React Hooks

김희윤

React Hooks 란?

React 16.8.0에서 새로 도입된 기능

함수형 컴포넌트에서 상태를 관리하기 위해 클래스 컴포넌트 새로 작성

현재: 함수형 컴포넌트에서도 상태를 가질 수 있게 됨

React Hooks의 장점

1. 상태에 의존하는 로직을 재사용하기 어렵던 것을 해결

 - 그동안 API 레벨에서 컴포넌트에 재사용 가능한 로직을 추가하는 방법을 제공하지 않음

 - render props 나 higher-order components 같은 로직을 추가해야만 했음

- 컴포넌트 계층을 다시 구성하거나, 이해를 어렵게 만들었음

 - 결국 Wrapper가 추가되어야 하기 때문에 Hell이 펼쳐짐

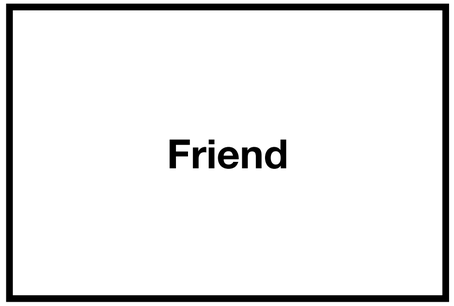
 - 상태를 가진 로직을 추출해서 독립적인 테스트 가능

 - 로직을 재사용하기 위해서 컴포넌트 계층을 굳이 변경하지 않아도 됨

React Hooks 의 장점

2. 컴포넌트의 복잡함을 해소

클래스 컴포넌트에 lifecycle 메서드를 사용하면 컴포넌트가 복잡해진다



친구가 접속했는지 실시간으로 구독하고, 데이터를 가져오는 컴포넌트

Friend 컴포넌트가 제거될 때는 접속여부를 해제해야 함

- 친구의 상태를 구독하는 로직과 데이터를 가져오는 로직이 함께 있을 수 밖에 없음
(나중에 다른 기능이 추가되면 더 복잡해 질 것)

→ **Hooks는 하나의 컴포넌트를 여러 개의 작은 함수로 나눔**
(메서드 단위가 아니라 연관된 것에 기반!)

- 또한 나중에 접속을 해제하는 기능은 연관된 로직인데 따로 작성할 수 밖에 없었음

React Hooks 의 장점

3. 클래스는 사람, 컴퓨터 모두 이해하기 어렵다, Hook는 쉽다!

- 클래스의 존재가 리액트를 배우는데 장애물이 됨
- 클래스 컴포넌트가 의도치 않은 패턴을 조장할 수 있다 -> 최적화를 방해한다
- 클래스는 코드 압축이 잘 안된다
- 클래스 컴포넌트들은 hot reloading을 신뢰할 수 없게 만든다

Hooks 는 클래스 없이 리액트의 기능들을 사용할 수 있게 해준다!

Hooks API는 10가지로 구성되어 있음 -> 이 PPT에서는 6가지만 소개

1. useState
2. useEffect
3. useContext
4. useReducer
5. useCallback
6. useMemo

1. useState

- 함수형 컴포넌트에서 state를 가질 수 있게 하는 Hook

- 두 개의 요소가 담긴 배열을 반환
(현재상태, 상태를 설정할 수 있는 함수)

index.js

```jsx
import React, { useState } from "react";
import ReactDOM from "react-dom";

function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Current COUNT : {count}</p>
      <button onClick={() => setCount(prev => prev - 1)}>-</button>
      <button onClick={() => setCount(prev => prev + 1)}>+</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}

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

2. useEffect

- 사이드이펙트를 일으킬 수 있는 함수를 전달받음

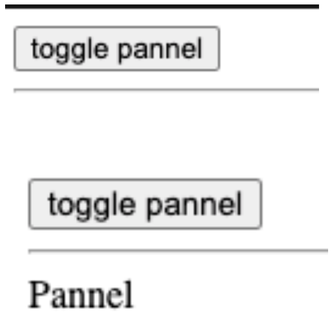
(사이드이펙트 = 외부 데이터 변경, 특정 데이터 구독, 타이머 설정, 로깅 등)

- 함수형 컴포넌트의 body에 작성되면 안됨

(컴포넌트가 렌더링될 때마다 실행되기 때문에 UI버그 발생 가능)

- 첫번째 인자로 이펙트를 일으킬 함수,

두 번째 인자로 디펜던시 목록




```
component will unmount
```

```
component did mount
```

index.js

```jsx
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

function App() {
  const [isShowPannel, setIsShowPannel] = useState(true);
  return (
    <div>
      <button onClick={() => setIsShowPannel(!isShowPannel)}>
        toggle pannel
      </button>
      <hr />
      {isShowPannel && <Pannel />}
    </div>
  );
}

function Pannel() {
  useEffect(() => {
    console.log("component did mount");

    return () => {
      console.log("component will unmount");
    };
  }, []);
  return <div>Pannel</div>;
}

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

3. useContext

- React.createContext 로 생성된 컨텍스트 객체를 인자로 받고 컨텍스트 현재 값을 반환

- 현재의 컨텍스트 값은 useContext를 호출하는 컴포넌트의 상위 컴포넌트 중 가장 가까운 <MyContext.Provider>의 value에 의해 결정

- 함수형 컴포넌트에서 컨텍스트를 더 쉽게 사용 가능

```jsx
import React, { createContext, useContext } from 'react';

const ThemeContext = createContext('black');
const ContextSample = () => {
  const theme = useContext(ThemeContext);
  const style = {
    width: '24px',
    height: '24px',
    background: theme
  };
  return <div style={style} />;
};
```

4. useReducer

- useState의 대안
(여러가지의 부수적인 값들을 포함하거나, 이전의 상태에 다음 상태가 의존하는 경우)

- 더 다양한 상황에 따라 다양한 상태를 다른 값으로 업데이트 해주고 싶을 때

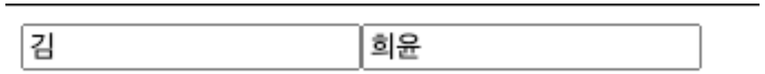
```
function init(initialCount) {
  return {
    count: initialCount
  };
}

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return {
        count: state.count + 1
      };
    case "decrement":
      return {
        count: state.count - 1
      };
    case "reset":
      return init(action.payload);
    default:
      throw new Error("not valid action type");
  }
}

function Counter({ initialCount }) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  console.log(initialCount);
  console.log(state);

  return (
    <div>
      <p>Current COUNT : {state.count}</p>
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
      <button
        onClick={() => dispatch({ type: "reset", payload: initialCount })}
      >
```

5. useCallback

- 콜백을 반환함

- 뒤의 useMemo를 편하게 사용하기 위한 shortcut

- useMemo에서 함수를 반환하는 상황에서 더 편하게 사용가능




```
previous lastname 희윤
handleLastnameChange recreated by lastname change
previous firstname 김
handleFirstnameChange is recreated
```

```jsx
function App() {
  const [firstname, setFirstname] = useState("");
  const [lastname, setLastname] = useState("");

  const handleFirstnameChange = useMemo(() => {
    console.log("handleFirstnameChange is recreated");
    return e => {
      setFirstname(e.target.value);
      console.log("previous firstname", firstname);
    };
  }, [firstname]);

  const handleLastnameChange = useCallback(
    e => {
      setLastname(e.target.value);
      console.log("previous lastname", lastname);
    },
    [lastname]
  );

  if (prev !== undefined && prev !== handleLastnameChange) {
    console.log("handleLastnameChange recreated by lastname change");
  }

  prev = handleLastnameChange;

  return (
    <div>
      <input onChange={handleFirstnameChange} value={firstname} />
      <input onChange={handleLastnameChange} value={lastname} />
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

6. useMemo

- 렌더링 과정에서 넘겨받은 함수 값을 실행

- 최적화의 수단으로 사용해야 함
(함수형 컴포넌트 내부에서 발생하는 연산을 최적화 할 수 있다)

- 숫자, 문자열, 객체 같은 일반 값을 재사용 = useMemo

- 함수를 재사용 = useCallback