2024\_1학기 프론트엔드프레임워크 6/3 10~13

### children props의 타입

children props의 타입은 JSX.Element | JSX.Element[] 이다.

JSX.Element는 virtual DOM 태그 객체의 타입이다.

즉 children props의 타입은

virtual DOM 태그 객체 한 개이거나

virtual DOM 태그 객체 배열이다.

sass: scss의 다른 이름

프로젝트를 빌드할 때, \*.scss 파일이 \*.css 파일로 트랜스파일된다.

sass 트랜스파일러는 빌드할 때만 필요하고, 실행할 때는 필요없다.

따라서 -D 옵션으로 설치한다.

예를 들어 java 컴파일러는 프로젝트를 빌드할 때만 필요하고 서버에서 운영할 때는 필요없다.

확장 문법 \*.scss 파일을 표준 문법 \*.scc 파일로 트랜스 파일 할 때, 즉 프로젝트를 빌드할 때만 필요하다.

이렇게 프로젝트를 개발할 때만(빌드할 때만) 필요하고

서버에서 실행할 때는 필요없는 npm 패키지를 설치할 때 -D 옵션을 준다.

프로젝트에 설치한 패키지들 목록은 package.jon 파일에서 확인할 수 있다.

이 파일의 "dependencies" 항목은, 언제나 필요한 패키지 목록이고

"devDependencies" 항목은, 개발할 때만 필요한 패키지 목록이다.

& -> 현재 태그를 의미함.

&:active { ... } 부분은 앞페이지의 .btn:active { ... } 부분과 동일한 서식이다.

그런데 &:active { ... } 부분은 .btn { ... } 내부에 있어야 한다.

$color: #88f;

변수 이름이 $color

변수에 대입된 값이 #88f

darken(색, %) lighten(색, %) 기본 함수

예: darken(#88f, 10%) #88f 색 보다 10% 더 어두운 색을 리턴함

lighten($color, 15%) $color 변수에 들어있는 색 보다 15% 더 밝은 색을 리턴함.

리액트에서 함수형 컴포넌트는

Virtual DOM 태그 객체들을 생성하여 리턴하는 함수이다.

동일한 구조의 태그 객체를 여기 저기서 생성해야 한다면,

그 부분을 함수로 추출하여 구현하고 필요한 곳에서 그 함수를 호출하자.

function colorDiv(name: string) {

  return ... }

이 소스 코드는 태그 객체를 생성하여 리턴하는 함수를 구현한 것이다.

{colorDiv("blue")}

이 소스 코드는 colorDiv 함수가 생성하여 리턴한 태그 객체를 이 부분에 삽입한다.

lightness($color)

ligntness(...) 함수의 파라미터는 색상값(color) 이어야 한다.

이 색의 밝은 정도를 0부터 100까지 정수로 표현하여 리턴한다.

0: 검정색, 100: 흰색

$font\_color: white;

@if (lightness($color) >= 80) {

  $font\_color: black;}

위 코드는 $color 값이 밝은 정도가 80% 이상이라면 $font\_color 값을 black 으로 설정하고,

그렇지 않다면 $font\_color 값을 white로 설정한다.

## useRef 훅

useRef 훅의 용도는 두 개 이다.

용도 #1: Physical DOM 객체의 속성이나 메소드를 사용하기. (웹브라우저의 DOM 객체)

용도 #2: 상태값이 변경될 경우에 즉시 렌더링 될 필요는 없는 상태값 만들기.

평범한 자바스크립트 지역 변수는,

함수가 호출될 때 다시 만들어지고 함수가 리턴될 때 없어진다.

함수형 컴포넌트 내부의 평범한 자바스크립트 변수는

그 함수가 호출될 때 마다

즉 랜더링 될 때 마다 다시 만들어지고 함수가 리턴될 때 없어진다.

따라서 값이 유지되지 않고, 렌더링될 때 마다 초기화된다.

함수형 컴포넌트 내부에서 값이 계속 유지되어야 한다면

평볌한 지역 변수에 그 값을 담을 수는 없고

useState 상태를 만들던가

useRef 변수를 만들어야 한다.

useState나 useRef에 대입한 값은,

컴포넌트가 unmount될 때까지 유지된다.

값이 변할 때 마다 즉시 다시 렌더링되어야 한다면 useState

그렇지 않다면 useRef

## 용도 #1

리액트이 virtual DOM 태그 객체에는 없고

웹브라우저 창 내부 pysical DOM 객체에만 있는 기능을 사용하려면 (속성이나 메소드를 사용하려면)

그 pysical DOM 객체에 대한 참조 주소가 필요하다.

이 참조 주소를 저장하기 위한 변수를 만들 때 useRef 훅을 사용한다.

const divRef = useRef<HTMLDivElement>(null)

위 코드는 useRef 훅을 이용해서 divRef 변수를 만든다.

divRef 변수는 Pysical DOM 객체(웹브라우저의 DOM 객체)를 참조하기 위한 리액트 변수이다.

웹브라우저의 Pysical DOM 객체의 타입이 HTMLDivElement 이다.

divRef 변수의 초기 값으로 null을 대입한다.

예를 들어 웨브라우저에서 스크롤바의 스크롤 위치를 설정하는 메소드는

virtual DOM 객체에 없고, pysical DOM 객체에 있다.

따라서 그 메소드를 호출하려면 pysical DOM 객체에 대한 참조 변수가 필요하다.

**참조변수.스크롤메소드();**

만약 참조변수의 값이 null 이면, 위 코드는 null 에러가 발생한다.

**참조변수?.스크롤메소드();**

참조변수의 값이 null 이어도 에러가 발생하지 않는다.

const Div = styled.div<{w: string, h: string}>`

  border: 1px solid #ccc;

  padding: 5px;

  overflow-y: scroll;

  width: ${props => props.w};

  height: ${props => props.h};

Div 컴포넌트의 props는 w: string, h: string 이다.

이 props 값들이 각각 ${props => props.w} 부분과 ${props => props.h} 부분에 출력된다.

<Div w={width} h={height}>

Div 컴포넌트의 w, h props 값 전달하기

const divRef = useRef<HTMLDivElement>(null)

divRef 참조 변수를 만든다.

이 변수에 대입된 값은 컴포넌트가 unmount 될 때까지 계속 유지된다.

divRef 참조 변수에 처음 대입되는 값은 null 이다.

<div ref={divRef}></div>

여기서 ref={divRef} 부분은이 div 태그에 해당하는 Pysical DOM 객체에 대한 참조를

나중에 divRef 변수에 대입해 달라는 선언이다.

왜 나중에 대입할 수 밖에 없냐면 이 함수형 컴포넌트가 만들어서 리턴하는 것은

virual DOM 자바스크립트 객체이다

함수형 컴포넌트들이 리턴한 virtual DOM 객체들과 똑같은 구조로

리액트 엔진이 웹브라우저 내부에 Pysical DOM 객체들을 만든다.

그렇게 만들어질 Pysical DOM 객체에 대한 참조 주소를

이 divRef 변수에 대입해 달라는 선언이 ref={divRef} 이다.

<div ref={divRef}></div>

이 코드는 Virual DOM 자바스크립트 객체를 만들어 리턴한다.

그렇게 리턴된 Virtual DOM 객체와 동일한 구조로

리액트 엔진이 웹브라우저창 내부에 Pysical DOM 객체를 만든다.

그렇게 만들어진 Pysical DOM 객체에 대한 참조 주소를 divRef 변수에 대입해 준다.

이렇게 대입할 때는 이미 함수형 컴포넌트 함수가 리턴된 후 이므로

리턴될 때 마다 없어지는 평범한 자바스크립트 지역 변수에 대입해 줄 수 없고

useRef 훅으로 만든 변수에 대입해야 한다.

이렇게 divRef에 대입된 값을 사용하는 코드는 다음과 같다.

divRef.current

.current를 뒤에 붙여서 사용해야 한다.

useEffect(() => divRef.current?.scrollIntoView());

위 코드는 useEffect 훅의 호출형태 (#C: update) 이므로,

() => divRef.current?.scrollIntoView() 이 콜백 함수는

이 컴포넌트가 렌더링 될 때 마다 호출된다.

정확히 말하면, 화면이 렌더링 된 직후에 호출된다.

divRef.current 부분이 divRef 참조 변수에 대입된 값이다.

이것은 div 태그 Pysical DOM 객체이다.

이 div 태그에 overflow-y: scroll; 서식을 적용했기 때문에,

스크롤바가 달려 있을 것이다.

이 스크롤바 위치를 설정하는 메소드는 Pysical DOM 객체에만 있고, Virtual DOM 객체에는 없다.

scrollIntoView() 메소드는 Pysical DOM 객체의 메소드이다.

리액트의 메소드가 아니고, 대부분의 웹브라우저가 지원하는 표준 HTML 메소드이다.

이 메소드를 호출하면 그 태그가 화면에 보이도록 자동으로 스크롤된다.

참조변수.스크롤메소드()

divRef.current?.scrollIntoView()

참조 값이 null 일 수도 있을 때는 . 대신 ?. 을 사용해야 한다.

화면이 렌더링 될 때 마다 렌더링 된 직후에,

() => divRef.current?.scrollIntoView() 콜백함수가 호출된다.

<div ref={divRef}></div> 태그가 화면에 보이도록

웹브라우저 창의 스크롤바, 부모 태그의 스크롤바들이 자동으로 스크롤된다.

이 div 태그는 <Div>...</Div> 태그로 구현한 스크롤박스의 마지막 항목이므로

화면이 렌더링 될 때 마다

이 스크롤 박스의 마지막 항목이 보이도록 스크롤바가 자동으로 스크롤된다.

const colorRef = useRef<string>("#ffa");

colorRef 상태 변수를 만든다.

이 변수의 값은 string 이다. 이 변수의 최초 값은 "#ffa"이 변수의 값은 앞으로 추가될 항목의 배경 색이다.

const [list, setList] = useState<Item[]>([]);

데이터 목록 상태 변수

const addItem = () => {

    const time = new Date().toLocaleTimeString();

    setList([...list, {time, color: colorRef.current}]); }

add 버튼이 클릭되면, 이 메소드가 호출된다.

현재 시각 문자열과, colorRef 상태 변수의 현재 값울 배경색으로 데이터 항목 객체를 만들어서

데이터 목록에 추가한다. 이때 화면이 다시 렌더링 된다.

<select onChange={e => colorRef.current = e.target.value}>

select 태그에서 선택된 항목이 변경되면, 그 항목의 value 값을 colorRef 상태 변수에 대입된다.

useRef 참조 변수의 값이 변경될 때는 화면이 다시 랜더링 되지 않는다.

colorRef 값은, 앞으로 추가될 항목의 배경색일 뿐이므로

이 값이 바뀌어도 현재 화면에서 달라질 부분이 없다.

따라서 이 값이 바뀔 때 화면을 다시 렌더링 할 필요 없으므로

useState 보다는 useRef 훅으로 구현하였다.

useState 훅으로. color 상태를 구현하면,

select 태그의 값이 변경될 때마다 다시 렌더링되는 비효율이 있다.

**컴포넌트의 상태를 지역 변수로 구현할 수 없는 이유**

1. App 컴포넌트가 렌더링 될 때 마다, App() 함수가 호출된다.
2. App 함수가 호출 될 때 마다, color 지역 변수가 다시 생성되고 초기화 된다.
3. 즉 color 지역 변수의 값이 렌더링될 때 마다 초기화 된다.

예를 들어 TodoItem 컴포넌트는 todo 항목의 갯수 만큼 컴포넌트 객체가 생성되어야 한다.

TodoItem 컴포넌트의 상태를 전역 변수로 구현하면,

생성된 TodoItem 컴포넌트 객체들이 그 전역 변수 한 개를 공유하게 된다.

useRef 훅의 기능은 useState 훅과 큰 차이는 없다.

둘 다 지역 변수에 저장할 수 없는 값을 저장하기 위한 공간을 제공한다.

지역 변수: 함수가 호출 될 때 마다 새로 만들어지고, 리턴될 때 없어진다.

useState, useRef: 컴포넌트가 마운트될 때 처음 저장 공간이 만들어지고, 언마운트될 때 없어진다.

그 저장 공간의 값이 변경될 때

즉시 다시 렌더링 되어야 한다면 useState

다시 렌더링 될 필요가 없다면 useRef

import { counterState } from './counterState';

counterState recoil 상태를 사용하기 위해 import 한다.

import { useRecoilState } from 'recoil';

const [count, setCount] = useRecoilState<number>(counterState);

useRecoilState 훅은 useState 훅과 매우 비슷하다.

차이점은 다음과 같다.

- useState는 컴포넌트 내부에 상태 변수를 만드는 것이라고 생각할 수 있다.

컴포넌트 인스턴스 마다 그 상태 변수를 따로 소유한다.

- useRecoilState는 recoil 전역 상태를 공유하며 사용한다.

counterState 전역 상태는 한 개이다. 이 상태를 사용하는 컴포넌트들은

한 개의 전역 상태를 공유한다.

그러면 두 컴포넌트 인스턴스가 하나의 recoil 전역 상태를 공유하기 때문에,

화면의 두 숫자가 같이 변한다.

get: ({get}) => {

  const message = get(messageState);

  return message.length; }

화살표 함수 문법으로 getter를 정의하였다.

이 화살표 함수의 파라미터는 {get} 이다. 여기서 get은 다른 상태값을 읽기 위한 get 함수이다.

const message = get(messageState);

messageState 상태값을 읽어서 message 지역 변수에 대입한다.

return message.length;

message 문자열의 길이를 리턴한다.

useState로 input 태그 입력 기능을 구현한 것과 매우 유사한 구현이다.

useState 대신 useRecoilState 훅으로 구현했으므로,

input 태그에 입력된 값은 messageState 전역 상태 변수에 대입된다.

상태값을 읽고/쓰고 할 때는 useRecoilState 훅을 사용한다.

이 훅은 [상태값, setter함수] 배열을 리턴한다.

recoil 전역 상태에 접근하는 컴포넌트에, 그 전역 상태에 대한 작업 코드도 구현하는 방식으로 구현된

recoil 예제들을 흔히 볼 수 있다.

위와 같은 구현 방식은 별로 바람직하지 않다.

주요 데이터와, 그 데이터에 대한 작업 코드는

소스코드 파일 한 곳에 모아서 구현하는 것이 바람직하기 때문이다.

todo 앱의 주요 데이터는 todo 목록이다.

todo 목록에 대한 add, delete, toggle 작업은

소스 파일 한 곳에 모아서 구현하는 것이 바람직하다.

useState로 구현한 todo 앱 예제에서는 TodoContainer 컴포넌트에 모아서 구현했고,

이 예제에서는 useTodoListState 커스텀 훅에 모아서 구현했다.

주요 데이터와, 그 데이터에 대한 작업 코드는

소스코드 파일 한 곳에 모아서 구현하는 것이 바람직한 이유는

1. 디버깅 하기 편하고

2. 유지보수 하기 편하기 때문이다.

type StatusState = {

  running: boolean,

  errMsg**?**: string}

export const statusSate = atom<StatusState>({

  key: 'status',

  default: { running: false }});

running : 백엔드 서버에 요청 보냈고, 아직 응답이 도착하지 않았으면 true

errMsg : 에러 메시지.

속성의 타입을 선언할 때, 속성 이름 옆에 **?** 문자를 붙이면, 생략할 수 있는 속성이 된다.

&& 연산자 왼쪽 항의 값이 false 이면, 즉 status.running 값이 false 이면,

&& 연산자 오른쪽 항을 실행하지 않는다.

이때 위 표현식의 값은 왼쪽항의 값이다. 즉 false 이다.

따라서 위 코드는 { false } 와 같다. 즉 false 값이 출력된다.

리액트는 출력할 값이 false/true/null/NaN 이면, 화면에 아무것도 출력하지 않는다.

&& 연산자 왼쪽 항의 값이 true 일 때만, && 연산자 오른쪽 항을 실행한다.

이때 위 표현식의 값은 오른쪽항의 값이다. 즉 <p className="msg">Loading...</p> 이다.

따라서 위 위 코드는 { <p className="msg">Loading...</p> } 이다.

<p className="msg">Loading...</p> 부분이 화면에 출력된다.

functionClock() {

  const [time, setTime] = useState<Date>(newDate())

  useEffect(() => {

    constid = setInterval(() =>setTime(newDate()), 1000)

    return () =>clearInterval(id)

  }, [])

1) 위 연두색 코드에서 useState 함수는 어떤 때 or 어떤 때마다 실행(호출) 되는가?

Clock 컴포넌트가 렌더링 되기위해 Clock 함수가 호출될 때마다

2) 위 노란색 코드에서 setInterval 함수는 어떤 때 or 어떤 때마다 실행(호출) 되는가?

Clock 컴포넌트가 mount 될 때

3) 위하늘색 코드에서 clearInterval 함수는어떤때 or 어떤때마다실행(호출)되는가?

Clock 컴포넌트가 unmount 될 때

axios.get('/students')

    .then(response => {

      students = response.data;

      return students[0].id;

     })

    .then(id => {

      axios.get('/students/' + id)

        .then(response => {

          student = response.data;

          console.log('test2', student);

        })

    })

4) 노란색콜백함수는언제호출되는가?

get('/students') 요청에 대한 백엔드 서버의 응답이 도착했을 때

5) 하늘색 콜백함수는 언제 호출되는가?

return students[0].id 이 값이 리턴되자 마자 즉시

6) 연두색 콜백함수는 언제 호출되는가?

get('/students/' + id) 요청에 대한 백엔드 서버의 응답이 도착했을 때