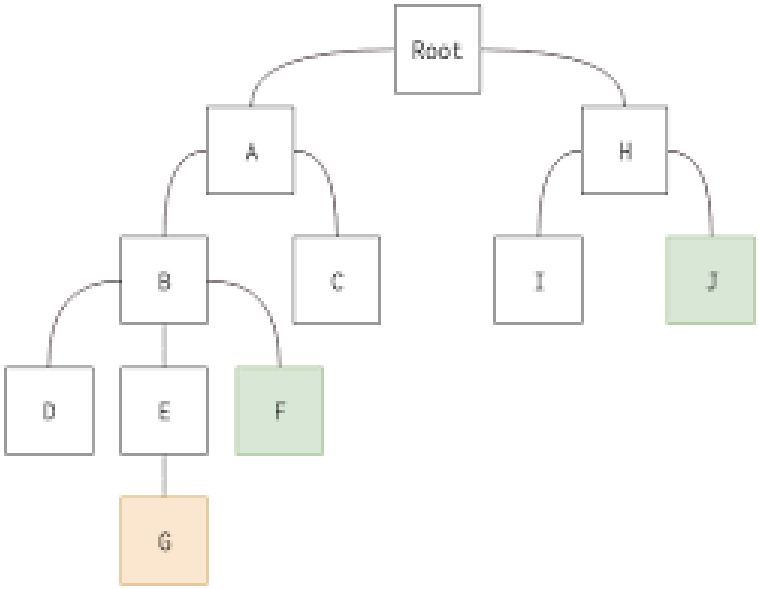
context API, Redux개념

Context API란?

리액트 프로젝트에서 전역적으로 사용할 데이터가 있을 때 유용한 기능이다.

리액트 어플리케이션은 컴포넌트 간에 데이터를 props로 전달하기 때문에 컴포넌트 여기저기서 필요할 경우 최상위 컴포넌트인 App의 state에 넣어서 관리한다.



리액트를 다루는 기술 참조(일반적인 전역 상태)

```
function App() {
  const [myName, setMyName] = useState("")

  const onChangeNameHandle = (e) => {
    setMyName(e.target.value)
  }

  return (
      <div>
        이름 : <input
          value={myName}
          onChange={onChangeNameHandle} />
      </div>
  );
}

export default App;
```

App이 가지고 있는 myName값을 f컴포넌트와 j컴포넌트에 전달하려면

F의 경우 : App → A → B → F

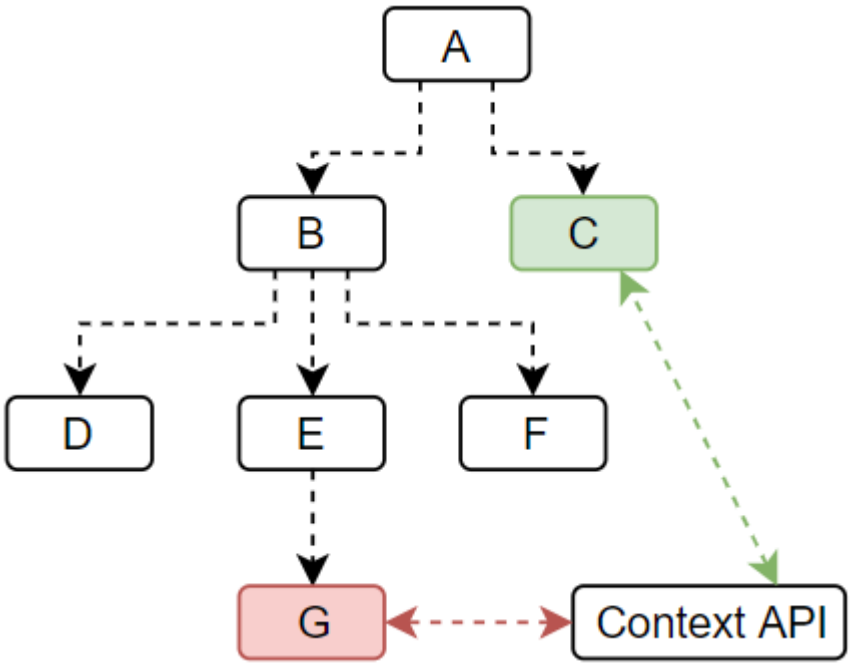
J의 경우 : App → H → J

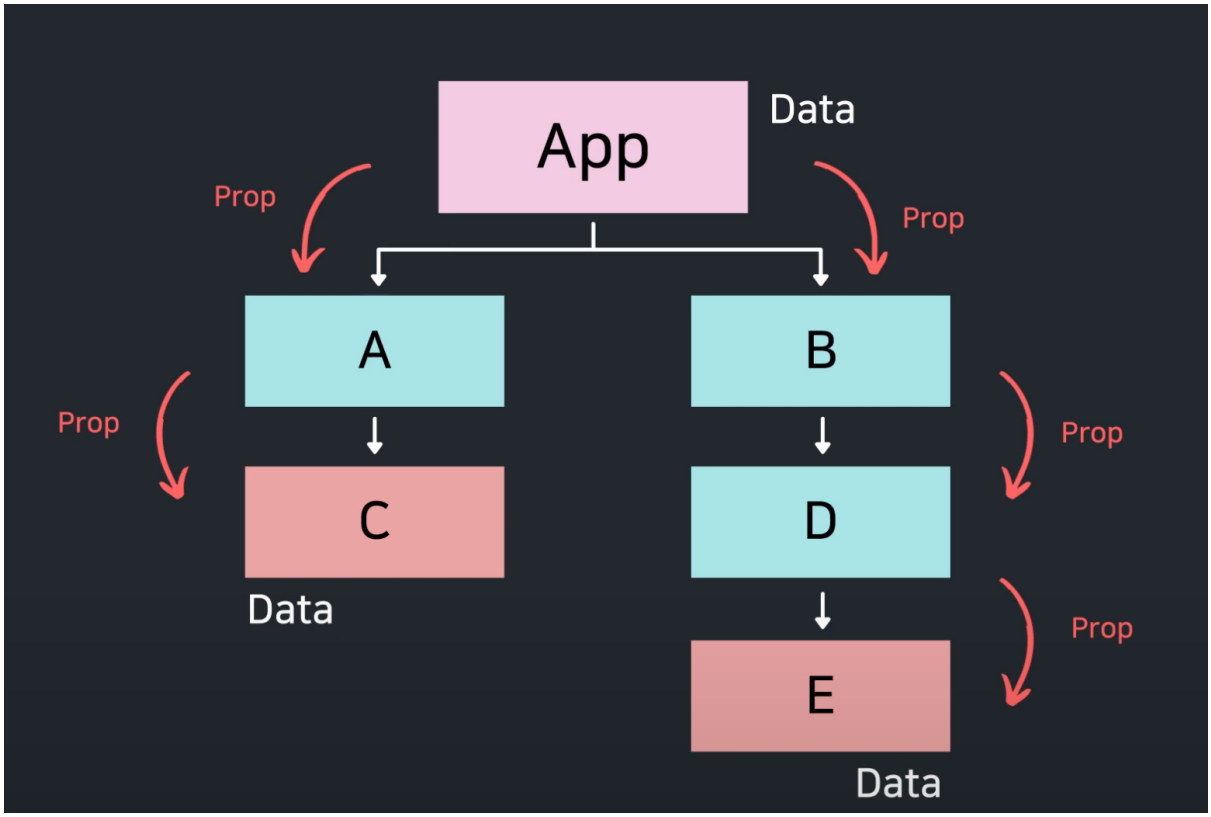
이렇게 복잡하게 여러 번 거쳐서 전달해야한다.

이렇게 리액트 코드를 짜다보면 더 많은 컴포넌트를 거치거나 다루어야 하는 데이터가훨씬 많아질 경우 유지 보수성이 낮아질 가능성이 있다.

그렇기 때문에 리덕스 같은 라이브러리들을 많이 이용하는데 리액트 v16.3 업데이트 이후에는 Context API가 많이 개선되어 별도의 라이브러리들을 사용하지 않아도 리액트 자체에서 제공하는 Context Hook을 사용하여 전역 상태를 손쉽게 관리할 수 있다.

기존에는 최상위 컴포넌트에서 여러 컴포넌트를 거쳐 props로 원하는 상태와 함수를 전달했지만 Context API를 사용하면 Context를 만들어 단 한 번에 원하는 값을 받아와서 사용할 수 있다.





▼ dark mode(props)

```
const App = () => {
  const [isDark, setIsDark] = useState(false)
  return (
    <>
      <Page isDark={isDark} setIsDark={setIsDark} />
    </>
  )
}
```

```
const Page ={isDark, setIsDark}) => {
  return (
    <div className='page'>
      <Header isDark={isDark}/>
      <Content isDark={isDark}/>
      <Footer isDark={isDark} setIsDark={setIsDark}/>
    </div>
  )
}
```

```
const Header = ({isDark}) => {
  return (
    <header
```

```
    className='header'
    style={{
      backgroundColor:isDark?'black':'lightgray',
      color:isDark?'white':'black'
    }}>header</header>
  )
}
```

```
const Content = ({isDark}) => {
  return (
    <div className='content'
    style={{
      backgroundColor:isDark?'black':'white',
      color:isDark?'white':'black'
    }}>유리님 이건 props입니다.</div>
  )
}
```

```
const Footer = ({isDark,setIsDark}) => {
  const toggleTheme = ()=>{
    setIsDark(!isDark)
  }

  return (
    <footer
    className='footer'
    style={{
      backgroundColor:isDark?'black':'lightgray',
    }}>
      <button className='button'
      onClick={toggleTheme}>Dark Mode</button></footer>
  )
}
```

props로 useState 상태값을 내려주고 그걸 이용해서 토글을 이용한다.

```
const [isDark, setIsDark] = useState(false)
```

이렇게 되면 page.jsx파일에서는 해당 내용을 받지 않아도 되는데 그 하위 컴포넌트 들에게 전달하기 위해 props를 받을 수 밖에 없다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/99a2d38e-2efc-4d8c-b35b-6d58638ca3f9/%E1%84%92%E1%85%AA%E1%84%86%E1%85%A7%E1%86%AB_%E1%84%80%E1%85%B5%E1%84%85%E1%85%A9%E1%86%A8_2023-04-25_%E1%84%8B%E1%85%A9%E1%84%92%E1%85%AE_4.08.18.mov

▼ dark mode(useContext)

1. Context를 관리할 파일을 만든다.

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

export const ThemeContext = createContext(null)
```

createContext를 import해준다.

```
createContext(null)
초기값에 null을 해준이유?
만약 App.jsx에서 provider로 감싸주지 않았을 경우 value가 없기 때문에
ThemeContext의 초기값을 지정을 해줘야한다.
지금 하려는 코드에서는 App.jsx에서 value에 초기값을 넣어줬기 때문에
ThemeContext에서는 초기값을 지정해주지 않아도 되서 null로 적어줘도 된다.
```

-

2. App.js에서 리턴문 내용을 <ThemeContext.provider>로 감싸준다.

```
const App = () => {
  const [isDark, setIsDark] = useState(false)
  return (
    <ThemeContext.Provider value={{isDark, setIsDark}}>
      <Page />
    </ThemeContext.Provider>
  )
}
```

provider에는 value라는 props을 받는데 하위컴포넌트들에게 전달할 오브젝트로 {{isDark, setIsDark}}를 넣어주면 하위컴포넌트들은 props로 전달해 주지 않아도 isDark와 setIsDark를 받을 수 있다.

3. page1.jsx에서 useContext를 통해 데이터를 받아올 수 있는데 콘솔에 해당 값을 찍어보면 isDark와 setIsDark가 포함되어 있는걸 볼 수 있다.
   하지만 우리는 page1.jsx에서는 해당 내용을 받을 필요가 없기 때문에 그 하위컴포넌트에서 useContext로 받아주면 된다.

```
const {isDark} = useContext(ThemeContext)
```

💡useContext와 ThemeContext는 꼭 import 해줘야 한다.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/22103ac9-dbad-48c1-b3ea-3d037dc39dbf/%E1%84%92%E1%85%AA%E1%84%86%E1%85%A7%E1%86%AB_%E1%84%80%E1%85%B5%E1%84%85%E1%85%A9%E1%86%A8_2023-04-25_%E1%84%8B%E1%85%A9%E1%84%92%E1%85%AE_4.08.37.mov

useContext() 장점

1. Prop drilling을 방지할 수 있습니다.
2. 코드의 가독성을 높일 수 있습니다.
3. 코드의 재사용성을 높일 수 있습니다.
4. 상태 관리를 효율적으로 할 수 있습니다.
5. 중복된 코드를 줄일 수 있습니다.

useContext() 단점

1. Context API의 복잡성
2. 성능 문제

3. 디버깅의 어려움
4. 다른 라이브러리와의 호환성
5. 과도한 사용

useContext를 사용하면 상위 컴포넌트의 변경이 하위 컴포넌트까지 전파될 수 있으므로, 어떤 상태의 변화에 의해 불필요한 렌더링이 발생할 수 있습니다. 또한, useContext를 남용하면 코드의 추적이 어려워질 수 있으며, 불필요한 성능 저하를 초래할 수도 있다.

이런 단순히 전역 상태 관리만 한다면 Context API를 사용하는 것만으로 충분하다.

다만 리덕스를 사용하면 상태를 더욱 체계적으로 관리할 수 있기 때문에 프로젝트의 규모가 클 경우에는 리덕스를 사용하는 편이 좋다.

Redux

Redux는 JavaScript 애플리케이션을 위한 상태 관리 라이브러리이다. Redux는 애플리케이션의 모든 상태를 하나의 객체로 관리하고, 이를 예측 가능한 방식으로 업데이트하는 데 사용된다. Redux는 React와 함께 사용되지만, 다른 라이브러리나 프레임워크와도 함께 사용이 가능하다. Redux는 Flux 아키텍처의 영향을 받았으며, 단방향 데이터 흐름을 따르는 패턴을 사용하다. Redux는 애플리케이션의 복잡한 상태 관리를 용이하게 하며, 개발자들이 상태를 예측 가능하고 디버그 가능한 방식으로 조작할 수 있도록 도와준다.

리덕스 설치 방법

```
yarn add redux react-redux
```

리덕스의 개념 정리

1. 액션
   상태에 어떠한 변화가 필요하면 액션(action)이란 것이 발생하고, 이는 하나의 객체로 표현된다.

```
{
  type : 'TOGGLE_VALUE'
}
```

2. 액션 생성 함수
   액션 객체를 만들어주는 함수

```
const addTodo = (data) =>{
  return {
    type: 'ADD_TODO'
    data
  }
}
```

액션 생성 함수를 만드는 이유?

매번 액션 객체를 직접 작성하기 번거러울 수 있고, 만드는 과정에서 실수로 정보를 놓칠 수도 있어서 이러한 일을 방지하기 위해 이를 함수로 만들어서 관리한다.

3. 리듀서

   변화를 일으키는 함수
   액션을 만들어서 발생시키면 리듀서가 현재 상태와 전달받은 액션 객체를 파라미터로 받아온다. 그리고 두 값을 참고하여 새로운 상태를 만들어서 반환해준다.

```
const initialState = {
  conter:1
};
const reducer(state=initialState, action){
  switch(action.type){
    case INCREMENT:
      return{
        counter: state.counter + 1
      }
    defaule:
      return state;
  }
}
```

4. 스토어
   프로젝트에 리덕스를 적용하기 위해 스토어(store)를 만든다. 한 개의 프로젝트는 단 하나의 스토어만 가질 수 있다. 스토어 안에는 현재 어플리케이션 상태와 리듀서가 들어가 있으며, 그외에도 몇 가지 중요한 내장 함수를 지닌다.

5. 디스패치
   스토어의 내장 함수 중 하나이며 '액션을 발생시키는 것'이라고 이해하면 된다.
   dispatch(action)와 같은 형태로 액션 객체를 파라미터로 넣어서 호출한다.

```
리덕스 상태는 읽기 전용이다. 상태를 업데이트할 때 기존의 객체는 건드리지 않고
새로운 객체를 생성해 주어야 한다.
객체의 변화를 감지할 때 객체의 깊숙한 안쪽까지 비교하는 것이 아니라
겉핥기 식으로 비교하여 좋은 성능을 유지할 수 있다
```

Redux 장점

1. 상태 관리가 용이합니다: Redux는 중앙 상태 저장소를 사용하여 애플리케이션의 상태를 관리합니다. 이렇게 함으로써 애플리케이션 전체에서 일관된 방식으로 상태를 관리할 수 있으며, 개발자들은 상태 변경을 추적하고 디버깅하는 것이 쉬워집니다.
2. 예측 가능한 상태 변경: Redux는 "단방향 데이터 흐름" 아키텍처를 따르므로 상태 변경이 예측 가능합니다. 이것은 디버깅이 더 쉬우며, 예측할 수 없는 버그를 줄일 수 있습니다.
3. 테스트 용이성: Redux는 테스트가 용이한 코드를 작성하는 것을 촉진합니다. 애플리케이션의 상태를 변경하는 모든 액션은 순수 함수로 구현되어 있기 때문에 테스트 케이스를 작성하기 쉽습니다.
4. 확장성: Redux는 높은 수준의 확장성을 제공합니다. 애플리케이션이 커지더라도 상태 관리를 위한 코드의 양이 늘어나는 것을 방지할 수 있습니다.

Redux 단점

1. 러닝 커브: Redux를 이해하는 것은 초기에는 어려울 수 있습니다. 상태 관리 라이브러리로서 복잡성이 증가할 수 있기 때문입니다.
2. 보일러플레이트 코드: Redux를 사용하면 일부 보일러플레이트 코드가 필요합니다. 애플리케이션의 규모가 작을 경우에는 이것이 불필요한 작업으로 느껴질 수 있습니다.

3. 파일 수: Redux를 사용하면 상태를 관리하는 데 사용되는 파일의 수가 늘어납니다. 이는 애플리케이션의 규모가 작을 경우 불필요한 파일이 추가되는 것으로 느껴질 수 있습니다.

useContext와 redux의 가장 큰 차이점

- useContext는 React의 Context API를 사용하여 컴포넌트 간에 상태를 공유하고 전달할 수 있습니다. 이는 보통 중첩된 컴포넌트 간에 상태를 전달하거나, 여러 컴포넌트에서 공통으로 사용하는 데이터를 관리할 때 유용합니다. useContext는 주로 상태 관리가 필요한 부분을 선택적으로 적용할 수 있기 때문에, 애플리케이션의 규모가 작거나 단순한 경우에 적합합니다.
- 리덕스는 중앙 집중화된 상태 관리를 제공하는 라이브러리로, 애플리케이션 전체에서 상태를 관리합니다. 이는 여러 컴포넌트 간에 데이터를 공유하고, 상태 변경 시에 액션과 리듀서를 통해 상태를 업데이트하는 방식입니다. 리덕스는 상태를 변화시키는 모든 액션을 로깅하고, 성능 최적화를 위해 상태를 비교하여 재렌더링을 방지하는 등 다양한 기능을 제공합니다. 따라서, 애플리케이션 규모가 크고 복잡한 경우에는 리덕스를 사용하는 것이 적합합니다.