



React Hook

1. React Hooks

리액트의 생명주기(Life Cycle)

2. 자주 사용되는 훅

useEffect

useNavigate

useRef

3. Custom hook

1. React Hooks

지금까지 스테이트 관리를 위해 `useState` 를 사용했는데, 사실 이 함수는 리액트 훅입니다. 훅이란 건 뭘까요? 이름이 어렵게 느껴지지만, 사실은 기존에 사용하던 클래스 컴포넌트의 여러 기능들을 자연스럽게 함수형 컴포넌트에서 사용하기 위한 장치입니다.


조금 더 자세히 얘기하자면, 기존의 클래스형 컴포넌트는 스테이트를 직접 다룰 수 있을 뿐만 아니라 컴포넌트의 생명주기, 즉 라이프사이클도 직접적으로 다룰 수 있었습니다. 라이프사이클에 대해서는 바로 다음에 배우도록 하겠습니다. 다시 본론으로 돌아와서, 클래스형 컴포넌트는 자바스크립트 클래스로 구성되어있기 때문에 기본적인 자바스크립트 문법을 잘 알아야 하기 때문에 진입 장벽이 존재했습니다. 게다가 간결하지 못한 문법 때문에 코드를 알아보기도 힘들었습니다.

하지만 함수형 컴포넌트를 사용하면 쉽고 간결하게 컴포넌트를 만들 수 있지만, 스테이트와 생명주기 관리가 어려워지는 문제가 있었습니다. 리액트 16.8버전 이후부터는 이 문제를 리액트 훅을 추가해 해결했습니다. 즉 함수형 컴포넌트가 기존의 클래스형 컴포넌트를 완전히 대체할 수 있도록 하는 기능입니다.

여기서는 몇 가지 자주 사용되는 리액트 훅에 대해서 다루겠습니다. 이 외에도 리액트에서는 정말 다양한 훅이 정의되어 있습니다. 전체 훅에 대한 자세한 설명은 공식 문서를 참고하세요.

Hook의 개요 - React

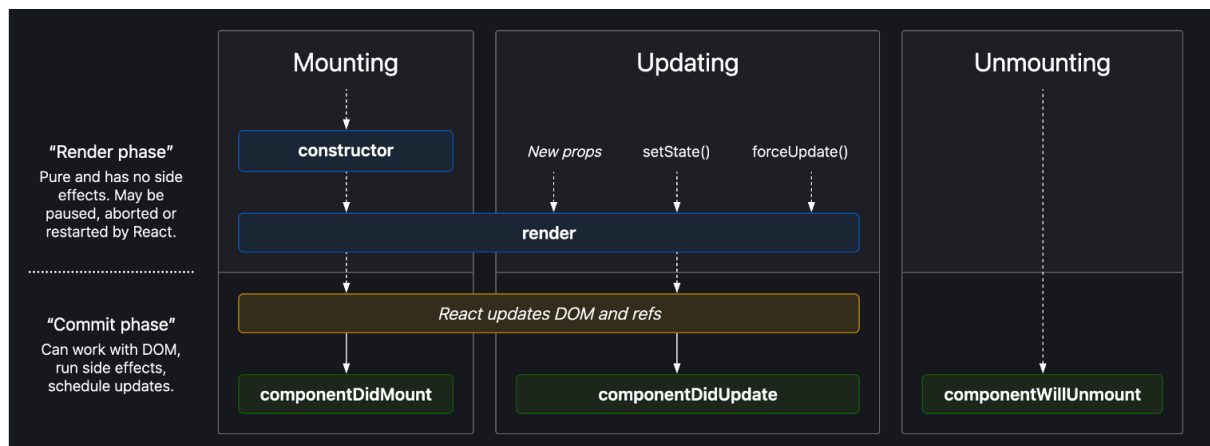
Hook 은 React 버전 16.8부터 React 요소로 새로 추가되었습니다. Hook을 이용하여 기존 Class 바탕의 코드를 작성할 필요 없이 상태 값과 여러 React의 기능을 사용할 수 있습니다. useState 는 우리가

 <https://ko.reactjs.org/docs/hooks-intro.html>



리액트의 생명주기(Life Cycle)

리액트는 컴포넌트가 화면에 렌더링되는 과정을 생명주기를 이용해 관리합니다. 실제로는 이 생명주기가 굉장히 복잡하게 구성되어 있지만, 간단하게 정리하면 다음과 같이 정리할 수 있습니다.



출처: <https://projects.wojtekmaaj.pl/react-lifecycle-methods-diagram/>

먼저 생명주기는 크게 3개의 페이지로 구성됩니다. 마운팅, 업데이트, 언마운팅. 마운팅은 DOM에 컴포넌트가 등록되는 과정, 업데이트는 등록된 컴포넌트의 내용이 업데이트되는 과정, 언마운팅은 DOM에서 사라지는 과정입니다.

마운팅 페이지에서 우리가 만든 컴포넌트가 DOM에 등록되기 전에 렌더링을 거치고, 렌더링 과정에서 업데이트가 진행되어 props나 스테이트의 내용이 컴포넌트에 업데이트됩니다. 그 다음 실제 브라우저의 DOM에 컴포넌트가 등록됩니다. 그러다 더이상 컴포넌트가 사용되지 않게 되면 언마운팅되게 됩니다.

리액트의 모든 컴포넌트는 이러한 생명주기를 가지고 있습니다. 이 생명주기를 알아야 하는 이유는, 렌더링이 끝나기 전, 즉 브라우저 DOM에 업데이트 되기 전까지는 리액트에서 관리하는 영역이지만, 그 이후에는 브라우저 그리고 화면에 나타나는 과정이기 때문입니다.

2. 자주 사용되는 훅

useEffect

컴포넌트가 마운트되어 DOM이 변경된 다음에 실행됩니다. 즉 컴포넌트 마운트(DOM 변경) → 렌더링 → `useEffect` 실행 의 순서로 이루어집니다.

`useEffect` 는 보통 다음과 같은 작업을 수행합니다.

- `props` 에 속한 값을 컴포넌트의 로컬 변수로 선언
- API 호출
- 서드파티 라이브러리 사용

▼ 참고

`useEffect` 는 파라미터로 함수를 입력받는데, 이 함수를 `effect`라고 부릅니다. 두번째 입력으로 `deps` 라고 불리는 배열을 받습니다. 이 배열은 의존값 목록인데, 비어있다면 컴포넌트가 처음 렌더링될 때만 `useEffect` 가 호출됩니다. 그리고 첫 번째 입력으로 받은 함수는 다시 함수를 리턴할 수 있습니다. 이를 `cleanup` 함수라고 부르고, 컴포넌트에 대한 뒷정리를 해주는 부분입니다. 만일 `deps` 배열이 비어있다면 컴포넌트가 unmount 되어 사라질 때 `return` 하는 함수 부분이 실행됩니다.

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

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

  useEffect(() => {
    console.log("useEffect");
    document.title = `You clicked ${count} times`;
  });
  return (
    <>
      <p>You clicked {count} times</p>
      <button
        onClick={() => {
          console.log("Click");
          setCount(count + 1);
        }}
      >
        Click me
      </button>
    </>
  );
}

export default Input;
```

useNavigate

여기서부터는 `react-router-dom` 을 설치해줘야 합니다.

```
npm install react-router-dom
```

아래와 같이 예제 코드를 작성합니다.

```
import React from "react";
import {
  BrowserRouter,
  Routes,
  Route,
  Link,
  useNavigate,
} from "react-router-dom";

export default function BasicExample() {
  return (
    <BrowserRouter>
      <button>
        <Link to="/home">Home</Link>
      </button>
      <button>
        <Link to="/">Index</Link>
      </button>
      <Routes>
        <Route exact path="/" element={<Index />} />
        <Route path="/home" element={<Home />} />
      </Routes>
    </BrowserRouter>
  );
}

function Index() {
  let navigate = useNavigate();
  function handleClick() {
    navigate("/home");
  }

  return (
    <>
      <h2>Index</h2>
      <button type="button" onClick={handleClick}>
        Go home
      </button>
    </>
  );
}

function Home() {
  return (
    <>
```

```

    <h2>Home</h2>
    <p>This is home</p>
  </>
);
}

```

useRef

자바스크립트에서는 특정 DOM을 선택할 때 다음과 같이 하면 됩니다.

```

document.getElementById("#foo.bar")
document.querySelector('#foo.bar')

```

리액트에서 비슷한 작업을 해야 하는 경우 `useRef` 함수를 사용해 다음과 같이 할 수 있습니다. 아래 예제에서는 특정 함수가 호출되는 경우, 포커스를 해당 엘리먼트에 맞추고 있습니다.

```

//Input.js
import { useState, useRef } from "react";

export default function App() {
  const [text, setText] = useState("");
  const onChange = (event) => {
    const target = event.target;
    setText(target[1]);
  };

  const newInput = useRef();
  const onReset = () => {
    setText("");
    newInput.current.focus();
  };
  return (
    <div>
      <input ref={newInput} onChange={onChange} value={text}></input>
      <button onClick={onReset}>Reset</button>
    </div>
  );
}

```

3. Custom hook

Custom hook이란, 리액트에 정의되어있지 않은 나만의 Hook을 만들어 사용하는 것을 의미합니다. 아래 예제 코드를 보겠습니다.

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

function App() {
  const [click1, setState1] = useState(false);
  const toggle1 = () => setState1((click1) => !click1);
  const [click2, setState2] = useState(false);
  const toggle2 = () => setState2((click2) => !click2);
  const [click3, setState3] = useState(false);
  const toggle3 = () => setState3((click3) => !click3);
  const [click4, setState4] = useState(false);
  const toggle4 = () => setState4((click4) => !click4);
  const [click5, setState5] = useState(false);
  const toggle5 = () => setState5((click5) => !click5);

  return (
    <>
      <button onClick={toggle1}>{click1 ? "Hello" : "Goodbye"}</button>
      <button onClick={toggle2}>{click2 ? "Hello" : "Goodbye"}</button>
      <button onClick={toggle3}>{click3 ? "Hello" : "Goodbye"}</button>
      <button onClick={toggle4}>{click4 ? "Hello" : "Goodbye"}</button>
      <button onClick={toggle5}>{click5 ? "Hello" : "Goodbye"}</button>
    </>
  );
}

export default App;
```

버튼을 클릭하면, state가 반전되는 역할을 수행하는 setter 함수를 사용하고 있습니다. 문제는 이 패턴이 5개의 버튼에 동일하게 반복된다는 점입니다. 물론 이렇게 간단한 경우에는 단순히 반복해서 코드를 작성해도 됩니다. 하지만 코드가 복잡해지고, 훨씬 많은 곳에서 이 코드를 사용한다면 이 기능을 별도의 함수, 즉 Hook으로 만들어서 관리하는 것이 훨씬 좋은 방법입니다.

위의 내용을 custom hook으로 만들어 적용해보면 아래와 같습니다.

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

function App() {
  const [click1, setClick1] = useToggle();
  const [click2, setClick2] = useToggle();
  const [click3, setClick3] = useToggle();
  const [click4, setClick4] = useToggle();
  const [click5, setClick5] = useToggle();

  return (
    <>
      <button onClick={setClick1}>{click1 ? "Hello" : "Goodbye"}</button>
      <button onClick={setClick2}>{click2 ? "Hello" : "Goodbye"}</button>
      <button onClick={setClick3}>{click3 ? "Hello" : "Goodbye"}</button>
      <button onClick={setClick4}>{click4 ? "Hello" : "Goodbye"}</button>
      <button onClick={setClick5}>{click5 ? "Hello" : "Goodbye"}</button>
    </>
  );
}
```

```
    );  
  }  
  
  const useToggle = (initialState = false) => {  
    const [state, setState] = useState(initialState);  
    const toggle = () => setState((state) => !state);  
  
    return [state, toggle];  
  };  
  
  export default App;
```

`useToggle` 이라는 Hook 덕분에 코드가 훨씬 직관적이고 간결해졌습니다. 게다가 Hook의 기능을 수정하면 모든 버튼에 변경 내용이 적용되기 때문에 유지보수가 훨씬 편리합니다.