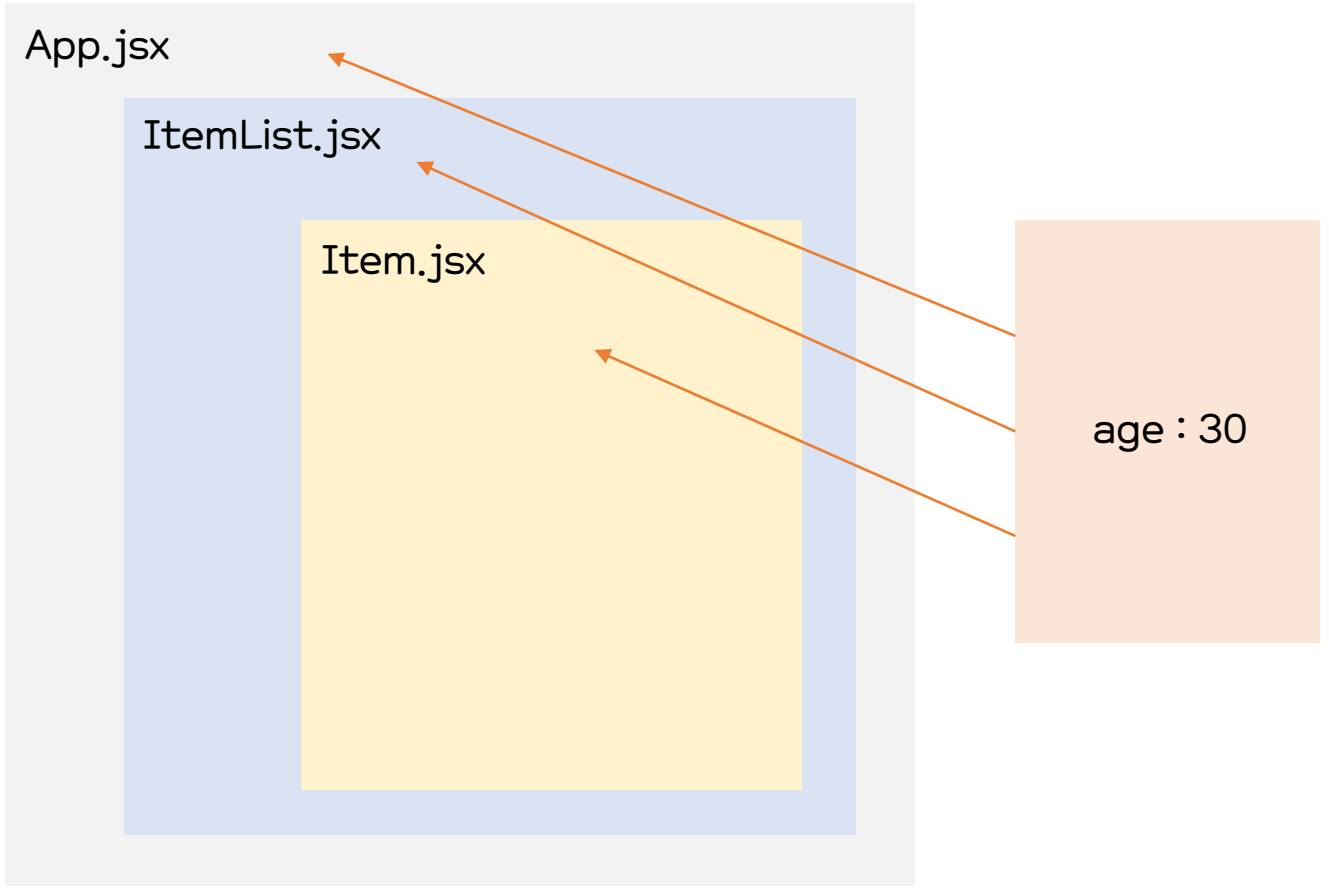
React redux

Redux(리덕스)

Redux는 상태(state)를 전역으로 관리할게 있게 도와주는 라이브러리다. 간단하게 말하면 Redux를 사용하면 부모컴포넌트에서 자식컴포넌트로 props를 사용해 전달하던 데이터를 props를 사용하지 않아도 모든 컴포넌트가 함께 사용할 수 있다는 말이다.

부모컴포넌트의 데이터를 props로 자식컴포넌트로 전달



redux는 공용 데이터 저장소에 state 변수값을 저장하고 모든 컴포넌트가 공용으로 사용

Redux vs Redux Toolkit

우리는 Redux toolkit을 사용할 것이다. Redux와의 차이점은 Redux에서 복잡하고 많은 량의 코드는 좀 더 간결하게 사용할 수 있도록 나온 라이브러리가 Redux toolkit이다. Redux toolkit은 Redux의 기본 개념을 그대로 보존하기 때문에 많이 사용되고 있다.

Redux의 기본 용어

- Store : 공유해서 사용할 state변수들의 저장소
- Slice : store에서 관리할 데이터의 단위. 데이터 및 그 데이터의 상태 변경 로직을 포함한다.
- Action : 상태의 변화를 일으키기 위한 정보를 갖고 있는 객체(데이터)
- Reducer : 상태를 업데이트하는 함수
- Dispatch : 상태를 업데이트하는 함수를 호출하는 행위

Redux와 Redux toolkit을 사용하기 위해 먼저 아래의 명령어를 입력하여 라이브러리를 설치한다.

npm install @reduxjs/toolkit react-redux

본격적으로 Redux를 사용해보겠다. 먼저 공용으로 사용할 state를 slice로 만들어보겠다. slice를 만든다는 것은 상태 관리할 데이터를 만든다는 것이다.

가장 간단하게 정수 하나를 상태 관리할 slice를 만들어보자. counterSlice.js 파일을 만들고 다음과 같이 작성한다.

```
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: 'counter',
  initialState : 5
});

export default counterSlice;
```

- createSlice() 함수는 slice를 생성하는 명령어이다. slice는 데이터와 데이터의 상태를 변경하는 함수로 이루어져 있다.
- createSlice() 함수의 인자는 객체타입으로 전달되며, name key는 변수의 이름, initialState는 해당 변수의 초기값이다.
- 결국, 위 슬라이스를 해석하면 counter라는 state변수를 생성하고 초기값으로 5를 주겠다는 의미이다.
- 마지막으로 외부 파일에서 counterSlice를 사용하기 위해 export 하였다.
- 슬라이스에는 데이터 및 그 데이터의 상태를 변경하는 함수가 포함되는데, 현재 코드는 데이터만 존재하고, 상태변경 함수는 작성하지 않은 상태다.
- 필요하다면 이러한 슬라이스 파일은 필요한 만큼 생성할 수 있다.

slice로 상태를 관리할 데이터를 만들었다면, 이러한 slice 데이터를 관리할 저장소(store)를 만들어야 한다. store에 저장한 state들은 props로 전달하지 않아도 필요한 컴포넌트에서 손 쉽게 사용할 수 있다. store.js 파일을 생성하고 다음과 같이 코드를 작성한다.

```
import { configureStore } from "@reduxjs/toolkit";
import counterSlice from "./countSlice";

export const store = configureStore({
  reducer:{
    counter : counterSlice.reducer,
  }
});
```

- configureStore() 함수는 중앙 저장소를 설정하는 함수이다. 이 함수를 사용하여 state들을 저장할 저장소를 만든다.
- reducer : {} 에는 중앙 저장소에서 사용할 state 값들을 작성한다. 이때 주의 사항은 import한 슬라이스 뒤에 반드시 .reducer를 붙여야 한다.

중앙 저장소인 store까지 만들었다면 이 store를 모든 컴포넌트에 사용하게 위해 main.jsx를 다음과 같이 작성한다.

```
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
import { BrowserRouter } from 'react-router'
import { Provider } from 'react-redux'
import { store } from './redux/store.js'

createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
)
```

- <Provider> 컴포넌트의 store에 우리가 만든 store.js를 연결해주면 Provider 컴포넌트 모든 자식 컴포넌트는 store.js에서 등록한 모든 state데이터에 접근 할 수 있다.
- 우리가 만드는 모든 컴포넌트는 App 컴포넌트의 자식 컴포넌트로 이루어지기 때문에, App 컴포넌트만 Provider로 감싸면 모든 컴포넌트에서 store.js에 등록한 모든 state들을 사용할 수 있다.

이제 컴포넌트에서 store에 등록한 state를 사용해보자. 예제는 App 컴포넌트에 작성하지만 모든 컴포넌트에서 동일하게 사용하면 된다.

작성한 개발자모드의 콘솔을 확인해보자.

```
function App() {
  const allData = useSelector((state) => {return state});
  console.log(allData);

  const counter1 = useSelector((state) => {return state.counter})
  console.log(counter1);

  const counter2 = useSelector(state => state.counter);
  console.log(counter2);

  return (
    <div>
      {allData.counter} / {counter1} / {counter2}
    </div>
  )
}
```

- useSelector()는 store에 저장된 state값을 불러온다.
- allData는 store에 저장한 모든 state를 불러오는 코드이다. 물론 지금은 store에 counter 데이터만 존재한다.
- counter1은 sotre에 저장된 state 중 counter 데이터만 받아오는 코드이다.
- counter2도 counter데이터만 받아오는 코드이며, 좀 더 축약해서 작성한 코드이다(추천)

슬라이스를 하나 더 만들고 store에 등록해보자. nameSlice.js 파일을 만들고 다음과 같이 작성한다.

```
import { createSlice } from "@reduxjs/toolkit";

const nameSlice = createSlice({
  name : 'name',
  initialState : 'hong'
});

export default nameSlice;
```

해당 slice는 'hong'이라는 초기값을 갖는 name state 변수를 생성한 코드이다.

그리고 방금 만든 nameSlice도 store에 등록한다.

```
export const store = configureStore({
  reducer:{
    counter : counterSlice.reducer,
    name : nameSlice.reducer
  }
});
```

위와 같이 state 변수당 하나씩 slice 파일을 만들어주며, 모든 slice를 store.js 파일에 등록하면 된다.

다시 말하지만 store.js 에 slice를 등록할 때는 반드시 slice 이름 뒤에 .reducer를 붙이도록 하자.

App.jsx 에서 새롭게 등록한 name state 값도 사용해보자.

```
function App() {
  const allData = useSelector((state) => {return state});
  console.log(allData);

  const counter1 = useSelector((state) => {return state.counter})
  console.log(counter1);

  const counter2 = useSelector(state => state.counter);
  console.log(counter2);

  const name = useSelector(state => state.name);
  console.log(name);

  return (
    <div>
      {allData.counter} / {counter1} / {counter2} / {name}
    </div>
  )
}
```

- allData는 store에 등록된 모든 데이터(counter, name)을 받아옴을 확인할 수 있을 것이다.
- name은 store 저장된 nameSlice의 데이터를 받아온다.

이번에는 state 값을 변경하는 reducer를 만들어보겠다. state 값 변경 reducer는 slice에 작성한다.

counterSlice에 counter 값을 변경하는 reducer를 등록하면 아래와 같다.

```
const counterSlice = createSlice({
  name: 'counter',
  initialState : 5,
  reducers : {
    increase : (state) => {
      return state + 1
    },
    handleCounter : (state, action) => {
      return state + action.payload
    }
  }
});

export const {increase, handleCounter} = counterSlice.actions;
export default counterSlice;
```

- counter 슬라이스에 reducer : {} 부분이 추가되었다. 이 부분이 state 값을 변경하는 함수를 정의하는 영역이다.
- 현재 counterSlice에는 increase, handleCounter라는 두 함수를 정의한 것이다.
- 함수에서 retur되는 데이터가 새로운 state 값으로 지정된다.
- 각 함수의 매개변수인 state는 현재 state 변수의 값을 의미한다.
- 매개변수 action은 state값을 변경할 데이터다. 변경할 값은 action.payload로 접근한다.
- 마지막으로 우리가 작성한 함수도 expor한다. 문법이니 익숙해지자.

App.jsx에서 이전 슬라이드에서 만든 state 변경 함수를 사용해보자.

```
const dispatch = useDispatch();

return (
  <div>
    <button type='button' onClick={() =>
      dispatch(increase())
    }>counter + 1</button>
    <button type='button' onClick={() =>
      dispatch(handleCounter(10))
    }>counter 증가</button>
    {allData.counter} / {counter1} / {counter2} / {name}
  </div>
)
```

- 슬라이스에서 정의한 state 변경 함수는 직접 호출해서 사용하는 것이 아니라, '함수를 호출해주세요'라는 느낌으로 함수 호출을 위임해야 한다.
- const dispatch = useDispatch() 는 dispatch 생성하는 것이다. dispatch는 위에서 말한 우리가 실행할 함수를 호출해주는 역할을 한다.
- dispatch(호출할함수)는 dispatch에게 함수 호출을 해달라는 코드이다.
- handleCounter 함수의 인자로 전달한 10은 함수 정의 부분의 action.payload값으로 전달된다.

이번에는 객체 데이터를 redux를 활용해 다루어보자. 먼저 slice를 다음과 같이 생성한다.

```
const memberSlice = createSlice({
  name : 'member',
  initialState : {
    memName : 'kim',
    memAge : 20
  },
  reducers :{
    changeMemberName : (state, action) => {
      return {...state, memName : action.payload}
    },
    handleMemberName : (state, action) => {
      state.name = action.payload
    }
  }
});

export const {changeMemberName} = memberSlice.actions
export default memberSlice;
```

- 여기서 중요한 것은 ChangeMemberName 함수와 handleMeberName 함수는 동일하게 매개변수로 전달된 데이터로 객체의 name 값을 변경하는 함수라는 것이다.
- ChangeMemberName 함수는 우리가 알고 있는 방식으로 state 값을 변경하고 있다.
- handleMeberName 함수처럼 값을 변경하는 것은 실제 리액트에서 값 변경으로 인지하지 못한다는 것을 우리는 알고 있다. 객체와 배열 데이터를 변경할 때 항상 신경썼던 부분이다. 하지만 redux에서는 객체, 배열 데이터를 일반 데이터 변경 문법처럼 변경하면 state값을 변경해준다.

redux에서 객체와 배열의 값을 편하게 바꿀 수 있다는 것을 알게 되었다. 그래서 배열과 객체 데이터도 편하게 변경하기 위해 redux에서는 일반 데이터도 일부러 객체로 사용하는 편이다. nameSlice를 예로 들어보겠다.

```
const nameSlice = createSlice({
  name : 'name',
  initialState : 'hong',
  reducers :{
    changeName : (state, action) => {
      return state + action.payload
    }
  }
});

export const {changeName} = nameSlice.reducer
export default nameSlice;
```

```
const nameSlice = createSlice({
  name : 'name',
  initialState : {data : 'hong'},
  reducers :{
    changeName : (state, action) => {
      state.data += action.payload
    }
  }
});

export const {changeName} = nameSlice.reducer
export default nameSlice;
```

redux로 관리되는 state 변수의 값이 변경되면, state 변수를 구독하는 컴포넌트는 재렌더링된다. store에서 가져온 데이터를 선언한 컴포넌트를 재렌더링

여기서 구독이란, const data = useSelector(state => state.data) 코드를 의미한다.