컴포넌트의 재사용성과 가독성

관심사의 분리가 필요하다. 관심사의 분리란 복잡한 코드를 비슷한 것끼리 관리하는 것. UI / API / DB 등의 관심사 분리를 통해 작성.

하나의 컴포넌트 안에서 모든 기능을 구현할 수 없기 때문에 / 여러 개의 분리된 컴포넌트를 하나의 폴더 안에 넣어서 관리하면 시간이 흐를수록 컴포넌트가 늘어나면서 관리가 힘듦. 따라서 컴포넌트 성격끼리 폴더를 분리. 페이지끼리 나눌 수도 있다. home 페이지 안에서, product 페이지 안에서 쓰이는 컴포넌트끼리 분리.

상태값이 컴포넌트 여기저기에 흩어져있을 수 있다. 자식컴포넌트에서 부모의 데이터를 별도의 상태값으로 관리하는 것은 좋지않다. 상태값은 일부 컴포넌트로 한정해서 관리한다. 컴포넌트가 비즈니스로직이나 상태값을 갖고 있으면 재사용하기가 힘들다.

비즈니스 로직과 상태값의 유무로 컴포넌트를 구분. 재사용성이 높은 컴포넌트와 복잡한 컴포넌트로 구분.

비즈니스 로직이 없다 + 상태값이 없다 -> 재사용성이 높은 컴포넌트. (UI담당 컴포넌트)

useEffect와 의존성배열

API를 받아서 부수효과를 사용하는 코드.

의존성배열에 userId나 다른 변수가 업데이트 됐을 때만 새로 렌더링하도록 하기.

eslint에서 의존성 배열을 제대로 입력하지 않았을 때 경고를 해 주는 패키지 제공. cra에는 기본 탑재.

만약에 값이 바뀌지 않을 것이라는 확신이 있다면 마운트 됐을때만 작동을 하는 별도의 훅을 만들어서 useEffect 대신 사용

import { useEffect } from "react";

export default function useOnMounted(effect) {

  useEffect(effect, []);

}

useOnMounted(() => fetchUser(userId).then(data => setUser(data)));

의존성 배열에 필요한 값이 입력되지 않았을 때:

function MyComponenet(){

    const [value1, setValue1] = useState(0);

    const [value2, setValue2] = useState(0);

    useEffect(() => {

        const id = setInterval(() => console.log(value1, value2), 1000);

        return () => clearInterval(id);

    }, [value1]);

    return(

        <div>

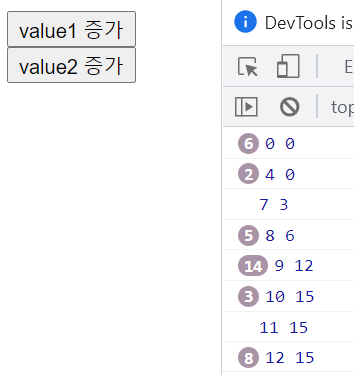
            <button onClick={() => setValue1(value1 + 1)}>value1 증가</button>

            <button onClick={() => setValue2(value2 + 1)}>value2 증가</button>

        </div>

    );

}

 value1을 증가시킨 데이터가 렌더링 이후 부수효과 함수가 실행될 때 반영된다. 하지만 value2가 새로 부수효과 함수를 만들 때에는 그것을 무시하고 value1이 변경될 때 생성된 함수를 바라보고 있다. value1 값이 변했을 때만 다시 데이터를 반영하여 부수효과 함수가 호출된다.

async await는 부수효과 함수가 될 수 없다.

부수효과 함수 안에서 실행 시점을 의존성배열을 사용하지 않고 조건문으로 실행시점을 조절할 수 있다. 이럴 경우 사용하는 모든 변수는 최신화된 값을 사용하므로 안심할 수 있다.

이전 상태값을 기준으로 다음 상태값을 계산하기 위해 의존성배열을 사용할 수 있다. 이 경우 상태값 변경함수에 함수를 입력함으로써 의존성배열을 대체할 수 있다. 해당 함수의 매개변수로 이전 상태값이 들어오기 때문

function MyComponenet() {

  const [count, setCount] = useState(0);

  useEffect(() => {

      function onClick() {

          setCount(count + 1);

      }

      window.addEventListener('click', onClick);

      return () => window.removeEventListener('click', onClick);

  }, [count]);

  // ...

}

//이렇게 변경

function MyComponenet() {

  const [count, setCount] = useState(0);

  useEffect(() => {

      function onClick() {

          setCount(prev => prev + 1);

      }

      window.addEventListener('click', onClick);

      return () => window.removeEventListener('click', onClick);

  }, []);

  // ...

}

렌더링

리액트에서 씨피유를 가장 많이 쓰는 부분은 렌더링

function MyComponent(props){

    //...

}

function isEqual(prevProps, nextProps){

    // true 또는 false를 반환

}

React.memo(MyComponent, isEqual);

컴포넌트의 속성값이나 상태값이 변경되면 리액트는 그 컴포넌트를 다시 그릴 준비를 한다. 만약 위처럼 리액트 메모로 감싼 컴포넌트라면 속성값 비교 함수를 호출한다. 이전 이후 속성값을 매개변수로 받아서 참 또는 거짓을 반환한다. 참을 반환하면 이전 렌더링 결과를 재사용하고 거짓이면 컴포넌트 함수를 호출해서 가상돔을 업데이트하고 바뀐 부분만 다시 렌더링한다.

isEqual부분에 속성값 비교함수를 기입하지 않으면 얕은 비교를 하는 함수가 자동 호출, 메모로 감싸놓지 않으면 항상 거짓을 반환하는 속성값 비교함수를 매개변수로 사용하는 것.

컴포넌트 실행과 가상돔 계산을 건너뛸 수 있어서 성능 향상.

// MyComponent.js

import React, { useState, useEffect } from "react";

function MyComponent({ value1, value2 }) {

  return (

    <div>

      <p>{`value1: ${value1}`}</p>

      <p>{`value2: ${value2}`}</p>

    </div>

  );

}

function isEqaul(prevProps, nextProps) {

  return prevProps.value1 === nextProps.value1;

}

export default React.memo(MyComponent, isEqaul);

import React, { useState, useEffect } from "react";

import MyComponent from "./MyComponent";

export default function App() {

  const [value1, setValue1] = useState(0);

  const [value2, setValue2] = useState(0);

  return (

    <div>

      <button onClick={() => setValue1(value1 + 1)}>value1 증가</button>

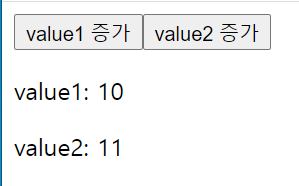
      <button onClick={() => setValue2(value2 + 1)}>value2 증가</button>

      <MyComponent value1={value1} value2={value2} />

    </div>

  );

}

 메모함수에 의해 value1이 변경될 때만 새로 렌더링

불변객체를 사용하면 비교가 빨라지는 이유

객체 자체를 변화시키는 것이 아니라 새 객체를 만드는 것이라서.

const prevTodos = [1, 2, 3];

const nextTodos = [...todos, 4];

prevTodos === nextTodos;

단순 비교만으로 가능

함수를 컴포넌트 내에서 이벤트에 함수를 넣어주면 함수가 매번 생성됨

function Parent(){

    // ...

    const onChangeFruit = useCallback(fruit => {

        // ...

    }, []);

    return (

        <div>

            // ...

            <SelectedFruit onChange = {onChangeFruit} />

        </div>

    );

}

반면 useCallback을 사용해서 위와 같이 작성하면 함수는 매번 바뀌지 않는다.

어레이 관련 함수 사용에 따른 매번 새로운 어레이 생성은 useMemo를 사용한다.

const fruits = useMemo(() => fruits.filter(item => item.price < maxPrice),[]);

return(

    <div>

        <Selected options = {fruits} />

    </div>

)

다만 미리 성능을 걱정해서 선제적으로 사용하기 보다는 성능 문제가 발생하면 추후 리팩토링 하는 방식 추천

값 변경은 전개연산자

setFruits([…fruit, newFruit])으로 사용한다.

리액트는 key속성값을 입력하여 같은 key를 가진 값끼리 비교하여 가상돔에서 실제돔에 반영한다.

또한 key속성을 사용할 때 객체의 중간 값이 변하는 경우에는 인덱스로 부여하지 않도록 한다.

리덕스: 상태 관리 라이브러리, 애플리케이션 전체의 상태를 관리할 수 있다.

컴포넌트 코드로부터 상태 관리 코드를 분리할 수 있다.

미들웨어를 활용한 다양한 기능 추가

* 강력한 미들웨어 라이브러리(ex. redux-saga)
* 로컬 스토리지에 데이터 저장하기 및 불러오기

SSR시 데이터 전달이 간편하다.

리액트 콘텍스트보다 효율적인 렌더링 가능

리덕스의 구조

액션 – 미들웨어 – 리듀서 - 스토어 🡪 뷰(프론트엔드 애플리케이션을 만들 때)

1.액션

type 속성값을 갖고 있는 객체

{ type: "todo/ADD", title: "영화 보기", priority: "high" }

type속성 외에도 원하는 대로 나머지 필요한 데이터(속성)을 전달할 수 있다.

이 데이터를 리듀서에서 받아서 처리를 한다. 액션을 구별하기 위해 type을 사용하며 따라서 유니크해야한다. todo/…와 같이 프리픽스를 붙여 잘 사용함.

일반적으로 액션크리에이터 함수를 만들어서 사용함.

각 액션의 구조를 일관성있게 만들기 위함이다. 속성 등이

액션타입은 액션크리에이터 뿐만 아니라 리듀에서도 사용하기 때문에 상수로 선언해두는 것이 좋다.

2. 미들웨어 는 스토어와 넥스트를 사용한다.

미들웨어1, 미들웨어2, 리듀서, 스토어 선언 후 액션 실행 예시

import { createStore, applyMiddleware } from "redux";

const middleware1 = store => next => action => {

    console.log('middleware1 start');

    const result = next(action);

    console.log('middleware1 end');

    const result;

};

const middleware2 = store => next => action => {

    console.log('middleware2 start');

    const result = next(action);

    console.log('middleware2 end');

    const result;

};

const myReducer = (state, action) => {

    console.log('myReducer');

    return state;

}

const store = createStore(myReducer, applyMiddleware(middleware1, middleware2));

store.dispatch({type: 'someAction'});

실행되는 순서:

초기 설정으로 인한 Reducer -> 액션 실행 후 middleware1 -> middleware2 -> reducer -> mw1 end -> mw2 end

next를 호출하며 다음 미들웨어를 호출하고 마지막 미들웨어에서 reducer를 호출한다.

트라이캐치를 사용하여 리듀서에서 발생한 에러를 서버로 전송하는 기능 사용 가능.

액션의 메타에 딜레이 데이터가 있을 때 셋타임아웃을 줘서 리듀서를 늦게 실행하는 기능 사용 가능.

//JS문법 action.meta?.delay 만약 meta가 undefined일 경우에도 에러가 발생하지 않음

//action.meta && action.meta.delay와 같다.

3. 리듀서는 액션이 발생했을 때 새로운 상태값을 만드는 함수이다. 리덕스의 상태값을 변경하는 유일한 방법은 액션 객체와 함께 dispatch함수를 호출하는 것.

상태값은 불변객체로 관리한다. 장점: 단순 비교로 객체의 변화를 파악할 수 있다.

전개연산자를 사용하여 불변객체로 관리.

리듀서 코드 예시

function reducer(state = INITIAL\_STATE, action) {

  switch (action.type) {

    case REMOVE\_ALL:

      return {

        ...state,

        todos: [],

      };

    case REMOVE:

      return {

        ...state,

        todos: state.todos.filter((todo) => todo.id !== action.id),

      };

    default:

      return state;

  }

}

const INITIAL\_STATE = { todos: [] };

수정하려는 값이 깊은 곳에 있다면 매번 전개연산자를 사용하는 것이 번거로울 수 있다.

이 때 여러 라이브러리 중 immer라는 패키지가 좋다.

produce함수를 사용하여 변경하고 싶은 값을 매개변수로, 상태값을 변경하는 로직을 매개변수로 작성하여 수정 가능.

import produce from "immer";

const person = { name: "mike", age: 22 };

const newPerson = produce(person, draft => {

  draft.age = 32;

});

리듀서는 순수함수로만 작성. 서버 API를 리듀서에서 호출하면 안됨

createReducer라는 함수를 자주 만들어서 사용을 함.

객체의 키는 액션타입, 밸류는 함수로 작성

4.스토어

createStore라는 함수 사용. 인자에 리듀서를 넣어서 호출 및 store객체 생성.

스토어는 상태값을 저장하는 역할도 있으며 액션처리가 끝났음을 외부에서 알려주는 역할도 한다. subscribe 메서드 사용.

각 액션에 대한 처리가 끝나면 subscribe 내부에 작성한 함수가 호출됨.

const store = createStore(reducer);

let prevState;

store.subscribe(() => {

  const state = store.getState();

  if (state === prevState) {

    console.log("상태값 같음");

  } else {

    console.log("상태값 변경됨");

  }

  prevState = state;

});

combineReducers 여러 개의 리듀서를 하나로 합치는 리덕스의 메서드

react-redux 패키지

리덕스에서 변수를 가져올 때에는 useSelector를 사용함

shallowEqaul을 사용하여 얕은 비교를 통한 배열의 변화를 확인할 수 있다. 매번 useSelector 안에 shallowEqual을 사용하는 것은 번거롭기 때문에 커스텀훅을 만들어서 사용하기를 추천.

dispatch 함수 또한 useDispatch 훅을 사용해서 만들고 사용.

리덕스에 저장된 데이터를 화면에 보여줄 때 다양한 형식으로 가공이 필요 - reselect패키지가 도움이 된다.

예를 들어 프렌드리스트를 화면에 구현할 때 필터연산이 필요할 때. 15~20세까지 등등 성별 등등

import {createSelector} from 'reselector';

const getFriends = state => state.friend.friends;

export const getAgeLimit = state => state.friend.ageLimit;

export const getShowLimit = state => state.friend.showLimit;

export const getFriendsWithAgeLimit = createSelector(

    [getFriends, getAgeLimit],

    (friends, ageLimit) => friends.filter(item => item.age <= ageLimit),

);

export const getFriendsWithAgeShowLimit = createSelector(

    [getFriendsWithAgeLimit, getShowLimit],

    (friends, showLimit) => FriendsWithAgeLimit.slice(0, showLimit),

);

selector.js파일로 선택자를 미리 만들어 데이터를 가공하는 로직을 만들어 두고, 본문에서 useSelector로 호출해 준다.

리덕스에서 새로운 상태값을 추가하려면 액션- 상태값 생성함수 – 등 모두를 작성해야한다.

리덕스 사가: 제너레이터 문법을 사용하고 테스팅이 쉬운 비동기 액션 처리 패키지

import { all, call, put, takeLeading } from "redux-saga";

import { callApiLike } from //...

export function\* fetchData(action) {

  yield put(actions.setLoading(true));

  yield call(callApiLike);

  // ...

}

export default function\* () {

  yield all([takeLeading(types.REQUEST\_LIKE, fetchaData)]);

}

\*, yield 등은 제너레이터 문법

put, call 등은 리덕스 사가의 부수효과함수 같은…

put 등의 부수효과 함수는 JS객체를 반환한다. 이것을 이용하여 yield로 호출했을 때 사가 미들웨어로 객체를 넘기고 프로세스 진행…

제너레이터 이해하기

제너레이터 함수를 호출하면 제너레이터 객체가 반환된다. 제너레이터 객체에는 next()라는 메서드가 있다.

function\* f1() {

    console.log('f1-1');

    yield 10;

    console.log('f1-2');

    yield 20;

    console.log('f1-3');

    return 'finished';

}

const gen = f1();

console.log(gen.next());

console.log(gen.next());

console.log(gen.next());

f1-1

{value: 10, done: false}

f1-2

{value: 20, done: false}

f1-3

{value: 'finished', done: true}

이터레이터와 이터러블 개념과 관련 있다. 결국 제너레이터는 이터레이터라고 생각하면 된다. next메서드의 반환값으로는 밸류와 던이 있고 던은 끝나면 true를 반환한다.

게다가 제너레이터는 이터레이터이면서 이터러블이다.

console.log(gen[Symbol.iterator]() === gen); // true

위 Symbol.iterator 속성값으로 함수를 실행하면 이터레이터가 반환되는데, 제너레이터는 이터레이터이므로 위 코드는 true 출력.

function\* naturalNumbers(){

    let v = 1;

    while (true){

        yield v++;

    }

}

일반함수 였다면 무한루프 때문에 아마 먹통이 되었을 것. 하지만 제너레이터는 값을 하나씩 던져주기 때문에 구현 가능. 실행을 멈추고 재개할 수 있다.

따라서 함수간 협업이 수월하다

function\* minsu(){

    const myMsgList = [

        '안녕 나는 민수야',

        '만나서 반가워',

        '내일 영화 볼래?',

        '시간 안 되니?',

        '내일 모레는 어때',

    ];

    for (const msg of myMsgList){

        console.log('수지:', yield msg);

    }

}

function suji(){

    const myMsgList = ['', '안녕 나는 수지야', '그래 반가워', '...'];

    const gen = minsu();

    for (const msg of myMsgList){

        console.log('민수:', gen.next(msg).value);

    }

}

suji();

민수: 안녕 나는 민수야

VM46:10 수지: 안녕 나는 수지야

VM46:18 민수: 만나서 반가워

VM46:10 수지: 그래 반가워

VM46:18 민수: 내일 영화 볼래?

VM46:10 수지: ...

VM46:18 민수: 시간 안 되니?

디바운스: 같은 함수가 연속해서 호출될 때 첫번째 혹은 마지막 호출만 실행하는 것

프로젝트

JS도구로 타입 정의 하는 예시

/\*\*

 \*

 \* @param {object} param

 \* @param {() => void} param.logout

 \*/

입력인자 없고 리턴 보이드인 로그아웃 함수

/\*\*

 \*

 \* @param {object} param

 \* @param {'get' | 'post' =} param.method

 \* @param {string} param.url

 \* @param {object=} param.params

 \* @param {object=} param.data

 \* @param {object=} param.totalCount

 \*/

예를들어 메서드는 겟이나 포스트만 가능하며 옵셔널로 값을 입력하지 않아도 된다.

api통신과 관련된 액션은 fetch~로 작성