React.js

status 와 hook

first coding

first coding

React.js

1. status
2. Hook

first
coding

State와 생명주기

State

- State 란
  - 리액트에서의 state는 리액트 컴포넌트의 상태를 의미 (리액트 컴포넌트의 데이터)
  - **리액트 컴포넌트의 변경 가능한 데이터를 state라고 한다.**
  - state는 사전에 미리 정해진 것이 아니라 리액트 컴포넌트를 개발하는 각 개발자가 직접 정의해서 사용
    - state를 정의할 때 중요한 점은 꼭 렌더링이나 데이터 흐름에 사용되는 값만 state에 포함시켜야 한다
    - state가 변경될 경우 컴포넌트가 재렌더링되기 때문에 렌더링과 데이터 흐름에 관련 없는 값을 포함하면 컴포넌트가 다시 렌더링되어 성능을 저하 시킬 수 있다.

State

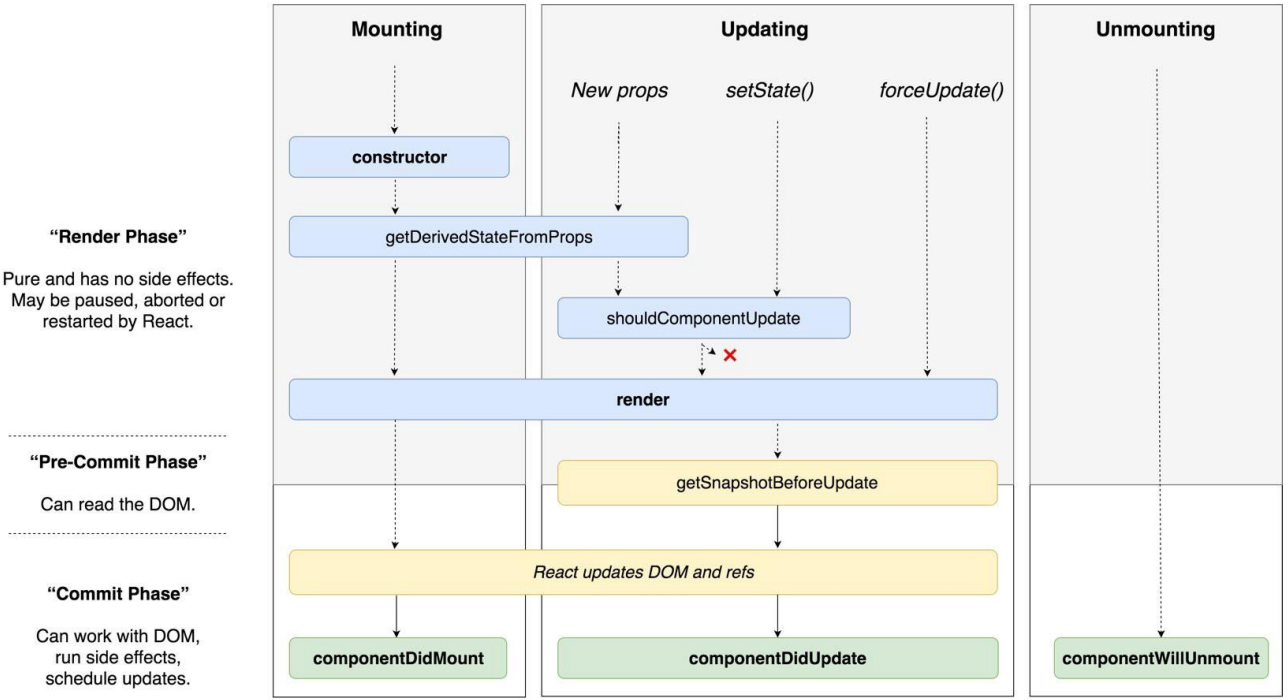
- State의 특징
  - 리액트의 state는 따로 복잡한 형태가 있는 것이 아니라, 그냥 하나의 자바스크립트 객체

```
class LikeButton extends React.Component {

  constructor(props){
    super(props)

    this.state = {
      liked : 'false'
    }

  }

}
```

- `constructor(props){}` 생성자 코드를 보면 this.state라는 부분이 현재 컴포넌트의 state를 정의하는 부분
- state는 정의된 이후 일반적인 자바스크립트 변수를 다루듯이 **직접 수정하면 안되고** setState () 함수를 이용해 수정
- state는 컴포넌트의 렌더링과 관련 있기 때문에 state를 변경하고자 할 때에는 꼭 setState ()라는 함수를 사용해야 함.

리액트 컴포넌트의 생명주기

- 리액트 컴포넌트의 생명주기
  - 리액트 컴포넌트도 생명주기를 갖고 있있으며 컴포넌트가 생성되는 시점과 소멸 시점이 정해져 있다



리액트 컴포넌트의 생명주기

- 리액트 컴포넌트의 생명주기
  - 각 과정의 하단에 초록색으로 표시된 부분은 생명주기에 따라 호출되는 클래스 컴포넌트의 Lifecycle method
    - **컴포넌트가 생성되는 시점**
      - Mounting

        ➔ 이때 컴포넌트의 constructor (생성자)가 실행, 컴포넌트의 state를 정의, componentDidMount () 호출
    - **컴포넌트 렌더링**
      - Updating ➔ componentDidUpdate () 함수가 호출
      - 컴포넌트의 props가 변경
      - setState () 함수 호출에 의해 state가 변경
      - forceUpdate()라는 강제 업데이트 함수 호출로 인해 컴포넌트가 다시 렌더링
    - **언마운트**
      - 상위 컴포넌트에서 현재 컴포넌트를 더 이상 화면에 표시하지 않게 될 때 언마운트

        ➔ 언마운트 직전에 componentWillUnmount () 호출

리액트 컴포넌트의 생명주기

- Notification.jsx

```jsx
import React from "react";

const styles = {

    wrapper: {
        margin: 8,
        padding: 8,
        display: "flex",
        flexDirection: "row",
        border: "1px solid grey",
        borderRadius: 16,
    },
    messageText: {
        color: "black",
        fontSize: 16,
    },

};
```

```jsx
class Notification extends React.Component {
    constructor(props) {
        super(props);
        this.state = {};
    }
    componentDidMount() {
        console.log(`${this.props.id} componentDidMount() called.`);
    }
    componentDidUpdate() {
        console.log(`${this.props.id} componentDidUpdate() called.`);
    }
    componentWillUnmount() {
        console.log(`${this.props.id} componentWillUnmount() called.`);
    }
    render() {
        return (
            <div style={styles.wrapper}>
                <span style={styles.messageText}>{this.props.message}</span>
            </div>
        );
    }
}
export default Notification;
```

리액트 컴포넌트의 생명주기

- Notification.jsx

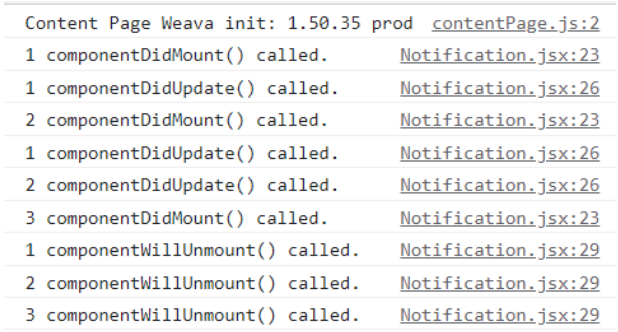
```
import React from "react";
import Notification from
"./Notification";

const reservedNotifications = [
    {
        id: 1,
        message: "안녕하세요, 오늘 일정을
알려드립니다.",
    },
    {
        id: 2,
        message: "점심식사 시간입니다.",
    },
    {
        id: 3,
        message: "이제 곧 미팅이 시작됩니
다.",
    },
];
var timer;
```

```
class NotificationList extends React.Component {
    constructor(props) {
        super(props);
        this.state = { notifications: [],  };
    }
    componentDidMount() {
        const { notifications } = this.state;
        timer = setInterval(() => {
            if (notifications.length < reservedNotifications.length) {
                const index = notifications.length;
                notifications.push(reservedNotifications[index]);
                this.setState({  notifications: notifications,  });
            } else {
                this.setState({  notifications: [], });
                clearInterval(timer);
            }
        }, 5000);
    }
    componentWillUnmount() {
        if (timer) {  clearInterval(timer);  }
    }
    render() {
        return (
            <div>
                {this.state.notifications.map((notification) => {
                    return (
                        <Notification
                            key={notification.id}
                            id={notification.id}
                            message={notification.message}
                        />
                    );
                })}
            </div>
        );
    }
}
export default NotificationList;
```

리액트 컴포넌트의 생명주기

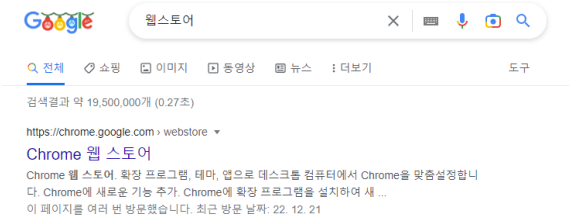
- Index.jsx

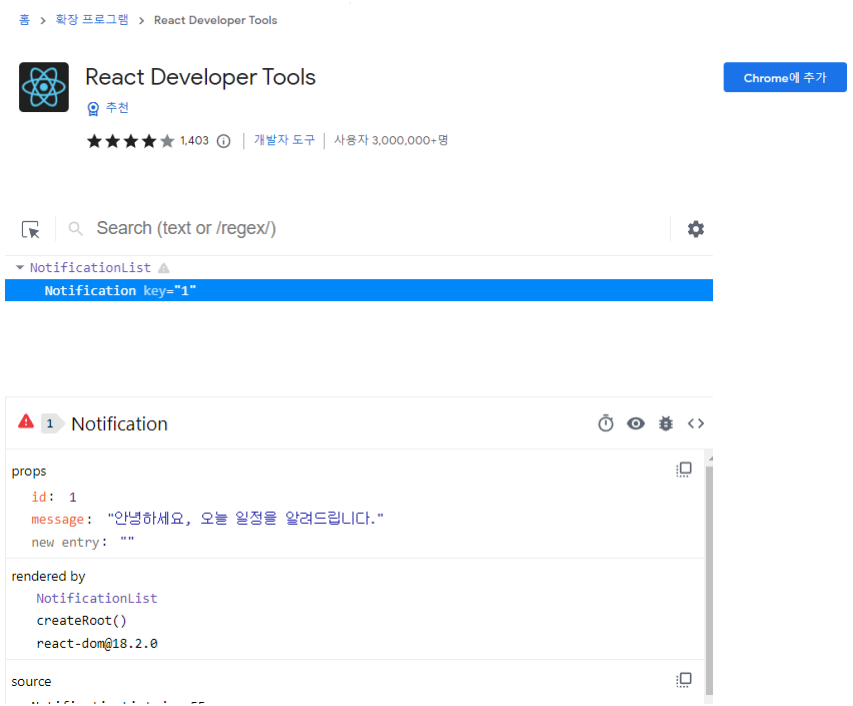
```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import reportWebVitals from './reportWebVitals';
import NotificationList from './test/NotificationList';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <NotificationList  />
);

// If you want to start measuring performance in your app, pass a
function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more:
https://bit.ly/CRA-vitals
reportWebVitals();
```



State와 생명주기 함수 사용하기

- React Developer Tools 설치





State와 생명주기 함수 사용하기

- 생명주기 정리
  - - 마운트
    - 컴포넌트가 생성될 때
    - componentDidMount()
  - 업데이트
    - 컴포넌트의 props가 변경될 때
    - setState() 함수 호출에 의해 state가 변경될 때
    - forceUpdate()라는 강제 업데이트 함수가 호출될 때
    - componentDidUpdate()
  - 언마운트
    - 상위 컴포넌트에서 현재 컴포넌트를 더 이상 화면에 표시하지 않게 될 때
    - componentWillUnmount()
  - 컴포넌트는 계속 존재하는 것이 아니라 시간의 흐름에 따라 생성되고 업데이트되다가 사라지는 과정을 겪음

first
coding

학

훅

- 훅
  - 함수 컴포넌트는 클래스 컴포넌트와는 다르게 코드도 굉장히 간결하고, 별도로 state를 정의해서 사용하거나 컴포넌트의 생명주기에 맞춰 어떤 코드가 실행되도록 할 수 없었다.
  - 함수형 컴포넌트에 이런 기능을 지원하기 위해서 나온 것이 바로 훅
  - 훅을 사용하면 함수 컴포넌트도 클래스 컴포넌트의 기능을 모두 동일하게 구현할 수 있다.
  - Hook
    - 프로그래밍에서는 원래 존재하는 어떤 기능에 마치 갈고리를 거는 것처럼 끼어 들어가 같이 수행되는 것을 의미
    - 리액트의 state와 생명주기 기능에 갈고리를 걸어 원하는 시점에 정해진 함수를 실행 하도록 만드는 것
      ➔ 이때 실행되는 함수를 훅이라고 한다.
    - 훅의 이름은 모두 use로 시작
    - 훅이 수행하는 기능에 따라서 이름을 짓게 되었는데 각 기능을 사용하겠다는 의미로 use를 앞에 붙여 개발자가 직접 커스텀 훅을 만들어서 사용할 수 있다.

훅

- 훅
  - 자주 사용되는 훅의 종류
    - useState ()
    - useEffect()
    - useMemo ()
    - useCallback()
    - useRef()

useState

- useState ()
  - 가장 대표적이고 많이 사용되는 훅으로 이름에서 알 수 있듯이 state를 사용하기 위한 훅
  - 함수 컴포넌트에서는 기본적으로 state를 제공하지 않기 때문에 state를 사용하고자 한다면 useState() 사용

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

function Counter(props){

    var count = 0;

    return (
        <di>
            <p>총 {count}번 클릭했습니다.
</p>
            <button onClick={()=> count++)}>
                Click
            </button>
        </di>
    )
}
```

- 버튼 클릭 시 카운트 값을 증가시킬 수 있다.
- **재렌더링 re-rendering이 일어나지 않아 새로운 카운트 값이 화면에 표시되지 않음.**
- 이런 경우에는 state를 사용해서 값이 바뀔때마다 재렌더링이 되도록 해야 함

useState

- useState ()

const [변수명, set함수명] = useState(초기값);

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

function Counter(props){

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

    return (
        <di>
            <p>총 {count}번 클릭했습니다. </p>
            <button onClick={()=> setCount(count+1)}>
                Click
            </button>
        </di>
    )
}
```

useEffect

- useEffect()
  - 사이드 이펙트side effect를 수행하기 위한 훅
  - 리액트에서의 사이드 이펙트는 그냥 효과 혹은 영향을 뜻하는 이펙트의 의미
    - 예를 들면 **서버에서 데이터를 받아오거나 수동으로 DOM을 변경하는 등의 작업을 의미**
    - 이런 작업을 이펙트라고 부르는 이유는 이 작업들이 다른 컴포넌트에 영향을 미칠 수 있고, 렌더링 중에는 작업이 완료될 수 없기 때문에 사이드 이펙트라 칭함.
    - **렌더링이 끝난 이후에 실행되어야 하는 작업들이다**.
  - useEffect()는 리액트의 함수 컴포넌트 생명주기 안에서 사이드 이펙트를 실행할 수 있도록 해주는 훅
    - useEffect()는 클래스 컴포넌트에서 제공하는 생명주기 함수인 componentDidMount(), componentDidUpdate() 그리고 componentWillUnmount()와 동일한 기능을 하나로 통합해서 제공
    - 그래서 useEffect() 훅만으로 위의 생명주기 함수와 동일한 기능을 수행할 수 있다.

useEffect

- useEffect()

```
useEffect(이펙트 함수, 의존성 배열);
```

  - 첫 번째 파라미터 ➔ 이펙트 함수 effect function
  - 두 번째 파라미터 ➔ 의존성 배열 an array of dependencies
    - 의존성 배열은 말 그대로 이 이펙트가 의존하고 있는 배열인데 배열 안에 있는 변수 중에 하나라도 값이 변경되었을 때 이펙트 함수가 실행.
  - 기본적으로 이펙트 함수는 처음 컴포넌트가 렌더링된 이후와 업데이트로 인한 재 렌더링 이후에 실행
    - **만약 이펙트 함수가 마운트와 언마운트시에 단 한 번씩만 실행되게 하고 싶으면, 의존성 배열에 빈 배열([])을 넣으면 된다.**
      → 해당 이펙트가 props state에 있는 어떤 값에도 의존하지 않는 것이 되므로 여러 번 실행되지 않음.
    - **의존성 배열은 생략할 수도 있는데 생략하게 되면 컴포넌트가 업데이트될 때마다 호출**
  - **useEffect() 혹은 하나의 컴포넌트에 여러 개를 사용할 수 있다.**

useEffect

- useEffect()

```
useEffect(() => {

    // 컴퍼넌트가 마운트 된 후
    // 의존성 배열에 있는 변수들 중 하나라도 값이 변경되었을 때 실행
    // 의존성 배열에 빈 배열([])을 넣으면 마운트와 언마운트시에 단 한 번씩만 실행됨
    // 의존성 배열 생략 시 컴포넌트 업데이트 시마다 실행됨

}, [의존성 변수1, 의존성 변수2, …]);
```

useEffect

- useEffect()

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


function UserStatus(props) {

    const [isOnline, setIsOnline] = useState(null);

    function handleStatus(status) {
        setIsOnline(status.isOnline);
    }

    useEffect(() => {
        ServerAPI.loginStatus(props.user.id, handleStatusChange);
        return () => {
            ServerAPI.logoutStatus(props.user.id, handleStatusChange);
        };
    });

    if (isOnline === null) {
        return '대기중 ...'
    }
    return isOnline ? '온라인' : '오프라인';
}
```

useEffect()에서 먼저 ServerAPI를 사용하여 사용자의 상태를 확인

이후 함수를 하나 리턴하는데
해당 함수 안에는 로그아웃 하는 API를 호출하도록 되어 있다.

useEffect()에서 리턴하는 함수는
컴포넌트가 마운트 해제될 때 호출

➔ useEffect()의 리턴 함수의 역할은 componentWillUnmount () 함수가 하는 역할과 동일합니다.

메모이제이션

- **메모이제이션 (Memoization)**
  - 뒤에서 다룰 useMemo () 와 useCallback() 훅에서는 메모이제이션 개념이 반영된 기능.
  - 메모이제이션
    - 은 최적화를 위해서 사용하는 개념
    - 비용이 높은(연산량이 많이 드는) 함수의 호출 결과를 저장해 두었다가, 같은 입력 값으로 함수를 호출하면 새로 함수를 호출하지 않고 이전에 저장해놨던 호출 결과를 바로 반환하는 것
    - 이렇게 하면 결과적으로 함수 호출 결과를 받기까지 걸리는 시간도 짧아질뿐더러 불필요한 중복 연산도 하지 않기 때문에 컴퓨터의 자원(CPU, Memory 등)을 적게 쓰게 된다.
    - Memoized value ➔ 메모이제이션이 된 결과 값

useMemo

- useMemo ()
  - Memoized value를 리턴하는 훅
    - 파라미터로 Memoized value 를 생성하는 create 함수와 의존성 배열을 받는다.
    - 의존성 배열에 들어있는 변수가 변했을 경우에만 새로 create 함수를 호출하여 결과값을 반환하며, 그렇지 않은 경우에는 기존 함수의 결과값을 그대로 반환.
    - useMemo() 훅을 사용하면 컴포넌트가 다시 렌더링될 때마다 연산량이 높은 작업을 반복하는 것을 피할 수 있다.

      ➔ 빠른 렌더링 속도 가능

```
const memoizedValue = useMemo(
    () => {
        // 연산량이 높은 작업을 수행하여 결과를 반환
        return computeExpensiveValue(의존성 변수1, 의존성 변수);
    },
    [의존성 변수 1, 의존성 변수2]
);
```

useMemo

- useMemo ()
  - useMemo () 훅을 사용할 때 기억 할 점은 **useMemo ()로 전달된 함수는 렌더링이 일어나는 동안 실행된다는 점**
    - 일반적으로 렌더링이 일어나는 동안 실행돼서는 안될 작업을 useMemo ()의 함수에 넣으면 안 됨.
    - 예를 들면 서버에서 데이터를 받아오거나 수동으로 DOM을 변경하는 작업 등은 렌더링이 일어나는 동안 실행돼서는 안되기 때문에 useMemo () 훅의 함수에 넣으면 안 되고 useEffect() 훅을 사용해야 합니다.
  - 다음 코드와 같이 의존성 배열을 넣지 않을 경우 렌더링이 일어날 때마다 매번 함수가 실행된다.

```
const memoizedValue = useMemo(
    () => {
        return computeExpensiveValue(a, b);
    }
);
```

  - 의존성 배열에 빈 배열을 넣게 되면 컴포넌트 마운트 시에만 함수가 실행된다.

```
const memoizedValue = useMemo(
    () => {
        return computeExpensiveValue(a, b);
    },    []
);
```

useCallback

- useCallback()
  - useMemo () 훅과 유사한 역할을 함.
  - 차이점은 값이 아닌 함수를 반환한다는 것
  - 함수와 의존성 배열을 파라미터로 받고, useCallback () 훅에서는 파라미터로 받는 함수를 콜백callback이라 함.
  - 의존성 배열에 있는 변수 중 하나라도 변경되면 Memoized (메모이제이션이 된) 콜백 함수를 반환합니다.

```
const memoizedCallback = useCallback(
    () => {
          doSomething(의존성 변수1, 의존성 변수);
    },
    [의존성 변수 1, 의존성 변수2]
);
```

- 의존성 배열에 따라 Memoized 값을 반환하기 때문에
  useCallback(function, dependencies)은 useMemo(() => function, dependencies)와 동일하다고 볼 수 있습니다.

useCallback

- useCallback()
  - 만약 useCallback() 훅을 사용하지 않고 컴포넌트 내에 함수를 정의한다면 매번 렌더링이 일어날 때마다 함수가 새로 정의됩니다.
    - 따라서 useCallback() 훅을 사용하여 특정 변수의 값이 변한 경우에만 함수를 다시 정의하도록 해서 불필요한 반복 작업을 없애주는 것입니다.
    - 예를 들어 useCallback() 훅을 사용하지 않고 컴포넌트 내에서 정의한 함수를 자식 컴포넌 트에 props로 넘겨 사용하는 경우, 부모 컴포넌트가 다시 렌더링이 될 때마다 매번 자식 컴포 넌트도 다시 렌더링됩니다.
    - 하지만 useCallback() 훅을 사용하면 특정 변수의 값이 변한 경우에만 함수를 다시 정의하게 되므로, 함수가 다시 정의되지 않는 경우에 자식 컴포넌트도 재 렌더링이 일어나지 않습니다.

useRef

- useRef()
  - 레퍼런스Reference를 사용하기 위한 훅
    - 리액트에서 레퍼런스란 특정 컴포넌트에 접근할 수 있는 객체를 의미.
    - useRef () 혹은 바로 레퍼런스 객체ref object를 반환
    - 레퍼런스 객체에는 .current라는 속성이 있는데 이것은 현재 레퍼런스(참조)하고 있는 엘리먼트를 가리킨다.

```
const refContainer = useRef(초기값);
```

      - 위와 같이 useRef () 훅을 사용하면 파라미터로 들어온 초기값 initial value 으로 초기화된 레퍼런스 객체를 반환
      - 초기값이 null이라면 .current의 값이 null 인 레퍼런스 객체가 반환
      - 반환된 레퍼런스 객체는 컴포넌트의 라이프타임 전체에 걸쳐서 유지.
        - » 즉, 컴포넌트가 마운트 해제 전까지는 계속 유지된다
    - useRef () 혹은 변경 가능한 .current 라는 속성을 가진 하나의 상자.

useRef

- useRef()
  - useRef () 훅을 사용하여 버튼 클릭 시 <input>에 포커스focus를 하도록 하는 예제 코드

```
import React, { useState, useEffect, useRef } from "react"

function TextInputWithFocusButton(prps) {

    const inputElem = useRef("null");

    const onButtonClick = () => {
        // current`는 마운트된 input element 를 가리킴
        inputElem.current.focus();
        inputElem.current.value='new Text'
    }

    return (
        <>
            <input ref={inputElem} type="text" />
            <button onClick={onButtonClick}>Focus the input</button>
        </>
    );
}

export default TextInputWithFocusButton;
```

훅의 규칙

- **첫 번째 규칙은 혹은 무조건 최상위 레벨Top Level에서만 호출해야 한다는 것**
  - 여기에서 말하는 최상위 레벨은 리액트 함수 컴포넌트의 최상위 레벨을 의미.
    ➔ 반복문이나 조건문 또는 중첩된 함수들 안에서 훅을 호출하면 안 된다.
  - 이 규칙에 따라서 혹은 컴포넌트가 렌더링될 때마다 매번 같은 순서로 호출되어야 한다.
  - 그렇게 해야 리액트가 다수의 useState () 훅과useEffect() 훅의 호출에서 컴포넌트의 state를 올바르게 관리할 수 있게 된다.
- **두 번째 규칙은 리액트 함수 컴포넌트에서만 훅을 호출해야 한다는 것입니다.**
  - 그렇기 때문에 일반적인 자바스크립트 함수에서 훅을 호출하면 안 된다.
  - 혹은 리액트 함수 컴포넌트에서 호출하거나 직접 만든 커스텀 Custom Hook에서만 호출할 수 있다.
  - 이 규칙에 따라 리액트 컴포넌트에 있는 state와 관련된 모든 로직은 소스코드를 통해 명확하게 확인이 가능해야 한다.

커스텀 훅 만들기

- **컴스텀 훅**
  - 추가적으로 필요한 기능이 있다면 직접 훅을 만들어서 사용할 수 있습니다.
  - 이것을 커스텀 혹 이라고 부르는데 커스텀 훅을 만드는 이유는 여러 컴포넌트에서 반복적으로 사용되는 로직을 혹으로 만들어 재사용하기 위함

커스텀 훅 만들기

- 컴스텀 훅이 필요한 상황

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

function UserStatus(props) {

    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {

        function handleStatus(status) {
            setIsOnline(status.isOnline);
        }

        ServerAPI.loginStatus(props.user.id, handleStatusChange);
        return () => {
            ServerAPI.logoutStatus(props.user.id, handleStatusChange);

        };
    });
    if (isOnline === null) {
        return '대기중 ...'
    }
    return isOnline ? '온라인' : '오프라인';
}

export default UserStatus;
```

커스텀 훅 만들기

- 컴스텀 훅이 필요한 상황

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

function UserStatus(props) {

    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {
        function handleStatus(status) {
            setIsOnline(status.isOnline);
        }
        ServerAPI.loginStatus(props.user.id, handleStatusChange);
        return () => {
            ServerAPI.logoutStatus(props.user.id, handleStatusChange);
        };
    });

    return (
        <li style={{ color: isOnline ? 'green' : 'black'}}>{props.user.name}</li>
    );

}

export default UserStatus;
```

커스텀 훅 만들기

- 커스텀 훅 추출하기
  - UserStatus와 useState(), useEffect() 훅을 사용하는 부분이 동일한 것을 볼 수 있다.
    - 두 개의 자바스크립트 함수에서 하나의 로직을 공유하도록 하고 싶을 때 새로운 함수를 하나 만드는 방법을 사용합니다.
    - **커스텀 훅은 use로 시작하는 이름으로 만들고 내부에서 다른 훅을 호출하는 하나의 자바스크립트 함수**
    - 아래 코드는 중복되는 로직을 useUserStatus ()라는 커스텀 훅으로 추출해낸 것입니다.

```
function useUserStatus(userId){
    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
            }
    });

    ServerAPI.loginStatus(userId, handleStatusChange);
    return() => {
        ServerAPI.logoutStatus(userId, handleStatusChange);
    };

    return isOnline;
}
```

다른 컴포넌트 내부에서와 마찬가지로 다른 훅을 호출하는 것은 무조건 커스텀 훅의 최상위 레벨에서만 해야 한다.

커스텀 훅은 특별한 규칙이 없다.
➔ 파라미터로 무엇을 받을지, 어떤 것을 리턴해야 할지를 개발자가 직접 정의

커스텀 훅 만들기

- 커스텀 훅 추출하기
  - UserStatus와 useState(), useEffect() 훅을 사용하는 부분이 동일한 것을 볼 수 있다.

```
function UserStatus(props) {
    const isOnline = useUserStatus(props.user.id);
    if (isOnline === null) {
        return '대기중 ...'
    }
    return isOnline ? '온라인' : '오프라인';
}

function UserListItem(userId){
    const isOnline = useUserStatus(props.user.id);
    return (
        <li style={{ color: isOnline ? 'green' : 'black'}}>{props.user.name}</li>
    );
}
```

- 커스텀 훅의 이름은 꼭 use로 시작해야 한다.

  ➔ 이름이 use로 시작하지 않는다면 특정 함수의 내부에서 훅을 호출하는지 를 알 수 없기 때문이다

커스텀 훅 만들기

- **커스텀 훅 추출하기**
  - 같은 커스텀 훅을 사용하는 두 개의 컴포넌트는 state를 공유하는 것이 아니라 단순히 state와 연관된 로직을 재사용이 가능하게 만든 것
  - 여러 개의 컴포넌트에서 하나의 커스텀 훅을 사용할 때에 컴포넌트 내부에 있는 모든 state와 effects는 전부 분리되어 있다.
  - 커스텀 훅의 state를 분리 방법
    - 각각의 커스텀 훅 호출에 대해서 분리된 state를 얻게 된다.
    - 위의 예제 코드에서 useUserStatus () 훅을 직접 호출하는 것 처럼 리액트의 관점에서는 컴포넌트에서 useState () 와 useEffect () 훅을 호출하는 것과 동일한 것.
    - 하나의 컴포넌트에서 useState () 와 useEffect() 훅을 여러 번 호출 할 수 있는 것처럼 각 커스텀 훅의 호출 또한 완전히 독립적이라고 볼 수 있다.

커스텀 훅 만들기

- 훅들 사이에서 데이터를 공유
  - 훅을 호출함에 있어 각 호출은 완전히 독립적

```
function ChatUserSelector(props) {

    const [userId, setUserId] = useState(1);
    const isUserOnline = useUserStatus(userId);

    return (
        <>
            {/* <Circle color={isUserOnline ? 'green' : 'red'} /> */}
            <select
                value={userId}
                onChange={event => setUserId(Number(event.target.value))}
            >
                {userList.map(user => (
                    <option key={user.id} value={user.id}>
                        {user.name}
                    </option>
                    ))}
                astCom </select>
        </>
    );

}
```

```
const [userId, setUserId] = useState(1);
const isUserOnline = useUserStatus(userId);
```

useState () 훅을 사용해서 현재 선택된 사용자의 아이디를 저장하기 위해 userId라는 state를 만듬.
→ 이후 useUserStatus 훅의 파라미터로 사용

이렇게 하면 setUserId 함수를 통해 userId가 변경될 때마다, useUserStatus 혹은 이전에 선택된 사용자를 로그인 상태연결을 취소하고 새로 선택된 사용자의 온라인 여부를 확인하게 됩니다. 훅들 사이에서는 이러한 방법으로 데이터를 공유할 수 있습니다.

first coding

훅을 사용한 컴포넌트 개발 : useCounter() 커스텀 훅 만들기

훅을 사용한 컴포넌트 개발 : useCounter() 커스텀 훅 만들기

- useCounter.jsx
  - 초기 카운트 값을 받아서 count라는 이름의 state를 생성하고, count의 증 가/감소 처리를 하는 함수를 제공하는 훅 ➔ useCounter() 훅으로 어떤 함수 컴포넌트에서든 카운트 기능을 쉽게 사용

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

function useCounter(initialValue) {

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

    const increaseCount = () => setCount((count) => count + 1);

    const decreaseCount = () => setCount((count) => Math.max(count - 1, 0));

    return [count, increaseCount, decreaseCount];
}

export default useCounter;
```

훅을 사용한 컴포넌트 개발 : Accormodate 컴포넌트 만들기

- Accommodate.jsx

```jsx
import React, { useState, useEffect } from "react";
import useCounter from "./useCounter";

const MAX_CAPACITY = 10;

function Accommodate(props) {
    const [isFull, setIsFull] = useState(false);
    const [count, increaseCount, decreaseCount] = useCounter(0);

    useEffect(() => {
        console.log("=======================");
        console.log("useEffect() is called.", `isFull: ${isFull}`);
    });

    useEffect(() => {
        setIsFull(count >= MAX_CAPACITY);
        console.log(`Current count value: ${count}`);
    }, [count]);

    return (
        <div style={{ padding: 16 }}>
            <p>{`총 ${count}명 입장했습니다.`}</p>
            <button onClick={increaseCount} disabled={isFull}> 입장 </button>
            <button onClick={decreaseCount}>퇴장</button>
            {isFull && <p style={{ color: "red" }}>정원이 가득찼습니다.</p>}
        </div>
    );
}

export default Accommodate;
```

훅을 사용한 컴포넌트 개발 : Accormodate 컴포넌트 만들기

- Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import Accommodate from './hooktest/Accommodate';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Accommodate  />
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```