**Immer를 사용한 더 쉬운 불변성 관리**

1. 리액트에서 배열이나 객체를 업데이트 해야 할 때에는 직접 수정하면 안되고 불변성을 지켜주면서 업데이트해줘야한다. 따라서 객체를 수정할 때는

const object = {

a: 1, b:2

};

object.b = 3;

로 해주면 안되고

const nextObject = {

…object,

b:3

};

처럼 새로운 객체를 만들어줘야한다.

1. 배열도 마찬가지로 push, splice 등의 함수를 사용하거나 n번째 항목을 직접 수정하면 안되고, 다음과 같이 concat, filter, map 등의 함수를 사용해야 합니다.

const todos = [

{

id: 1,

text: ‘할 일 #1’,

done: true

},

{

id: 2,

text: ‘할 일 #2’,

done: false

}

]

const inserted = todos.concat({

id: 3,

text: ‘할 일 #3’,

done: false

});

const filtered = todos.filter(todo => todo.id !== 2);

const toggled = todos.map(

todo => todo.id === 2

? {

…todo,

done: !todo.done,

}

: todo

};

1. 대부분의 경우 … 연산자와 배열 내장 함수를 사용하는 건 그렇게 어렵지는 않지만, 데이터의 구조가 까다로워지면 힘들어진다.
2. 이때 immer를 사용해준다. immer를 사용하면 우리가 상태를 업데이트 할 때, 불변성을 신경쓰지 않으면서 업데이트를 해주면 immer가 불변성 관리를 대신 해준다.

**Immer 사용법**

1. 먼저 사용하기 전에 immer를 설치해야 한다.

yarn add immer

1. immer를 import한다. 보통 produce란 이름으로 불러옴

import produce from ‘immer’;

1. produce 함수를 사용할 때에는 첫번째 파라미터에는 수정하고 싶은 상태, 두번째 파라미터에는 어떻게 업데이트하고 싶을지 정의하는 함수는 넣어주면 됨.
2. 이때 두번째에는 그냥 꼴리는대로 업데이트해주면 알아서 해준다.

const state = {

number : 1,

dontChangeMe: 2

};

const nextState = produce(state, draft => {

draft.number += 1;

});

console.log(nextState);

**reducer에서 immer 사용하기**

1. 이번 예제에선 users 배열이 깊은 곳에 위치하지 않아서 오히려 immer를 사용하면 코드가 더 길어지만, 일단 연습해보자.

function reducer(state, action) {

switch (action.type) {

case 'CREATE\_USER':

return produce(state, draft => {

draft.users.push(action.user);

});

case 'TOGGLE\_USER':

return produce(state, draft => {

const user = draft.users.find(user => user.id === action.id);

user.active = !user.active;

});

case 'REMOVE\_USER':

return produce(state, draft => {

const index = draft.users.findIndex(user => user.id === action.id);

draft.users.splice(index, 1);

});

default:

return state;

}

//이전 reducer 함수

function reducer(state, action) {

  switch (action.type) {

    case 'CREATE\_USER':

      return {

        users: state.users.concat(action.user)

      };

    case 'TOGGLE\_USER':

      return {

        ...state,   //inputs는 그대로 가져가기 위함

        users: state.users.map(user =>

          user.id === action.id ? { ...user, active: !user.active} : user

        )

      };

    case 'REMOVE\_USER':

      return {

        ...state,

        users: state.users.filter(user => user.id !== action.id)

      };

    default:

      return state;

  }

}

1. 더 복잡해진 게 있고, 더 짧아진 게 있다. 잘 골라서 사용하자.

**immer과 함수형 업데이트**

1. 이전 시간에 함수형 업데이트를 통해, useCallback 사용시 두번째 파라미터에 deps 배열에 todo(다음 예제일 때)를 안넣어도 된다고 배움.

const [todo, setTodo] = useState({

text: 'Hello',

done: false

});

const onClick = useCallback(() => {

setTodo(todo => ({

...todo,

done: !todo.done

}));

}, []);

이때 함수형 업데이트를 하는 대신에, immer를 사용하면 상황에 따라 코드가 더 간결해질 수 있다.

1. 만약 produce 함수에 두개의 파라미터를 넣게 된다면 첫번째 파라미터에 넣은 상태의 불변성을 유지하면서 새로운 상태를 만들어주지만, 만약에 첫번째 파라미터를 생략하고 바로 업데이트 함수를 넣어주게 되면 반환값은 새로운 상태가 아닌 상태를 업데이트 해주는 함수가 됨.

const todo = {

text: 'Hello',

done: false

};

const updater = produce(draft => {

draft.done = !draft.done;

});

const nextTodo = updater(todo);

console.log(nextTodo);

* produce가 반환하는 것이 업데이트 함수가 되서 updater(todo)하면 바로 상태가 변한다.
* 따라서 useState의 업데이트 함수를 사용할 때 다음과 같이 구현할 수 있게됨.

const [todo, setTodo] = useState({

text: 'Hello',

done: false

});

const onClick = useCallback(() => {

setTodo(

produce(draft => {

draft.done = !draft.done;

})

);

}, []);

* produce가 반환한 “이렇게 업데이트하세요”함수가 setTodo에 들어가 불변성을 지키면서 업데이트 됨.

**주의점**

1. 성능적으로 immer를 사용하지 않은 코드가 조금 더 빠르다.

* 하지만 아주많은 데이터를 처리할 생각이 아니면, 크게 신경 쓰지 않아도 된다.
* 다만, immer는 JS엔진의 Proxy라는 기능을 사용하는데, 구형 브라우저 및 react-native 환경에서는 지원되지 않으므로 ES5 fallback을 하는데, 이 경우 시간이 많이 느려지게 된다.
* 역시 데이터가 별로 없다면 크게 걱정은 안해도 된다.

1. 정리 : 무조건 사용은 금물, 데이터의 구조를 최대한 간결하게 하다가, 어쩔 수 없이 복잡해지면 그때 사용하라.