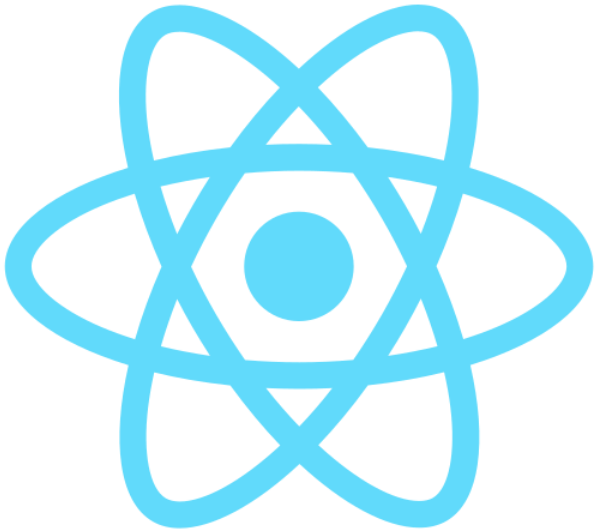


React 勉強会 - React の思想からパフォーマンス最適化まで -

リードエンジニア 鈴木 慎太郎

1. React の特徴

1.1. 宣言的 UI: `UI = f(state)`

1.2. コンポーネントベース

1.1. 宣言的 UI: `UI = f(state)`

React において、UI (見た目) は state (状態) の写像です。

有機的に見える UI についても、内部的には状態の変更を検知してゼロから画面が再構築されています。

つまり、UI の変更の手続きを記述する必要がなく、状態に対する結果としての UI を記述するだけでよいのです。

1.2. コンポーネントベース

コンポーネントとは画面におけるパーツのことを指します。

ブラウザは、DOM ツリーと呼ばれる HTML 要素(`<div>` , `<ul>` , `<p>` など) とテキストノードからなる木構造を解釈して画面を描画しますが、React に代表されるコンポーネントベースの UI ライブラリを使うと、DOM ツリーの一部分のみを出力することができます。

そのため、巨大な DOM ツリーを管理する必要がなく、分割して部分ごとに管理ができるので、開発が容易になります。

また、作成したコンポーネントを複数の画面で使い回すことができるので、開発効率の向上も見込めます。

2. React のレンダリング

2.1. レンダリングプロセスの概要

2.2. レンダリングが発生する条件

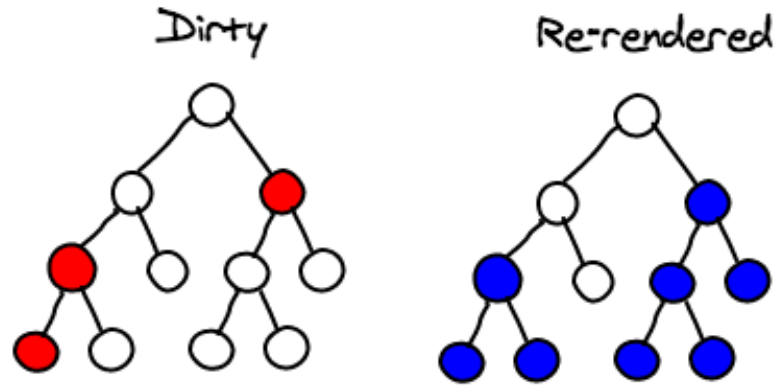
2.1. レンダリングプロセスの概要

```
// jsx
return <Component label="dummy">Text</Component>;

/* ↓ JS にコンパイル ↓ */

// js
return React.createElement(Component, { label: "dummy" }, "Text");

/* ↓ React DOM の出力 ↓ */

// React DOM
{ type: Component, props: { label: "dummy" }, children: ["Text"]}
```

2.1. レンダリングプロセスの概要

React では React DOM ツリーの変更を検知して、差分のある箇所についてのみ再レンダリングを効率的に行います。

[Performance Calendar >> React's diff algorithm]



2.2. レンダリングが発生する条件

コンポーネントは **props** と **state** を持っています。どちらもレンダリングに影響を及ぼしますが、下記のような違いがあります。

- **props**: コンポーネントの外部から渡される
- **state**: コンポーネントの内部で制御される

2.2. レンダリングが発生する条件

下記の例では、`label` が **props**、`name` が **state** となります。

```
const InputField = ({ label }) => {
    const [name, setName] = useState(name);

    return (
        <>
            <label>{label}</label>
            <input onChange={(e) => setName(e.target.value)} />
        </>
    );
};
```

2.2. レンダリングが発生する条件

React のレンダリングのタイミングは基本的に下記の 3 つです。

レンダリングの回数を必要最小限にし、パフォーマンスを最適化するためには、いつレンダリングが発生するのかを意識することが重要です。

1. **state が更新されたとき**
2. **props が更新されたとき**
3. **親コンポーネントが再レンダリングされたとき**

3. React Hooks と パフォーマンス最適化

3.1. useState

3.2. useEffect

3.3. React.memo

3.4. useMemo

3.5. useCallback

3.6. useReducer

3.7. useContext

3.8. カスタムフック

3.1. useState

useState は state を内部で制御するためのフックです。

下記の例では count が **state** となり、setCount が **state** を制御するための関数になっています。

```
const Component = () => {
    const [count, setCount] = useState(0);
    const handleClick = () => {
        setCount(count + 1);
    };
    return (
        <>
            <p>{count}</p>
            <button onClick={handleClick}>+1</button>
        </>
    );
};
```

3.1. useState

しかし、先ほどの例では、setCount 内で count(state) を参照しているため、ボタンを高速でクリックすると、正しく動作しません。

再レンダリングが完了する前に 2 回目の setCount を呼ぶと、更新前の count を参照してしまうためです。

```
const handleClick = () => {
    setCount(count + 1);
};
```

3.1. useState

React 要素はイミュータブルです。UI = f(state) を思い出してください。

UI(見た目) は state(状態) に応じて完全にゼロから再構築されるためです。

つまり、厳密には state が「更新される」という表現は正しくありません。state が 0 → 1 に更新されているように見えても、state = 0 の画面を廃棄して state = 1 の画面を再描画しているだけです。

※ なので state を宣言するときは `const` を使うわけです。

3.1. useState

これを回避するために、いくつか方法がありますが、setCount の引数にコールバックを渡すという方法があります。

こうすることで、その画面の count に依存せず、前の状態を参照して画面を更新することができます。

```
const handleClick = () => {
    setCount((prev) => prev + 1);
};
```

3.1. useState

state を増やすことはそのままレンダリングの条件を複雑化させることに繋がります。そのため、state はなるべく少なく保つべきです。

※ props の変更による再レンダリングは state に比べ限定的なので、そこまで props を減らすことに注力する必要はないです。

3.2. useEffect

useEffect はレンダリング時に副作用を起こすためのフックです。

レンダリングを検知して、任意の処理を行うことができます。

```
const Example = () => {
    const [conut, setCount] = useState(0);

    useEffect(() => {
        // Update the document title using the browser API
        document.title = `You clicked ${count} times`;
    });

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>Click me</button>
        </div>
    );
};
```

3.2. useEffect

useEffect は新しい公式ドキュメントにて React のパラダイムからのエスケープハッチであると明言されています。

ネットワークやブラウザの DOM などの外部システムが関与しない場合は、**useEffect** は必要ありません。

不要な **useEffect** をなくすことで、より保守性・パフォーマンスが高いコードを書くことができます。

3.2. useEffect

❌ **useEffect** が不要な例 1

既存の state や props から計算できる値を state にする必要はありません。また、外部システムが関与しない場合 **useEffect** は不要です。

```
const Comnponent = () => {
    const [firstName, setFirstName] = useState('Daniel')
    const [lastName, setLastName] = useState('Radcliffe')

    const fullName, setFullName] = useState('')
    useEffect(() => {
        setFullName(firstName + ' ' + lastName);
    }, [firstName, lastName])

    return (...)
}
```

3.2. useEffect

⭕ useEffect が不要な例 1

state の更新に伴って、コンポーネントが再レンダリングされるため、コンポーネント内の変数は再定義されます。

```
const Component = () => {
    const [firstName, setFirstName] = useState("Daniel");
    const [lastName, setLastName] = useState("Radcliffe");

    const fullName = firstName + " " + lastName;

    return (...)
}
```

3.2. useEffect

❌ **useEffect** が不要な例 2

先ほどと同様に受け取った props を操作するのに **useEffect** を使う必要はありません。

```
const Component = ({ todos }) => {
    const [visibleTodos, setVisibleTodos] = useState([]);

    useEffect(() => {
        setVisibleTodos(getFilteredTodos(todos, filter));
    }, [todos, filter]);

    return (...)
};
```

3.2. useEffect

⭕ useEffect が不要な例 2

コンポーネント内のローカル変数は、再レンダリングと同時に再評価されるので、下記のコードで実現できます。

```
const Component = ({ todos }) => {
    const visibleTodos = getFilteredTodos(todos, filter);

    return (...)
};
```

3.3. React.memo

コンポーネントのメモ化を行い、親コンポーネントの再レンダリングによって子コンポーネントが再レンダリングしないようにします。

props の変更があった場合や、メモ化したコンポーネント内部の state が更新された場合は、通常通り再レンダリングされます。

```
const MemoizedComponent = React.memo(({ label, onClick }) => {
    <button onClick={onClick}>{label}</button>;
});
```

3.4. useMemo

useMemo はコンポーネント内部の値をメモ化するフックです。

下記のようなレンダリングのたびに重い計算を行うコンポーネントを考えてみましょう。

```
const Component = () => {
    const [name, setName] = useState('')
    const [count, setCount] = useState(0);

    const square = () => {
        let i = 0;
        while (i < 200000000000) i++;
        return count * count;
    };

    return (...)
};
```

3.4. useMemo

count が変わらない限り、何度計算をしても結果は変わらないのにも関わらず、レンダリングのたびに重い計算を行うのは非効率なので、**useMemo** を使って値をメモ化します。

```
const Component = () => {
    const [name, setName] = useState('')
    const [count, setCount] = useState(0);

    const square = useMemo(() => {
        let i = 0;
        while (i < 200000000000) i++;
        return count * count;
    }, [count]);

    return (...)
};
```

3.4. useMemo

useMemo は計算結果をキャッシュするだけでなく、**React.memo** と併用することでレンダリングを制御することもできます。むしろこの使い方がよく見られます。

下記の例において `MemoizedChild` はメモ化されたコンポーネントですが、この場合 `Parent` が再レンダリングされるたびに `MemoizedChild` が再レンダリングされてしまいます。

```
const Parent = () => {
    const [age, setAge] = useState(0);
    const [name, setName] = useState("");

    const user = {
        name: name,
    };

    return <MemoizedChild user={user} />;
};
```

3.4. useMemo

JavaScript を含むほとんどの言語の等価比較はメモリアドレスの比較です。

プリミティブ値（`0`, `"text"` など) が持つメモリアドレスは、データそのものを指すため、同じ値を持つ 2 つの変数は等価と判定されます。

```
const a = 0;
const b = 0;

console.log(a === b);
// true
```

3.4. useMemo

一方で、オブジェクト値(`[0, 1, 2]`, `{ a: 1 }` など)が持つメモリアドレスは、データの保存先を指します(参照渡し)。

そのため、内部の値が同じでも、別々に宣言されていたなら、等価でないと判定されます。

```
const a = {};
const b = {};

console.log(a === b);
// false
```

3.4. useMemo

コンポーネント内の変数は、コンポーネントの再レンダリングに伴い再定義されることを思い出してください。

先程の例で `user` はオブジェクトだったので、レンダリングによって変数が再定義されてしまうと、中身が変わってなかったとしても、「前の状態と props が異なる」と判定され、メモ化しているにも関わらず常に再レンダリングが走ります。

```
const user = {
    name: name,
};

return <MemoizedChild user={user} />;
```

3.4. useMemo

useMemo を用いてキャッシュを行うことにより、変数が再定義されるのを回避することができ、前の props と今回の props が等価であると判定され、意図しない子コンポーネントのレンダリングをスキップすることができます。

```
const Parent = () => {
    const [age, setAge] = useState(0);
    const [name, setName] = useState("");

    const user = useMemo(
        () => ({
            name: name,
        }),
        [name]
    );

    return <MemoizedChild user={user} />;
};
```

3.5. useCallback

useCallback は関数のメモ化を行うフックです。JavaScript においては関数もオブジェクトなので、**useCallback** を使わないと、メモ化した子コンポーネントが意図せず再レンダリングされることに繋がります。

```
const Parent = () => {
    const [count, setCount] = useState(0);

    const handleClick = useCallback(() => {
        setCount((prev) => prev + 1);
    }, [setCount]);

    return <MemoizedChildButton onClick={handleClick} />;
};
```

3.5. useCallback

お気づきかもしれませんが、**useCallback** は **useMemo** の特殊形です。関数に特化した **useMemo** とも言い換えられます。

先程の例は **useMemo** を使って、下記のように書き換えられます。

```
const handleClick = useMemo(
    () => () => {
        setCount((prev) => prev + 1);
    },
    [setCount]
);
```

3.6. useReducer

useReducer は **useState** と同様に state を管理するためのフックです。

```
const initialState = { count: 0 };

const reducer = (state, action) =>  {
    switch (action.type) {
        case 'increment':
            return { count: state.count + 1 };
        case: 'decrement':
            return { count: state.count - 1 };
        default:
            throw new Error();
    }
}

const Counter = () => {
    const [state, dispatch] = useReducer(reducer, initialState)
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
            <button onClick={() => dispatch({ type: 'increment' })}>+</button>
        </>
    )
}
```

3.6. useReducer

useReducer は **useState** の一般形です。

第一引数に `(state, action) => newState` という型の reducer を、第二引数に initialState (初期値) を渡します。

下記の 2 つのコードは同値です。

```
const [count, setCount] = useState(0);

const [count, setCount] = useReducer((_, newCount) => newCount, 0);
```

3.6. useReducer

下記のようなコンポーネントを考えます。

```
const Combobox = ({ options }) => {
    const [items, setItems] = useState([]);
    const [isCreating, setCreating] = useState(false);

    const onChange = (e) => {
        const filteredItems = options.filter((option) =>
            option.startsWith(e.target.value)
        );
        if (filteredItems.length > 0) {
            setItems(filteredItems);
            setCreating(false);
        } else {
            setItems([e.target.value]);
            setCreating(true);
        }
    };

    return (
        <>
            <input onChange={onChange} />
            {items.map((item) => {
                <p key={item}>{`${item}${isCreating ? "を新規作成" : ""}`}</p>;
            })}
        </>;
    )
};
```

3.6. useReducer

このコードは、入力値が変わるたびに 2 回 state を更新してしまているため、パフォーマンスが悪いです。

```
const onChange = (e) => {
    const filteredItems = options.filter((option) =>
        option.startsWith(e.target.value)
    );
    if (filteredItems.length > 0) {
        setItems(filteredItems);
        setCreating(false);
    } else {
        setItems([e.target.value]);
        setCreating(true);
    }
};
```

3.6. useReducer

この例に限った話だと `isCreating` を消して、一致するものがない場合は下記のように `items` を更新すれば解決しますが、新規作成時のみに発火させたい処理がある場合を考慮し、やはり `isCreating` を state として保持しておきたいです。

```
setItems([`${e.target.value}を新規作成`]);
```

3.6. useReducer

useRef を使って、再レンダリングを起こさない state を作成する、という方法によって解決しますが、これは React18 から **非推奨** です。❌

```
const [items, setItems] = useState([]);
const isCreating = useRef(false);

const onChange = (e) => {
    const filteredItems = options.filter((option) =>
        option.startsWith(e.target.value)
    );
    if (filteredItems.length > 0) {
        setItems(filteredItems);
        isCreating.current = false;
    } else {
        setItems([e.target.value]);
        isCreating.current = true;
    }
};
```

3.6. useReducer

そのため、2 つの state を一元管理する、という方法をとります。

```
const initialState = {
    items: [],
    isCreating: false,
};

const reducer = (_state, action) => {
    const filteredItems = action.options.filter((option) =>
        option.startsWith(action.value)
    );
    if (filteredItems.length > 0) {
        return {
            items: filteredItems,
            isCreating: false,
        };
    } else {
        return {
            items: [action.value],
            isCreating: true,
        };
    }
};
```

3.6. useReducer

ロジックを reducer に隠蔽したため、コンポーネントは簡単に記述できます。

```
const Combobox = ({ options }) => {
    const [{ items, isCreating }, dispatch] = useReducer(reducer, initialState);

    const onChange = (e) => {dispatch({
        options,
        value: e.target.value
    })};

    return (
        <>
            <input onChange={onChange} />
            {items.map((item) => {
                <p key={item}>{`${item}${isCreating ? "を新規作成" : ""}`}</p>;
            })}
        </>;
    )
}
```

3.6. useReducer

実際には useState を使っても同じことを再現できますが、ロジックを reducer に隠蔽できるという点で **useReducer** が優れています。

この「state に依存するロジックを、state に非依存な関数 (reducer) で表現できる」というのが **useReducer** の本質です。

これにより、コンポーネント内部のロジック同士の依存度を下げ、パフォーマンスを向上させることができます。

3.6. useReducer

また、1 度だけ変更される state には useState より **useReducer** が優れています。

finished は `false` → `true` のみを期待していますが、誤って `true` → `false` という実装を行い、バグを起こす可能性があります。

```
const Component () => {
    const [finished, setFinished] = useState(false)
    const onClick = () => {
        setFinished(true)
    }

    if (finished) return <p>送信しました</p>

    return (
        <button onClick={onClick}>送信する</button>
    )
}
```

3.6. useReducer

useReducer を使うと、`false` → `true` という単方向の更新しかないことを明示できます。

こうすると、誤って `true` → `false` の実装をする可能性が減り、より堅牢なコンポーネントが書けます。

```
const Component () => {
    const [finished, setFinished] = useReducer(() => true, false)
    const onClick = () => {
        setFinished()
    }

    if (finished) return <p>送信しました</p>

    return (
        <button onClick={onClick}>送信する</button>
    )
}
```

3.7. useContext

useContext は、コンポーネント間で state を共有したいときに使用するフックです。

```
const CountContext = createContext();

const Component = () => {
    const [count, setCount] = useState(0);

    return (
        <CountContext.Provider value={{ count, setCount }}>
            <TextComponent />
            <ButtonComponent />
        </CountContext.Provider>
    );
};
```

3.7. useContext

`TextComponent`, `ButtonComponent` の中身は下記のようになっています。

useContext を使うことで、props のバケツリレーを回避できています。これはコンポーネントの階層が深くなったときに特に有効です。

`CountContext` に変更があったとき、これを参照しているコンポーネントは再レンダリングされます。

```
const TextComponent = () => {
    const { count } = useContext(CountContext);

    return <p>{count}</p>;
};

const ButtonComponent = () => {
    const { setCount } useContext(CountContext);

    return <button onClick={() => setCount((prev) => prev + 1)}>Click Me!</button>;
};
```

3.7. useContext

しかし、例によって先般のコードはパフォーマンスが悪いです。

`count` と `setCount` をまとめて管理しているため、`count` しか更新されていないのに、`setCount` のみを使用している `ButtonComponent` も再レンダリングが走ってしまいます。

```
return (
    <CountContext.Provider value={{ count, setCount }}>
        <TextComponent />
        <ButtonComponent />
    </CountContext.Provider>
);
```

3.7. useContext

Context を適切に分解することで、この問題を回避できます。

```
const CountContext = createContext();
const SetCountContext = createContext();

const Component = () => {
    const [count, setCount] = useState(0);

    return (
        <CountContext.Provider value={count}>
            <SetCountContext.Provider value={setCount}>
                <TextComponent />
                <ButtonComponent />
            </SetCountContext.Provider>
        </CountContext.Provider>
    );
};
```

3.7. useContext

しかし、Provider が再レンダリングされると、子コンポーネントが再レンダリングされてしまうので、Provider 配下のコンポーネントはすべて再レンダリングされてしまう、という問題が残っています。

とはいえ、Provider 配下のすべてのコンポーネントを **React.memo** をラップするのは現実的ではありません。

親コンポーネント起因の再レンダリングを回避する方法は、もう一つあります。

3.7. useContext

`props.children` を使えば、子コンポーネントの再レンダリングは発生しません。

そのため、Context ごとに Provider コンポーネントを作成することが有効です。

```
const CountProvider = ({ children }) => {
    return (
        <CountContext.Provider value={count}>{children}</CountContext.Provider>
    );
};

const SetCountProvider = ({ children }) => { ... };

const AppProvider = ({ children }) => {
    return (
        <CountProvider>
            <SetCountProvider>{children}</SetCountProvider>
        </CountProvider>
    );
};
```

3.7. useContext

useContext を使う際に気をつける点

- Context を適切に分割すること
- Context 毎に Provider コンポーネントを作成すること

パフォーマンスに留意しながら **useContext** を使うと、Context を無数に作らなければならず、負荷が高いです。このようなことを考えたくなければ、**Redux** や **Recoil** を使ったほうがよいでしょう。

3.8. カスタムフック

カスタムフック は React Hooks を含むロジックをコンポーネントから切り離して、再利用可能な関数をします。

```
const useCombobox = (options) => {
    const [{ items, isCreating }, dispatch] = useReducer(reducer, {
        items: options
        isCreating: false
    });

    const onChange = (value) => {dispatch({
        options,
        value
    })};

    return {
        items,
        isCreating,
        onChange
    }
}
```

3.8. カスタムフック

カスタムフック を使うと、ロジックをカスタムフック内に隠蔽しているので、コンポーネントを簡潔に記述することができます。

```
const Combobox = ({ options }) => {
    const {items, isCreating, onChange: onComboChange } = useCombobox(options);

    return (
        <>
            <input onChange={(e) => onComboChange(e.target.value)} />
            {items.map((item) => {
                <p key={item}>{`${item}${isCreating ? "を新規作成" : ""}`}</p>;
            })}
        </>;
    )
}
```

3.8. カスタムフック

`<input>` は state (`items` , `isCreating`) に依存していないので、無駄な再レンダリングを避けるためにこれをメモ化します。

しかし、先般の **カスタムフック** の実装だと、これはうまく機能せず、`<MemoizedInput>` も、state の更新のたびに再レンダリングされてしまいます。

```
const Combobox = ({ options }) => {
    const {items, isCreating, onChange: onComboChange } = useCombobox(options);

    return (
        <>
            <MemoizedInput onChange={onComboChange} />
            {items.map((item) => {
                <p key={item}>{`${item}${isCreating ? "を新規作成" : ""}`}</p>;
            })}
        </>;
    )
}
```

3.8. カスタムフック

カスタムフック内の `onChange` がメモ化されておらず、state の更新のたびに参照が変わっていた(変数が再定義された)ことが原因です。

```
const useCombobox = (options) => {
    const [{ items, isCreating }, dispatch] = useReducer(reducer, {
        items: options
        isCreating: false
    });

    const onChange = useCallback((value) => {dispatch({
        options,
        value
    })}, [options]);

    return {
        items,
        isCreating,
        onChange
    }
}
```

3.8. カスタムフック

ここでの示唆は「**カスタムフックの返り値は、必ずメモ化せよ**」ということです。

カプセル化・汎用化を目的としたカスタムフックにおいて、再利用可能性と独立性を高めることは重要です。

メモ化しないと、カスタムフック側のロジックによって、意図しない再レンダリングが発生してしまうので、カスタムフックの再利用可能性が低くなってしまいます。

参考文献

- You Might Not Need an Effect - React
- 【React】1 度だけ変更される state には useState より useReducer を使う方が最適
- useContext + useState 利用時のパフォーマンスは Provider の使い方で決まる！かも。。。？ - Qiita
- useCallback はとにかく使え！特にカスタムフックでは - uhyo/blog