Hooks

Status	Not started
Assign	React.js

REACT Hooks

useState

React에서 컴포넌트의 동적인 값을 상태라고 부른다.
이전에는 함수 컴포넌트에서는 state 사용이 불가능하였다.
이후 React hooks의 useState 함수로 state 사용이 가능해졌다.
useState는 state를 함수 컴포넌트 안에서 사용할 수 있게 해준다.

사용방법

useState를 호출하면 배열을 반환하는데,
배열의 0번째 요소는 현재 state 변수이고,
1번째 요소는 이 변수를 갱신할 수 있는 함수다.
useState 의 인자로 넘겨주는 값은 state의 초기값이다.

```
const [state 저장 변수, state 갱신 함수] = useState(상태 초기 값);
```

코드를 실제 적용시키면 다음과 같다.
state 저장 변수를 이용해 데이터를 표현하고,
onClick을 통해 state 갱신 함수를 사용하여 state를 갱신한다.

```
import React, { useState } from 'react';
import './App.css';

const [count, setCount] = useState(0)

const increaseCount () => {
  setCount(count+1)
}

function App() {
  return (
   <div>
       <div className = "react">The number is : {count}</div>
       <button onClick={() => { increaseCount }}>plus</button>
   </div>
  )
}

export default App;
```

useEffect

useEffect는 Side Effect를 다루기 위한 기본 내장 API 함수다.
React의 함수 컴포넌트는 Pure Function으로 작동한다.
그저 props로 값을 받고 그에 대한 값의 출력만이 있을 뿐이다.

하지만 React로 애플리케이션을 만들때는 외부 API에 요청이 필요할 경우가 생긴다.
이 경우에는 React 입장에서는 모두 Side Effect다.

Side Effect란 함수 내의 구현이 함수 외부에 영향을 끼치는 경우를 말한다.
Pure Function이란 함수를 입력했을때 반환되는 값을 예측 가능한 함수, 즉 똑같은 값이 나오는 함수를 말한다.

사용방법

useEffect에 첫번째 인자는 함수이고,
해당 함수 내에서 Side Effect를 실행하면 된다.

useEffect의 기본형은 전달된 함수를 3가지 조건에 실행시키는데,
렌더링이 완료된 직후
새로운 props를 전달받았을 때
state값이 변경되었을 때 가 있다.

```
useEffect (() => {
  //sideEffect
})
```

두 번째 인자에 배열을 넣어 useEffect를 실행할 조건도 설정 가능하다.
빈 배열을 넣으면 컴포넌트가 처음 생성될때만 useEffect가 실행된다.
빈 배열에 state를 넣게되면 지정한 state값이 변경될때마다 useEffect가 실행된다.

```
useEffect (() => {
  //sideEffect
}, [])
...

useEffect (() => {
  //sideEffect
}, [state])
```

useEffect 에서는 함수를 반환 할 수 있는데 이를 cleanup 함수라고 부른다.
cleanup 함수는 useEffect 에 대한 뒷정리라고 말할 수 있다.
의존성 배열이 비어있는 경우에는 컴포넌트가 사라질 때 cleanup 함수가 호출된다.

```
useEffect (() => {
  console.log('컴포넌트가 화면에 나타남');
  //sideEffect
  return () => {
   console.log('컴포넌트가 화면에서 사라짐');
  };
}, [])
```

useMemo와 useCallback을 배우기 전에 알아야 하는 것

함수형 컴포넌트는 그냥 함수다. 다시 한 번 강조하자면 함수형 컴포넌트는 단지 jsx를 반환하는 함수이다.

컴포넌트가 렌더링 된다는 것은 누군가가 그 함수(컴포넌트)를 호출하여서 실행되는 것을 말한다.
함수가 실행될 때마다 내부에 선언되어 있던 표현식(변수, 또다른 함수 등)도 매번 다시 선언되어 사용된다.
컴포넌트는 자신의 state가 변경되거나, 부모에게서 받는 props가 변경되었을 때마다 리렌더링 된다.
(심지어 하위 컴포넌트에 최적화 설정을 해주지 않으면 부모에게서 받는 props가 변경되지 않았더라도 리렌더링 되는게 기본이다.)

useMemo

useMemo는 메모리제이션된 값을 반환한다는 문장이 핵심이다.
만약 컴포넌트가 2개의 props를 전달받을때,
한가지 props가 변경되어도 두가지 props를 동시에 가공시켜야한다.
변경 되지않는 props는 useMemo에 저장시켜서 이전에 계산된 값을 쓰면 된다.

```
// App.js

import Info from "./Info";

const App = () => {
  const [color, setColor] = useState("");
  const [movie, setMovie] = useState("");

  const onChangeHandler = e => {
    if (e.target.id === "color") setColor(e.target.value);
```

```
  else setMovie(e.target.value);
};

return (
  <div className="App">
    <div>
      <label>
        What is your favorite color of rainbow ?
        <input id="color" value={color} onChange={onChangeHandler} />
      </label>
    </div>
    <div>
      What is your favorite movie among these ?
      <label>
        <input
          type="radio"
          name="movie"
          value="Marriage Story"
          onChange={onChangeHandler}
        />
        Marriage Story
      </label>
      <label>
        <input
          type="radio"
          name="movie"
          value="The Fast And The Furious"
          onChange={onChangeHandler}
        />
        The Fast And The Furious
      </label>
      <label>
        <input
          type="radio"
          name="movie"
          value="Avengers"
          onChange={onChangeHandler}
        />
        Avengers
      </label>
    </div>
    <Info color={color} movie={movie} />
  </div>
);
};

export default App;
```

```
// Info.js

const getColorKor = color => {
    console.log("getColorKor");
    switch (color) {
      case "red":
        return "빨강";
      case "orange":
        return "주황";
      case "yellow":
        return "노랑";
      case "green":
        return "초록";
      case "blue":
        return "파랑";
      case "navy":
        return "남";
      case "purple":
        return "보라";
      default:
        return "레인보우";
    }
};

const getMovieGenreKor = movie => {
  console.log("getMovieGenreKor");
  switch (movie) {
    case "Marriage Story":
      return "드라마";
    case "The Fast And The Furious":
      return "액션";
    case "Avengers":
      return "슈퍼히어로";
    default:
      return "아직 잘 모름";
  }
```

```
  };


const Info = ({ color, movie }) => {
  const colorKor = getColorKor(color);
  const movieGenreKor = getMovieGenreKor(movie);

  return (
    <div className="info-wrapper">
      제가 가장 좋아하는 색은 {colorKor} 이고, <br />
      즐겨보는 영화 장르는 {movieGenreKor} 입니다.
    </div>
  );
};

export default Info;
```

해당 코드에서 계산함수 useMemo 내부 콜백함수에 저장시킨다.
의존성 배열에 넘겨준 값이 변경되었을 때만 메모리제이션된 값을 다시 계산시켜준다.

```
import React, { useMemo } from "react";

const colorKor = useMemo(() => getColorKor(color), [color]);
const movieGenreKor = useMemo(() => getMovieGenreKor(movie), [movie]);
```

useCallback

useCallback은 메모리제이션된 함수를 반환한다라는 문장이 핵심이다.
리액트는 컴포넌트가 렌더링 될 때마다 내부에 선언되어 있던 표현식(변수, 또다른 함수 등)도 매번 다시 선언되어 사용된다.
App.js의 onChangeHandler 함수는 내부의 color, movie 상태값이 변경될 때마다 재선언된다는 것을 의미한다.
이 함수를 굳이 매번 선언하기보다는 한번만 선언하고 재사용하기 위해서 useCallback을 사용한다.

```
// App.js

import React, { useState, useCallback } from "react";

const onChangeHandler = useCallback(e => {
    if (e.target.id === "color") setColor(e.target.value);
    else setMovie(e.target.value);
  }, []);
```

이벤트 핸들러 함수나 api를 요청하는 함수를 주로 useCallback 으로 선언한다.

React.memo를 이용하여 최적화한 컴포넌트에 이러한 함수를 props로 전달하는 경우일때
상위 컴포넌트에서 useCallback을 사용하면 좋다.
왜냐면 매번 재선언 된 함수는 하위 컴포넌트가 변경된 값이라고 인식하기때문이다.

useCallback과 useMemo의 차이점

useMemo는 복잡한 함수의 return값을 기억해야 할 때
useCallback은 함수 자체를 기억해야 할 때

useContext

React의 props를 전역적으로 사용할 수 있게 도와주는 Hook.
부모 -> 자식으로 불필요하게 여러번 넘겨줘 props drilling이 발생하는 경우
Redux를 사용하기도 하지만 useContext를 사용해도 이 문제를 해결 할 수 있다.

사용방법

```
import { createContext, useState } from 'react';

export const TestContext = createContext({
  name: '',
  setNameHandler: (name: string) => {},
});

const TestContextProvider: React.FC<React.ReactNode> = ({ children }) => {
  const [firstName, setFirstName] = useState('');
```

```
  const setFirstNameHandler = (name: string) => setFirstName(name);

  return (
  <TestContext.Provider
    value={{ name: fristName, setNameHandler: setFirstNameHandler }}>
    {children}
  </TestContext.Provider>
  );
}

exprot default TestContextProvider;
```

createContext 를 이용해 단일 export 할 수 있는 변수 생성, 그 안에 내게 필요한 함수 작성한다.
Provider 에서 value 상태 입력 (value 값은 무조건 필요하며 작성안하면 오류뜸)
상태를 적용해줄 때 제일 최상단에(App.js 같은)에Provider 적용된 함수 작성해주면 된다.

```
import TestContextProvider from './utils/context/TestContext';

const App = () => {
  return (
    <TestContextProvider>
      <div className='App'></div>
    </ TestContextProvider>
  );
};

export default App;
```

다른 컴포넌트에서 생성한 Context를 import 해준뒤, 사용 가능하다..
설정값인 name 혹은 setNameHandler를 사용할 때 testContext.name 혹은 .
testContext.setNameHandler와 같이 사용할 수 있다..

```
import { useContext } from 'react';
import { TestContext } from './utils/context/TestContext';

const Slider = () => {
  const { name, setNameHandler } = useContext(TestContext)

  return <div></div>;
};

export default Slider;
```

useContext를 이용해서도 사용 가능하다..

useContext를 이용할때 주의할점

useContext를 쓸 때 주의할 사항은, Provider에 제공한 value가 달라지면 useContext를 쓰고 있는 모든 컴포넌트가 리렌더링 된다.
value 안의 상태가 하나라도 바뀌면 객체로 묶여있으므로 전체가 리렌더링 됨.
따라서 잘못 쓰면 엄청난 렉을 유발할 수 있다.

useRef

JavaScript 에서 특정 DOM 을 선택 시 getElementById, querySelector 같은 DOM Selector 함수를 사용해서 DOM 을 선택한다.
React안에서 DOM reference를 활용 할 수 있게 해주는 Hook.
useRef 로 관리하는 변수는 값이 바뀐다고 해서 컴포넌트가 리렌더링되지는 않는다.
기본 React 문법을 벗어나 useRef를 남용하는 것은 부적절하고,
특정 엘리먼트의 크기를 가져와야 한다던지, 스크롤바 위치를 가져오거나 설정해야된다던지, 또는 포커스를 설정.
추가적으로 Video.js, JWPlayer 같은 HTML5 Video 관련 라이브러리,
또는 D3, chart.js 같은 그래프 관련 라이브러리 등의 외부 라이브러리를 사용해야 할 때에도
특정 DOM 에다 적용하기 때문에 useRef를 사용하면 적절하다.
React의 특징이자 장점인 선언적 프로그래밍 원칙과 배치되기 때문에, 조심해서 사용해야 한다.

사용방법

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

```
export default function App() {
  const videoRef = useRef(null);

  const playVideo = () => {
    videoRef.current.play();
    console.log(videoRef.current);
  };

  const pauseVideo = () => {
    videoRef.current.pause();
    videoRef.current.remove();
  };

  return (
    <div className="App">
      <div>
        <button onClick={playVideo}>Play</button>
        <button onClick={pauseVideo}>Pause</button>
      </div>
      <video ref={videoRef} width="320" height="240" controls>
        <source
          type="video/mp4"
          src="<https://player.vimeo.com/external/544643152.sd.mp4?s=7dbf132a4774254dde51f4f9baabbd92f6941282&profile_id=165>"
        />
      </video>
    </div>
  );
}
```

vedio요소에 접근하기 위해 useRef를 사용하여 변수에 참조값을 저장하고,
해당 변수를 이용해 vedio요소에 접근하여 vedio요소에 있는 prototype을 사용할 수 있다.
