Context API 사용하기

용도

- Context API는 **전역 상태(변수)를 제공**하는 용도로 사용됨
  - ex: 로그인한 유저 정보, 설정(언어, 테마(다크모드) 등등) 정보
- Context(맥락)이라는 개념을 제공하고 **Context로부터 필요한 전역 데이터를 저장하고, 읽어온다는 개념 부여**
- 보통 전역적인 정보는 모든 컴포넌트에 필요한데, 이론적으로 가장 최상위 컴포넌트에서 최하위 컴포넌트(leaf 컴포넌트)로 계속해서 **prop 값을 전달하는 prop drilling 현상** 없이도 (Context는 전역적으로 존재하므로) 값을 바로 컴포넌트에 전달 가능
- 서드파티 라이브러리인 **redux**를 이용해서도 전역 상태 저장 가능 (redux를 많이 사용하는 추세)
- react-router, redux 라이브러리에서 내부적으로 Context API를 사용

prop drilling 현상

```
class App extends React.Component {
  render() {
    // theme 값 전달
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // theme 값 전달
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

class ThemedButton extends React.Component {
  // theme 값 사용 (여기서는 2단계를 통해 값을 전달하지만, 단계가 늘어난다면??)
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

Context 생성 및 적용

- **React.createContext 함수**를 이용하여 컨텍스트 생성
  - 함수를 호출하며 **인자로 기본값을 전달** 가능
    - 해당 기본값은 **Provider 컴포넌트로 감싸지지 않은 컴포넌트 내부에서 Consumer를 통해 값을 참조하려 할 때 제공**됨

- 저장할 수 있는 값의 **타입에는 제한이 없음** (숫자, 문자열, 객체 등 모두 전달 가능)

```
// 문자열을 저장하는 Context 생성 (기본값은 'global')
const MyContext = createContext('default');
```

- 값을 전달받고자 하는 컴포넌트를 Context의 **Consumer 컴포넌트를 이용하여 감싸면** 내부에서 함수를 통해 전달된 Context 값 참조 가능

```
class ChildComponent extends Component {
    render() {
        return (
            {/* Consumer 컴포넌트로 전체 컴포넌트 내용을 감싸기 */}
            <MyContext.Consumer>
                {/* 내부에서 콜백 함수를 정의하여 값(valueFromContext)을 전달받음 */}
                {valueFromContext => {
                    return (
                        <div>
                            <p>id : {this.props.id}</p>
                            <p>value : {valueFromContext}</p>
                        </div>
                    )
                }}
            </MyContext.Consumer>
        );
    }
}

const Nested = (props) => <>{props.children}</>

class App extends Component {
    render() {
        let globalValue = 'global'

        return (
            <div>
                <MyContext.Provider value={globalValue}>
                    {/* props 값을 전달하지 않아도 내부에서 Provider를 통해 전달한 값
에 접근 가능 */}
                    <ChildComponent id='child 1' />
                    <Nested>
                        <Nested>
                            <Nested>
                                {/* 비록 많은 중첩 컴포넌트의 내부에 포함되어 있다고
하더라도, (props를 통해 값을 전달받지 않아도) Context 값에 접근 가능 */}
                                <ChildComponent id='child 2' />
                            </Nested>
                        </Nested>
                    </Nested>
                </MyContext.Provider>
                {/*
                    Provider 컴포넌트로 감싸지 않은 ChildComponent는 디폴트값
('default')을 전달받게 됨
                */}
                <ChildComponent id='child 3' />
            </div>
        );
```

```
    }
}

ReactDOM.render(<App />, document.getElementById("root"))
```

- 함수도 전달할 수 있으므로 **Context 값을 변경할 수 있는 함수**도 전달 가능

context-api-theme-context.js

```
import React, { Component, createContext } from 'react'
import ReactDOM from 'react-dom'

const ThemeContext = createContext({
    theme: 'light',
    // 저장할 수 있는 값에 제한이 없으므로, Context에 함수도 저장 가능
    // 여기서는 아무 작업도 하지 않는 함수를 디폴트값으로 전달
    toggleTheme: () => {}
});

// 이 컴포넌트는 theme값과 상태를 바꾸는 함수(toggleTheme)도 필요로 함
class ThemeUsingButton extends Component {
    render() {
        return (
            <ThemeContext.Consumer>
                {/* theme 값, toggleTheme 함수 모두 전달 받기 */}
                {({ theme, toggleTheme }) => {
                    const buttonStyle = { width: '100px', height: '100px',
fontSize: '50px' }
                    if(theme === 'light') buttonStyle['background'] = '#000'
                    if(theme === 'dark') buttonStyle['background'] = '#fff'

                    return (
                        <button onClick={toggleTheme} style={buttonStyle}>
                            {theme === 'light' ? '🌙' : '🌞' }
                        </button>
                    )
                }}
            </ThemeContext.Consumer>
        )
    }
}

// 이 컴포넌트는 오직 theme 값만 필요로 함
class ThemeUsingContainer extends Component {
    render() {
        const lightTheme = { background: '#fff' }
        const darkTheme = { background: '#000' }

        return (
            {/* theme 값만 전달 받기 (자바스크립트에서는 함수에서 일부 파라미터값만 전달
받아도 무방) */}
            <ThemeContext.Consumer>
                {({ theme }) => {
                    return (
                        <div style={theme === 'light' ? lightTheme : darkTheme}>
                            <ThemeUsingButton>Toggle theme</ThemeUsingButton>
```

```
                        </div>
                    )
                }}
            </ThemeContext.Consumer>
        );
    }
}

class ThemeApp extends Component {
    constructor(props) {
        super(props)

        this.state = {
            theme: 'light'
        }
    }

    toggleTheme = () => {
        this.setState(state => ({
            theme : state.theme === 'light' ? 'dark' : 'light'
        }));
    };

    render() {
        return (
            <div>
                {/* 컴포넌트 내부의 theme 상태값과 토글 메소드를 Consumer로부터 전달받
을 수 있도록 Provider 컴포넌트에 value 값을 설정 */}
                <ThemeContext.Provider value={{
                    theme: this.state.theme,
                    toggleTheme: this.toggleTheme
                }}>
                    <ThemeUsingContainer />
                </ThemeContext.Provider>
            </div>
        );
    }
}

ReactDOM.render(<ThemeApp />, document.getElementById("root"))
```

static contextType

- 클래스 컴포넌트 내부에 사용 가능
  - 간편하게 context에 저장된 값에 접근할 수 있다는 장점
  - 한 개의 Context 만 static하게 접근할 수 있다는 단점
- https://reactjs.org/docs/context.html#classcontexttype

여러개의 Context Consumer 사용하기

- **중첩된 Provider 컴포넌트를 사용**하여 여러개의 Context 값을 전달받아야 하는 컴포넌트들에 값 전달 가능

```
{/* 중첩된 Provider 사용 */}
<ThemeContext.Provider value={theme}>
    <UserContext.Provider value={signedInUser}>
        <Layout />
    </UserContext.Provider>
</ThemeContext.Provider>
```

- **복수개의 Consumer 컴포넌트와 값을 전달받는 함수를 추가**하여 값을 전달받을 수 있음

```
{/* 여러 Context 값 전달 받기 (복수개의 Consumer 컴포넌트 사용) */}
<ThemeContext.Consumer>
    {theme => (
    <UserContext.Consumer>
        {user => (
            <ProfilePage user={user} theme={theme} />
        )}
    </UserContext.Consumer>
    )}
</ThemeContext.Consumer>
```

주의사항

- Provider를 통해 제공할 value 값이 바뀌었는지 여부를 **참조가 달라졌는지 여부로 판단**하므로 value 값을 전달할 때 **의도치 않게 새 객체 생성이 이루어지지 않도록 주의**해야 함

아래 예제의 경우 value 값을 확인할 때마다 새 객체 생성 코드가 실행되므로 매번 참조가 바뀌고 해당 Provider를 통해 값을 전달받는 모든 Consumer 컴포넌트가 반응하게 되어 비효율적인 상황 발생 가능

```
class App extends React.Component {
    render() {
        return (
            {/* render 함수 호출시마다 매번 새로운 객체가 생성됨 */}
            <MyContext.Provider value={{something: 'something'}}>
                <Toolbar />
            </MyContext.Provider>
        );
    }
}
```

- 이를 해결하기 위해서 컴포넌트의 state에 값을 저장

```
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            value: {something: 'something'},
        };
    }

    render() {
        return (
            {/* 상태가 바뀌지 않는 이상 value 값 변경이 이루어지지 않음 */}
            <Provider value={this.state.value}>
```

```
                <Toolbar />
            </Provider>
        );
    }
}
```

리퍼런스

- https://velopert.com/3606
- https://medium.com/javascript-in-plain-english/how-to-avoid-prop-drilling-in-react-using-component-composition-c42adfcdde1b
- https://reactjs.org/docs/context.html#consuming-multiple-contexts