

React - 5, 6, 8장

🕒 생성일	@2022년 5월 9일 오후 12:08
▼ 분류	React-리액트를 다루는 기술 책
☰ 주제	React
📅 날짜	

Ref

- 리액트 프로젝트 내부에서 DOM에 이름을 다는 방법
- reference의 줄임말 - 변수간의 참조 관계

React에서의 id값

- id값은 사이트 전체에서 공유됨
- 사용을 권장하진 않는다.

Ref사용

- DOM에 작업을 해야할때 ref사용 - DOM을 직접 건드려야 할때 사용

DOM을 꼭 사용해야하는 상황

가끔 state만으로 해결할수 없는 기능이 있을때 DOM에 직접적으로 접근해야 하기 때문에 ref를 사용

- 특정 input에 포커스 추가 해야할때
- 스크롤 박스 조작할때
- Canvas요소에 그림 그릴때 등

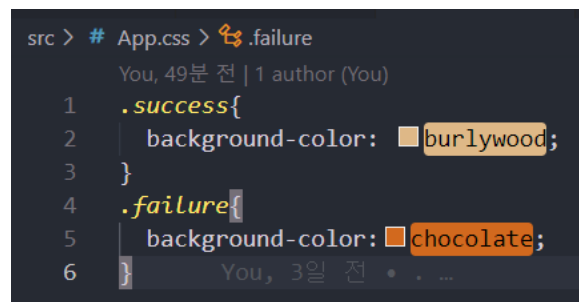
React.useRef() 사용

- 리액트에 내장되어있는 React.useRef() 함수를 사용
- 컴포넌트 내부에서 변수로 React.Ref()를 담아줘야함
- 해당 변수를 ref를 달고자 하는 요소에 ref props로 넣어주면 ref설정 완료
- 설정한뒤 나중에 ref를 설정해준 DOM에 접근하려면 ref참조변수.current를 조회하면됨
(뒷부분에 .current를 붙임)
 - ref참조변수.current는 일반 JS의 Html Element객체와 동일
 - = document.querySelector('...');
 - 바닐라 스크립트에서 배운 내용이 current뒤에 쭉 이어질수 있음
 - 사용 예시

```
import React from 'react';

const input= React.useRef();
const RefSample= ()=>{
  const handleFocus= ()=> {
    input.current.focus();
  }
  return(
    <div><input ref={input}/></div>
  );
};
export default RefSample;
```

Ref사용 전



```
src > # App.css > .failure
You, 49분 전 | 1 author (You)
1  .success{
2    background-color: burlywood;
3  }
4  .failure{
5    background-color: chocolate;
6  }
You, 3일 전 • . ...
```

```

import React from "react";
import "../App.css";

const Validation = () => {
  const [password, setPassword] = React.useState("");
  const [clicked, setClicked] = React.useState(false);
  const [validated, setValidated] = React.useState(false);

  const handleChange = (e) => {
    setPassword(e.target.value);
  };
  const handleClick = () => {
    setClicked(true);
    setValidated(password === "0000");
  };
  return (
    <div>
      <input
        type="password"
        value={password}
        onChange={handleChange}
        className={clicked ? (validated ? "success" : "failure") : ""}
      />
      <br/>
      <button onClick={handleClick}>검증하기!</button>
    </div>
  );
};

export default Validation;

```

Change함수: input태그 값 변경시 password가 변경됨

ButtonClick함수- 0000(o)⇒true, 0000(x)⇒false→ validated로 전달

⇒ 입력값이 0000이면 validated 수정함

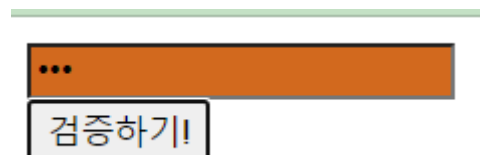
className: css를 결정하는 이중 삼항 연산자 사용

출력 결과

0000 입력 후 버튼 클릭



0000이 아닌 값 입력 후 버튼 클릭



Ref사용

```

import React from "react";
import "../App.css";

const Validation = () => {
  const [password, setPassword] = React.useState("");
  const [clicked, setClicked] = React.useState(false);
  const [validated, setValidated] = React.useState(false);
  const input = React.useRef();
  const handleChange = (e) => {
    setPassword(e.target.value);
  };
  const handleClick = () => {
    setClicked(true);
    setValidated(password === "0000");
    input.current.focus();
  };
  return (
    <div>
      <input
        type="password"
        value={password}
        onChange={handleChange}
        className={clicked ? (validated ? "success" : "failure") : ""}
        ref={input}
      />
      <br/>
      <button onClick={handleClick}>검증하기!</button>
    </div>
  );
};
export default Validation;

```

CSS파일과 결과는 동일함

컴포넌트에 ref달기

- 컴포넌트 내부에 있는 DOM을 컴포넌트 외부에서 사용할때 사용
- 사용법: <컴포넌트이름 ref={참조변수이름}/>
- 메서드와 멤버 변수는 클래스 컴포넌트에서만 성립, 함수형 컴포넌트에서는 성립 X

```

> JS App.js > App > <function>
You: 4시간 전 [1 author (You)]
1 import React from 'react';
2 import ScrollBox from './ScrollBox';
3
4 function App() {
5   const scrollBoxRef = React.useRef();
6   return (
7     <div>
8       <ScrollBox ref={scrollBoxRef}/>
9       <br/>
10      <button onClick={()=>{
11        const {scrollHeight, clientHeight} = scrollBoxRef.current;
12        scrollBoxRef.current.scrollTop = scrollHeight - clientHeight;
13      }}>맨 밑으로!</button>
14    </div>
15  );
16 }
17
18 export default App;
19

```

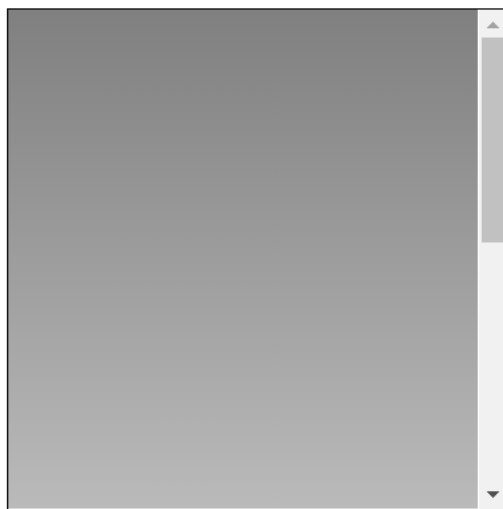
```

5 ScrollBox.js > ScrollBox > React.forwardRef() callback > innerStyle
const style={
  border: '1px solid black',
  height: '300px',
  width: '300px',
  overflow: 'auto',
  position: 'relative'
};
const innerStyle={
  width: '100%',
  height: '650px',
  background: 'linear-gradient(gray, white)'
}
return(
  <div style={style} ref={ref}>
    <div style={innerStyle}></div>
  </div>
);
export default ScrollBox;

```

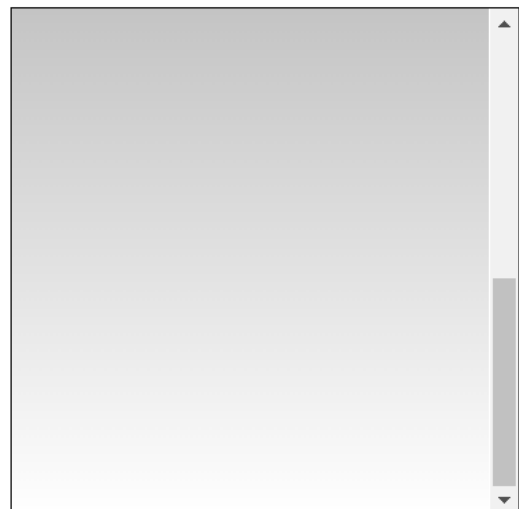
출력 결과

- 초기 화면



맨 밑으로!

- 버튼 클릭시



맨 밑으로!

- 컴포넌트 내부에서 DOM에 직접 접근해야할때 ref사용
- ref를 사용하지 않고도 원하는 기능을 구현할수 있는지 반드시 고려한 후 활용 할 것

- 서로 다른 컴포넌트끼리 데이터를 교류할때 ref를 사용한다면 잘못 사용된것
- 할수는 있지만 컴포넌트에 ref를 달고 그 ref를 다른 컴포넌트에 전달하고, 다른 컴포넌트에서 ref로 전달받은 컴포넌트의 메소드를 실행하는 방식은

리액트 사상에 어긋난 설계

⇒ 앱의 규모가 커진다면 구조가 꼬여 유지보수가 불가능할수 있음

- 컴포넌트 끼리 데이터를 교류할때는 언제나 데이터를 부모↔자식 흐름으로 교류해야함 - 리덕스를 사용해 효율적으로 교류하는 방법을 사용할 것

컴포넌트 반복

map() 함수

- JS배열 객체의 내장 함수map을 활용- 반복되는 컴포넌트를 렌더링
- 문법 - arr.map(callback, [thisArg])
 - 파라미터
 - callback: 새로운 배열의 요소를 생성하는 함수
-파라미터: currentValue- 현재 처리하고 있는 요소
index- 현재 처리하고 있는 요소의 index값
array- 현재 처리하고 있는 원본 배열
 - thisArg(선택항목): callback함수 내부에서 사용할 this 레퍼런스
- 기존의 배열(return값)로 새로운 배열을 만드는 역할

```
src > ex > JS Map.js > default
1 import React from 'react'
2
3 const Iteration= () => {
4   const names=['논사람', '얼음', '눈', '바람'];
5   const nameList=names.map(name=> <li>{name}</li>);
6   return <ul>{nameList}</ul>
7 };
8 export default Iteration;
```

```
> ex > JS Map.js > Iteration
You: 16초 전 | 1 author (You)
1 import React from 'react'
2
3 const Iteration= () => {
4   const [names, setNames]=React.useState(
5     [(id: 1, text: '논사람'),
6       (id: 2, text: '얼음'),
7       (id: 3, text: '눈'),
8       (id: 4, text: '바람')
9     ]);
10   const [inputText, setInputText]= React.useState('');
11
12   // 새로운 항목을 추가할때 id
13   const [nextId, setNextId]= React.useState(5);
14   const namesList= names.map(name => <li key={name.id}>{name.text}</li>);
15   return <ul>{namesList}</ul>
16 };
17 export default Iteration;
```

출력 결과
과 같음

- 눈사람
- 얼음
- 눈
- 바람

- `...` JSX코드로 된 배열을 새로 생성후 `nameList`에 담음
 - `map`함수에서 JSX작성할때는 DOM요소를 작성해도, 컴포넌트를 사용해도 가능
-

key

- 컴포넌트 배열을 렌더링했을때 어떤 원소에 변동이 있었는지 알아내려고 사용
 - 설정: `map`함수의 인자로 전달되는 함수내부에서 컴포넌트 props를 설정하듯 설정
 - key값은 언제나 유일- 데이터가 가진 고유값을 key값으로 설정
 - 게시판의 게시물을 렌더링한다면 게시물 번호를 key값으로 설정해야함
 - 중복된다면 렌더링 과정에서 오류 발생
 - 컴포넌트 배열을 렌더링 할때는 key값 설정에 항상 주의
-

데이터 추가 기능

```

x / JS Maps / Iteration
You, 9초 전 | 1 author (You)
import React from "react";

const Iteration = () => {
  const [names, setNames] = React.useState([
    { id: 1, text: "눈사람" },
    { id: 2, text: "얼음" },
    { id: 3, text: "눈" },
    { id: 4, text: "바람" },
  ]);
  const [inputText, setInputText] = React.useState("");

  // 새로운 항목을 추가할 때
  const [nextId, setNextId] = React.useState(5);
  const onChange = (e) => {
    setInputText(e.target.value);
  };
  const onClick = () => {
    const names = {
      id: number;
      text: string;
    }[]
    const nextNames = names.concat([
      {
        id: nextId, // nextId 값을 id로 설정
        text: inputText
      }
    ]);
    setNextId(nextId + 1); // nextId 값에 1을 더함
    setNames(nextNames); // names 값을 업데이트
    setInputText(''); // inputText를 비움
  };

  const namesList = names.map((name) => <li key={name.id}>{name.text}</li>);
  return (
    <>
      <input value={inputText} onChange={onChange} />
      <br />
      <button onClick={onClick}> 추가 </button>
      <ul>{namesList}</ul>
    </>
  );
};

export default Iteration;

```

출력 결과

추가

- 눈사람
- 얼음
- 눈
- 바람
- ○ ○

- 배열에 새 항목을 추가할때 배열의 push함수 대신 concat사용
 - names배열에 원소가 추가된 복사본을 nextName으로 반환
 - nextName= 사용자가 입력한 값
 - push함수는 기존 배열 자체를 변경
 - concat함수는 새로운 배열을 만들어줌
 - 불변성 유지: 상태를 업데이트 할때는 기존 상태를 그대로 두면서 새로운 값을 상태로 설정
-

데이터 제거 기능 추가

```

import React from "react";

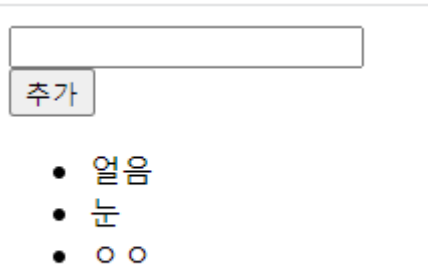
const Iteration = () => {
  const [names, setNames] = React.useState([
    { id: 1, text: "눈사람" },
    { id: 2, text: "얼음" },
    { id: 3, text: "눈" },
    { id: 4, text: "바람" },
  ]);
  const [inputText, setInputText] = React.useState("");

  // 새로운 항목을 추가할때 id
  const [nextId, setNextId] = React.useState(5);
  const onChange = (e) => setInputText(e.target.value);
  const onClick = () => {
    const nextNames = names.concat({
      id: nextId, //nextId값을 id로 설정
      text: inputText,
    });
    setNextId(nextId + 1); //nextId값에 1을 더함
    setNames(nextNames); //names 값을 업데이트
    setInputText(""); //inputText를 비움
  };
  const onRemove = (id) => {
    const nextNames = names.filter((name) => name.id !== id);
    setNames(nextNames);
  };
  const namesList = names.map((name) => (
    <li key={name.id} onDoubleClick={() => onRemove(name.id)}>
      {name.text}
    </li>
  ));
  return (
    <>
      <input value={inputText} onChange={onChange} />
      <br />
      <button onClick={onClick}> 추가 </button>
      <ul>{namesList}</ul>
    </>
  );
};

export default Iteration;

```

출력 결과



- 상태 안에서 배열을 변경할때는 배열에 직접 접근하여 수정하는것이 아니라 concat, filter 등 의 배열 내장 함수를 사용해 새로운 배열을 만든후 이를 새로운 상태로 설정해야함(참조 복사X)

```
let k= [...] or {...}  
let x= k; //->참조
```

Hooks

함수 컴포넌트에서도 상태 관리를 할수있는 useState와 렌더링 직후 작업을 설정하는 useEffect 등 기능을 제공

useState

- 가장 기본적인 Hook
- 함수 컴포넌트에서도 가변적인 상태를 지닐수있게 해줌
- useState함수의 파라미터에는 상태의 기본값을 넣어줌
- 배열을 반환- 첫번째 원소: 상태값
 - 두번째 원소: 상태를 설정하는 함수
- 호출하면 전달받은 파라미터로 값이 바뀌고 컴포넌트가 정상적으로 리렌더링됨

```

JS App.js > ...
You, 34초 전 | 1 author (You)
import React from 'react';
import { useState } from './ex/useState';

function App() {
  return (
    <div>
      <UseState />
    </div>
  );
}
export default App;

```

```

ex > JS UseState.js > default
import React from 'react'

const useState=()=> {
  const [value, setValue] = React.useState(0);

  return (
    <div>
      <p>현재 카운터 값은!:: <b>{value}</b></p>
      <br />
      <button onClick={() => setValue(value+ 1)}> +1 </button>
      <button onClick={() => setValue(value- 1)}> -1 </button>
    </div>
  )
}

export default useState;

```

출력 결과

현재 카운터 값은!:: 4

+1 -1

- setValue함수를 통해 상태값이 갱신되면 자동으로 화면에 실시간 반영됨

useState여러번 사용하기

- 하나의 useState함수는 하나의 상태값만 관리할수 있음
- 컴포넌트에서 관리해야할 상태가 여러개라면 useState를 여러번 사용

```
import React from 'react'

const useState=()=> {
  const [name, setname]= React.useState('');
  const [nickname, setNickname]= React.useState('');
  const onChangeName= e =>{
    setname(e.target.value);
  };
  const onChangeNickname= e =>{
    setNickname(e.target.value);
  }

  return (
    <div>
      <div>
        <input value={name} onChange={onChangeName}/>
        <input value={nickname} onChange={onChangeNickname}/>
      </div>
      <br />
      <div>
        <b>이름:: </b><span>{name}</span>
      </div>
      <div>
        <b>닉네임:: </b><span>{nickname}</span>
      </div>
    </div>
  )
}

export default useState;
```

출력 결과

박세영 냐꾸

이름:: 박세영
닉네임:: 냐꾸

useEffect

리액트 컴포넌트가 렌더링 될때마다 특정 작업을 수행하도록 설정

- 렌더링: 화면에 그림

```
import React from "react";

const UseEffect = () => {
  const [name, setName] = React.useState("");
  const [nickname, setNickname] = React.useState("");
  React.useEffect(() => {
    console.log("렌더링이 완료되었습니다");
    console.log({ name, nickname });
  });
  const onChangeName = (e) => {
    setName(e.target.value);
  };
  const onChangeNickname = (e) => {
    setNickname(e.target.value);
  };
  return (
    <div>
      <div>
        <input value={name} onChange={onChangeName} />
        <input value={nickname} onChange={onChangeNickname} />
      </div>
      <br />
      <div>
        <b>이름:: </b>
        <span>{name}</span>
      </div>
      <div>
        <b>닉네임:: </b>
        <span>{nickname}</span>
      </div>
    </div>
  );
};

export default UseEffect;
```

출력 결과

이름:: 박

닉네임::

요소

콘솔

소스

▶

🛑

top ▼

👁

필터

Download the React DevTools for [act-devtools](#)

렌더링이 완료되었습니다

▶ {name: '', nickname: ''}

렌더링이 완료되었습니다

▶ {name: 'ㅂ', nickname: ''}

렌더링이 완료되었습니다

▶ {name: '박', nickname: ''}

렌더링이 완료되었습니다

▶ {name: '박', nickname: ''}

>

- 첫번째 로그: 컴포넌트가 화면에 최초 등장시

- 두번째 이후 로그: 상태값이 변경되었을시

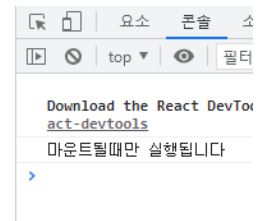
마운트될때만 실행하고 싶을때

useEffect에서 설정한 함수를 컴포넌트가 화면에 맨 처음 렌더링 될때만 실행하고, 업데이트될때는 실행하지 않으려면 두번째 파라미터에 비어있는 배열을 사용

```
React.useEffect(() => {  
  console.log("마운트될때만 실행됩니다");  
}, []);
```

출력 결과

이름:: ㄴ ㄴ
닉네임::



특정값이 업데이트 될때만 실행하고 싶을때

- 두번째 파라미터로 전달되는 배열안에 검사하고 싶은 값을 넣음
- useState를 통해 관리하고있는 상태값이나 props로 전달받은 값도 가능

```
React.useEffect(() => {  
  console.log(name);  
}, [name]);
```

출력 결과

이름:: ㄴ ㄴ
닉네임::



뒷정리

컴포넌트가 언마운트되기 전이나 업데이트 직전에 어떠한 작업을 수행하고싶다면 useEffect에서 **cleanup(뒷정리)** 함수를 반환해야함

```

JS Appjs > App
You, 17초 전 | 1 author (You)
import React from 'react';
import UseEffect from './ex/UseEffect';

function App() {
  const [visible,setVisible]= React.useState(false);
  return (
    <div>
      <button onClick={()=> {setVisible(!visible);}}>
        {visible ? '숨기기' : '보이기'}
      </button>
      <br />
      <hr />
      {visible && <UseEffect/>}
    </div>
  );
}

export default App;

```

```

import React from "react";

const UseEffect = () => {
  const [name, setName] = React.useState("");
  const [nickname, setNickname] = React.useState("");
  React.useEffect(() => {
    // 화면이 로딩될때 실행
    console.log('effect');
    console.log(name);

    // 화면의 로딩이 종료될때 실행
    return () => {
      console.log('cleanup'+ name)
    };
  },[name]);
  const onChangeName = (e) => {
    setName(e.target.value);
  };
  const onChangeNickname = (e) => {
    setNickname(e.target.value);
  };
  return (
    <div>
      <div>
        <input value={name} onChange={onChangeName} />
        <input value={nickname} onChange={onChangeNickname} />
      </div>
      <br />
      <div>
        <b>이름:: </b>
        <span>{name}</span>
      </div>
      <div>
        <b>닉네임:: </b>
        <span>{nickname}</span>
      </div>
    </div>
  );
};

export default UseEffect;

```

출력 결과

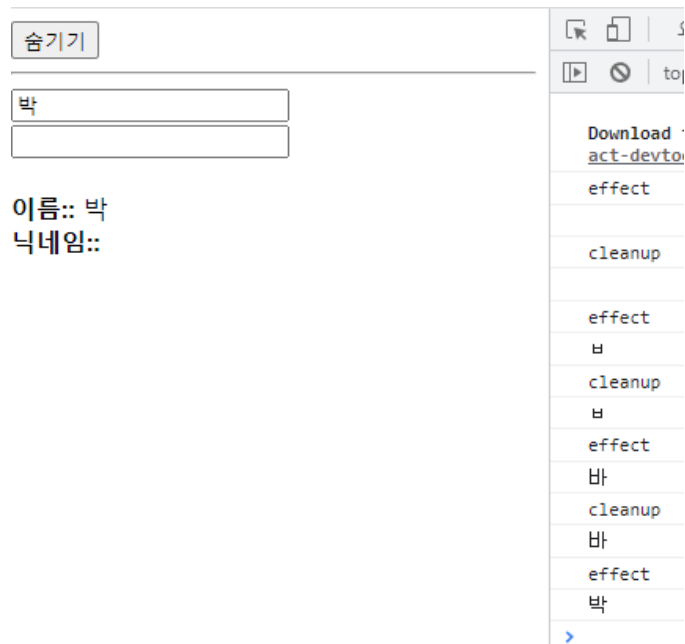
초기 화면



보이기 클릭시



값 입력시



값 입력후 숨기기
클릭시



- `!visible : false → true, true → false`
- `{visible && <UseEffect/>}: true`일때만 표시
 - css의 `display`속성을 제어하는것이 아닌 실제 DOM자체가 사라졌다 생성되기를 반복함

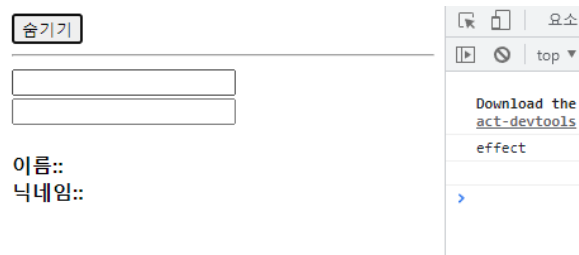
언마운트 될때만 뒷정리 함수 호출하기

-useEffect함수의 두번째 파라미터에 비어있는 배열 넣기

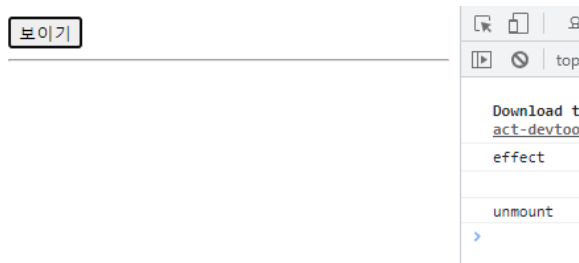
```
React.useEffect(() => {  
  // 화면이 로딩될때 실행  
  console.log('effect');  
  console.log(name);  
  
  // 화면의 로딩이 종료될때 실행  
  return () => {  
    console.log('unmount');  
  };  
}, []);
```

출력 결과

보이기 클릭
시



숨기기 클릭
시



useReducer

- useState보다 더 다양한 컴포넌트 상황에 따라 다양한 상태를 다른값으로 업데이트할때 사용하는 Hook
- 현재 상태와 업데이트를 위해 필요한 정보를 담은 action값을 전달받아 새로운 상태를 반환하는 함수
- action값은 그 어떤 값으로도 사용 가능

- 새로운 상태를 만들때는 반드시 불변성을 지켜야함
 - 불변성: 상태값에 대한 복사본을 생성한 후 복사본을 활용해 상태값을 갱신해야함
- 가장 큰 장점은 컴포넌트 업데이트 로직을 컴포넌트 바깥으로 빼낼수 있다는것

```

ex > JS UseReducer.js > UseReducer
import React from "react";

function render(state, action) {
  //action.type에 따라 다른 작업 수행
  switch (action.type) {
    case "INCREMENT":
      return { value: state.value + 1 };
    case "DECREMENT":
      return { value: state.value - 1 };
    default:
      // 아무것도 해당되지 않을때
      return state;
  }
}

const UseReducer = () => {
  const [state, dispatch] = React.useReducer(render, { value: 0 });
  return <div>
    <p>현재 카운터 값은 <b>{state.value}</b> 입니다</p>
    <br/>
    <button onClick={() => dispatch({type: 'INCREMENT'})}> + 1 </button>
    <br />
    <button onClick={() => dispatch({type: 'DECREMENT'})}> - 1 </button>
  </div>;
};

export default UseReducer;

```

출력 결과

현재 카운터 값은 3 입니다



- ~.type과 ~.value로 ~의 값은 json이라고 유추 가능
 - action.type⇒ action= {type: 'INCREMENT'(or 'DECREMENT')}
 - state.value⇒ state= {value: 0}
- useReducer의 파라미터
 - 첫번째 파라미터: reducer함수

- 두번째 파라미터: 해당 reducer함수의 기본값
- state: 현재 가리키고있는 상태
- dispatch: action을 발생시키는 함수
 - dispatch(action)과 같은 형태로 함수 안에 파라미터로 action값을 넣어주면 reducer함수가 호출됨

```

import React from "react";

function render(state, action) {
  return {
    ...state,
    [action.name]: action.value //입력값 그자체- <input>
  };
}

const UseReducer = () => {
  const [state, dispatch] = React.useReducer(render, {
    name: '',
    nickname: ''
  });
  const {name, nickname} = state;
  const onChange= e=>{dispatch(e.target)};
  return <div>
    <div>
      <input name="name" value={name} onChange={onChange}/>
      <input name="nickname" value={nickname} onChange={onChange}/>
    </div>
    <br />
    <div>
      <b>이름:: </b><span>{name}</span>
    </div>
    <div>
      <b>닉네임:: </b><span>{nickname}</span>
    </div>
  </div>;
};

export default UseReducer;

```

출력 결과

이름::

닉네임::

- ...state: 기존 상태값 복사- 비구조 문법

- [action.name]: json의 key를 동적으로 명시

```
const key= 'b';  
const data={[key]: 100};  
//=>{b: 100}
```

- const {name, nickname} = state; : 한번더 비구조 문법으로 분리
- action: <input name= 'name' or 'nickname'> 이벤트가 발생한 하나
- dispatch(e.target): e.target= <input>태그
- <input>태그 중에서 이벤트가 발생했을경우 동적 제어에 의해 name과 nickname의 입력값이 바뀜

-05.09-

useMemo

함수 컴포넌트 내부에서 발생하는 연산을 최적화 할수있음

```

import React from 'react'

const getAverage= numbers=>{
  console.log('평균값 계산중..');
  if(numbers.length === 0) return 0;
  const sum= numbers.reduce((a, b)=> a+ b);
  return sum/ numbers.length;
};

const UseMemo = () => {
  const [list, setList] = React.useState([]);
  const [number, setNumber] = React.useState('');
  const onChange= e =>{
    setNumber(e.target.value);
  };
  const onInsert= e =>{
    //기존의 list배열에 number의 값을 추가한 새로운 복사본 생성
    const nextList= list.concat(parseInt(number));

    //새롭게 생성된 복사본을 list에 적용
    setList(nextList);

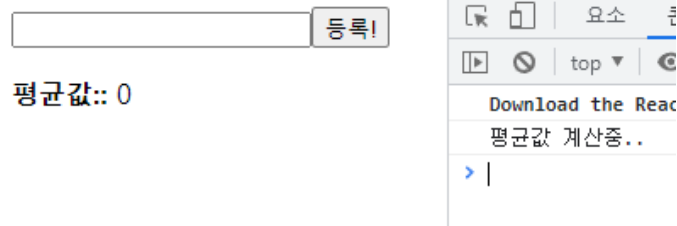
    //적용후 내용 초기화
    setNumber('');
  };
  return (
    <div>
      <input value={number} onChange={onChange}/>
      <button onClick={onInsert}> 등록! </button>
      <ul>
        {list.map((v, i)=>(
          <li key={i}>{v}</li>
        ))}
      </ul>
      <div><b>평균값:: </b> {getAverage(list)}</div>
    </div>
  );
};

export default UseMemo

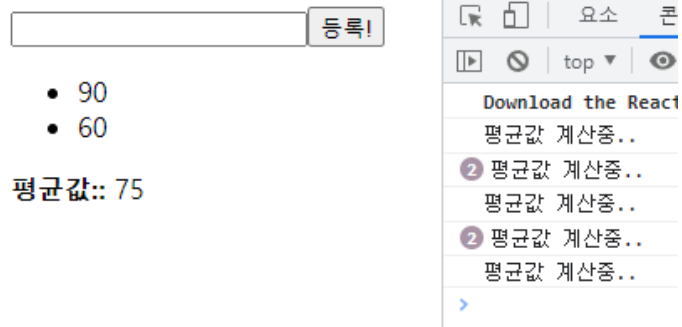
```

출력 결과

초기 화면



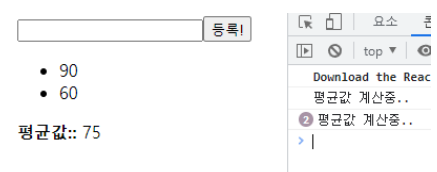
90과 60을 버튼으로 추가함



- `numbers.length`: 배열이라는것을 유추할수 있음
- 버튼 클릭→ 배열의 원소 추가→ 평균값 다시 계산
 - 입력값이 변경되는것만으로는 평균을 다시 계산할 필요없음
- `getAverage(list)`
 - 버튼을 클릭하여 상태값 `list`의 내용이 추가되면 변경된 배열을 사용해 `getAverage()`함수의 리턴값을 출력
 - `list`배열의 변경에 따라 실시간 처리됨
- 숫자를 등록할때 뿐만아니라 `input`의 내용이 수정될때도 `getAverage()`함수가 호출되는것을 확인할수 있음⇒ 렌더링할때마다 계산하는것은 낭비

렌더링을 하는 과정에서 **특정값이 바뀌었을때만 실행**하고, 원하는 값이 바뀌지 않았다면 이전에 연산했던 결과를 다시 사용하는 방식 ⇒ **useMemo Hook**

출력 결과



```

const avg = React.useMemo(() => getAverage(list), [list]);
return (
  <div>
    <input value={number} onChange={onChange}/>
    <button onClick={onInsert}> 등록! </button>
    <ul>
      {list.map((v, i) => (
        <li key={i}>{v}</li>
      ))}
    </ul>
    <div><b>평균값:: </b> {avg}</div>
  </div>
);
};

```

- `useMemo()`: `getAverage()`의 return값을 반환
- `avg`: `getAverage()`값이 저장= `list`의 평균
- `..., [list]`: 상태값인 `list`가 변경되었을때만 `getAverage()`호출함

useCallback

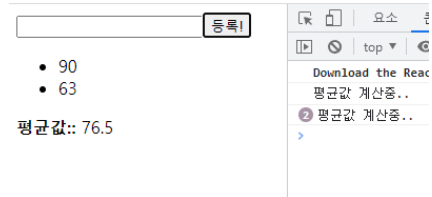
- 렌더링 성능을 최적화해야하는 상황에 사용
- 컴포넌트가 리렌더링될때마다 새로 만들어진 함수를 사용함
- 컴포넌트의 렌더링이 자주 발생하거나 렌더링해야할 컴포넌트의 개수가 많아지면 최적화해주는것이 좋음
- 파라미터
 - 첫번째 파라미터: 생성하고 싶은 함수
 - 두번째 파라미터: 배열 - 어떤값이 바뀌었을때 함수를 새로 생성해야하는지 명시
 - 빈배열: 컴포넌트가 렌더링될때 만들었던 함수를 계속해서 재사용
 - 상태값: 값이 바뀌거나 새로운 항목이 추가될때 새로 만들어진 함수를 사용함
 - 함수내부에서 상태값에 의존해야할때는 그값을 반드시 두번째 파라미터안에 포함시켜야함

출력 결과


```
import React from 'react'

const getAverage= numbers=>{
  console.log('평균값 계산중..');
  if(numbers.length === 0) return 0;
  const sum= numbers.reduce((a, b)=> a + b);
  return sum / numbers.length;
};

const UseMemo = () => {
  const [list, setList] = React.useState([]);
  const [number, setNumber] = React.useState('');
  const onChange= React.useCallback(e=> {
    setNumber(e.target.value);
  },[]); // 컴포넌트가 처음 렌더링될때만 함수 생성
  const onInsert= React.useCallback(() =>{
    const nextList= list.concat(parseInt(number));
    setList(nextList);
    setNumber('');
  },[number,list]); //number or list가 바뀌었을때만 함수 생성
  const avg= React.useMemo(()=> getAverage(list), [list]);
  return (
    <div>
      <input value={number} onChange={onChange}/>
      <button onClick={onInsert}> 등록! </button>
      <ul>
        {list.map((v, i)=>[
          <li key={i}>{v}</li>
        ])}
      </ul>
      <div><b>평균값:: </b> {avg}</div>
    </div>
  );
};
```



- onInset는 기존의 number와 list를 조회하여 nextList를 생성하기 때문에 배열안에 number와 list를 꼭 넣어줘야함

useMemo:: 숫자, 문자열, 객체처럼 일반값을 재사용할때 사용

useCallback:: 함수를 재사용할때 사용

useRef

등록버튼 클릭시 포커스가 input태그로 넘어가게 작성함

```
import React from 'react'

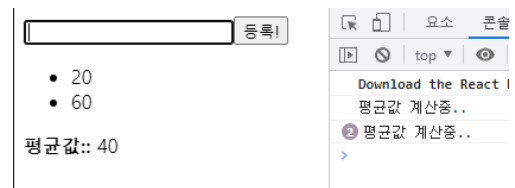
const getAverage= numbers=>{
  console.log('평균값 계산중..');
  if(numbers.length === 0) return 0;
  const sum= numbers.reduce((a, b)=> a + b);
  return sum / numbers.length;
};

const UseMemo = () => {
  const [list, setList] = React.useState([]);
  const [number, setNumber] = React.useState('');
  const inputEl= React.useRef(null);
  const onChange= React.useCallback(e=> {
    setNumber(e.target.value);
  },[]); // 컴포넌트가 처음 렌더링될때만 함수 생성
  const onInsert= React.useCallback(() =>{
    const nextList= list.concat(parseInt(number));
    setList(nextList);
    setNumber('');
    inputEl.current.focus();
  },[number,list]); //number or list가 바뀌었을때만 함수 생성
  const avg= React.useMemo(()=> getAverage(list), [list]);
  return (
    <div>
      <input value={number} onChange={onChange} ref={inputEl}/>
      <button onClick={onInsert}> 등록! </button>
      <ul>
        {list.map((v, i)=>{
          <li key={i}>{v}</li>
        })}
      </ul>
      <div><b>평균값:: </b> {avg}</div>
    </div>
  );
};

export default UseMemo;
```

출력 결과

- useRef를 사용해 ref를 설정하려면
useRef를 통해 만든 객체안의
current값이 실제 element를 가리킴



커스텀 Hooks

여러 컴포넌트에서 비슷한 기능을 공유할 경우 커스텀Hooks를 작성해 사용할수있음

Hook정

의

```
import React from 'react'

function reducer(state, action) {
  return {
    ...state,
    [action.name]: action.value
  };
}

export default function useInputs(initialForm) {
  const [state, dispatch] = React.useReducer(reducer, initialForm);
  const onChange = e => {dispatch(e.target)};
  return [state, onChange];
}
```

- reducer의 return{..}⇒const [state,...]
- dispatch(e.target)⇒useReducer(reducer,..)
⇒reducer(..,action)

```
const Info = () => {
  const [state, onChange] = useInputs({
    name: '',
    nickname: ''
  });
  const {name, nickname} = state;
  return (
    <div>
      <div>
        <input name="name" value={name} onChange={onChange}/>
        <input name="nickname" value={nickname} onChange={onChange}/>
      </div>
      <br />
      <div>
        <b>이름:: </b><span>{name}</span>
      </div>
      <div>
        <b>닉네임:: </b><span>{nickname}</span>
      </div>
    </div>
  )
}

export default Info
```

- useInputs({name:'',nickname:''})⇒ initialForm
- <input>의 내용 변경시 실행→ onChange
→ function reducer(state,action)...을 통해 값을 갱신

-05.10-