



全4回で学ぶReact.js入門講座 3回目

2025/6/24

Kazuma SEKIGUCHI

前回のアジェンダ

- CSSModuleによるCSS適用の方法
- styled componentsの使い方
 - 外部のライブラリ（Emotion）の使い方
- Propsの扱い方
- State（useState）の扱い方

今回のアジェンダ

- useEffectの使い方
- 再レンダリングの起きる条件
- memoによるレンダリング最適化
- useCallbackによるレンダリング最適化
- 変数のmemo化
- グローバルなState管理の方法（Contextの利用）

State(useState)

- Reactの場合、画面に表示するデータ、可変の状態を全て Stateとして管理する
 - コンポーネントの状態を示す値
 - 状態を管理して、イベント実行時などに値を変更することでアプリケーションを実現する
- useState関数を利用して管理を行う
 - importする必要がある

```
import { useState } from "react";
```

useStateの利用

- useStateは配列の形で1つめにStateの変数、2つ目にその変数を更新するための関数を設定する

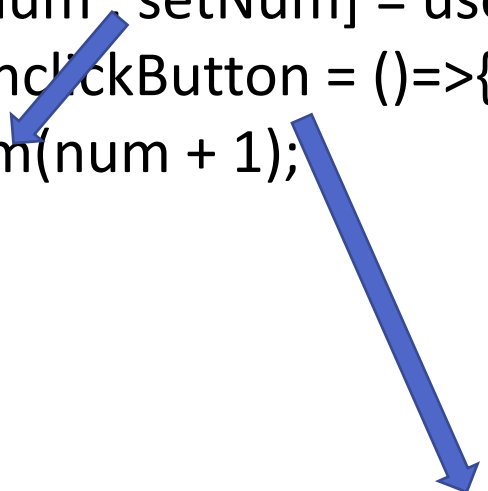
```
const [num , setNum] = useState();
```

- 変数名を付けて、暗黙的に更新側はset変数名とすることが多い
 - 変数の初期値はundefinedになるので、useState()の引数に値を指定すれば、初期化される

useStateの利用

```
import { useState } from "react";
export const App = ()=>{
  const [num, setNum] = useState(0);
  const onClickButton = ()=>{
    setNum(num + 1);
  }

  return (
    <>
      <button onClick={onClickButton}>クリック</button>
      <p>クリックした回数:{num}</p>
    </>
  );
}
```



- ボタンをクリックする度に
onClickButtonが動く
 - setNum() でnum値を増やしている
 - 動的に変更した値を表示

useState内の関数で更新

```
const onClickButton = ()=>{  
  setNum((prev) => prev + 1);  
}
```

- setNum内で関数を定義してしまう
 - 関数の引数に前のstate値が入ってくる
 - prevには実行される前の値が入ってくる
 - prevに1をプラスして戻す

値が変わったときに処理を実行

- Reactの機能であるuseEffectを利用することで、ある値が変わったときにだけ、ある処理を実行することができる
 - コンポーネントが初期化されたときにだけ実行する、ということも可能

numの値が変わったときにだけalert()を表示する

```
useEffect(() => {  
  alert();  
},[num]);
```

値を指定せずに、[]だけを記述すると、最初に1回(開発環境で副作用の検証のために2回)だけ実行される

配列で変更を検知する変数を指定する
→複数值を指定可能

再レンダリング

- 値が変わったときに画面の描画が更新される
 - Reactのstateを利用すれば、値の変更と更新が可能
 - 変更を検知して自動的にコンポーネントが再実行される
- 画面の描画＝レンダリング
 - これが再度行われるので、再レンダリング
 - CPUやGPU能力を利用してブラウザーが再レンダリングを行っている
- 多数の再レンダリングが生じると操作性に問題が生じる
 - 反応が遅くなる

再レンダリング

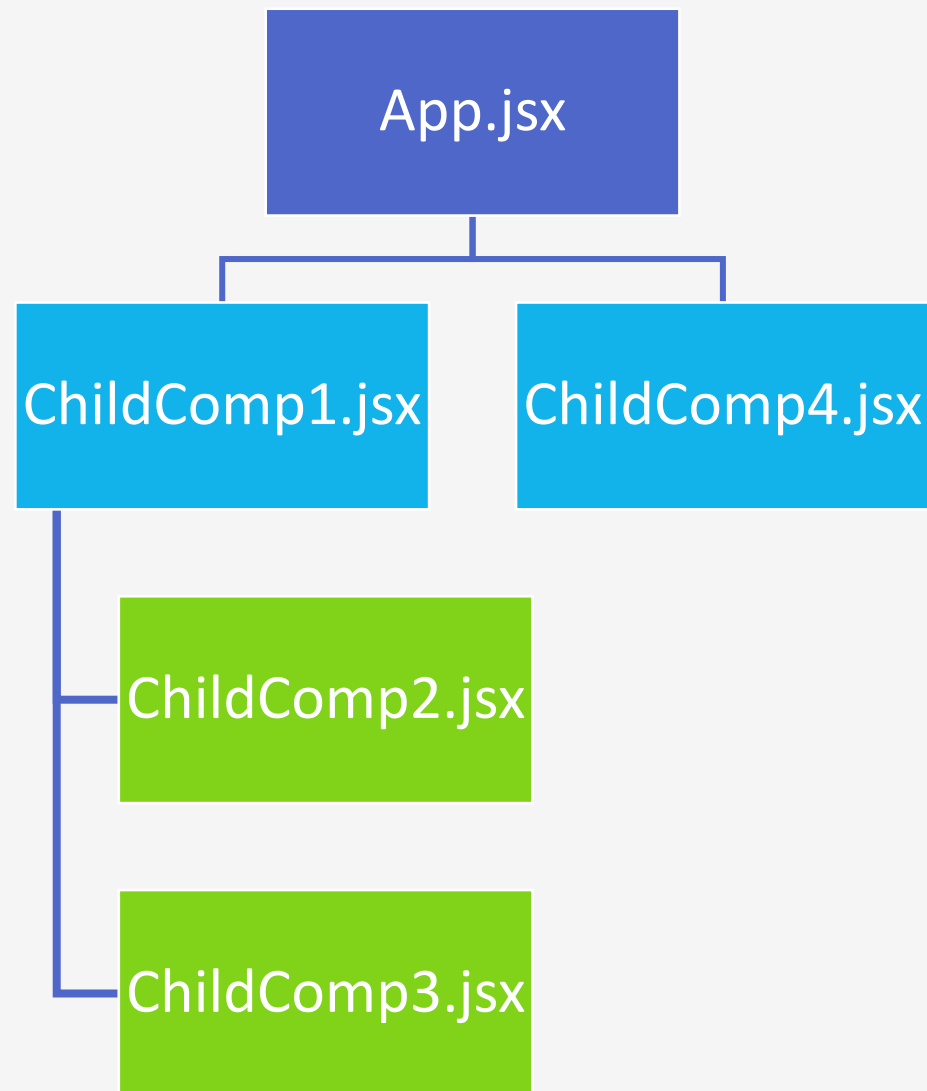
- プログラム自体が小さい場合は影響も少ない
 - そもそも影響を与える範囲が少ないため、再レンダリングのコストが低い
- プログラムが巨大化してくると影響が顕著に出てくる
 - 再レンダリングのコストが無視できないレベルに達してしまう
 - 必要に応じて再レンダリング制御をする必要がある
- 動的にUIを変更するReactでは再レンダリングされやすい
 - 値が変わるとき以外にも再レンダリングが生じる

再レンダリングの条件

- 実際に再レンダリングされるパターン
 1. Stateが更新されたコンポーネント
 2. Propsが更新されたコンポーネント
 3. 再レンダリングされたコンポーネントの配下のコンポーネントの全て
- StateとPropsが更新されたときに再描画されないと値が変更されたことが理解できない
 - ある程度これは必要な再レンダリング
- 問題は配下のコンポーネント全てが再レンダリングされること

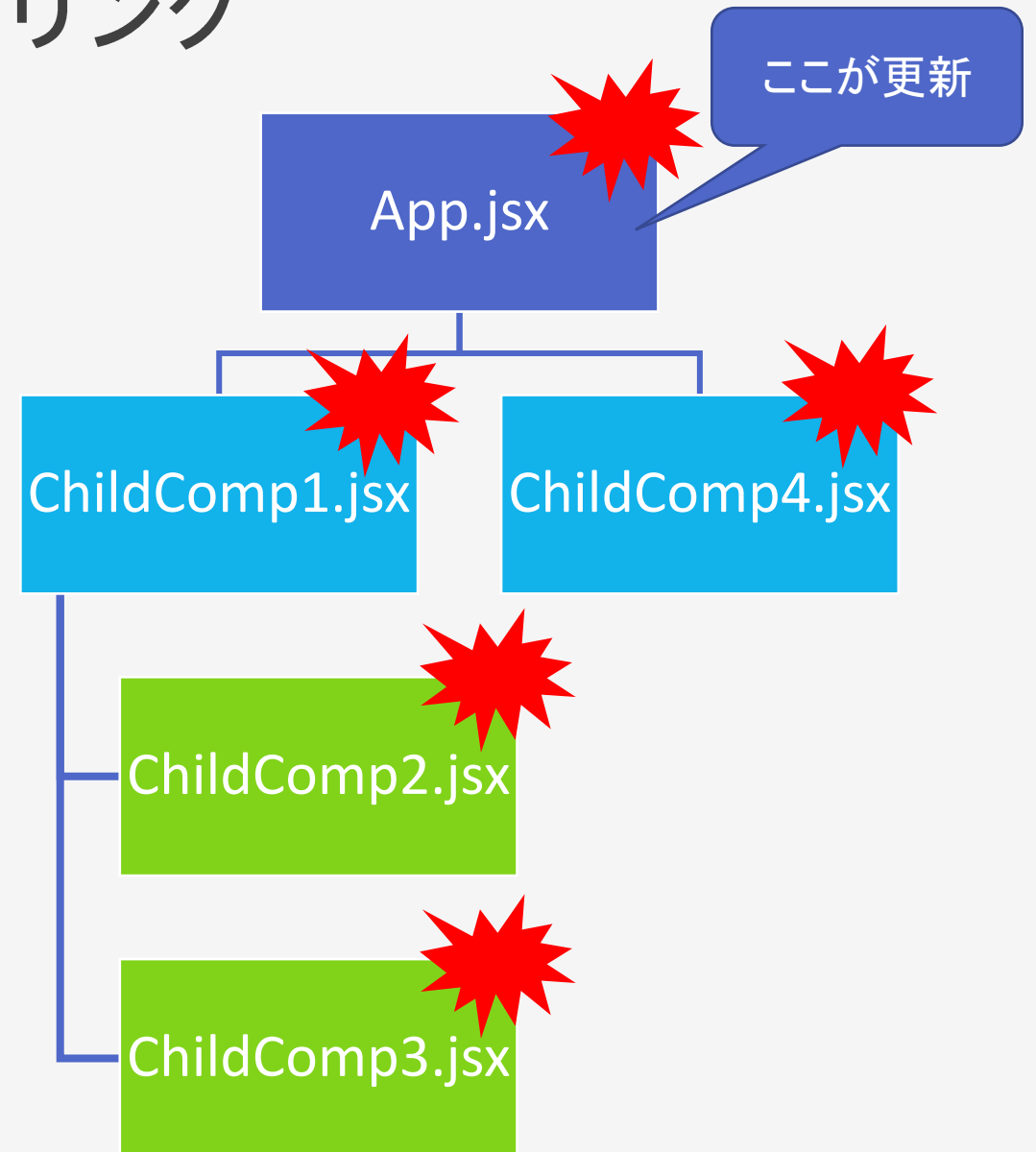
配下のコンポーネント

- コンポーネントは他のコンポーネントを読み込むことが可能
 - 親コンポーネントと子コンポーネントの関係
- App.jsxからみると、ChildComp1.jsx以外にChildComp2.jsxなども配下のコンポーネントになる



配下のコンポーネントの再レンダリング

- 配下のコンポーネントも再レンダリングされる
 - App.jsxのStateが更新されると配下全てのコンポーネントが再レンダリングされる
- コンポーネント数が増えれば増えるほど再レンダリングのコストが増大する



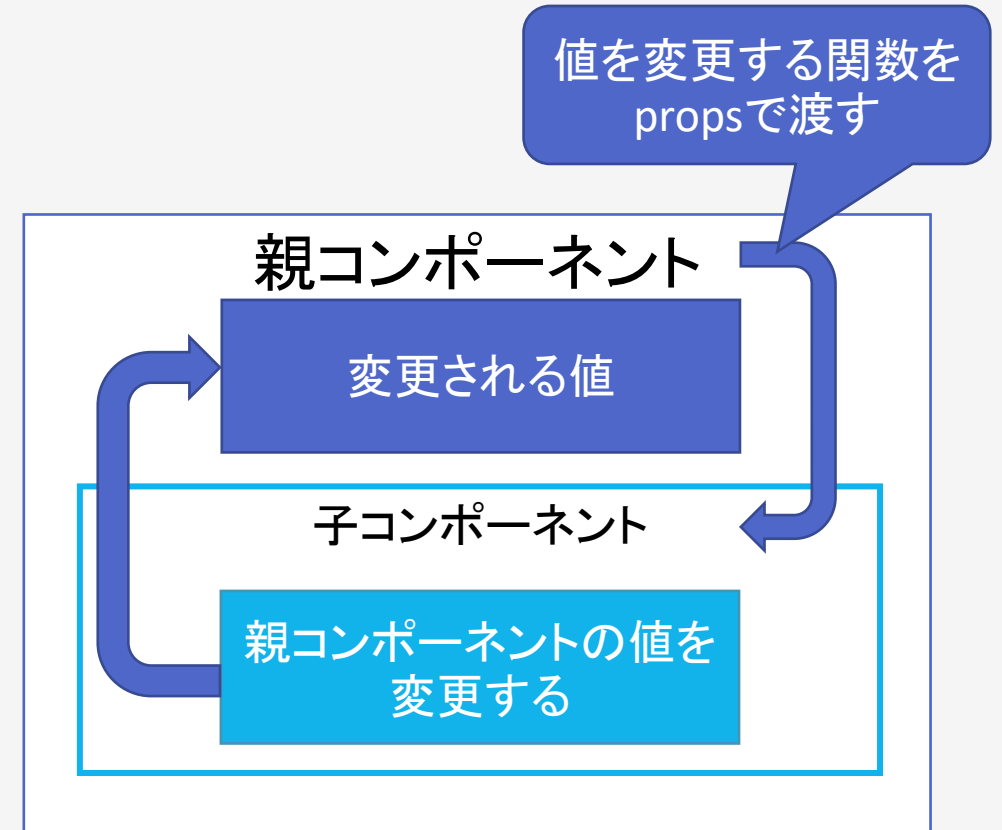
memoを利用して再レンダリング制御

- App.jsxのStateが変わったとしても配下のコンポーネントに影響を及ぼさない場合、配下の再レンダリングが不要
 - memoという機能を利用してメモ化し、再レンダリングを防ぐ
 - メモ化：前回の処理結果を保持しておくことで、処理を高速化する技術
- memo()で括ることで、propsに変更がないかぎり再レンダリングを防ぐことが可能

```
const Components = memo(() => {  
  //コンポーネント内部のプログラム  
});
```

useCallbackを利用して再レンダリング制御

- 関数の子コンポーネントに親コンポーネントからpropsで渡して配置し、機能を利用する場合
 - 子コンポーネントでは画面描画は変更が無く、親コンポーネントのみ描画内容が変更される
 - 子コンポーネントの機能を呼び出すと子コンポーネントも再レンダリングされる



useCallbackを利用して再レンダリング制御

- 関数をpropsで渡す場合、コンポーネントをメモ化していても再レンダリングされる
 - 親コンポーネントで再レンダリングが行われると、常に関数が再生成されるためpropsが変化すると判断される
 - propsが変化すると判断されると再レンダリングされる
- 関数をメモ化することで再レンダリングを防ぐことが可能
 - Reactに存在するuseCallbackを利用する

関数にuseCallbackを指定する

```
const onClickButton = useCallback(()=>{  
  alert('クリックされました');  
},[]);
```

関数が実行されたときにalert()を表示

値を設定すると、この値が変更されたときに関数を再生成するように指定可能。値を指定しなければ、最初に作成されたものが使い回される

変数のメモ化

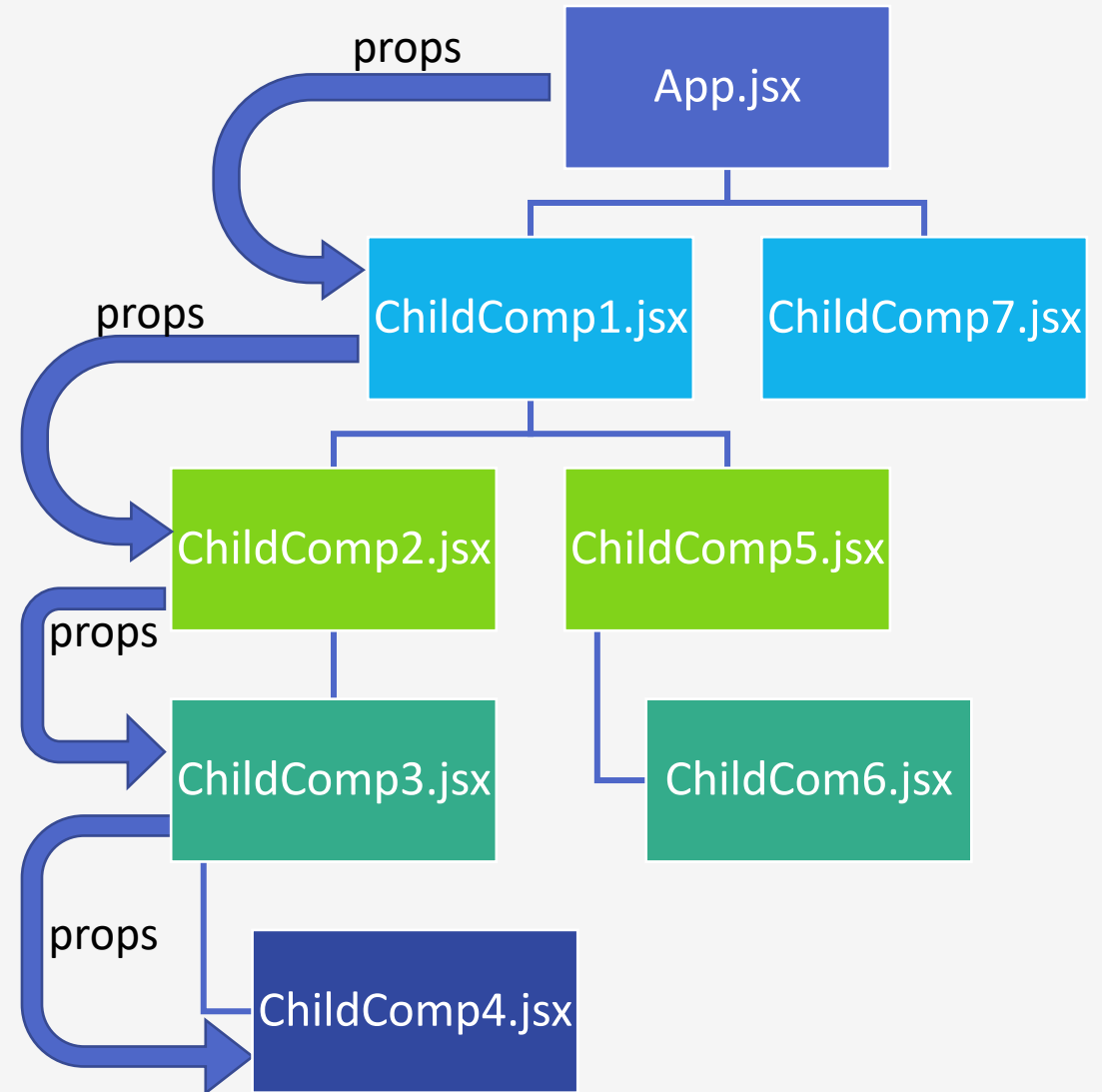
- それほど利用する機能でもない
 - Reactに存在するuseMemoを利用する
 - 関数的に記述し、returnで設定する変数の値を格納する
 - 配列に値を設定しておけば、その値が変わったときに再計算される
 - 何も記述していない場合は、最初だけ計算される

valuesの合計値が格納される

```
const sum = useMemo(() => {  
  return values.reduce((a, b) => a + b, 0);  
}, [values]);
```

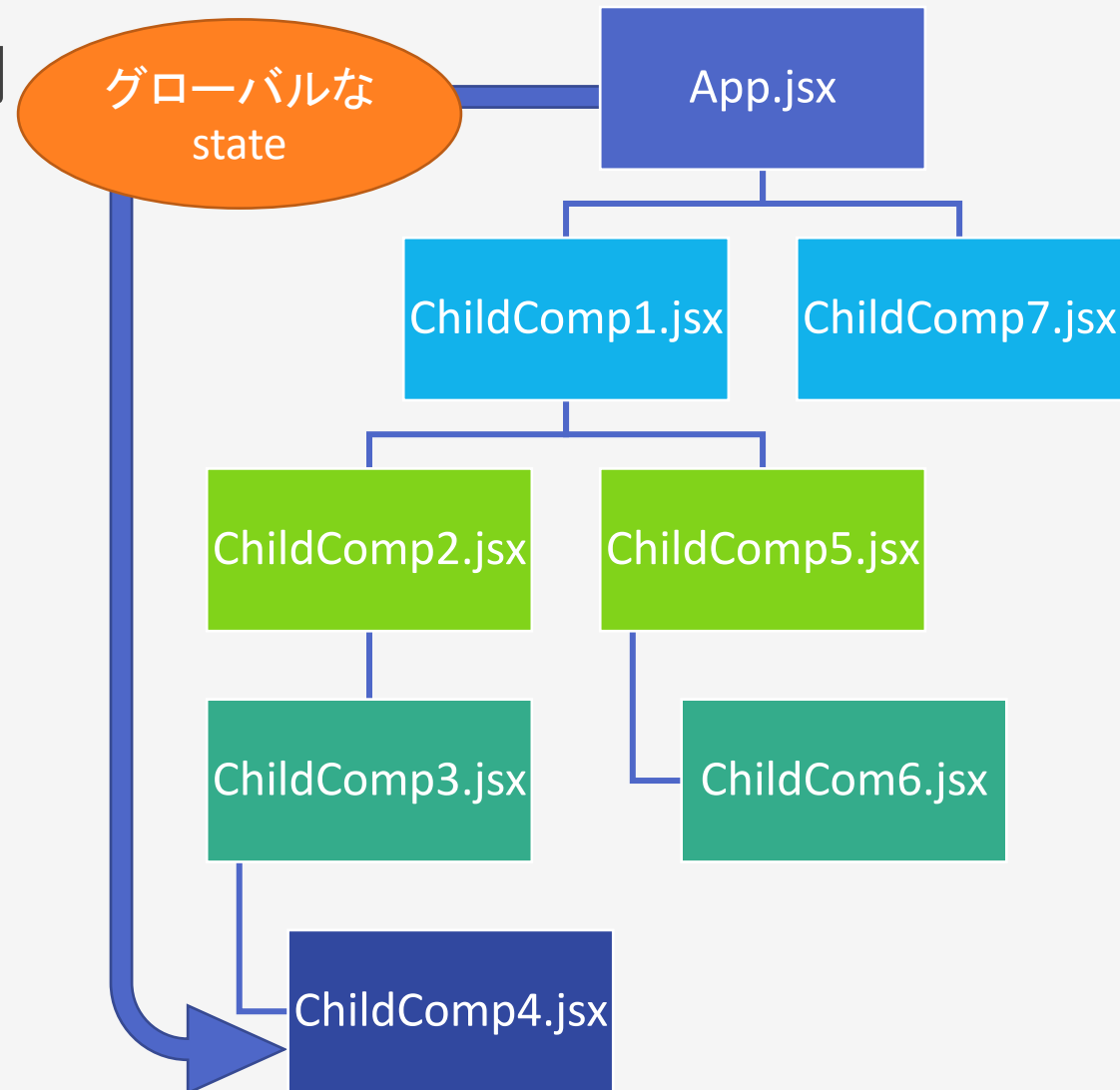
グローバルなstate管理

- 配下のコンポーネントが大量に存在する場合、stateの値をpropsで渡していく必要が生じる
 - バケツリレー
- propsが変更されると再レンダリングが生じるため、パフォーマンスも悪化する



グローバルなstate管理

- グローバルなstateに格納することで、propsによるバケツリレーを避けることができる
- 必要なコンポーネントだけで値を取得することでstateを扱うことができる



Contextを利用したグローバルなstate管理

- Reactが持っているContextによりグローバルなState管理が可能
 - 他にもRedux、Recoil、Jotaiなどの外部ライブラリが存在
- Contextの使い方
 - createContextでContextの枠を作成
 - Contextに存在するProviderを使い、グローバルStateを使いたいコンポーネントを囲む
 - Stateを参照したいコンポーネントでuseContextを利用する



Redux



Recoil

Contextを利用したグローバルなstate管理

- createContextを利用

contextの名前

```
export const LoginFlagContext = createContext(  
  {isLogin: false,  
    setIsLogin: () => {}  
});
```

初期値を指定することも可能
指定しないことも可能

- Contextの値を参照するためには、ProviderでContextの値を参照したいコンポーネントを囲む必要がある
 - Providerコンポーネントは何でも囲めるようにpropsとしてchildrenを受け取れるようにしておく

Contextを利用したグローバルなstate管理

- Providerを作成する

childrenを受け取る

```
export const LoginFlagProvider = (props) => {  
  const { children } = props;  
  const [isLogin, setIsLogin] = useState(false);  
  return(  
    <LoginFlagContext.Provider value={{isLogin, setIsLogin}}>  
      {children}  
    </LoginFlagContext.Provider>  
  );  
}
```

useStateを使いstateを作成して、false値を格納する
どのコンポーネントからも更新ができる

valueの中にグローバルで使う値を設定する

contextを作っているなので、providerが存在する。それで{children}を囲む

Contextを利用したグローバルなstate管理

- 作成したProviderを参照したい範囲のコンポーネントを囲む
 - 全体で利用したい場合は、main.jsx内でAppコンポーネントを囲む

```
const container = document.getElementById('root');  
const root = ReactDOMClient.createRoot(container);  
root.render(  
  <LoginFlagProvider>  
    <App />  
  </LoginFlagProvider>);
```

Appコンポーネントを作成した
Providerで囲む

Contextを利用したグローバルなstate管理

- stateを参照したいコンポーネントでuseContextを使って値を参照する

作成したProviderを読み込む

```
import { LoginFlagContext } from "../providers/LoginFlagProvider";
```

```
export const Editor = (props)=>{  
  const { isLogin } = useContext(LoginFlagContext);
```

useContextでContextで指定した値を取得

isLoginの値だけ取得

```
  return (  
    <textarea disabled={!isLogin}></textarea>  
  );  
}
```

isLoginの値を利用

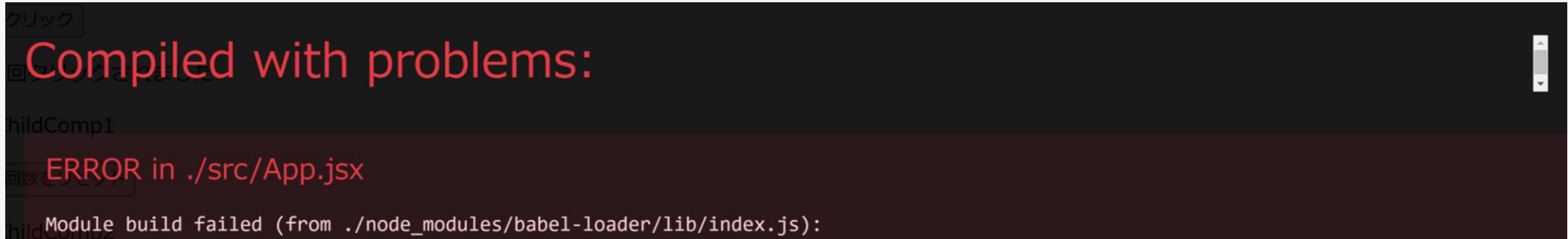
Contextを利用したグローバルなstate管理

- コンポーネントの階層が深くなってくる、深くなりそうなときにグローバルなState管理を利用する
- Contextを使うと、値が更新されたときにuseContextで参照しているコンポーネントは再レンダリングされる
 - 状況によっては、再レンダリングのコストが高くなるため、Context自体を分けるなどが必要

Reactの場合

- コンポーネントは入れ子にすることが多い
 - 親コンポーネントから子コンポーネントに対してデータを渡すケースが当然増える
 - データを渡すとそのままでは再レンダリングが生じる
 - パフォーマンスが悪化する
 - 再レンダリングを防ぐための仕組みがあるので、有効活用する
- コンポーネントが増えるとグローバルステートを使った方が楽
 - 外部ライブラリーを使うケースが多いがContextでもできる
 - Context自体を分けることを考慮しても良い

エラーの場合

A screenshot of a terminal window with a dark background. The text is displayed in a light color. The first line is "Compiled with problems:" in a larger font. The second line is "ERROR in ./src/App.jsx" in a slightly smaller font. The third line is "Module build failed (from ./node_modules/babel-loader/lib/index.js):" in a smaller font. There are some faint, partially visible text elements on the left side of the terminal, such as "クリック", "hildComp1", and "hild".

```
Compiled with problems:  
ERROR in ./src/App.jsx  
Module build failed (from ./node_modules/babel-loader/lib/index.js):
```

- コードのどこかに間違いが合った場合は、ブラウザー上でエラー表示になるので、エラー内容に従って修正をする
 - ほとんどの場合はコードミス
 - ある程度エラー内容は記述されているので、参考にして修正をする

今回の手順 (1)

1. フォルダーを作成 (英数字で作成のこと)
2. ターミナルでcdコマンドを使い、作成したフォルダーにアクセス
3. `npm create vite@latest childapp -- --template react`
を入力して実行
4. cdコマンドで作成されたフォルダーにアクセス
5. `npm install`
を入力してreactをインストール
6. `npm run dev`を入力してReactを起動
7. srcフォルダー内にApp.jsx,main.jsxを作成しつつ、コードを記述
8. components内にChildComp1.jsx, ChildComp2.jsx, ChildComp3.jsx, ChildComp4.jsxを作成しつつコードを記述
9. ブラウザーで随時動作を確認できるので、確認しつつ進めていく

今回の手順 (2)

1. フォルダーを作成（英数字で作成のこと）
2. ターミナルでcdコマンドを使い、作成したフォルダーにアクセス
3. `npm create vite@latest reactglobalapp -- --template react`
を入力して実行
4. cdコマンドで作成されたフォルダーにアクセス
5. `npm install`
を入力してreactをインストール
6. `npm run dev`
を入力してReactを起動
7. srcフォルダー内にApp.jsx,main.jsxを作成しつつ、コードを記述
8. components内（フォルダーを作成）にprovidersフォルダーを作成し、LoginFlagProvider.jsxを作成
9. components内にEditor.jsxとLogin.jsxを作成
10. ブラウザーで随時動作を確認できるので、確認しつつ進めていく

ありがとうございました。
また次回。