3rd React

useState

useEffect

- 컴포넌트가 리렌더링 될 때마다 (렌더링 직후) 특정 작업을 수행하도록 설정
- 컴포넌트가 언마운트 되기 전이나 업데이트 되기 직전에 어떠한 작업을 수행하고 싶다면 뒷정리 함수를 return 해주면 된다. 오직 언마운트 될 때만 뒷정리 함수를 호출하고 싶다면 useEffect함수의 두번째 파라미터에 비어있는 배열을 넣으면 된다.
- 마운트 될 때만 실행하고 싶으면 함수의 두 번째 파라미터로 비어있는 배열을 넣어주면 됨.
- 특정 값이 업데이트 될 때만 실행하고 싶으면, useEffect 의 두번째 파라미터로 전달되는 배열 안에 검사하고 싶은 값을 넣어주면 됨.

useReducer

컴포넌트 상황에 따라 다양한 상태를 다른 값으로 업데이트해 주고 싶을 때 사용하는 Hook

리듀서는 현재 상태, 그리고 업데이트를 위해 필요한 정보를 담은 액션값을 전달받아 새로운 상태를 반환하는 함수다. 리듀서 함수에서 새로운 상태를 만들 때는 반드시 불변성을 지켜줘야 한다.

```
function reducer(state, action) {
  return { ... }; // 업데이트한 새로운 상태를 반환
}
```

▼ 예시1 - Counter

useReducer 의 첫번째 파라미터에는 reducer 함수, 두번째 파라미터에는 해당 리듀서의 기본 값을 넣어준다.

```
function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { value: state.value + 1};
    case 'DECREMENT':
      return { value: state.value - 1};
    default:
      return state;
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, { value: 0 });

  return (
    <div>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>+1</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>-1</button>
    </div>
  )
}

export default Counter;
```

▼ 예시 2 - 인풋 상태 관리하기

e.target 값 자체를 action 값으로 사용

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

const Info = () => {
  const [state, dispatch] = useReducer(reducer, {
    name: '',
    nickname: ''
  });
  const { name, nickname } = state;
  const onChange = e => {
    dispatch(e.target);
  };

  return (
    <div>
      <input name="name" value={name} onChange={onChange} />
      <input name="nickname" value={nickname} onChange={onChange} />
```

```
    </div>
    <div>
      <div>
        <b>이름:</b> {name}
      </div>
      <div>
        <b>닉네임:</b> {nickname}
      </div>
    </div>
  );
};

export default Info;
```

useMemo

함수형 컴포넌트 내부에서 발생하는 연산 최적화 가능

숫자, 문자열, 객체처럼 일반값을 재사용 할 때 사용

useCallback

렌더링 성능을 최적화해야 하는 상황에서 사용.

이벤트 핸들러 함수를 필요할 때만 생성이 가능하다.

useRef

함수형 컴포넌트에서 ref를 쉽게 사용할 수 있도록 해준다

컴포넌트 로컬 변수를 사용해야 할 때도 useRef를 활용할 수 있다. (ref 안의 값이 바뀐다고 컴포넌트가 렌더링 되지 않는다)

커스텀 Hooks

여러 컴포넌트에서 비슷한 기능을 공유할 경우, 커스텀 Hook으로 작성하여 재사용이 가능하다

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

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

// Info.js
const Info = () => {
  const [state, onChange] = useInputs({
    name: '',
    nickname: ''
  });
  const { name, nickname } = state;

  return (
    <div>
      <input name="name" value={name} onChange={onChange} />
      <input name="nickname" value={nickname} onChange={onChange} />
    </div>
    <div>
      <div>
        <b>이름:</b> {name}
      </div>
      <div>
        <b>닉네임:</b> {nickname}
      </div>
    </div>
  );
};

export default Info;
```

컴포넌트 스타일링

▼ 일반 CSS

▼ Sass

CSS 전처리기중 하나로 확장된 CSS문법을 사용하여 CSS 코드를 더욱 쉽게 작성할 수 있도록 해준다.

.sass, .scss 확장자 모두 지원. 단, .sass 확장자는 중괄호와 세미콜론을 사용하지 않음

주로 .scss를 더 많이 사용한다.

`yarn add node-sass로` node-sass 라이브러리 설치 (sass 를 css로 변환해줌)

```
// 변수
$red: #fa5252;
$orange: #fd7e14;

// 믹스인 만들기(재사용되는 스타일 블록을 함수처럼 사용)
@mixin square($size) {
  $caculated: 32px * $size;
  width: $calculated;
  height: $calculated;
}

.SassComponent {
  display: flex;
  .box { // 일반 css에서의 .SassComponet .box 와 같은 의미
    background: red;
    cursor: pointer;
    transition: all 0 0.3s ease-in;
    &.red {
      // .red 클래스가 .box 와 함께 사용되었을 때
      background: $red;
      @include square(1);
    }
    &.orange {
      background: $orange;
      @include square(2);
    }
  }
}
```

```
// SassComponent.js
import './SassComponent.scss';

const SassComponent = () => {
  return (
```

```
    <div className="SassComponent">
      <div className="box red"/>
      <div className="box orange"/>
    </div>
  )
}
```

여러 파일에 사용될 수 있는 Sass 변수와 믹스인은 주로 다른 파일(utils.scss)로 분리하여 사용한다. 또한 sass-loader 설정을 커스터마이징 해두면 편하다

▼ CSS Module

스타일 작성 시 CSS 클래스가 다른 CSS 클래스의 이름과 절대 충돌하지 않도록 파일마다 고유한 이름(**[파일이름]_[클래스이름]__[해시값]** 형태)을 자동으로 생성해 중첩 현상을 방지해주는 옵션

.module.css 확장아로 파일을 저장하면 자동으로 적용됨.

▼ styled-components

컴포넌트 성능 최적화

▼ 리렌더링이 일어나는 상황

- 자신이 전달받은 props가 변경되었을 때
- 자신의 state 가 바뀌었을때
- 부모 컴포넌트가 리렌더링 될 때
- forceUpdate 함수가 실행되었을 때

→ **불필요한 리렌더링을 최소화** 해야한다... **HOW?**

▼ 컴포넌트의 props가 바뀌지 않았다면 리렌더링 되지 않도록 설정

1. 클래스형 컴포넌트일 경우 shouldComponentUpdate 함수 이용
2. 함수형 컴포넌트일 경우 React.memo 함수를 이용한다.

```
export default React.memo(TodoListItem);
```

▼ 함수가 계속 만들어지는 상황을 방지하기

1. useState 의 함수형 업데이트 기능 사용

   setNumber(number + 1) 이라고 하는 대신, `number ⇒` 를 붙여준다

```
const onIncrease = useCallback(
  () ⇒ setNumber(prevNumber ⇒ prevNumber + 1),
  [],
)
```

2. useReducer 사용

   두번째 파라미터에 undefined 를 넣고, 세번째 파라미터에 초기 상태를 만들어 주는 함수인 createBulkTodos 를 넣어주면, 컴포넌트가 맨 처음 렌더링 될 때에만 createBulkTodos 함수가 호출된다.

→ 불변의 중요성

불변성을 지킨다 === 기존의 값을 직접 수정하지 않으면서 새로운 값을 만들어 내는 것

```
const onToggle = useCallback(id ⇒ {
  setTodos(todos =>
    todos.map(todo =>
      todo.id === id ? { ...todo, checked: !todo.checked } : todo,
    ),
  );
}, []);
```

불변성이 지켜지지 않으면 객체 내부의 값이 새로워져도 바뀐 것을 감지하지 못한다.
React.memo 로 서로 비교하며 최적화 하는것도 불가능하짐.

+) 전개연산자를 사용하면, 가장 바깥쪽 값만 복사되므로, 객체나 배열이라면 내부의 값은 따로 복사해주어야 한다. → 즉, 배열 혹은 객체의 구조가 복잡해지면 불변성을 유지하기가 매우 까다로워진다. → immer 라이브러리를 사용하자

→ react-virtualized 를 사용한 렌더링 최적화

스크롤 되기 전 보이지 않는 컴포넌트는 렌더링 하지 않고 크기만 차지하게끔. 그리고 만약 스크롤되면 해당 스크롤 위치에서 보여주어야 할 컴포넌트를 렌더링.

(코드는 책 참고)