React Hook

1. useState

(1) 정의

useState는 가장 기본적인 hook, 렌더링을 다시하기 위해서 사용

또 함수 컴포넌트에서 가변적인 상태를 가지게 해준다.

→ 가변적인 상태란? **< 바꿀 수 있거나 바뀔 수 있는 상태 >**

```
const [state, setState] = useState(초기값); // 기본형태

const [초기값을 가지고 있는 변수, set + 초기값을 가지고 있는 변수를 변경할 수 있는 메소드]
```

(2) 특징

1) 함수형 업데이트

```
// 기존에 우리가 사용하던 방식
setState(number + 1);

// 함수형 업데이트
setState((currentNumber) => {return currentNumber + 1});
```

2) 차이점

```
      {/* 버튼을 누르면 1씩 플러스된다. */}
      <div>{number}</div>
      <button
        onClick={() => {
          setNumber(number + 1); // 첫번째 줄
          setNumber(number + 1); // 두번째 줄
          setNumber(number + 1); // 세번째 줄
        }}

        // 1씩 증가한다.

        {/* 버튼을 누르면 3씩 플러스 된다. */}
        <div>{number}</div>
        <button
          onClick={() => {
            setNumber((previousState) => previousState + 1);
            setNumber((previousState) => previousState + 1);
            setNumber((previousState) => previousState + 1);
          }}
```

```
        // 3씩 증가한다.
```

일반 업데이트 방식은 버튼을 클릭했을 때 setNumber가 각각 실행되는 것이 아니라, 배치(batch)로 처리.

onClick을 했을 때 setNumber 라는 명령을 세번 내리지만, 리액트는 그 명령을 하나로 모아 최종적으로 한번만

실행

반면에 **함수형 업데이트 방식**은 **3번을 동시에 명령을 내리면, 그 명령을 모아 순차적으로 각각 1번씩 실행**

→ 0에 1더하고, 그 다음 1에 1을 더하고, 2에 1을 더해서 3이라는 결과가 우리 눈에 보이는 것이죠.

3) 단일 업데이트(batch update)

불필요한 **리-렌더링**을 방지(렌더링 최적화)하기 위해.

→ 즉, 리액트의 성능을 위해 한꺼번에 state를 업데이트

2.useEffect

(1) 정의

useEffect는 리액트 컴포넌트가 렌더링될 때마다 특정 작업을 수행하도록 설정할 수 있는 Hook.

< 예시 >

```
import React, { useEffect, useState } from "react";

const App = () => {
  const [value, setValue] = useState("");

  useEffect(() => {
    console.log("hello useEffect");
  });

  return (
    <div>
      <input
        type="text"
        value={value}
        onChange={(event) => {
          setValue(event.target.value);
        }}
      />
    </div>
  );
}
```

```
export default App;
```

1. input에 값을 입력한다.
2. value, 즉 state가 변경된다.
3. state가 변경되었기 때문에, App 컴포넌트가 리렌더링 된다.
4. 리렌더링이 되었기 때문에 useEffect가 다시 실행된다.
5. 1번 → 5번 과정이 계속 순환한다.

(2) 특징

1) 의존성 배열

- **의존성 배열(dependency array) 이란?**

  → **배열에 값을 넣으면 그 값이 바뀔 때만 실행한다는 것**

```
useEffect(()=>{
  // 실행하고 싶은 함수
}, [의존성배열])
```

- 의존성 배열을 사용하는 방법 2가지

-배열에 값을 넣을때 : **그 값이 바뀔 때만 실행**

-배열에 빈 상태로 넣을때 : **컴포넌트가 렌더링 될 때 단 한번만 실행**

```
const [value, setValue] = useState("");
  useEffect(() => {
    console.log("hello useEffect");
  }, [@]); // @가 바뀔때만 실행



const [value, setValue] = useState("");
  useEffect(() => {
    console.log("hello useEffect");
  }, []); // 컴포넌트가 렌더링 될 때 단 한번만 실행
```

2) 클린업(clean up)

- 클린업이란?

  → **컴포넌트가 사라졌을 때 실행**

```
  useEffect(()=>{
    // 화면에 컴포넌트가 나타났을 때 실행하고자 하는 함수를 넣어주세요.

    return ()=>{
      // 화면에서 컴포넌트가 사라졌을 때 실행하고자 하는 함수를 넣어주세요.
    }
  }, [])
```

```
// src/SokSae.js

import React, { useEffect } from "react";
import { useNavigate } from "react-router-dom";

const 속세 = () => {
  const nav = useNavigate();

  useEffect(() => {
    return () => {
      console.log(
        "안녕히 계세요 여러분! 전 이 세상의 모든 굴레와 속박을 벗어 던지고 제 행복을 찾아 떠납니다! 여러분도 행복하세요~~!"
      );
    };
  }, []);

  return (
    <button
      onClick={() => {
        nav("/todos");
      }}
    >
      속세를 벗어나는 버튼
    </button>
  );
};

export default 속세;
```

3.useRef

(1) 정의

DOM 요소에 접근할 수 있도록 하는 Hook

(2) 특징

```
function App() {
  const ref = useRef("초기값");
  console.log("ref 1", ref);

  ref.current = "바꾼 값";
  console.log("ref 1", ref);

// current : "초기값"
// current : "바꾼 값"
```

(중요) 이렇게 설정된 ref 값은 컴포넌트가 계속해서 렌더링 되어도 컴포넌트가 삭제되기 전까지 값을 유지

1) useRef는 다음 용도로 사용

1. **저장공간**
   a. state와 비슷한 역할.
      i. state는 리렌더링이 꼭 필요한 값을 다룰 때 사용
      ii. ref는 리렌더링을 발생시키지 않는 값을 저장할 때 사용
   b. state와 ref의 차이점
      i. state는 변경되면 렌더링이 된다.
      ii. ref는 변경되면 렌더링이 안된다

< 예시 >

```
import "./App.css";
import { useRef, useState } from "react";

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

  const plusStateCountButtonHandler = () => {
    setCount(count + 1);
  };

  const plusRefCountButtonHandler = () => {
    countRef.current++;
  };

  return (
    <>
      <div>
        state 영역입니다. {count} <br />
        <button onClick={plusStateCountButtonHandler}>state 증가</button>
      </div>
      <div>
        ref 영역입니다. {countRef.current} <br />
        <button onClick={plusRefCountButtonHandler}>ref 증가</button>
      </div>
    </>
  );
}

export default App;
```

2. **DOM**
   a. 렌더링 되자마자 특정 input이 focusing 돼야 할 때

```
import { useEffect, useRef } from "react";
import "./App.css";

function App() {
  const idRef = useRef("");

  // 렌더링이 될 때
  useEffect(() => {
    idRef.current.focus();
  }, []);

  return (
    <>
      <div>
        아이디 : <input type="text" ref={idRef} />
      </div>
      <div>
        비밀번호 : <input type="password" />
      </div>
    </>
  );
}

export default App;
```

4.useContext

(1) Context란?

보통 props를 사용해 부모 → 자식 컴포넌트로 정보를 전달.

하지만 정보들을 전역적으로 사용하기 위해서 사용, 이를 사용하기 위한 기능을 **context API**

context API 필수 개념

- `createContext` : context 생성
- `Consumer` : context 변화 감지
- `Provider` : context 전달(to 하위 컴포넌트)

(2) 정의

Context를 이용해 전역데이터를 쉽게 관리할 수 있는 Hook

→ 전역데이터 : 컴퓨터 프로그램 내의 모든 모듈 및 요소에서 접근이 가능한 데이터

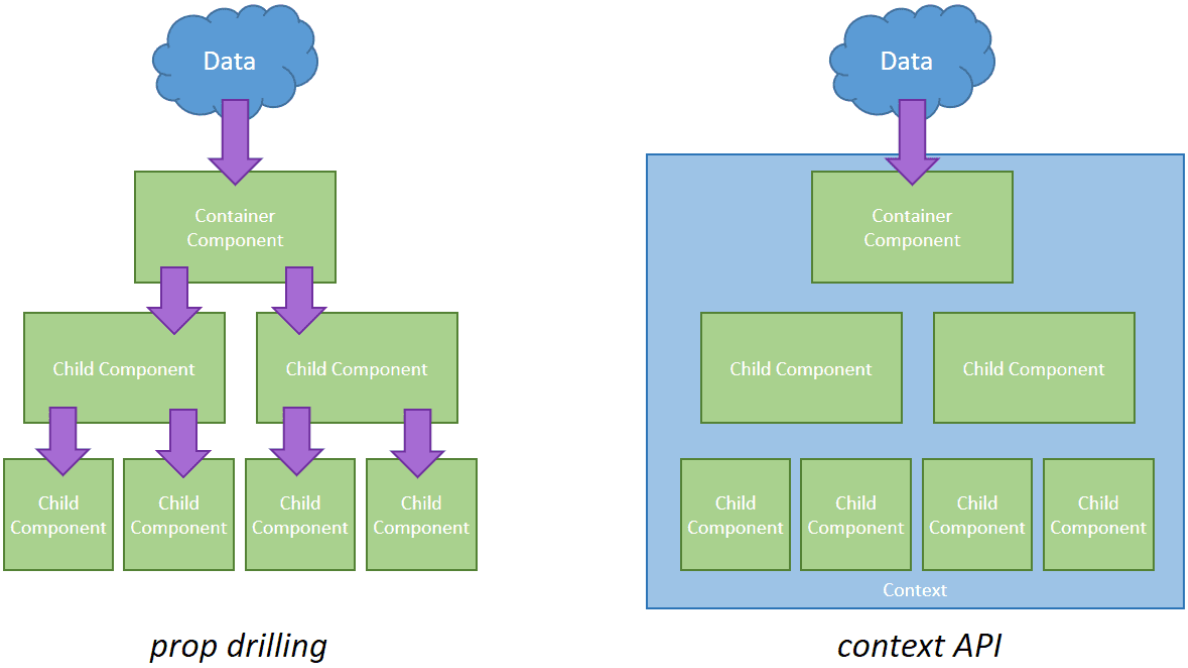
(3) 특징

1) **prop drilling 개선할 수 있다**

props로 부모 → 자식으로 자료를 넘겨주었다. 하지만 깊어지게 되면 prop drilling 현상이 발생

- **prop drilling의 문제점은**

1. 깊이가 너무 깊어지면 이 prop이 어떤 컴포넌트로부터 왔는지 파악이 어렵다
2. 어떤 컴포넌트에서 오류가 발생할 경우 추적이 힘들어 대처가 늦다.



prop drilling *context API*

출처 : https://www.copycat.dev/blog/react-context/

< 예시 > **useContext를 사용하지 않았을 때**

GrandFather.jsx

```
import React from "react";
import Father from "./Father";

function GrandFather() {
  const houseName = "스파르타";
  const pocketMoney = 10000;

  return <Father houseName={houseName} pocketMoney={pocketMoney} />;
}

export default GrandFather;
```

Father.jsx

```
import React from "react";
import Child from "./Child";

function Father({ houseName, pocketMoney }) {
  return <Child houseName={houseName} pocketMoney={pocketMoney} />;
}

export default Father;
```

Child.jsx

```
import React from "react";

function Child({ houseName, pocketMoney }) {
  const stressedWord = {
    color: "red",
    fontWeight: "900",
  };
  return (
    <div>
      나는 이 집안의 막내에요.
      <br />
      할아버지가 우리 집 이름은 <span style={stressedWord}>{houseName}</span>
      라고 하셨어요.
      <br />
      게다가 용돈도 <span style={stressedWord}>{pocketMoney}</span>원만큼이나
      주셨답니다.
    </div>
  );
}

export default Child;
```

< 예시 > **useContext를 사용했을 때**

. context > FamilyContext.js 생성

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

// 여기서 null이 의미하는 것은 무엇일까요? 초기값
export const FamilyContext = createContext(null);
```

#2. GrandFather.jsx 수정

```
import React from "react";
import Father from "./Father";
import { FamilyContext } from "../context/FamilyContext";

function GrandFather() {
  const houseName = "스파르타";
```

```
  const pocketMoney = 10000;

  return (
    <FamilyContext.Provider value={{ houseName, pocketMoney }}>
      <Father />
    </FamilyContext.Provider>
  );
}

export default GrandFather;
```

#3. Father.jsx 수정(props를 제거해요!)

```
import React from "react";
import Child from "./Child";

function Father() {
  return <Child />;
}

export default Father;
```

#4. Child.jsx 수정

```
import React, { useContext } from "react";
import { FamilyContext } from "../context/FamilyContext";

function Child({ houseName, pocketMoney }) {
  const stressedWord = {
    color: "red",
    fontWeight: "900",
  };

  const data = useContext(FamilyContext);
  console.log("data", data);

  return (
    <div>
      나는 이 집안의 막내에요.
      <br />
      할아버지가 우리 집 이름은 <span style={stressedWord}>{data.houseName}</span>
      라고 하셨어요.
      <br />
      게다가 용돈도 <span style={stressedWord}>{data.pocketMoney}</span>원만큼이나
      주셨답니다.
    </div>
  );
}

export default Child;
```

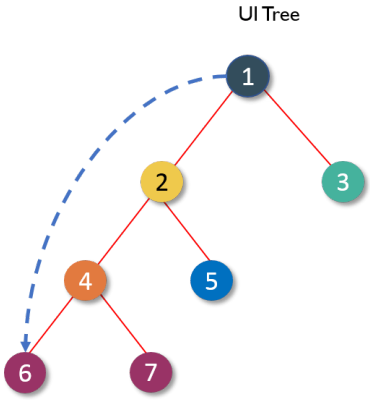
5.memo(React.memo)

(1) **memo란?**

컴포넌트를 메모리에 임시로 저장했다가 필요할때 가져다가 사용하는 것. 메모리에 저장하는 것을 캐싱

> → 캐싱 : 파일 복사본을 캐시 또는 임시 저장 위치에 저장하여 보다 빠르게 액세스할 수 있도록 하는 프로세스

부모 컴포넌트가 렌더링 되면 모든 자식 컴포넌트 또한 렌더링 되는데, props가 변경되지 않았다면 자식 컴포넌트는 렌더링 될 필요가 없기 때문에, 이때 React.memo 함수를 사용해 불필요한 렌더링을 방지



- 1번 컴포넌트가 리렌더링 된 경우, 2~7번이 모두 리렌더링
- 4번 컴포넌트가 리렌더링 된 경우, 6, 7번이 모두 리렌더링

< 예시 > memo 사용 전

App.jsx

```
function App() {
  console.log("App 컴포넌트가 렌더링되었습니다!");

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

  // 1을 증가시키는 함수
  const onPlusButtonClickHandler = () => {
    setCount(count + 1);
  };

  // 1을 감소시키는 함수
  const onMinusButtonClickHandler = () => {
    setCount(count - 1);
  };

  return (
    <>
      <h3>카운트 예제입니다!</h3>
      <p>현재 카운트 : {count}</p>
      <button onClick={onPlusButtonClickHandler}>+</button>
      <button onClick={onMinusButtonClickHandler}>-</button>
```

```
      <div style={boxesStyle}>
        <Box1 />
        <Box2 />
        <Box3 />
      </div>
    </>
  );
}

export default App;
```

Box1~3.jsx

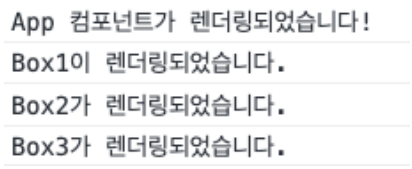
```
function Box1() {
  console.log("Box1이 렌더링되었습니다.");
  return <div style={boxStyle}>Box1</div>;
}
export default Box1;

//-----------------------

function Box2() {
  console.log("Box2가 렌더링되었습니다.");
  return <div style={boxStyle}>Box2</div>;
}
export default Box2;

//-----------------------

function Box3() {
  console.log("Box3가 렌더링되었습니다.");
  return <div style={boxStyle}>Box3</div>;
}
export default Box3;
```



모든 하위 컴포넌트가 리렌더링 되었다. 하지만 실제로 변한 것은 부모컴포넌트, `App.jsx` 뿐

< 예시 > **memo를 통해 해결해보기**

Box1~3.jsx

```
export default React.memo(Box1);
export default React.memo(Box2);
export default React.memo(Box3);
```

최초 렌더링 이외에는 App.jsx 컴포넌트의 state가 변경되더라도 자식 컴포넌트들은 렌더링이 되지 않는다. App.jsx 컴포넌트만 렌더링이 됐었다.

6.useCallback

(1) 정의

useCallback이란?

함수 자체를 기억했다가, 필요할때 가져다가 사용하는 것

< 예시 >

```
...

  // count를 초기화해주는 함수
  const initCount = () => {
    setCount(0);
  };

  return (
    <>
      <h3>카운트 예제입니다!</h3>
      <p>현재 카운트 : {count}</p>
      <button onClick={onPlusButtonClickHandler}>+</button>
      <button onClick={onMinusButtonClickHandler}>-</button>
      <div style={boxesStyle}>
        <Box1 initCount={initCount} />  // props로 자료를 전달
        <Box2 />
        <Box3 />
      </div>
    </>
  );
}

...
```

```
...

function Box1({ initCount }) {
  console.log("Box1이 렌더링되었습니다.");

  const onInitButtonClickHandler = () => {
    initCount();
  };
```

```
  return (
    <div style={boxStyle}>
      <button onClick={onInitButtonClickHandler}>초기화</button>
    </div>
  );
}

...
```

```
App 컴포넌트가 렌더링되었습니다!
Box1이 렌더링되었습니다.
```

`App` 컴포넌트와 `Box1` 컴포넌트가 **리렌더링** 되는 것을 볼 수 있습니다.

React.memo를 통해서 Box1.jsx는 메모이제이션을 했는데도 리렌더링이 된다?? 왜??

→ 함수형 컴포넌트를 사용하기 때문이고 App.jsx가 리렌더링 되면서 코드가 다시 만들어지기 때문

```
const initCount = () => {
   setCount(0);
 };
```

- 메모리에 직접 저장이 되는것이 아니라, 별도의 공간에 저장이 된다. 그리고 별도의 공간을 바라보는 주소값을 저장한다. 즉, 리-렌더링이 되면서 initCounter가 새로운 공간에 저장을 하면서 새로운 주소값을 저장하기 때문에 props는 변경이 되었다고 인식을 하는 것이다.

< 예시 > **useCallback 사용을 통한 함수 메모이제이션**

이 함수를 메모리 공간에 저장해놓고, 특정 조건이 아닌 경우엔 변경되지 않도록 해야겠어요.

```
// 변경 전
const initCount = () => {
  setCount(0);
};

// 변경 후
const initCount = useCallback(() => {
  setCount(0);  // 0 값을 계속 유지한다.
}, []);

// 변경 후
const initCount = useCallback(() => {
  setCount(0);
}, [@]);  // 배열안에 @가 변경될때 마다 함수를 반영
```

7. useMemo

(1) 정의

값(value)을 기억했다가, 필요할때 가져다가 사용하는 것

```
// as-is
const value = 반환할_함수();

// to-be
const value = useMemo(()=> {
  return 반환할_함수()
}, [dependencyArray]);
```

< 예시 > useMemo 사용

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

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

  const heavyWork = () => {
    for (let i = 0; i < 1000000000; i++) {}
    return 100;
  };

  // CASE 1 : useMemo를 사용하지 않았을 때
  const value = heavyWork();  // 100이 찍히지만 무거운 작업때문에 속도가 느리다.

  // CASE 2 : useMemo를 사용했을 때
  const value = useMemo(() => heavyWork(), []); // 값을 기억했다가 사용하기 때문에 빠르다.

  // CASE 3 : useMemo와 의존성배열 사용했을 때
  const value = useMemo(() => heavyWork(), [@]);
      // 값을 기억했다가 사용하기 때문에 빠르며, @가 변할때 마다 그 값을 기억한다.


  return (
    <>
      <p>나는 {value}을 가져오는 엄청 무거운 작업을 하는 컴포넌트야!</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        누르면 아래 count가 올라가요!
      </button>
      <br />
      {count}
    </>
  );
}

export default HeavyButton;
```