상태를 관리(업데이트)하게 될 때 사용하게 되는 함수 useState

이 함수를 사용하면 컴포넌트의 상태 업데이트 로직을 컴포넌트에서 분리 시킬 수있습니다.

상태 업데이트 로직을 컴포넌트 바깥에 작성할 수도 있고, 심지어 다른 파일에 작성 후 불러와서 사용 할 수도 있다.

+ 실습폴더 업로드

+ 상수는 변하지않는 변수를 의미(메모리 위치) 메모리 값을 변경할수 없다.

+ 리터럴은 변수의 값이 변하지 않는 데이터(메모리 위치안의 값)를 의미한다.

1. 커스텀 Hooks 만들기

* Ex) const [{username, email}, onChange, reset ] =useInputs({
  + Username:’’,
  + Email:’’
* });

이런 식으로

1. Context API 를 사용한 전역 값 관리

* contextAPI와 이전 섹션에서 배웠던 dispatch를 함께 사용하면 복잡한 구조를 해결 할 수있습니다.
* contextAPI를 사용하면, 프로젝트 안에서 전역적으로 사용 할 수있는 값을 관리할 수있습니다.
* contextAPI를 사용해서 새로운 Context를 만드는 방법.
* Ex) const UserDispatch = React.createContext(null);
* <UserDispatch.Provider value={dispatch}>...</UserDispatch>
* Context를 만들면, context 안에 provider라는 컴포넌트가 들어있는데, 이 컴포넌트를 통해 context의 값을 정할 수 있습니다.
* 이렇게 설정해주고 나면, Provider에 의하여 감싸진 컴포넌트 중 어디서든지 우리가 Context의 값을 다른 곳에서 바로 조회해서 사용 할 수있습니다.
* Export const UserDispatch = React.createContext(null);
* Return(
  + <UserDispatch.Provider value={dispatch}>
    - ...
  + </UserDispatch.Provider>

);

useReducer를 사용하면 이렇게 dispatch를 ContextAPI를 사용해서 전역적으로 사용 할 수있게 해주면 컴포넌트에게 함수를 전달해줘야하는 상황에서 코드의 구조가 훨씬 깔끔해질 수있습니다.

1. Immer를 사용한 더 쉬운 불변성 관리

* 리액트에서 배열이나 객체를 업데이트 해야 할 때에는 직접 수정하면 안되고 불변성 을 지켜주면서 업데이트를 해주어야 합니다.
* const nextState = produce(state, draft => {
  + Const post = draft.posts.find(post => post.id===1);
  + Post.comments.push({
    - Id:3,
    - Text : ‘ㅎㅇㅎㅇ’
    - });
  + });

Produce 함수에 두개의 파라미터를 넣게 된다면, 첫번째 파라미터에 넣은 상태를 불변성 을 유지하면서 새로운 상태를 만들어주지만, 만약에 첫번째 파라미터를 생략하고 바로 업데이트 함수를 넣어준다면, 반환 값은 새로운상태가 아닌 상태를 업데이트 해주는 함수가 됩니다.

const todo = {

    text: 'Hello',

    done: false

  };

const updater = produce(draft => {

    draft.done = !draft.done;

});

const nextTodo = updater(todo);

console.log(nextTodo);

//첫번째 파라미터를 생략하고 바로 업데이트 함수를 넣어주게된다면, 반환 값은 새로운 상태가 아닌 상태를 업데이트 해주는 함수가 됩니다.

const [todo, setTodo] = useState({

    text:'Hello',

    done:false

});

const onClick = useCallback(()=> {

    setTodo(

        produce(draft => {

            draft.done = !draft.done;

        })

    );

},[]);

//immer 을 사용하지않았더라면

Const onClick = useCallback(() => {

setTodo(todo => ({

...todo,

Done : !todo.done

}));

},[]);

Const onClick = useCallback(() => {

setTodo({...todo, done: !todo.done}),[todo]);

}

Sfadfasdf

1. 클래스형 컴포넌트

* 클래스형 컴포넌트에서는 render() 메서드가 꼭 있어야 합니다
* 이 메서드에서 렌더링하고 싶은 JSX를 반환하시면 됩니다. 그리고, props를 조회 해야 할 때에는 this.props를 조회하시면 됩니다.
* ㅇdefaultProps를 설정하는 것은 이전 함수형 컴포넌트에서 했을 때와 똑같이 해도되고, 다음과 같이 클래스 내부에 static 키워드와 함께 선언 할 수도있습니다.
* Ex) class Hello extends Component {
  + Static defaultProps = {
    - Name: ‘이름없음’
  + };
  + Render() {
    - const { color, name, isSpecial } = this.props;
    - Return(
      * <div style={{color}}>
        + {isSpecial && <b>\*</b>}
        + 안녕하세요{name}
      * </div>
    - );
    - 클래스형 컴포넌트에서는 클래스 안에 커스텀 메서드를 선언합니다.
    - Ex)
    - Class Counter extends Component {
      * hadleIncrease(){
        + Console.log(‘increase’);
      * }
      * Render(){
        + Return(

<div>

<h1>0</h1>

<button onClick={this.handleIncrease}>+1</button>

</div>

* + - * + );
      * }
    - }
    - 클래스 내부에 종속된 함수를 “메서드”라고 부릅니다.
    - Ex)

class Counter extends Component {

handleIncrease() {

console.log('increase');

console.log(this);

}

handleDecrease() {

console.log('decrease');

}

render() { ...}

* + - 이 코드에서 this는 컴포넌트 인스턴스를 가르켜야 하는데, 현재 구현한 handleIncrease 와 handleDecrease에서는 this를 조회하려고 하면, 컴포넌트 인스턴스를 가르키지 않게 됩니다. (콘솔로 출력하면 undefined가 나타나게 된다)
    - 우리가 만든 메서드들을 각 button 들의 이벤트로 등록하게 되는 과정에서 메서드와 컴포넌트 인스턴스의 관계가 끊겨버리기 떄문입니다.
    - 이를 해결하기 위한 3가지 방법

1. 클래스의 생성자 메서드 constructor에서 bind 작업을 해주는 것

Ex) class Counter extends Component {

Constructor(props){

Super(props);

This.handleIncrease = this.handleIncrease.bind(this);

This.handleDecrease= this.handleDecrease.bind(this);

}

//함수의 bind를 사용하면, 해당 함수에서 가르킬 this 를 직접 설정해줄 수있습니다. constructor에서는 props 파라미터로 받아오고 super(props)를 호출해주어야하는데, 이 클래스가 컴포넌트로서 작동 할 수있도록 해주는 component 쪽에 구현되어있는 생성자 함수를 먼저 실행해주고, 우리가 할 작업을 하겠다 라는 것을 의미합니다.

handleIncrease(){

Console.log(‘increase’);

Console.log(this);

}

...

1. 커스텀 메서드를 선언 할 때 화살표 함수 문법을 사용해서 작성

* handleIncrease = () => {
  + Console.log(‘increase’);
  + Console.log(this);
* };

1. onClick 에서 새로운 함수를 만들어서 전달을 하는 것

* Render(){
  + Return(
    - <div>
      * <button onClick={()=>this.handleIncrease()}>+1</button>
    - </div>

);

* 상태 선언하기
  + 클래스형 컴포넌트에서 상태를 관리 할 떄에는 state라는 것을 사용합니다.
  + State를 선언 할 때에는 constructor내부에서 this.state를 설정해주면 됩니다.
  + Ex)
    - Class Counter extends Component {
      * Constructor(props){
        + Super(props);
        + This.state = {

Counter:0

* + - * + };

}

Render(){

Return(

<div>

<h1>{this.state.counter}</h1>

</div>

);

* 클래스 컴포넌트의 state는 무조건 객체형태여야 합니다.
* Render 메서드에서 state를 조회하려면 this.state를 조회하면 됩니다.
* 화살표함수 문법을 사용하여 메서드를 작성할 수있게 해주었던 class-properties 문법이 적용되어있다면 굳이 constructor를 작성하지 않고도 다음과 같이 state를 설정해줄 수있습니다.
* State ={ counter:0 };
* ... <h1>{this.state.counter}</h1>
* 상태 업데이트 하기
  + 상태를 업데이트 해야 할 때에는 this.setState 함수를 사용하면 됩니다.
  + handleIncrease = () => {
    - This.setState({
      * Counter : this.state.counter + 1
    - });
  + };
* 클래스형 컴포넌트의 state에서 객체 형태의 상태를 관리해야 한다면, 불변성을 관리해줘가면서 업데이트를 해야 합니다.
* setState의 함수형 업데이트
  + handleIncrease = () => {
    - This.setState( state => {
      * Counter : state.counter + 1
    - }
  + };
* 함수형 업데이트는 보통 한 함수에서 setState를 여러번에 걸쳐서 해야 하는 경우에 사용하면 유용합니다.
* setState는 단순히 상태를 바꾸는 함수가 아니라 상태로 바꿔달라고 요청해주는 함수로 이해를 해야 합니다. ( 비동기적으로 업데이트가 됩니다. )
* 상태가 업데이트 되고 나서 어떤 작업을 하고 싶다면 다음과 같이 setState의 두번째 파라미터에 콜백함수를 넣어줄 수도 있습니다.
* handleIncrease = () => {
  + This.setState(
  + {
    - Counter : this.state.counter + 1
  + },
  + ()=> {
    - Console.log(this.state.counter);
  + }
* );
* );
* };

1. LifeCycle Method

* 생명주기 메서드 (마운트, 언마운트, 업데이트에 호출되는 메서드, 에러 날떄 호출되는 메서드 )
* 생명주기 메서드는 클래스형 컴포넌트에서만 사용 할 수 있다.
* 마운트 : constructor, getDerivedStateFromProps, render, componentDidMount
* 업데이트 : getDerivedStateFromProops, shouldComponentUpdate, render, getSnapshotBeforeUpdate, componentDidUpdate
* 언마운트: componenetWillUnmount

1. componentDidCatch로 에러 잡아내기/Sentry 연동

* Function User({user}){ //user props가 제대로 전달되지않았을때 에러처리
  + If(!user){
    - Return null;
  + }
* } //이를 null checking이라고 부릅니다.
* Function Users({users, onToggle}){
  + If(!users) return null;
  + Return(
    - <ul>
      * {users.map(user => (
        + <li key = {user.id} onClick{()=> onToggle(user.id)}>

{user.username}

* + - * + </li>
      * ))}
    - </ul>
  + );
* }
* 위 코드에서 에러가 발생했을때 해결 법
* 1. defaultProps로 설정해주는 방법

Users.defaultProps = {

onToggle : () => {

Console.warn(‘onToggle is missing’);

}

};

* componentDidCatch 로 에러 잡아내기