、React Testing Libraryと組み合わせて使用されます。

```
// Button.tsx
import React from 'react';

interface ButtonProps {
  onClick: () => void;
  children: React.ReactNode;
}

const Button: React.FC<ButtonProps> = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);

export default Button;
```

```tsx
// Button.test.tsx
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import Button from './Button';

describe('Button', () => {
  it('renders children correctly', () => {
    const { getByText } = render(<Button onClick={() => {}}>Click me</Button>);
    expect(getByText('Click me')).toBeInTheDocument();
  });

  it('calls onClick when clicked', () => {
    const handleClick = jest.fn();
    const { getByText } = render(<Button onClick={handleClick}>Click me</Button>);
    fireEvent.click(getByText('Click me'));
    expect(handleClick).toHaveBeenCalledTimes(1);
  });
});
```

カスタムフックのテスト

カスタムフックは、`renderHook`を使用してテスト

```
// useCounter.ts
import { useState, useCallback } from 'react';

export const useCounter = (initialValue: number = 0) => {
  const [count, setCount] = useState(initialValue);
  const increment = useCallback(() => setCount(prev => prev + 1), []);
  const decrement = useCallback(() => setCount(prev => prev - 1), []);
  return { count, increment, decrement };
};

// useCounter.test.ts
import { renderHook, act } from '@testing-library/react-hooks';
import { useCounter } from './useCounter';

describe('useCounter', () => {
  it('should increment counter', () => {
    const { result } = renderHook(() => useCounter());
    act(() => {
      result.current.increment();
    });
    expect(result.current.count).toBe(1);
  });

  it('should decrement counter', () => {
    const { result } = renderHook(() => useCounter(10));
    act(() => {
      result.current.decrement();
    });
    expect(result.current.count).toBe(9);
  });
});
```

11. Next.jsの紹介

Next.jsとは

- Reactベースのフルスタックフレームワーク
- 簡単に言うと、Reactにサーバー機能を追加したもの

主な機能

1. ファイルベースのルーティング
2. サーバーサイドレンダリング（SSR）
3. 静的サイト生成（SSG）
4. APIルート
5. 自動コード分割

基本的なNext.jsアプリケーション

```tsx
// pages/index.tsx
import { NextPage } from 'next';
import Link from 'next/link';

const Home: NextPage = () => {
  return (
    <div>
      <h1>Welcome to Next.js!</h1>
      <Link href="/about">
        <a>About</a>
      </Link>
    </div>
  );
};

export default Home;

// pages/about.tsx
import { NextPage } from 'next';

const About: NextPage = () => {
  return <h1>About Page</h1>;
};

export default About;
```

データフェッチング

Next.jsは複数のデータフェッチング方法を提供する:

1. getStaticProps (静的サイト生成)
2. getServerSideProps (サーバーサイドレンダリング)
3. getStaticPaths (SSGの動的ルート)

1. getStaticProps (SSG)

- ビルド時にデータをフェッチしてページを生成
- 特徴:
  - ビルド時にHTMLが生成される
  - 高速なページロード
  - SEOに有利
  - データ更新頻度が低いページに適している

getStaticProps の使用例

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data')
  const data = await res.json()

  return {
    props: { data },
    revalidate: 60, // オプション：60秒ごとに再生成
  }
}
```

2. getServerSideProps (SSR)

- リクエストごとにサーバーサイドでデータをフェッチ
- 特徴:
  - リクエストのたびにデータがフェッチされる
  - 常に最新のデータを表示できる
  - サーバーの負荷が高くなる可能性がある
  - 初期ロードが遅くなる可能性がある

getServerSideProps の使用例

```
export async function getServerSideProps(context) {
  const { params, req, res } = context
  const { id } = params

  const response = await fetch(`https://api.example.com/data/${id}`)
  const data = await response.json()

  return { props: { data } }
}
```

3. getStaticPaths (SSGの動的ルート)

- 動的ルーティングを使用する静的生成ページのためのメソッド
- 特徴:
  - 動的ルートを持つページで使用
  - getStaticPropsと組み合わせて使用
  - ビルド時に指定されたパスの静的ページを生成

getStaticPaths の使用例

```
export async function getStaticPaths() {
  const res = await fetch('https://api.example.com/posts')
  const posts = await res.json()

  const paths = posts.map((post) => ({
    params: { id: post.id.toString() },
  }))

  return { paths, fallback: false }
}

export async function getStaticProps({ params }) {
  const res = await fetch(`https://api.example.com/posts/${params.id}`)
  const post = await res.json()

  return { props: { post } }
}
```

`getStaticProps` vs `getServerSideProps`

- 静的コンテンツや更新頻度が低いページ → getStaticProps
- ユーザー固有データや頻繁に更新されるデータ → getServerSideProps

12. まとめと次のステップ

まとめ

1. TypeScriptとReactの組み合わせにより、型安全性と開発者体験が向上
2. モダンなReactの機能（Hooks、コンポーネントパターン）の活用
3. 適切なスタイリング手法の選択
4. パフォーマンス最適化の重要性
5. テストによる品質保証
6. Next.jsによる本格的なアプリケーション開発

次のステップ

1. より高度なTypeScriptの機能の学習（Generics、Conditional Types等）
2. 状態管理ライブラリの探求（Redux, MobX, Recoil等）
3. GraphQLの学習とApollo Clientの使用
4. マイクロフロントエンドアーキテクチャの探求
5. CI/CDの構築
6. アクセシビリティ（a11y）, 多言語化（i18n）

多分実際に何か作ってみるのが一番効率的！！

本来はここで終わっても良かったのだが、もう少しReact.jsとTypeScriptについて深く学びたい方向けに、次のスライドに進みます。

React.jsとTypeScript: 次のステップ

1. より高度なTypeScriptの機能の学習

- Generics: 型の再利用性を高める

```
function identity<T>(arg: T): T {
  return arg;
}
```

- Conditional Types: 条件に基づいて型を選択

```
type Check<T> = T extends string ? "string" : "not string";
```

- Mapped Types: 既存の型から新しい型を作成

```
type Readonly<T> = { readonly [P in keyof T]: T[P] };
```

2. 状態管理ライブラリの探求

- Redux: 予測可能な状態コンテナ
  - 大規模アプリケーションに適している
  - 豊富なミドルウェアエコシステム
- MobX: シンプルで、スケーラブルな状態管理
  - 反応的プログラミングモデル
  - ボイラープレートが少ない
- Recoil: Reactに特化した状態管理
  - Atom-based
  - React Suspenseとの親和性が高い

3. GraphQLとRESTの学習

REST (REpresentational State Transfer)

- HTTPメソッドを使用してリソースを操作
- エンドポイントベースのAPI設計
- 利点:
  - シンプルで直感的
  - キャッシュが容易
- 欠点:
  - オーバーフェッチング/アンダーフェッチングの問題

例:

```
fetch('https://api.example.com/users/1')
  .then(response => response.json())
  .then(data => console.log(data));
```

3. GraphQLとRESTの学習 (続き)

GraphQL

- クエリ言語とランタイム
- クライアントが必要なデータを正確に指定可能
- Apollo Clientを使用した例:

```
const GET_USER = gql`
  query GetUser($id: ID!) {
    user(id: $id) {
      name
      email
    }
  }
`;

const { loading, error, data } = useQuery(GET_USER, {
  variables: { id: "1" },
});
```

4. マイクロフロントエンドアーキテクチャの探求

- 大規模フロントエンドアプリケーションを小さな部分に分割
- 利点:
  - チーム間の独立した開発が可能
  - 段階的なアップグレードが容易
- 実装方法:
  - iframeを使用
  - Web Componentsを使用(おすすめ)
  - Module Federationを使用 (Webpack 5)

5. CI/CDの構築

デプロイって毎回やるの面倒ですよね。CI/CDを導入することで、自動化しましょう！

- 継続的インテグレーション (CI):
  - 自動テスト
  - コード品質チェック
- 継続的デリバリー/デプロイメント (CD):
  - 自動ビルド
  - 自動デプロイ
- ツール:
  - GitHub Actions
  - GitLab CI/CD
  - CircleCI

6. アクセシビリティ（a11y）と多言語化（i18n）

アクセシビリティ (a11y)

- スクリーンリーダーの対応
- キーボードナビゲーション
- 色のコントラスト
- WAI-ARIAの使用

多言語化 (i18n)

- react-intlやnext-i18nextの使用
- 翻訳管理システムの導入
- 右から左へ書く言語 (RTL) のサポート

実践的アプローチ

実際に何か作ってみるのが一番効率的！

1. 小規模プロジェクトから始める
2. 学んだ技術を段階的に適用
3. オープンソースプロジェクトへの貢献
4. コードレビューを積極的に受ける
5. 技術ブログの執筆や勉強会での発表

まとめ

- TypeScriptの高度な機能を習得
- 適切な状態管理ライブラリを選択
- RESTとGraphQLの違いを理解し、適材適所で使用
- マイクロフロントエンドで大規模アプリケーションに対応
- CI/CDでデプロイメントを自動化
- アクセシビリティと多言語化で、より多くのユーザーに対応

継続的な学習と実践が、スキル向上の鍵となる！