내장 React Hook

*Hook*을 사용하면 컴포넌트에서 다양한 React 기능을 사용할 수 있습니다. 내장된 Hook을 이용하거나 이를 결합하여 자신만의 Hook을 만들 수 있습니다. 이 페이지에는 React에 내장된 모든 Hook이 나열되어 있습니다.

State Hooks

*State*를 통해 컴포넌트는 사용자 입력과 같은 정보를 "기억"할 수 있습니다. 예를 들어, 폼 컴포넌트는 State를 사용하여 입력값을 저장할 수 있고, 이미지 갤러리 컴포넌트는 State를 사용하여 선택한 이미지 인덱스를 저장할 수 있습니다.

컴포넌트에 State를 추가하려면, 다음 Hook 중 하나를 사용하세요.

- `useState` 는 직접 업데이트할 수 있는 State 변수를 선언합니다.
- `useReducer` 는 Reducer 함수 내부의 업데이트 로직을 사용하여 State 변수를 선언합니다.

```
function ImageGallery() {
  const [index, setIndex] = useState(0);
  // ...
```

Context Hooks

*Context*는 컴포넌트가 Props를 전달하지 않고도 멀리 있는 부모 컴포넌트로부터 정보를 받을 수 있게 해줍니다. 예를 들어, 애플리케이션의 최상위 컴포넌트는 현재 UI 테마를 아래의 모든 컴포넌트에 깊이와 상관없이 전달할 수 있습니다.

- `useContext` 는 Context를 읽고 구독합니다.

```
function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

Ref Hooks

*Ref*를 사용하면 컴포넌트가 DOM 노드나 Timeout ID와 같이 렌더링에 사용되지 않는 일부 정보를 보유할 수 있습니다. State와 달리, Ref는 업데이트를 해도 컴포넌트가 다시 렌더링 되지 않습니다. Ref는 React 패러다임의 "탈출구"입니다. 내장된 브라우저 API와 같이, React가 아닌 시스템으로 작업해야 할 때 유용합니다.

- `useRef` 는 Ref를 선언합니다. 여기에는 어떤 값이라도 담을 수 있지만, 대부분 DOM 노드를 담는 데 사용됩니다.
- `useImperativeHandle` 을 사용하면 컴포넌트에 노출되는 Ref를 커스텀할 수 있습니다. 이는 드물게 사용됩니다.

```
function Form() {
  const inputRef = useRef(null);
  // ...
```

Effect Hooks

*Effect*를 통해 컴포넌트를 외부 시스템에 연결하고 동기화할 수 있습니다. 여기에는 네트워크, 브라우저 DOM, 애니메이션, 다른 UI 라이브러리를 사용하여 작성된 위젯, 기타 React가 아닌 코드를 다루는 것이 포함됩니다.

- `useEffect` 는 컴포넌트를 외부 시스템에 연결합니다.

```
function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(roomId);
    connection.connect();
    return () => connection.disconnect();
```

```
  }, [roomId]);
  // ...
```

Effect는 React 패러다임의 "탈출구"입니다. 애플리케이션의 데이터 흐름을 조정하기 위해 Effect를 쓰지 마세요. 외부 시스템과 상호작용하지 않는다면, Effect가 필요하지 않을 수도 있습니다.

타이밍에서 차이가 있는 `useEffect` 의 두 가지 드물게 사용되는 변형이 있습니다.

- `useLayoutEffect` 는 브라우저가 화면을 다시 그리기 전에 실행됩니다. 여기에서 레이아웃을 계산할 수 있습니다.
- `useInsertionEffect` 는 React가 DOM을 변경하기 전에 실행됩니다. 라이브러리는 여기에 동적 CSS를 삽입할 수 있습니다.

Performance Hooks

재렌더링 성능을 최적화하는 일반적인 방법은 불필요한 작업을 건너뛰는 것입니다. 예를 들어, 이전 렌더링 이후 데이터가 변경되지 않은 경우 캐시된 계산을 재사용하거나 재렌더링을 건너뛰도록 React에 지시할 수 있습니다.

계산과 불필요한 재렌더링을 건너뛰려면 다음 Hook 중 하나를 사용하세요.

- `useMemo` 를 사용하면 비용이 많이 드는 계산 결과를 캐시할 수 있습니다.
- `useCallback` 을 사용하면 함수 정의를 최적화된 컴포넌트에 전달하기 전에 캐시할 수 있습니다.

```
function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

화면을 실제로 업데이트해야 하므로 재렌더링을 건너뛸 수 없는 경우도 있습니다. 이 경우, 동기식이어야 하는 Blocking 업데이트(예: Input에 입력)와 사용자 인터페이스를 차단할 필요가 없는 Non-Blocking 업데이트(예: 차트 업데이트)를 분리하여 성능을 향상시킬 수 있습니다.

렌더링 우선순위를 지정하려면, 다음 Hook 중 하나를 사용하세요.

- `useTransition` 을 사용하면 State 전환을 Non-Blocking으로 표시하고, 다른 업데이트가 이를 중단하도록 허용할 수 있습니다.
- `useDeferredValue` 를 사용하면 UI의 중요하지 않은 부분에 대한 업데이트를 지연하고, 다른 부분이 먼저 업데이트되도록 할 수 있습니다.

Other Hooks

다음 Hook은 대부분 라이브러리 작성자에게 유용하며 애플리케이션 코드에서는 일반적으로 사용되지 않습니다.

- `useDebugValue` 를 사용하면 커스텀 Hook에 대해 React 개발자 도구에 표시하는 레이블을 커스텀할 수 있습니다.
- `useId` 를 사용하면 컴포넌트가 고유 ID를 자신과 연결할 수 있습니다. 일반적으로 접근성 API와 함께 사용됩니다.
- `useSyncExternalStore` 를 사용하면 컴포넌트가 외부 저장소를 구독할 수 있습니다.
- `useActionState` 를 사용하면 액션을 통해 State를 관리할 수 있습니다.

나만의 Hooks

또한 자바스크립트 함수로 나만의 커스텀 Hook을 정의할 수도 있습니다.

이전
개요

다음
useActionState

커뮤니티

더 보기