16. 리덕스 라이브러리 이해하기

- 리덕스는 가장 많이 사용하는 리액트 상태관리 라이브러리 이다.
- 리덕스를 사용하면 컴퍼넌트의 상태 업데이트 관련 로직을 다른 파일로 분리시켜 효율적관리가 가능하다.
- 또한, 동일 상태를 공유해야 할 때도 여러 컴퍼넌트를 거치지 않고 손쉽게 상태값 전달 및 수정할 수 있다.
- 리덕스는 `잔약싱태를 관리할 떄 효과적` 이다.
- Context API를 통해서도 동일 작업을 할 수 있다. 단순히 전역상태관리만 한다면 Context API를 사용도 충분하다.
- 하지만, `리덕스를 사용하면 상태를 체계적으로 관리가 가능` 하기 때문에 대규모 프로젝트일 경우 리덕스를 사용

16.1 개념정리하기

16.1.1 액션

- 상태에 대한 변화가 필요하면 액션(action)이 발생한다.
- `액션객체는 type필드를 반드시 가지고 있어야 한다.`
- 이 값이 액션이름이고 그 외의 값들은 상태업데이트를 할 때 참고할 값이고 작성자가 임의로 사용할 수 있다.

액션예제

```
{
  type: 'ADD_TODO',
  data: {
    id:1,
    text: '리덕스 배우기'
  }
}

{
  type:'CHANGE',
  text: '변경하기'
}
```

16.1.2 액션생성함수

- `액션생성함수(action creator)는 액션객체를 만들어 주는 함수` 이다.
- 변화를 일으켜야 할 때마다 액션객체를 만들어야 하는데 매번 작성하는 것을 방지하기 위해 함수로 관리한다.

액션생성함수 예제

```
function addTodo(data) {
  return {
    type: 'ADD_TODO',
    data
  }
```

```
}

// 액션생성함수는 화살표함수로도 만들 수 있다.
const changeInput = text => {
  type: 'CHANGE',
  text
}
```

16.1.3 리듀서

- 리듀서(reducer)는 변화를 일으키는 함수 이다.
- 액션을 만들어서 발생시키면 리듀서가 현재상태와 액션객체를 파라미터로 전달 받는다
- 그리고 두 값을 참고로 새로운 상태를 만들어서 반환`해 준다.

리듀서 예시

```
const intialState = {
  counter: 1
}

function reducer(state=intialState, action) {
  switch(action.type) {
    case INCREMENT:
      return {
        counter: state.counter + 1
      };
    default:
      return state
  }
}
```

16.1.4 Store

- 프로젝트에 리덕스를 적용하기 위해서 Store를 만든다
- 한 개의 프로젝트는 단 하나의 Store를 가질 수 있다.
- Store안에는 현재 애플리케이션 상태와 리듀서가 들어가 있고 중요한 내장함수들을 가지고 있다.

16.1.5 dispatch

- 디스패치는 store내장함수중 하나 이다.
- 디스패치는 액션을 발생시키는 것, 이 함수는 dispatch(action) 과 같은 형태로 액션객체를 파라미터로 호출한다.
- 이 함수가 호출되면 스토어는 리듀서 함수를 실행시켜서 새로운 상태를 만들어 준다.

16.1.6 subscribe

- 구독 subscribe도 스토어의 내장함수 중 하나이다.
- subscribe함수 안에 리스너 함수를 파라미터로 넣어서 호출 해 주면,
- 이 리스너 함수가 액션이 디스패치되어 상태가 업데이트될 때마다 호출 된다.

subscribe 예제

```
const listner = () => {
  console.log('상태가 업데이트됨');
}

const unsubscribe = store.subscribe(listner);

unsubscribe(); // 추후 구독을 비활성화할 때 함수를 호출
```

16.2 리액트없이 리덕스 사용하기

- 리덕스는 리액트에 종속되는 라이브러리가 아니다.
- 리액트에서 사용하하기 위해 만들어 졌지만 다른 UI라이브러/프레임워크에서도 사용할 수 있다.
- 리덕스는 바닐라(vanilla) 지비스크립트와 함께 사용할 수 있다.

16.2.1 `Parcel`로 프로젝트 만들기

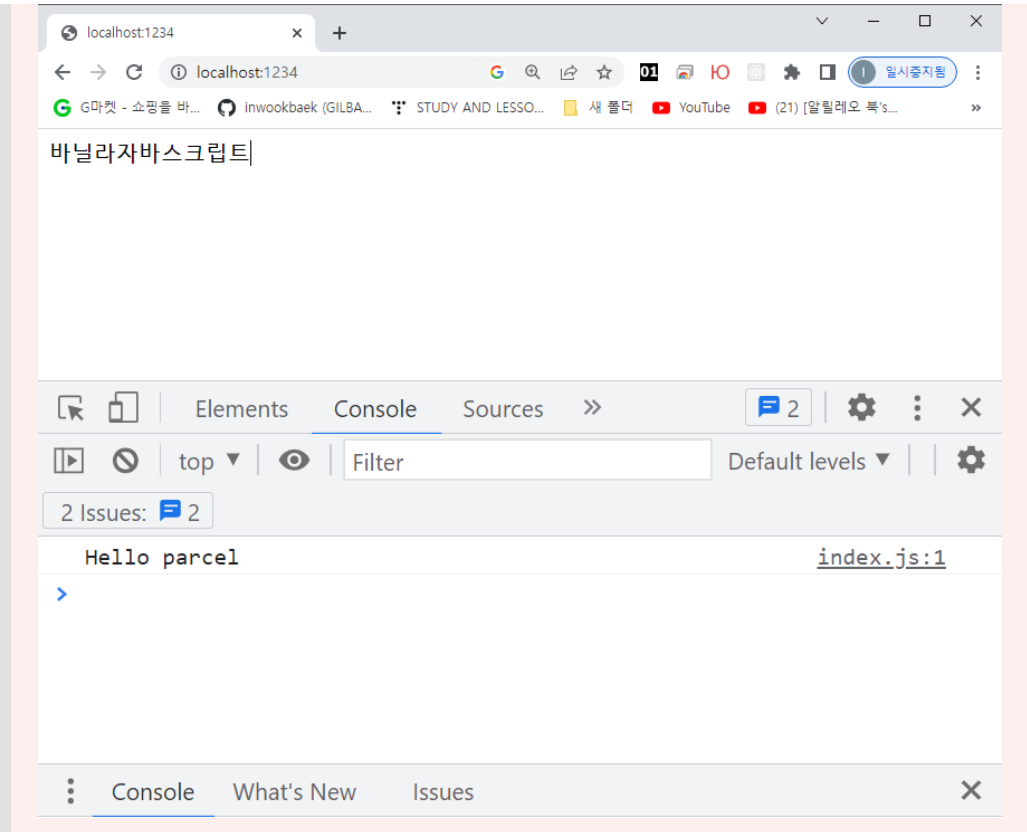
- Parcel을 사용하면 쉽고 빠르게 웹 애플리케이션을 구성할 수 있따.
- 설치 : yarn global add parcel-bundler
  - 인식이 잘 안되기 때문에 `npm install -g parcel-bundler`로 설치 할 것
- 폴더 16.vanilla_redux 생성
- 초기화 : `yarn init -y`
  - package.sjon 생성
- index.html 작성

```
<html>
<body>
  <div>바닐라자바스크립트</div>
  <script src="./index.js"></script>
</body>
</html>
```

- index.js

```
console.log('Hello parcel');
```

- 실행 : `parcel index.html`
  - http://localost:1234



- redux설치 : `yarn add redux`

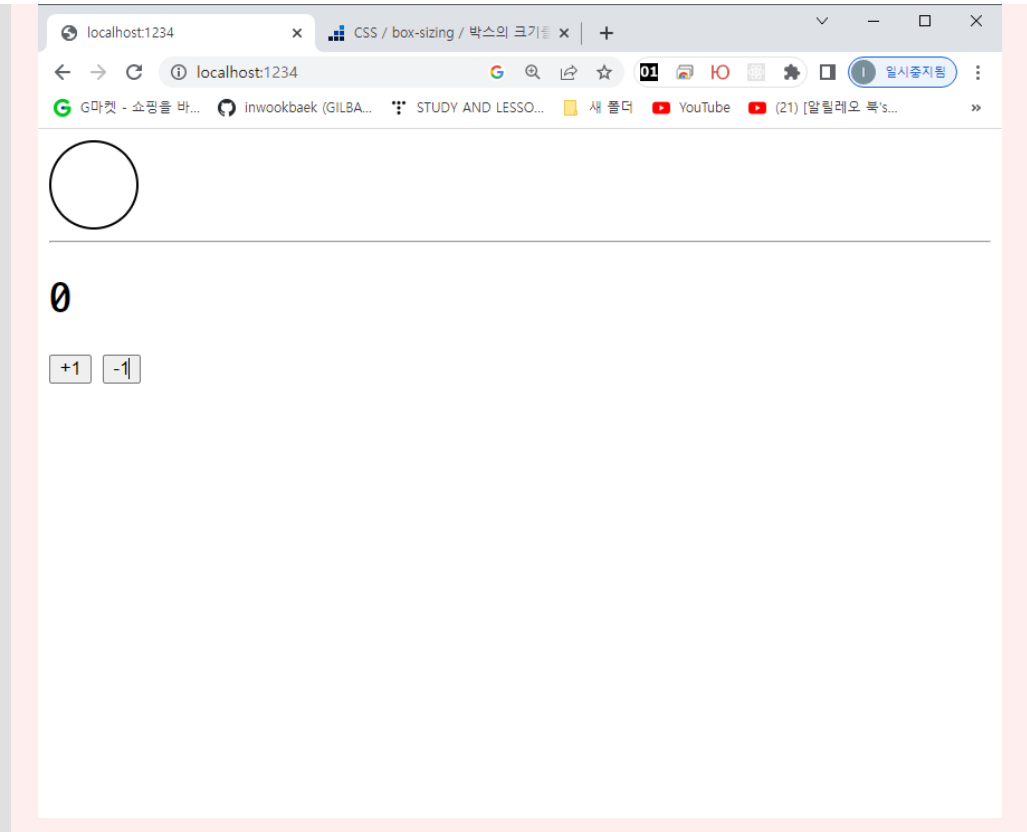
16.2.2 간단한 UI 구성하기

index.css

```css
.toggle {
  border: 2px solid black;
  width: 64px;
  height: 64px;
  border-radius: 32px;
  box-sizing: border-box;
}

.toggle.active {
  background: mediumseagreen;
}
```

index.html

```html
<html>
<head>
  <link rel="stylesheet" href="./index.css">
</head>
<body>
  <div class="toggle"></div>
  <hr/>
  <h1>0</h1>
  <button id="increase">+1</button>
  <button id="decrease">-1</button>
  <script src="./index.js"></script>
</body>
</html>
```



16.2.3 DOM 레퍼런스

index.js

```
// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');
```

16.2.4 액션타입, 액션함수 정의.

- 프로젝트의 상태변화를 일으키는 것이 액션이라고 한다.
- 먼저, `액션이름을 정의`, 액션이름은 문자열 대문자로 고유한 이름을 작성
- 다음, 액션이름을 사용하여 `액션객체를 만드는 액션함수를 작성`
- `액션객체는 반드시 type값을 가지고 있어야 하며` 추후 상태를 수정할 때 참고하고 싶은 값은 임의로 정의

index.js

```
// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';

// 액션생성함수
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'})
```

```
const increase = difference => ({ type: 'INCREASE', difference})
const decrease = () => ({ type: 'DECREASE'})
```

16.2.5 초기값설정

- 초기값을 정의, 초기값의 형태는 자유(숫자, 문자열 또는 객체)

index.js

```
// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';

// 액션생성함수
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'})
const increase = difference => ({ type: 'INCREASE', difference})
const decrease = () => ({ type: 'DECREASE'})

// 16.2.5 초기값설정
const initialState = {
  toggle: false,
  counter: 0
}
```

16.2.65 리듀서 함수 정의

- 리듀서함수가 처음 호출될 때 state값이 undefined이다. 초기값을 설정하기 위해 `state= initialState로 설정`
- 리듀서에서는 상태불변성을 유지하면서 데이터에 변화를 일으켜야 한다.
- 이 작업을 스프레드연산자(...)를 사용하면 편리하다.
- 단, 객체구조가 복잡해질 경우 스프레드연산자로 관리하는 것이 번거롭고 가독성이 나빠지기 때문에
- 리덕스의 상태는 최대한 간단한 구조로 진행하는 것이 좋다.
- 객체의 구조가 복잡해지거나 배열도 함께 처리하는 경우 `immer 라이브러리를 사용하면 쉽게 리듀서 작성이 가능`

index.js

```
// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';
```

```
// 액션생성함수
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'})
const increase = difference => ({ type: 'INCREASE', difference})
const decrease = () => ({ type: 'DECREASE'})

// 16.2.5 초기값설정
const initialState = {
  toggle: false,
  counter: 0
}

// 16.2.6 리듀서 함수 정의
// state가 undefined일 때는 initialState를 기본값으로
function reducer(state= initialState, action) {
  // action.type에 따라 처리
  switch(action.type) {
    case TOGGLE_SWITCH:
      return {
        ...state, // 불변성을 유지
        toggle: !state.toggle
      };
    case INCREASE:
      return {
        ...state,
        counter: state.counter + action.difference
      };
    case DECREASE:
      return {
        ...state,
        counter: state.counter - 1
      };
    default:
      return state;
  }
}
```

16.2.7 스토어 만들기

- store를 만들 때는 `createStore()함수를 사용`
- 상단에 createStore import해야 하고 팜수의 파라미터에 리듀서를 전달해야 한다.

index.js

```
import { createStore } from 'redux';

// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';
```

```
// 액션생성함수
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'})
const increase = difference => ({ type: 'INCREASE', difference})
const decrease = () => ({ type: 'DECREASE'})

// 16.2.5 초기값설정
const initialState = {
  toggle: false,
  counter: 0
}

// 16.2.6 리듀서 함수 정의
// state가 undefined일 때는 initialState를 기본값으로
function reducer(state= initialState, action) {
  // action.type에 따라 처리
  switch(action.type) {
    case TOGGLE_SWITCH:
      return {
        ...state, // 불변성을 유지
        toggle: !state.toggle
      };
    case INCREASE:
      return {
        ...state,
        counter: state.counter + action.difference
      };
    case DECREASE:
      return {
        ...state,
        counter: state.counter - 1
      };
    default:
      return state;
  }
}

// 16.2.7 스토어만들기
const store = createStore(reducer);
```

16.2.8 render함수 만들기

index.js

```
import { createStore } from 'redux';

// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';

// 액션생성함수
```

```
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'})
const increase = difference => ({ type: 'INCREASE', difference})
const decrease = () => ({ type: 'DECREASE'})

// 16.2.5 초기값설정
const initialState = {
  toggle: false,
  counter: 100
}

// 16.2.6 리듀서 함수 정의
// state가 undefined일 때는 initialState를 기본값으로
function reducer(state= initialState, action) {
  // action.type에 따라 처리
  switch(action.type) {
    case TOGGLE_SWITCH:
      return {
        ...state, // 불변성을 유지
        toggle: !state.toggle
      };
    case INCREASE:
      return {
        ...state,
        counter: state.counter + action.difference
      };
    case DECREASE:
      return {
        ...state,
        counter: state.counter - 1
      };
    default:
      return state;
  }
}

// 16.2.7 스토어만들기
const store = createStore(reducer);

// 16.2.8 render함수 만들기
const render = () => {
  const state = store.getState(); // 현재상태를 호출
  // 토글처리
  if(state.toggle) {
    divToggle.classList.add('active');
  } else {
    divToggle.classList.remove('active');
  }

  // counter처리
  counter.innerText = state.counter;
}

render();
```

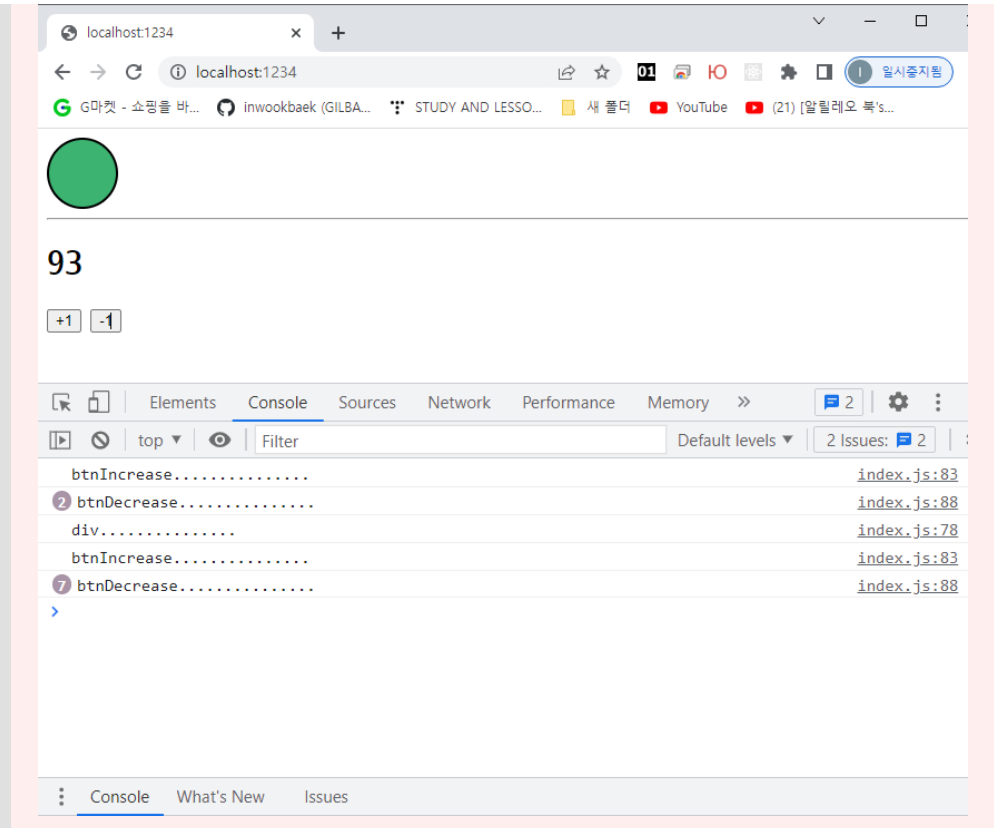
16.2.9 구독하기

- store상태가 변경될 때 마다 render()함수가 호출되도록 한다.
- 이 작업은 store의 내장함수 subscribe를 사용하여 수행한다.
- subscribe함수의 파라미터로는 함수 형태의 값을 전달해 준다.
- 바닐라 자바스크립트에서는 subscribe함수를 직접사용하지만 리덕스를 사용할 때는 직접 사용하지 않는다.
- 이는 `리덕스 상태를 조회하는 과정에서 react-redux 라이브러리가 이 작업을 대신 해 주기 때문` 이다.

index.js

```
import { createStore } from 'redux';

// 16.2.3 DOM reference
const divToggle = document.querySelector('.toggle');
const counter = document.querySelector('h1');
const btnIncrease = document.querySelector('#increase');
const btnDecrease = document.querySelector('#decrease');

// 16.2.4 액션타입과 액션생성함수 정의
const TOGGLE_SWITCH = 'TOGGLE_SWITCH';
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';

// 액션생성함수
const toggleSwitch = () => ({ type: 'TOGGLE_SWITCH'});
const increase = difference => ({ type: 'INCREASE', difference});
const decrease = () => ({ type: 'DECREASE'});

// 16.2.5 초기값설정
const initialState = {
  toggle: false,
  counter: 100
}

// 16.2.6 리듀서 함수 정의
// state가 undefined일 때는 initialState를 기본값으로
function reducer(state = initialState, action) {

  // action.type에 따라 처리
  switch(action.type) {
    case TOGGLE_SWITCH:
      return {
        ...state, // 불변성을 유지
        toggle: !state.toggle
      };
    case INCREASE:
      return {
        ...state,
        counter: state.counter + action.difference
      };
    case DECREASE:
      return {
        ...state,
        counter: state.counter - 1
      };
    default:
```

```
      return state;
    }
}

// 16.2.7 스토어만들기
// createStore에 취소선(deprecated되었기 때문) 없애는 방법
// https://velog.io/@xmun74/Q-createStore-취소선-왜-그어지나
const store = createStore(reducer);

// 16.2.8 render함수 만들기
const render = () => {
  const state = store.getState(); // 현재상태를 호출

  // 토글처리
  if(state.toggle) {
    divToggle.classList.add('active');
  } else {
    divToggle.classList.remove('active');
  }

  // counter처리
  counter.innerText = state.counter;
}

render();

// 16.2.9 구독하기
store.subscribe(render);

// 16.2.10 액션발생시키기
divToggle.onclick = () => {
  console.log('div..............')
  store.dispatch(toggleSwitch())
}

btnIncrease.onclick = () => {
  console.log('btnIncrease..............')
  store.dispatch(increase(1))
}

btnDecrease.onclick = () => {
  console.log('btnDecrease..............')
  store.dispatch(decrease(1))
}
```



16.3 리덕스의 세 가지 규칙

16.3.1 단일 스토어

- 하나의 애플리케이션 안에는 하나의 스토어가 있다.
- 사실, 여러 개의 스토어를 사용할 수는 있지만 상태관리가 복잡해 질 수 있기 때문에 권장하지는 않는다.

16.3.2 읽기 전용 상태

- `리덕스는 읽기 전용`이다.
- 리덕스에서 상태를 업데이트할 때 불변성을 유지하면서 새로운 객체를 생성해 주어야 한다.
- 불변성을 유지해야 하는 이유는 내부적으로 데이터가 변경되는 것을 감지하기 위해 얕은 비교를 하기 때문이다.

16.3.3 리듀서는 순수 함수

- 리듀서는 순수함수이어야 하고 순수함수의 조건은
  1. 리듀서함수는 이전상태와 액션객체를 파라미터로 받는다.
  2. 파라미터 이외의 값에 의존하면 않된다.
  3. 이전 상태는 건들이지 ㅇ낳고 변화를 준 새로운 상태 객체를 만들어 반환한다.
  4. 똑같은 파라미터로 호출된 리듀서 함수는 언제나 똑같은 결과 값을 반환해야 한다.
- 리듀서를 작성할 때는 위 네 가지 사항을 주의해야 한다.

- 리듀서 함수 내부에 랜덤값, Date함수등 네트워크 요청을 한다면 파라미터가 같아도 다른 결과가 발생되기 때문에 사용불가