6장 리덕스

* 상태관리 라이브러리입니다.
* 리덕스를 사용하는 것과 context API를 사용하는 것의 차이

1. 미들웨어

* 리덕스로 상태관리를 할 때에는 우리가 useReducer를 사용해볼때 접했던 개념인 리듀서 함수를 사용합니다.
* 리덕스의 미들웨어를 사용하면 액션 객체가 리듀서에서 처리되기 전에 우리가 원하는 작업들을 수행할 수있습니다.
  + 특정 조건에 따라 액션이 무시되게 만들 수있습니다.
  + 액션을 콘솔에 출력하거나, 서버쪽에 로깅을 할 수있습니다.
  + 액션이 디스패치 됐을 때 이를 수정해서 리듀서에게 전달되도록 할 수있습니다.
  + 특정 액션이 발생했을 때 이에 기반하여 다른 액션이 발생되도록 할 수있습니다.
  + 특정 액션이 발생했을 때 특정 자바스크립트 함수를 실행시킬 수 있습니다.

1. 유용한 함수와, Hooks

* useSelector, useDispatch, useStore 과 같은 Hooks를 사용하면 손쉽게 상태를 조회하거나 액션을 디스패치 할 수도 있습니다.

1. 하나의 커다란 상태

* 리덕스에서는 모든 글로벌 상태를 하나의 커다란 상태 객체에 넣어서 사용하는 것이 필수입니다. 때문에 매번 Context를 새로 만드는 수고로움을 덜 수있습니다.

1. 리덕스에 사용되는 키워드 숙지하기

* 액션(Action)
  + 상태에 어떠한 변화가 필요하게 될 땐, 우리는 액션이란 것을 발생시킵니다. 이는, 하나의 객체로 표현되는데, 액션 객체는 다음과 같은 형식으로 이뤄져 있습니다.
  + {type: “TOGGLE\_VALUE”}
  + 액션 객체는 type필드를 필수적으로 가지고 있어야하고 그 외의 값들은 개발자 마음대로 넣어줄 수있습니다.
  + { type : “ADD\_TODO”, data:{ id:0, text:”리덕스배우기”}}
* 액션 생성함수(Action Creator)
  + 액션 생성함수는, 액션을 만드는 함수 입니다. 단순히 파라미터를 받아와서 액션 객체 형대로 만들어줍니다.
  + Export const changeInput = text => ({ type: “CHANGE\_INPUT”, text});
  + -> 이러한 액션 생성함수를 만들어서 사용하는 이유는 나중에 컴포넌트에서 더욱 쉽게 액션을 발생시키기 위함 입니다. 그래서 보통 함수 앞에 expor t 키워드를 붙여서 다른 파일에서 불러와서 사용합니다.
  + 리덕스를 사용할 때 액션 생성함수를 사용하는 것이 필수적이진 않습니다. 액션을 발생시킬 때마다 직접 액션객체를 작성할 수도 있습니다.
* 리듀서 (Reducer)
  + 리듀서는 변화를 일으키는 함수입니다. 리듀서는 두가지의 파라미터를 받아옵니다.
  + Function reducer(state, action){
    - //상태 업데이트 로직
    - Return alteredState;
  + }
  + 리듀서는, 현재의 상태와, 전달 받은 액션을 참고하여 새로운 상태를 만들어서 반환합니다. 이 리듀서는 useReducer를 사용할 때 작성하는 리듀서와 똑같은 형태를 가지고 있습니다.
  + \*\*\*useReducer에선 일반적으로 default: 부분에 throw new Error(‘unhandled action’)과 같이 에러를 발생시키도록 처리하는게 일반적인 반면, 리덕스의 리듀서에서는 기존 state를 그대로 반환하도록 작성해야합니다.
  + 리덕스를 사용할 때에는 여러개의 리듀서를 만들고 이를 합쳐서 루트 리듀서(Root Reducer) 를 만들 수 있습니다.(루트 리듀서 안의 작은 리듀서들은 서브 리듀서라고 부릅니다.)
* 스토어(store)
  + 리덕스에서는 한 애플리케이션당 하나의 스토어를 만들게 됩니다. 스토어 안에는, 현재의 앱 상태와, 리듀서가 들어가있고, 추가적으로 몇가지 내장 함수들이 있습니다.
* 디스패치(dispatch)
  + 디스패치는 스토어의 내장함수 중 하나입니다. 디스패치는 액션을 발생 시키는 것이라고 이해하면 됩니다. dispatch 라는 함수에는 액션을 파라미터로 전달합니다. dispatch(action) 이런식으로
* 구독(subscrie)
  + 구독 또한 스토어의 내장함수 중 하나 입니다. subscribe 함수는, 함수 형태의 값을 파라미터로 받아옵니다. subscribe 함수에 특정 함수를 전달해주면, 액션이 디스패치 되었을 때 마다 전달해준 함수가 호출됩니다.

1. 리덕스의 3가지 규칙

* 1. 하나의 애플리케이션 안에는 하나의 스토어가 있습니다.
* 2. 상태는 읽기전용입니다.
  + 기존의 상태는 건들이지 않고 새로운 상태를 생성하여 업데이트해주는 방식으로 해주면, 나중에 개발자도구를 통해서 뒤로 돌릴 수도 있고, 다시 앞으로 돌릴 수도있습니다.
  + 리덕스에서 불변성을 유지해야하는 이유는 내부적으로 데이터가 변경되는 것을 감지하기 위하여 shallow equality 검사를 하기 때문입니다. 이를 통하여 객체의 변화를 감지 할 때 객체의 깊숙한 안쪽까지 비교를 하는 것이 아니라 겉핥기 식으로 비교를 하여 좋은 성능을 유지할 수 있는 것이죠
* 3. 변화를 일으키는 함수, 리듀서는 순수한 함수여야 합니다.
  + 순수한 함수
    - 리듀서 함수는 이전 상태와, 액션 객체를 파라미터로 받습니다.
    - 이전의 상태는 절대로 건들이지 않고, 변화를 일으킨 새로운 상태 객체를 만들어서 반환합니다.
    - 똑같은 파라미터로 호출된 리듀서 함수는 언제나 똑같은 결과값을 반환해야만 합니다.
  + 동일한 인풋이라면 언제나 동일한 아웃풋이 있어야 합니다. 그런데 일부 로직들 중에서는 실행 할 때마다 다른 결과값이 나타날 수도 있죠. new Date()를 사용한다던지, 랜덤 숫자를 생성한다던지.. 혹은, 네트워크에 요청을 한다던지, 그러한 작업을 결코 순수하지 않은 작업이므로, 리듀서 함수의 바깥에서 처리해줘야합니다. 그런것을 하기 위해서, 리덕스 미들웨어를 사용한다.

1. 리덕스 사용 할 준비하기

* Const initialState = {
  + Counter : 0,
  + Text : ‘’,
  + List : []
* };

//액션 타입 정의

//액션 타입은 주로 대문자로 작성합니다.

Const INCREASE = ‘INCREASE’;

Const DECREASE = ‘DECREASE’;

Const CHANGE\_TEXT = ‘CHANGE\_TEXT’;

Const ADD\_TO\_LIST = ‘ADD\_TO\_LIST’;

//액션 생성함수 정의

Function increase(){

Return {

Type: INCREASE

};

}

//화살표 함수로 작성하는 것이 더욱 코드가 간단하기에

//이렇게 쓰는 것을 추천합니다.

Const decrease = () => ({

Type: DECREASE

});

Const changeText = text => ({

Type: CHANGE\_TEXT,

Text //액션 안에는 type외에 추가적인 필드를 마음대로 넣을 수 있습니다.

});

Const addToList = item =({

Type: ADD\_TO\_LIST,

Item

});

//리듀서 만들기

//위 액션 생성함수들을 통해 만들어진 객체들을 참조하여 새로운 상태를 만드는 함수를 만들어봅시다. 주의: 리듀서에서는 불변성을 꼭 지켜줘야 합니다.

Function reducer(state = initaialState, action){

Switch(action.type){

Case INCREASE:

Return{

...state,

Counter : state.counter+1

};

Case DECREASE :

Return {

...state,

Counter : state.counter-1

};

Case CHANGE\_TEXT:

Return{

...state,

Text : action.text

};

Case ADD\_TO\_LIST:

Return{

...state,

List: state.list.concat(action.item)

};

Default:

Return state;

}

}

//스토어 만들기

Cosnt store = createStore(reducer);

//스토어안에 들어있는 상태가 바뀔 때마다 호출되는 listener함수

Const listener = () => {

Const state = store.getState();

Console.log(state);

}

Const unsubscribe = store.subscribe(listener);

// 구독을 해제하고 싶을 때는 unsubscribe()를 호출하면 됩니다.

//액션들을 디스패치

Store.dispatch(increase());

Store.dispatch(change\_text(‘안녕하세요’));

Store.dispatch(addToList({id:1, text:’와우’}));

1. 리덕스 모듈 만들기

* 리덕스 모듈
  + 액션 타입
  + 액션 생성함수
  + 리듀서
* Counter 모듈 만들기
  + Const SET\_DIFF = ‘counter/SET\_DIFF’;
  + Const INCREASE = ‘counter/INCREASE’;
  + Const DECREASE = ‘counter/DECREASE’;

Export const setDIff = diff => ({type:SET\_DIFF, diff});

Export const increase = () => ({type:INCREASE});

Export const increase = () => ({type:DECEASE});

Const initialState = {

Number : 0,

Diff : 1

};

//리듀서 선언

Export default function counter(state = initialState, action){

Switch(action.type){

Case SET\_DIFF:

Return {

...state,

Diff: action.diff

};

Case INCREASE:

Return{

...state,

Number : state.number + state.diff

};

Case DECREASE:

Retrun{

...state,

Number : state.number - state.diff

};

Default:

Return state;

}

}

Modules/todos.js

리액트 프로젝트에 리덕스 적용하기

* 리액트 프로젝트에 리덕스를 적용할 때에는 react-redux라는 라이브러리를 사용해야 합니다. ex) yarn add react-redux
* 그 다음에는 index.js에서 Provider라는 컴포넌트를 불러와서 App 컴포넌트를 감싸줍니다.
* Const store = createStore(rootReducer); //스토어를 만듭니다.
* ReactDom.render(
  + <Provider store={store}>
    - <App/>

</Provider>, ...);

* Provider로 store를 넣어서 App을 감싸게 되면 우리가 렌더링하는 그 어떤 컴포넌트던지 리덕스 스토어에 접근할 수있게 됩니다.

1. 카운터 구현하기

* 프리젠테이셔널 컴포넌트 만들기
  + 프리젠테이셔널 컴포넌트란, 리덕스 스토어에 직접적으로 접근하지 않고, 필요한 값 또는 props로만 받아와서 사용하는 컴포넌트 입니다.
  + 프레젠테이셔널 컴포넌트에선 주로 이렇게 UI를 선언하는 것에 집중하며, 필요한 값들이나 함수는 props로 받아와서 사용하는 형태로 구현합니다.
* 컨테이너 컴포넌트 만들기
  + 컨테이너 컴포넌트란, 리덕스 스토어의 상태를 조회하거나, 액션을 디스패치 할 수있는 컴포넌트를 의미합니다. 그리고, HTML 태그들을 사용하지 않고 다른 프리젠테이셔널 컴포넌트들을 불러와서 사용합니다.
  + useSelector는 리덕스 스토어의 상태를 조회하는 Hook입니다.
  + Ex) const {number, diff} = useSelect(state=> ({
    - Number : state.counter.number,
    - Diff : state.counter.diff
  + });
  + const dispatch = useDispatch();
  + //useDispatch는 리덕스 스토어의 dispatch를 함수에서 사용 할 수있게 해주는 Hook 입니다.
  + Const onIncrease = () => dispatch(increase());
  + Cosnt onDecrease = () => dispatch(decrease());
  + Const onSetDiff = diff => dispatch(setDiff(diff));
  + Return(
    - <Counter

Number={nuber}

Diff={diff}

//액션을 디스패치하는 함수들을 props로 넣어줍니다.

onIncrease={onIncrease}

onDecrease={onDecrease}

onSetDiff={onSetDiff}

/>

);

}

Export default CounterContainer;

1. 리덕스 개발자도구 적용하기

* Yarn add redux-devtools-extension
* Import {composeWithDevTools} fomr ‘redux -devtools-extension’;
* //리덕스 개발자 도구
* Const store = createStore(rootReducer, composeWIthDevTools());
* //스토어를 만들고, composeWithDevTools를 사용하여 리덕스 개발자 도구 활성화

1. 할 일 목록 구현하기

* 프리젠테이셔널 컴포넌트 구현하기
  + 이 파일에는 TodoItem,TodoList, Todos 이렇게 총 3가지의 컴포넌트를 작성할 것 입니다. 이렇게 여러개의 컴포넌트를 만드는 이유는 컴포넌트의 리렌더링 성능을 최적화 하기 위함입니다.