< React Hook >

Hook

1.1 Hook이란?

Hook은 React 버전 16.8부터 React 요소로 새로 추가되었습니다.

기존 Class 바탕의 코드를 작성할 필요 없이 **함수 컴포넌트에서 React state와 생명주기 기능(lifecycle features)을 "연동(hook into)"할 수 있게 해주는 함수**입니다. (* Hook은 class 안에서는 동작하지 않습니다.)

React는 useState 같은 내장 Hook을 몇 가지 제공합니다. 또한 컴포넌트 간에 상태 관련 로직을 재사용하기 위해 Hook을 직접 만드는 것(= Custom Hook)도 가능합니다.

1.2 Hook의 종류

- 기본 Hook
  - useState
  - useEffect

- 추가 Hooks
  - useMemo
  - useCallback

- ○ useRef
- ○ useImperativeHandle
- ○ useLayoutEffect

1.2 Hook의 규칙

Hook을 사용할 때 두 가지 규칙을 준수해야 합니다.

1. **최상위(at the Top Level) 에서만 Hook을 호출**해야 합니다.
    a. 반복문, 조건문 혹은 중첩된 함수 내에서 Hook을 호출하면 안 됩니다.

    useState, useEffect 같은 hook들이 여러 번 사용될 수 있는데, **리액트는 hook의 호출 순서를 저장**해 놓습니다. 만약 반목문이나 조건문 안에서 훅을 썼다면 훅이 조건에 따라 실행되지 않을 수 있습니다. 그럼 렌더링 때마다 리액트가 기억해 놓은 훅의 호출 순서가 꼬여서 실행될 것이고 이는 버그를 발생시키게 됩니다.

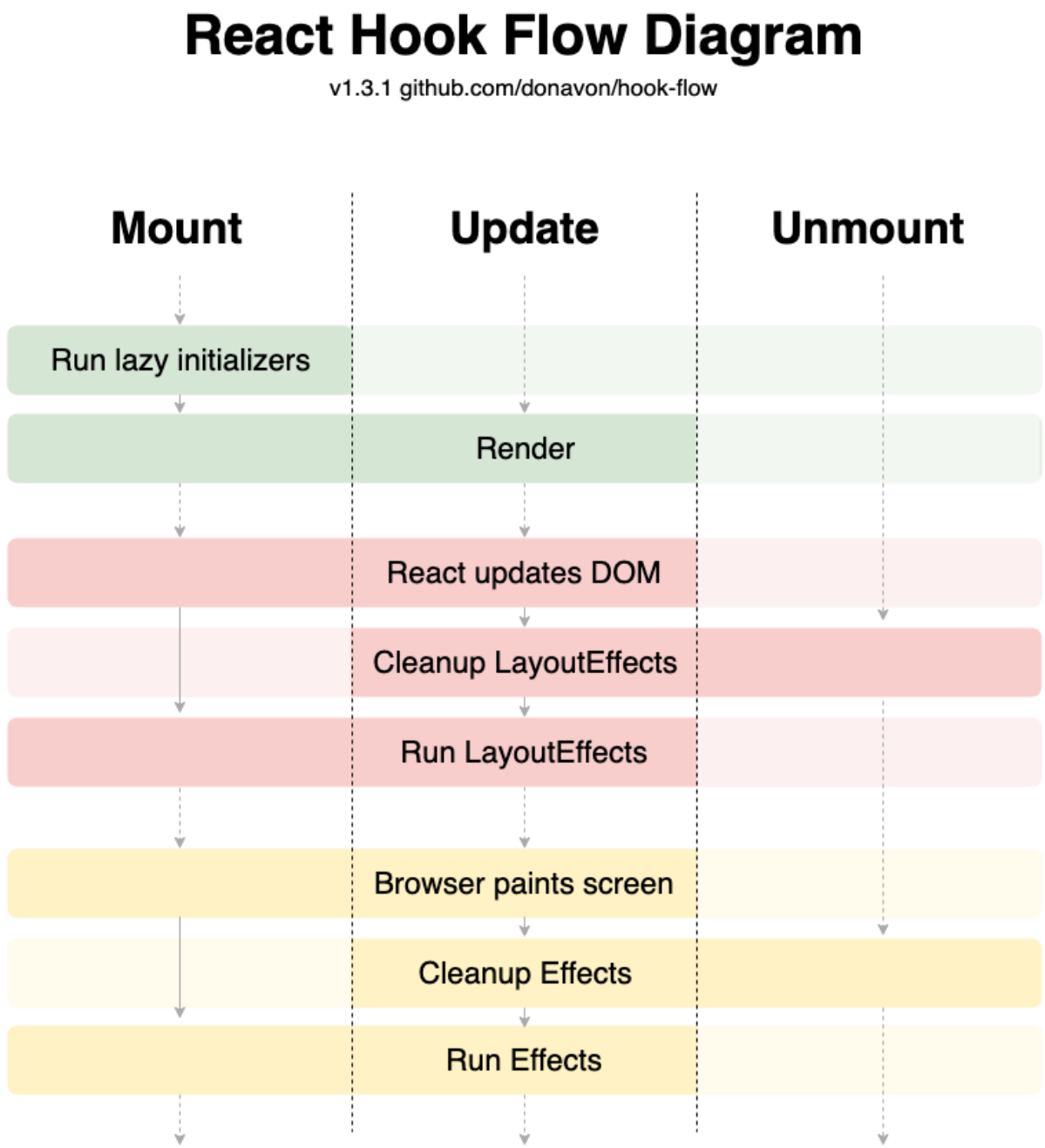
2. **오직 React 함수 내에서 Hook을 호출**해야 합니다.
    a. Hook을 일반적인 Javascript에서 호출하면 안 됩니다.
    b. React 함수형 컴포넌트 또는 Custom Hook에서 호출할 수 있습니다.

이 두가지 규칙을 강제하는 [eslint-plugin-react-hooks] 라는 ESLint 플러그인이 있는데 `Create React App` 에 기본적으로 포함되어 있습니다.

```
// ESLint 설정 파일
{
  "plugins": [
    // ...
```

```
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}
```

1.3 Hook-Flow



Notes:

1. Updates are caused by a parent re-render, state change, or context change.
2. Lazy initializers are functions passed to useState and useReducer.

2. useState

2.1 useState란?

함수형 컴포넌트에서 상태 관리를 위해 사용되며, 상태 유지 값과 그 값을 갱신하는 함수를 반환합니다. (= getter , setter)

* 상태(state)란? 동적인 데이터를 다루기 위한 객체로 컴포넌트 내부에서 변경될 수 있는 값.
* props는 (함수 매개변수처럼) 컴포넌트에 전달되는 반면, state는 (함수 내 선언된 변수처럼) 컴포넌트 안에서 관리된다는 차이가 있다.

2.2 useState 사용법

```
// 리액트 패키지에서 useState 불러오기
import React, { useState } from 'react';

/**
const numberState = useState(0);
const number = numberState[0];
const setNumber = numberState[1];

---
ES6 문법 비구조화 할당
const [상태 값 저장 변수 , 상태 값 갱신 함수] = useState(상태 초기 값);

**/

const Counter = () => {

 const [number, setNumber] = useState(0);
```

```
const onIncrease = () => {
  // Setter 함수는 파라미터로 전달 받은 값을 최신 상태로 설정
  setNumber(number + 1);
}

const onDecrease = () => {
  setNumber(number - 1);
}

return (
  <div>
    <h1>{number}</h1> // 갱신 된 현재 상태 값을 렌더링
    <button onClick={onIncrease}>+1</button>
    <button onClick={onDecrease}>-1</button>
  </div>
);
}

export default Counter;
```

1. 리액트 패키지에서 useState 를 불러옵니다.
2. useState 함수를 호출하면 길이가 2인 배열을 반환하는데 각각 상태변수, 상태변수 setter 함수입니다.
3. 최초 렌더링시 상태변수 초기값을 0으로 설정했으므로 0 이 렌더링 됩니다.

   {number} === 0
4. onclick 버튼의 함수가 작동이 되면 Setter함수를 통해 상태 값이 갱신됩니다. (setNumber 직후 화면에 state 반영 x)
5. state 변화로 리렌더링
6. 갱신 된 현재 상태 값이 브라우저에 반영됩니다.

*** state 변경 -> 리렌더링 -> state 반영**

리액트 리렌더링 조건 (함수형 컴포넌트 기준)

- state 변경이 있을 때 단, setState 함수가 아닌 직접 값을 변경하는 경우 리액트는 이를 감지하지 못하고 리렌더링 되지 않습니다. (state의 불변성)
- 전달받은 props 값이 업데이트 됐을 때
- 부모 컴포넌트가 렌더링 될 때

[함수형 업데이트]

setNumber에 값을 그대로 전달하는 것이 아니라 함수를 전달합니다.

```
// 함수형 업데이트는 주로 컴포넌트를 최적화할 때 사용
// 함수형 업데이트를 통해 setState를 동기적으로 사용할 수 있습니다.
const onIncrease = () => {
  // prevNumber는 parameter 명칭으로 개발자가 임의로 작성해주면 됩니다.
  // prevNumber는 setNumber가 호출되기 직전의 상태값을 의미합니다.
  setNumber(prevNumber => prevNumber + 1);
}
 const onDecrease = () => {
  setNumber(prevNumber => prevNumber - 1);
}
```

2.3 batch updating

리액트는 여러개의 setState()를 batch update(일괄 업데이트)로 처리하여 불필요한 렌더링을 방지하고 컴포넌트의 렌더링 횟수를 최소화합니다.

```
/*
      *** setState는 비동기적으로 작동합니다.***
      여러 state를 동시에 업데이트 하는 경우,
      리액트는 state를 일괄 업데이트합니다.(성능 향상)
```

```
        ----
        state가 변경되면 re-render를 하게 되는데
        한꺼번에 여러 번의 상태 변경이 발생하면 그만큼 많은 렌더링이 발생합니다.
        이를 방지하기 위해 state의 변경 사항을 즉시 반영하지 않고
        대기열(Queue)에 넣은 후 한꺼번에 적용합니다.

        이 과정에서 setState 호출 후의 state 값이 개발자가 의도하지 않는
        값이 될 수도 있습니다.

*/

const [value, setValue] = useState(0);
const [value2, setValue2] = useState(0);
const [value3, setValue3] = useState(0);

useEffect(() => {
  /*
   setState는 변경된 사항을 기억하지 않기 때문에 마지막 업데이트가
   상태값에 적용되어 렌더링에 쓰이게 됩니다.
  */
  setValue(value + 1); // 0(초기값) + 1 = 1
  setValue(value + 2); // 0(초기값) + 2 = 2
  setValue(value + 3); // 0(초기값) + 3 = 3
}, []);
// Expected value: 6
// Result value: 3


useEffect(() => {
  /*
    함수형 업데이트를 통해 setState를 동기적으로 사용
  */
  setValue((prev) => prev + 1); // prev:0(초기값), 0 + 1 = 1
  setValue((prev) => prev + 2); // prev:1, 1 + 2 = 3
  setValue((prev) => prev + 3); // prev:3, 3 + 3 = 6
}, []);
// Expected value: 6
```

```
// Result value: 6

/////////
useEffect(() => {
  setValue((prev) => prev + 1);
  setValue1((prev) => prev + 1);
  setValue2((prev) => prev + 1);
}, []);
// Result value: value: 1, value2: 1, value3: 1
// Component was rendered 2 times


useEffect(() => {
  setValue((prev) => prev + 1);
  setValue1((prev) => prev + 1);
  setValue2((prev) => prev + 1);
}, []);
// Result value: value: 1, value2: 1, value3: 1
// Component was rendered 2 times
```

하지만 모든 상황에서 똑같이 batch update가 일어나지는 않습니다.

이벤트 핸들러가 비동기 처리 방식으로 실행될 경우, 예를 들면 **async/await, then/catch, setTimeout, fetch** 등, 각각의 독립된 state값의 update는 일괄적으로 처리되지 않습니다.

```
const [counter1, setCounter1] = useState(0);
const [counter2, setCounter2] = useState(0);
const [counter3, setCounter3] = useState(0);

const handleClickAsync = async () => {
  await setCounter1(counter1 + 1);
  setCounter2(counter2 + 1);
  setCounter3(counter3 + 1);
};
```

```
// Result value: Counter1: 1, Counter2: 1, Counter3: 1
// Component was rendered 4 times

const handleClickThen = () => {
  Promise.resolve().then((res) => {
    setCounter1(counter1 + 1);
    setCounter2(counter2 + 1);
    setCounter3(counter3 + 1);
  });
};

// Result value: Counter1: 1, Counter2: 1, Counter3: 1
// Component was rendered 4 times
```

2.4 State의 불변성

리액트에서 State는 항상 불변성을 유지해야한다고 합니다.
불변성은 쉽게 말하면 상태를 변경하지 않는 것입니다. *즉, 메모리 영역의 값을 변경할 수 없는* 것입니다.

리액트에서 상태 값을 업데이트 할 때, **state 값을 직접적으로 변경하지 않고** 기존 값의 사본을 만든 후 원하는 값으로 변형 후 setState를 이용하여 새로운 값을 부여합니다.

왜 불변성을 지켜야하나?
원시타입은 애시당초 불변성 특징을 가지고 있지만 **참조타입인 객체나 배열의 경우 새로운 값을 변경할 때, 원본 데이터가 변경**됩니다.

```
// 원시 타입
let x = 100; // 원시 타입 데이터를 선언
let y = x; // 값을 새 변수에 복사
x = 99; // 'x'의 값을 변경
```

```
console.log(x); // 99
console.log(y); // 100, 'y'의 값은 변경되지 않음


// 참조 타입 object
let a = { count: 100 }; // 참조 타입 데이터를 선언
let b = a; // 참조를 새 변수에 복사
a.count = 99; // 참조 타입 데이터를 변경
console.log(a.count); // 99
// 'x'와 'y'는 동일한 참조를 담고 있기 때문에 동일한 객체를 가리킴
console.log(b.count);// 99


// 참조 타입 array
let array1 = [0, 1, 2, 3, 4];
let array2 = array1;
array2.push(5);
console.log(array1); //  0, 1, 2, 3, 4, 5
console.log(array2); //  0, 1, 2, 3, 4, 5
```

리액트에서 setState를 통해 상태값을 변경할 때 얕은 비교를 사용합니다. 얕은 비교란 객체의 프로퍼티를 하나하나 다 비교하지 않고, 객체의 참조 주소 값만 변경되었는지 확인 합니다. (React는 Object.is 비교 알고리즘을 사용) 얕은 비교는 계산 리소스를 줄여주기 때문에 리액트는 효율적으로 상태를 업데이트 할 수 있습니다.(성능 최적화)

때문에 불변성을 지키지 않으면 state가 바뀌어도 그 state 객체를 참조하는 참조값은 바뀌지 않기 때문에 리액트는 값의 변화를 인지하지 못하고 리렌더링을 하지않는 문제가 발생합니다.

어떻게 불변성을 지키는가?

setState를 이용할 때 원시타입 경우에는 값을 바로 넣어주어도 되지만
참조 타입인 경우 새로운 객체나 배열을 생성한 후 값을 넣어주어야 합니다.

```
// 원시타입
const [number, setNumber] = useState(0);
setNumber(3);
console.log('number', number); // 3

// 참조타입
const [person, setPerson] = useState({ name: '', age: 30 });
setPerson({ ...person, name: 'kim' });
console.log('person', person); // { name: 'kim', age: 30 }
```

spread operator, map, filter, slice, reduce, concat 등등 새로운 배열을 반환하는 메소드들을 활용해줍니다. (* splice, push 는 원본데이터를 변경합니다.)

3. useEffect

3.1 useEffect란?

Hook이 도입되기 전까지 라이프 사이클 제어를 위해 클래스형 컴포넌트를 주로 사용했지만, Hook이 도입되고 함수형 컴포넌트에서 **useEffect 를 통해 라이프 사이클 메서드를 대체**할 수 있게 되었습니다.

useEffect 는 Side-Effect 를 처리하기 위해 사용합니다. Side-Effect의 대표적인 예로 비동기 방식으로 api 호출하여 데이터 가져오기,

구독(subscription) 설정, 수동으로 DOM 조작, 타이머 설정, 로깅 등이 있습니다.

* react에서 Side-Effect란?
리액트 컴포넌트가 화면에 렌더링 된 이후에 비동기적으로 처리 되어야하는 부수적인 효과들을 뜻합니다. react에서는 화면에 렌더링할 수 있는 것은 먼저 렌더링하고 실제 데이터는 비동기로 가져오는 것이 권장됩니다. 요청 즉시 1차 렌더링을 함으로써 연동하는 API가 응답이 늦어지거나 응답이 없을 경우에도 영향을 최소화 시킬 수 있어서 사용자 경험 측면에서 유리합니다.

3.2 useEffect 사용법

useEffect는 **useEffect(callBackFunc, dependencies)** 로 두개의 인자를 넣어 호출할 수 있습니다.

- callBackFunc: 수행하고자 하는 작업
- dependencies: 배열 형태이며, 배열 안에는 검사하고자 하는 특정 값 or 빈 배열을 넣어줍니다.

dependencies에 특정 값 넣기

- deps 에 특정 값을 넣게 된다면 컴포넌트가 처음 마운트 될 때, 지정한 값이 바뀔 때, 언마운트 될 때, 값이 바뀌기 직전에 모두 호출합니다.
- useEffect 안에서 사용하는 상태나, props 가 있다면, useEffect 의 deps 에 넣어주어야 하는 것이 규칙입니다.
- 만약 사용하는 값을 넣어주지 않는다면, useEffect 안의 함수가 실행될 때 최신 상태, props를 가리키지 않습니다.
- deps 파라미터를 생략한다면, 컴포넌트가 리렌더링 될 때마다 useEffect 함수가 호출됩니다.

useEffect(callBackFunc);

렌더링 될 때마다 즉, 컴포넌트가 마운트 된 후, 업데이트 된 후, 언마운트 되기 전에 실행됩니다. 아래와 같은 코드는 [dependencies] 생략으로 *계속해서 렌더링이 발생하게 되기때문에 지양* 하는게 좋습니다.

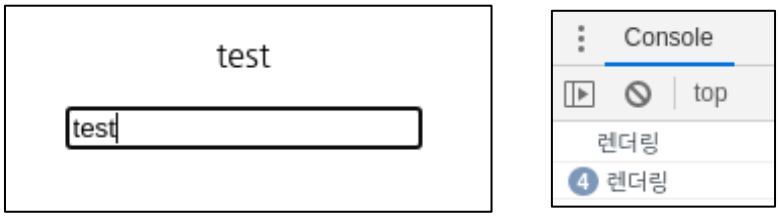
[예제 코드]

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

const App = () => {
 const [text, setText] = useState('initial value');

 useEffect(() => {
   console.log('렌더링');
 });

 return (
   <div className="App">
     <p>{text}</p>
     <input
       onChange={(e) => {
         setText(e.target.value);
       }}
     />
   </div>
 );
};

export default App;
```

결과 화면 (1)

컴포넌트가 마운트 된 후 화면 및 콘솔

결과 화면 (2)



컴포넌트가 마운트 된 후 한 번, input box에 텍스트 입력할 때마다 (컴포넌트가 업데이트된 후) 렌더링이 추가로 4번 더 발생

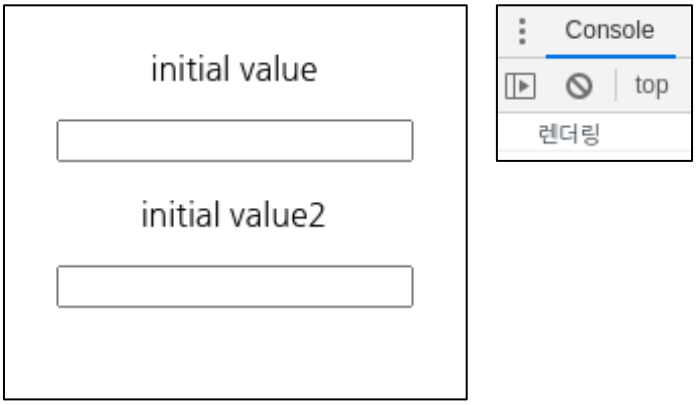
useEffect(callBackFunc, [deps1, deps2]);

최초 렌더링 됐을 때, dep1 또는 dep2가 변경되었을 때 실행됩니다.

[예제 코드]

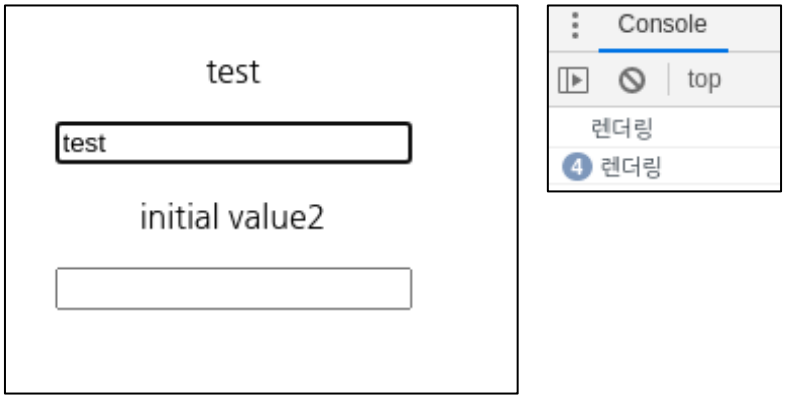
```
import React, { useState, useEffect } from 'react';
import './App.css';

const App = () => {
 const [text, setText] = useState('initial value');

 const [text2, setText2] = useState('initial value2');

 useEffect(() => {
   console.log('렌더링');
 }, [text]);

 return (
   <div className="App">
     <p>{text}</p>
     <input
       onChange={(e) => {
         setText(e.target.value);
       }}
     />
     <p>{text2}</p>
     <input
       onChange={(e) => {
         setText2(e.target.value);
       }}
     />
   </div>
 );
};
export default App;
```

결과 화면 (1)



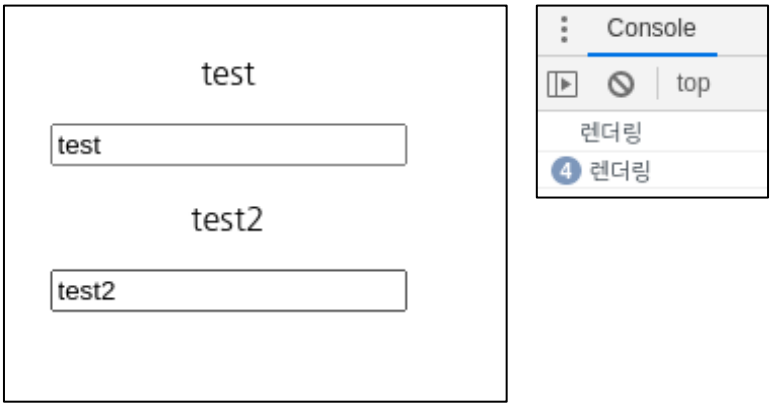
컴포넌트가 마운트 된 후 화면 및 콘솔

결과 화면 (2)



컴포넌트가 마운트 된 후 한 번, input box에 텍스트 입력할 때마다 (의존성 배열에 추가된 상태 값 text가 업데이트된 후) 렌더링이 추가로 4번 더 발생

결과 화면 (3)



컴포넌트가 마운트 된 후 한 번, input box에 텍스트 입력할 때마다 (의존성 배열에 추가된 상태 값 text가 업데이트된 후) 렌더링이 추가로 4번 더 발생, text2 값이 변경되었으나

의존성 배열에 text2가 추가되지 않아 렌더링 되지 않음. 만약 text2도 의존성 배열에 추가했을 경우 최초 렌더링 이후 발생한 렌더링 수는 총 9번이 된다. (test(4) + test2(5) =>9)

useEffect(callBackFunc, []);

최초 렌더링 됐을 때 한번 실행합니다.

```
useEffect(() => {
  console.log('첫 렌더링에만 호출')
}, [])
```

useEffect clean-up 함수

useEffect는 clean-up이라고 표현하는 함수를 return 할 수 있는데, clean-up 함수를 활용해 컴포넌트가 Unmout될 때 정리 해야 할 것을 처리합니다.

* useEffect의 이펙트 함수에서 반환되는 clean-up 함수는 컴포넌트가 unmount되는 시점에만 실행되지는 않습니다.

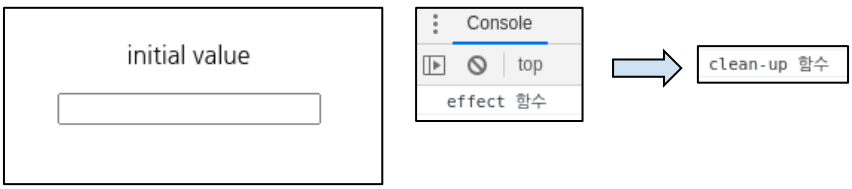
clean-up 함수는 이펙트가 실행되기 전에 매번 호출되며(update), 컴포넌트가 unmount 될 때 역시 실행됩니다.

[예제 코드]

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

const App = () => {
 const [text, setText] = useState('initial value');

 useEffect(() => {
   console.log('effect 함수');
   return () => {
     console.log('clean-up 함수');
```

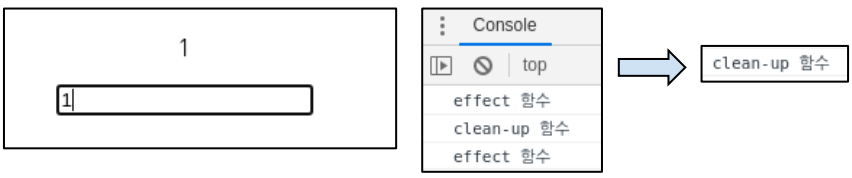
```
  };
}, [text]);

return (
  <div className="App">
    <p>{text}</p>
    <input
      onChange={(e) => {
        setText(e.target.value);
      }}
    />
  </div>
);
};

export default App;
```

결과 화면 (1)



컴포넌트가 마운트 된 후 한 번 effect 함수 실행 -> unmount되기 전 모든 clean-up 함수 실행

결과 화면 (2)



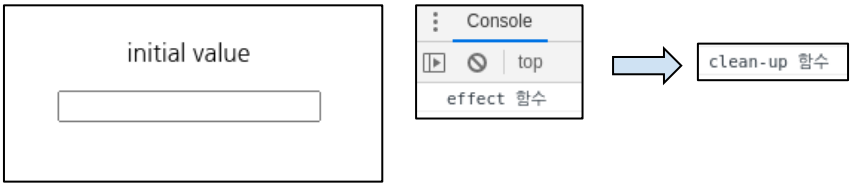
컴포넌트가 마운트 된 후 한 번 effect 함수 실행 -> 상태 값 변경 될 때 clean-up 함수 실행 후 effect 함수 실행 -> unmount되기 전 모든 clean-up 함수 실행

[예제 코드]

```
useEffect(() => {
```

```
  console.log('첫 렌더링에만 호출');
  return () => {
    console.log('마지막 언마운트 시 호출')
  }
}, [])
```

결과 화면 (1)



컴포넌트가 마운트 된 후 한 번 effect 함수 실행 -> unmount되기 전 모든 clean-up 함수 실행

4. useMemo

4.1 useMemo란?

메모이제이션된 **값을 반환**하는 Hook입니다.
useMemo는 함수의 연산량이 많을 때, 이전 값을 기억해두었다가 조건에 따라 재사용하여 성능을 최적화하는 용도로 사용됩니다.

* Memoization이란 기존에 수행한 연산의 결과값을 어딘가에 저장해두고 동일한 입력이 들어오면 재활용하는 프로그래밍 기법으로 적절히 사용하면 중복 연산을 피할 수 있기 때문에 메모리를 조금 더 쓰더라도 애플리케이션의 성능을 최적화할 수 있습니다.

* 컴포넌트가 렌더링될 때마다 내부에 선언되어 있던 표현식(변수, 함수 등)도 매번 다시 선언되어 사용됩니다.

4.2 useMemo 사용법

useMemo(**callBackFunc, dependencies**);

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

[예시 코드]

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

function MyComponent({ x, y }) {
  const z = useMemo(() => compute(x, y), [x, y]);
  return <div>{z}</div>;
}
```

x와 y값이 이 전에 렌더링했을 때와 동일할 경우, 이전 렌더링 때 저장해 두었던(메모이제이션) 결과값을 재활용합니다. 하지만, x와 y이 값이 이 전에 렌더링했을 때와 달라졌을 경우, () => compute(x, y)를 호출하여 결과값을 새롭게 구해 z에 할당해줍니다.

5. useCallback

5.1 useCallback이란?

메모이제이션된 **함수를 반환**하는 Hook입니다.
특정 함수를 새로 만들지 않고 재사용하고 싶을 때 사용합니다.
useCallback은 불필요한 렌더링을 방지하기 위하여 성능 최적화를 할 수 있습니다.

5.2 useCallback 사용법

useCallback(**callBackFunc, dependencies**);

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

useCallback(fn, deps)은 useMemo(() => fn, deps)와 같습니다.

[예시 코드]

```
import {useCallback} from 'react';
const [count, setCount] = useState({ num: 0 });

//수정 전
const increament = () => {
  setCount({ num: count.num + 1 });
};
//수정 후
const increament = useCallback(() => {
  setCount({ num: count.num + 1 });
},[count]);
```

의존성 배열에 넣어준 count 값이 변경될 때마다 콜백 함수를 새로 생성하게 됩니다.

메모이제이션을 할 때 어느 정도의 작업시간이 소요됩니다. 그렇기 때문에 재사용성이 거의 발생하지 않는 곳에 해당 기술을 적용하게 된다면 오히려 서비스 속도를 더 늦춰버리는 결과를 초래할 수 있습니다.

6. useRef

6.1 useRef란?

특정 DOM을 가리키거나, 컴포넌트 안에서 조회 및 수정할 수 있는 변수를 관리할 때 사용되는 Hook입니다.
포커스 설정, 특정 엘리먼트의 크기/색상 변경, 스크롤바 위치 가져오기 등에 사용됩니다.

useRef로 관리하는 **변수는 값이 바뀐다고 해서 컴포넌트가 리렌더링되지 않습니다.**
리액트 컴포넌트에서의 상태는 상태를 바꾸는 함수를 호출하고 나서 그 다음 렌더링 이후 업데이트 된 상태를 조회할 수 있는 반면,
useRef로 관리하고 있는 변수는 **설정 후 바로 조회**할 수 있습니다.

* ref 란?
ref는 JS의 getElementById()처럼, component의 어떤 부분을 선택할 수 있게 합니다.
리액트에 있는 모든 component는 reference element를 가지고 있어서,
어떤 component에 **ref = {변수명}** 을 넣어주면, 해당 component를 참조하게 됩니다.

6.2 useRef 사용법

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

const refContainer = useRef(initialValue);
```

useRef 함수는 current 속성을 가지고 있는 ref 객체를 반환하는데, 인자로 넘어온 초기값을 current 속성에 할당합니다. 이 current 속성은 값을 변경해도 상태를 변경할 때 처럼 React 컴포넌트가 다시 렌더링되지 않습니다. React 컴포넌트가 다시 랜더링될 때도 마찬가지로 이 current 속성의 값이 유실되지 않습니다.

[예시 코드]

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


// 버튼을 누르면 input 태그 focus 하기
const App = () => {
 const inputEl = useRef(null); // Ref 객체를 만들어준다.
 const onButtonClick = () => {
  // Ref 객체의 current 값은 선택하고자 하는 DOM을 가리킨다.
  // inputEl.current === input
  inputEl.current.focus(); // input 태그 포커스
 };
 return (
  <>
   {/* 선택하고 싶은 DOM에 속성으로 ref 값을 설정해준다. */}
   <input ref={inputEl} type="text" />
   <button onClick={onButtonClick}>Focus the input</button>
  </>
 );
};

export default App;
```

7. useImperativeHandle

7.1 useImperativeHandle란?

useImperativeHandle은 ref를 사용할 때 부모 컴포넌트에 노출되는 인스턴스 값을 사용자화(customizes)합니다.

즉, 부모 컴포넌트에서 자식 컴포넌트의 함수를 호출하여 사용하고 싶을 때 useImperativeHandle 을 사용합니다.

useImperativeHandle는 forwardRef 와 함께 사용합니다.

7.2 useImperativeHandle 사용법

```
useImperativeHandle(ref, createHandle, [deps])
```

[예시 코드]

```
// 자식 컴포넌트
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
```

```
FancyInput = forwardRef(FancyInput);
```

위의 예시에서 <FancyInput ref={inputRef} /> 를 렌더링한 부모 컴포넌트는 inputRef.current.focus()를 호출할 수 있습니다.

8. useLayoutEffect

8.1 useLayoutEffect란?

useLayoutEffect는 모든 DOM 변경 후에 동기적으로 발생, DOM에서 레이아웃을 읽고 동기적으로 리렌더링하는 경우에 사용합니다

useEffect 이용 시 렌더링이 될 때 화면이 깜빡이는 현상이 발생할 수 있는데 시각적으로 이를 방지하고 싶을 때 사용하면 됩니다.

우선 useEffect를 사용하고, 이슈가 있을 때 useLayoutEffect를 사용할 것을 권장합니다.

8.2 useLayoutEffect 사용법

```
useLayoutEffect(() => {
    effect
    return () => {
      cleanup
    };
  }, [input]);
```

이 함수의 시그니처는 useEffect와 동일한데 차이점이 있다면, "**실행 시점**"입니다.

useEffect : 화면이 그려진 이후 비동기적으로 실행.

컴포넌트 렌더링 -> 화면 업데이트 -> useEffect 실행

useLayoutEffect : 화면이 그려지기 전에 동기적으로 실행.

컴포넌트 렌더링 -> useLayoutEffect 실행 -> 화면 업데이트

9. 그 외 Hooks

9.1 useContext란?

useContext란 쉽게 말하면 , props를 글로벌하게 사용할 수 있게 도와줍니다.

9.2 useReducer란?

useState의 대체 함수로 컴포넌트 내에서 state 업데이트 로직 부분을 분리할 수 있고, 또 분리된 파일을 불러와서 사용할 수 있습니다.

9.3 useDebugValue란?

useDebugValue는 React 개발자도구에서 사용자 Hook 레이블을 표시하는 데에 사용할 수 있습니다.

10. Custom Hooks

10.1 Custom Hook이란?

함수형 컴포넌트에서 **로직을 재사용**하기 위해 만드는 Hook으로,
커스텀 훅은 코드의 중복을 줄이고 재사용성을 높여 효율적인 관리를 위해 필요합니다. 주로 Input을 관리하거나 Fetch 요청을 할 때 많이 사용됩니다.

커스텀 훅의 이름은 **use로 시작**해야하며, 다른 Hook을 호출할 수 있습니다.

* useSomething이라는 네이밍 컨벤션은 linter 플러그인이 Hook을 인식하고 버그를 찾을 수 있게 해줍니다.

또한 커스텀 훅은 기본 컴포넌트를 만들 때와 유사하지만 JSX가 아닌 **배열 혹은 오브젝트를 반환**합니다.

Array 로 리턴 시 return [value]; -> const [value] = useFetch('api/something');

Object 로 리턴 시 return {value}; -> const {value} = useFetch('api/something');

[예시 코드 1]

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

// 입력값 value, onChangeValue, setValue 만들어주는 커스텀 훅

export const useInput = (initialValue: any = null) => {
  const [value, setValue] = useState(initialValue);
  const handler = useCallback((e) => {
    setValue(e.target.value);
  }, []);

  return [value, handler, setValue];
};
```

[예시 코드 2]

```
import { useState, useEffect } from 'react';
import axios from 'axios';

const useFetch = (initialUrl: string) => { // 1. url을 넘겨 받습니다.
 const [url, setUrl] = useState(initialUrl);
 const [value, setValue] = useState('');
```

```
 useEffect(() => { // 2. useEffec가 실행됩니다.
   fetchData();
 }, [url]);

 const fetchData = () => axios.get(url).then(({ data }) => setValue(data)); // 3. 비동기로
value에 data를 set 해줍니다.

 return [value]; // 4. JSX가 아닌 Array or Object를 반환합니다.

};

export default useFetch;


---------------------------------

import useFetch from '../customs/useFetch.tsx';

const App = () => {
  const [value] = useFetch('/api/something');

  return <div>{value}</div>;

};

export default App;
```

11. Reference

1. https://ko.reactjs.org/docs/hooks-intro.html
2. https://react.vlpt.us/basic/12-variable-with-useRef.html
3. https://hsp0418.tistory.com/171
4. https://jforj.tistory.com/171?category=877028

5. https://ko-de-dev-green.tistory.com/71