



9장 후



9.1 리액트 후

후 추가 전 **리액트 16.8 버전** - 클래스 컴포넌트에서만 상태를 가질 수 있었다.

클래스 컴포넌트에서는 `componentDidMount`, `componentDidUpdate` 와 같이 하나의 생명주기 함수에서만 상태 업데이트에 따른 로직을 실행시킬 수 있었다.

! 모든 상태를 하나의 함수 내에서 처리

! 비슷한 로직을 가진 상태 업데이트 및 사이드 이펙트 처리의 불편함

▼ 클래스 컴포넌트 문제 코드

```
componentDidMount() {
  this.props.updateCurrentPage(routeName);
  this.didFocusSubscription = this.props.navigation.addListener(
    add focus handler to navigation */});
  this.didBlurSubscription = this.props.navigation.addListener(
    blur handler to navigation */});
}

componentWillUnmount() {
  if (this.didFocusSubscription !== null) {
    this.didFocusSubscription();
  }
  if (this.didBlurSubscription !== null) {
    this.didBlurSubscription();
  }
  if (this._screenCloseTimer !== null) {
    clearTimeout(this._screenCloseTimer);
    this._screenCloseTimer = null;
  }
}


componentDidUpdate(prevProps) {
  if (this.props.currentPage !== routeName) return;
```

```

    if (this.props.errorResponse !== prevProps.errorResponse) {/
*/}
    else if (this.props.logoutResponse !== prevProps.logoutRespo
response */}
    else if (this.props.navigateByType !== prevProps.navigateByT
navigateByType change */}

    // Handle other prop changes here
}

```

 리액트 훅

- 함수 컴포넌트에서도 클래스 컴포넌트와 같이 컴포넌트의 생명주기에 맞춰 로직을 실행할 수 있게 되었다.
- 비즈니스 로직 재사용, 작은 단위로 코드 분할하여 테스트, 사이드 이펙트와 상태를 관심사에 맞게 분리하여 구성

1. useState

리액트 함수 컴포넌트에서 상태를 관리하기 위한 훅

```

import { useState } from "react";

const MemberList = () => {
  const [memberList, setMemberList] = useState([
    {
      name: "KingBaedal",
      age: 10,
    },
    {
      name: "MayBaedal",
      age: 9,
    },
  ]);

  // 3. 🚀 addMember 함수를 호출하면 sumAge는 NaN이 된다
  const sumAge = memberList.reduce((sum, member) => sum + member

```

```
// 1. addMember 함수 호출
const addMember = () => {
  // 2. 리렌더링
  setMemberList([
    ...memberList,
    {
      name: "DokgoBaedal",
      agee: 11,
    },
  ]);
};
```

! addMember 함수가 호출되면, React는 상태 변경을 감지하고 컴포넌트를 **리렌더링**한다. 이때 새로운 memberList로 sumAge를 다시 계산한다. → 새 멤버의 **agee** 속성 때문에 age는 **undefined**가 되어 sumAge는 **NaN**이 된다.

👩 TS : 컴파일타임에 타입 에러 발견 가능

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

interface Member {
  name: string;
  age: number;
}

const MemberList = () => {
  const [memberList, setMemberList] = useState<Member[]>([]);

  // member의 타입이 Member 타입으로 보장된다
  const sumAge = memberList.reduce((sum, member) => sum + member

  const addMember = () => {
    // 🚨 Error: Type 'Member | { name: string; agee: number; }'
    // is not assignable to type 'Member'
    setMemberList([
      ...memberList,
      {
        name: "DokgoBaedal",
        agee: 11,
      },
    ],
```

```

    });
};

return (
    // ...
);
};

```

2. 의존성 배열을 사용하는 훅

의존성 배열 `deps`

`useEffect`와 `useLayoutEffect`

렌더링 이후 리액트 함수 컴포넌트에 어떤 일을 수행하는 지 알려주기 위해 사용한다.

```

function useEffect(effect: EffectCallback, deps?: DependencyList) {
  // ...
}

type DependencyList = ReadonlyArray<any>;
type EffectCallback = () => void | Destructor;

```

- `useEffect`의 콜백 함수에는 `경쟁 상태` 때문에 비동기 함수가 들어갈 수 X

경쟁 상태 : 멀티스레딩 환경에서 동시에 여러 프로세스나 스레드가 공유된 자원에 접근하려고 할 때 발생할 수 있는 문제. 이러한 상황에서 실행 순서나 타이밍을 예측할 수 X게 되어 프로그램 동작이 원하지 X는 방향으로 흐를 수 있다.

- `deps`가 변경되었는지를 `얕은 비교` 로만 판단한다.

얕은 비교 : 객체나 배열과 같은 복합 데이터 타입의 값을 비교할 때 내부의 각 요소나 속성을 재귀적으로 비교하지 않고, 해당 값들의 참조나 기본 타입 값만을 간단하게 비교하는 것

▼ 예시

```

// 1. 기본 타입(Primitive Types)의 비교
let a = 5;
let b = 5;
console.log(a === b); // true

// 2. 객체의 얕은 비교
const obj1 = { name: 'Kim' };
const obj2 = { name: 'Kim' };
console.log(obj1 === obj2); // false (서로 다른 객체)

const obj3 = obj1;
console.log(obj1 === obj3); // true (같은 참조)

// 3. React에서의 얕은 비교 예시
const ParentComponent = () => {
  const [count, setCount] = useState(0);

  // 매 렌더링마다 새로운 객체 생성
  const person = { name: 'Kim' };

  // 매 렌더링마다 새로운 배열 생성
  const numbers = [1, 2, 3];

  return (
    <div>
      <ChildComponent
        primitive={5}           // 기본값 - 값이
        person={person}         // 매번 새로운 참조
        numbers={numbers}       // 매번 새로운 참조
      />
    </div>
  );
};

```

```

    );
  };

  // memo 없는 경우 : ParentComonent 리렌더링될 때
  // React.memo로 감싸면 props의 얇은 비교 수행 ->
  const ChildComponent = React.memo(({ primit:
    console.log('Child rendered');
    return <div>{person.name}</div>;
  }));

```

- deps, 콜백함수
 - deps가 빈 배열 [] 이면, 콜백함수는 처음 렌더링될 때만 실행
 - deps가 빈 배열 [] 이면, 클린업 함수 **Destructor** 는 마운트 해제될 때 실행된다.
 - deps 배열이 존재하면, 배열의 값이 변경될 때마다 클린업 함수가 실행된다. (새로운 effect 실행 전에 이전 effect 정리)

클린업 함수 : useEffect, useLayoutEffect 같은 리액트 훅에서 사용되며, 컴포넌트가 해제되기 전에 정리 작업을 수행하는 함수

- useEffect는 레이아웃 배치와 화면 렌더링이 모두 완료된 후에 실행된다.

```


const [name, setName] = useState(""); // 1

// 3
useEffect(() => {
  // 매우 긴 시간이 흐른 뒤 아래의 setName()을 실행한다고 생각하자
  setName("배달이");
}, []);

return ( // 2
  <div>
    {`안녕하세요, ${name}님!`}
  </div>
);

```

! "안녕하세요, 님!"으로 name이 빈칸으로 렌더링된 후, 다시 "안녕하세요, 배달이님!"으로 변경되어 렌더링된다.

 `useLayoutEffect` : 화면에 해당 컴포넌트가 그려지기 전에 콜백함수를 실행하여 첫번째 렌더링 때 빈 이름이 뜨는 걸 방지한다.

useMemo와 useCallback

이전에 생성된 값 or 함수를 기억하며, 동일한 값과 함수를 반복해서 생성 X도록 해주는 훅

- 어떤 값을 계산하는 데 오랜 시간이 걸리거나 렌더링이 자주 발생하는 form에서 사용한다.
- `얕은 비교` 를 수행하므로 deps 배열이 변경되지 않았는데도 재계산되지 않도록 주의해야 한다.
- 과도한 `메모이제이션` 은 컴포넌트의 성능 향상이 보장되지 X게 한다.

메모이제이션 : 이전에 계산한 값을 저장함으로써 같은 입력에 대한 연산을 재수행하지 않도록 최적화하는 기술



▼ `useMemo`, `useCallback`

`useMemo` : 값 메모이제이션

```
// 복잡한 계산 결과를 메모이제이션
const expensiveValue = useMemo(() => {
  console.log("복잡한 계산 실행");
  return count * 2;
}, [count]); // count가 변경될 때만 재계산

// 객체 메모이제이션
const person = useMemo(() => ({
  name: name,
  age: 20
}), [name]); // name이 변경될 때만 새 객체 생성
```

`useCallback` : 함수 메모이제이션

```
// 일반적인 함수 정의 - 매 렌더링마다 새로 생성
const handleClick = () => {
  setCount(count + 1);
};

// useCallback 사용 - 의존성이 변경될 때만 새로 생성
const handleClickCallback = useCallback(() => {
  setCount(prev => prev + 1);
}, []); // 빈 배열: 함수가 재생성되지 않음
```

3. useRef

DOM을 직접 선택해야 하는 경우에 사용한다.

ex) `<input>` 요소에 포커스 설정, 특정 컴포넌트의 위치로 스크롤

- `useRef`는 세 종류의 타입 정의를 가진다.

`MutableRefObject` > `RefObject`


```
function useRef<T>(initialValue: T): MutableRefObject<T>;  
function useRef<T>(initialValue: T | null): RefObject<T>;  
function useRef<T = undefined>(): MutableRefObject<T | undefined>;  
  
interface MutableRefObject<T> {  
  current: T; // 값 변경 가능  
}  
  
interface RefObject<T> {  
  readonly current: T | null; // 값 변경 불가능  
}
```



MutableRefObject vs. RefObject

▼ MutableRefObject, RefObject 예시

```
// 1. MutableRefObject를 반환하는 경우
const mutableRef = useRef<HTMLInputElement | null>(null);
// 제네릭: T = HTMLInputElement | null
// initialValue = null
// T 타입과 initialValue 타입이 정확히 일치 -> MutableRef
mutableRef.current = someElement; // ✅ 값 변경 가능
```

```
// 2. RefObject를 반환하는 경우
const readOnlyRef = useRef<HTMLInputElement>(null);
// 제네릭: T = HTMLInputElement
// initialValue = null
// initialValue가 null -> RefObject 반환
readOnlyRef.current = someElement; // ❌ 읽기 전용 예
```

- DOM 요소 참조 → `RefObject` 사용

: 제네릭은 단일 타입, 초기값은 반드시 null

```
const inputRef = useRef<HTMLInputElement>(null);
// 메서드 호출만 가능
// inputRef.current?.focus();
```

- 값 저장/변경 필요 → `MutableRefObject` 사용

: 제네릭 타입과 초기값의 타입 일치

```
const countRef = useRef<number>(0);
// countRef.current 접근 가능
```

자식 컴포넌트에 ref 전달하기

기본 HTML요소 `<button/>` `<input/>` 가 아닌, 리액트 컴포넌트에 ref를 전달할 수 있다.

! ref를 일반적인 props로 넘겨주면 브라우저에서 경고 메시지를 띄운다.

👤 ref를 prop으로 전달하기 위해서는 `forwardRef`

- ref가 아니라 `inputRef`와 같이 다른 이름을 사용하면 `forwardRef` 사용 X해도 된다.

`forwardRef`

- 두번째 인자에 ref를 넣는다.

```
// 기본 형태
const ComponentName = forwardRef<RefType, Props>(() => {
  return JSX;
});
```

🌟 `usRef`와 다르게 **MutableRefObject**만 들어올 수 있다. → 부모 컴포넌트에서 ref를 어떻게 선언했는지와 관계없이 자식 컴포넌트가 해당 ref를 수용 가능하다.

```
interface Props {
  name: string;
}

const MyInput = forwardRef<HTMLInputElement, Props>((props, ref) => {
  return (
    <div>
      <label>{props.name}</label>
      <input ref= {ref} />
    </div>
  );
});
```

useImperativeHandle

`ForwardRefRenderFunction` 과 함께 쓸 수 있는 훅



▼ forwardRefRenderFunction

: forwardRef로 감싸는 함수의 타입을 정의하는 인터페이스

```
// 1. 직접 forwardRef 사용
const MyInput = forwardRef<HTMLInputElement, Props>((pr
  return <input ref={ref} />;
});

// 2. ForwardRefRenderFunction 사용
const renderInput: ForwardRefRenderFunction<HTMLInputEl
  return <input ref={ref} />;
};

const MyInput = forwardRef(renderInput);
```

- 이 훅을 활용하면 부모 컴포넌트에서 ref를 통해 자식 컴포넌트에서 정의한 커스터마이징된 메서드를 호출할 수 있게 된다.
→ 자식 컴포넌트는 내부 상태, 로직을 관리하면서 부모 컴포넌트와 연결도를 낮출 수 있다.



▼ useImperativeHandle

```

useImperativeHandle(
  ref, // 첫 번째 인자: 부모로부터 받은 ref
  () => ({ // 두 번째 인자: ref를 통해 노출할 메서드들을
    // 즉, 부모에 노출할 메서드들
    submit: () => {
      // 실제 구현할 로직
    },
    reset: () => {
      // 다른 메서드도 추가 가능
    }
  })
);

```

- 객체 메서드

부모 컴포넌트에서 `ref.current.submit()` 형태로 호출 가능

자식 컴포넌트에서는 `ref`로 정의된 `CreateFormHandle`을 통해 부모 컴포넌트에서 호출할 수 있는 함수를 생성하고, 부모 컴포넌트에서는 `current.submit()`을 사용하여 자식 컴포넌트의 특정 메서드를 실행할 수 있게 된다.

자식 컴포넌트

```

// (types.ts 또는 공유 파일에서 타입 정의)
// <form> 태그의 submit 함수만 따로 뽑아와서 정의한다
type CreateFormHandle = Pick<HTMLFormElement, "submit">;

type CreateFormProps = {
  defaultValues?: CreateFormValue;
};

// 컴포넌트 구현
const JobCreateForm: React.ForwardRefRenderFunction<CreateFormHandle, CreateFormProps> = (props, ref) => {
  // useImperativeHandle Hook을 사용해서 submit 함수를 커스터마이징한다

```

```

useImperativeHandle(ref, () => ({
  submit: () => {
    /* submit 작업을 진행 */
  }
}));

// ...
}

```

부모 컴포넌트

```

const CreatePage: React.FC = () => {
  // `CreateFormHandle` 형태를 가진 자식의 ref를 불러온다
  const refForm = useRef<CreateFormHandle>(null);

  const handleSubmitButtonClick = () => {
    // 불러온 ref의 타입에 따라 자식 컴포넌트에서 정의한 함수에 접근할 수 있
    refForm.current?.submit();
  };

  // ...
};

```

useRef의 여러 가지 특성

✓ useRef 사용법

- 자식 컴포넌트를 저장하는 변수로 활용 **자식 컴포넌트의 메서드나 속성에 접근하는 방법**
- useRef로 관리되는 **변수** 는 값이 바뀌어도 컴포넌트의 리렌더링 발생 X
- 리액트 컴포넌트의 상태는 상태 변경 함수를 호출하고 렌더링된 이후에 업데이트된 상태를 조회할 수 있다. 반면에 useRef로 관리되는 변수는 값을 설정한 후 즉시 조회할 수 있다.

```

// 이때 isAutoPlayPause는 UI 렌더링에 직접적인 영향을 주지 않으므로
// 리렌더링이 필요하지 않기 때문에 useRef를 사용이 성능상 효율적이다!

```

```

type BannerProps = {
  autoplay: boolean;
};

```

```

const Banner: React.FC<BannerProps> = ({ autoplay }) => {
  // 현재 자동 재생이 일시 정지되었는지 확인하는 ref
  const isAutoPlayPause = useRef(false);

  if (autoplay) {
    // keepAutoPlay 같이 isAutoPlay가 변하자마자 사용해야 할 때 쓸 수 있음
    const keepAutoPlay = !touchPoints[0] && !isAutoPlayPause.current;

    // ...
  }
  return (
    <>
      {autoplay && (
        <button
          aria-label="자동 재생 일시 정지"
          // isAutoPlayPause는 사실 렌더링에는 영향을 미치지 않고 로직에
          // 상태로 사용해서 불필요한 렌더링을 유발할 필요가 없음
          onClick={() => { isAutoPlayPause.current = true }}
        />
      )}
    </>
  );
};

const Label: React.FC<LabelProps> = ({ value }) => {
  useEffect(() => {
    // value.name과 value.id를 사용해서 작업한다
  }, [value]);

  // ...
};

```



훅의 2가지 규칙

1. 훅은 항상 **최상위 레벨**에서 호출되어야 한다.

조건문, 반복문, 중첩 함수, 클래스 등의 내부에서는 훅을 호출하지 X아야 한다.

반환문으로 함수 컴포넌트가 종료되거나, 조건문 or 변수에 따라 반복문 등으로 훅의 호출 여부가 결정되면 X된다.

2. 훅은 항상 함수 컴포넌트나 커스텀 훅 등의 리액트 컴포넌트 내에서만 호출되어야 한다.

➡ 리액트에서 훅은 호출 순서에 의존하기 때문에, 모든 컴포넌트 렌더링에서 훅의 순서 항상 동일하게 유지되어야 한다.

- 리액트는 규칙을 준수할 수 있도록 **Lint 플러그인** 제공

▼ **eslint-plugin-react-hooks** : ESLint 플러그인 중 하나로 리액트의 Hook 규칙을 검사한다.

```
// 설치
npm install eslint-plugin-react-hooks --save-dev

// .eslintrc 설정
{
  "plugins": ["react-hooks"],
  "rules": {
    "react-hooks/rules-of-hooks": "error", // Hook 사용 규칙
    "react-hooks/exhaustive-deps": "warn" // useEffect의 의존성
  }
}
```



9.2 커스텀 훅

1. 나만의 . 훅만들기

'useState, useEffect, useRef 훅 + 사용자 정의 훅'을 생성하여 컴포넌트 로직을 함수로 뽑아내 재사용한다.

🌟 커스텀 훅의 이름은 반드시 **use** 로 시작해야 한다.

useInput 커스텀 훅

: 인자로 받은 초기값을 useState로 관리하며, 해당 값을 수정할 수 있는 onChange 함수를 Input 값과 함께 반환하는 hook

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

const useInput = (initialValue) => {
  const [value, setValue] = useState(initialValue);

  const onChange = (e) => {
    setValue(e.target.value);
  };

  return { value, onChange };
};
```

- 컴포넌트 내에서 useInput 훅 사용 방법

```
const MyComponent = () => {
  const { value, onChange } = useInput("");

  return (
    <div>
      <h1>{value}</h1>
      <input onChange= {onChange} value= {text} />
    </div>
  );
};

export default App;
```

2. TS로 커스텀 훅 강화하기

useInput 커스텀 훅을 TS로 작성하기

! 이벤트 객체 e의 타입은 유추하기 힘들다.

👩 IDE 활용하면, TS 컴파일러 `tsc` 가 현재 사용하고 있는 이벤트 객체의 타입을 유추해서 알려준다.

```
(parameter) e: React.ChangeEvent<HTMLInputElement>  
<input onChange={(e) => { console.log(e)}} />
```

```
import { useState, useCallback, ChangeEvent } from "react";  
  
// ✅ initialValue에 string 타입을 정의  
const useInput = (initialValue: string) => {  
  const [value, setValue] = useState(initialValue);  
  
  // ✅ 이벤트 객체인 e에 ChangeEvent<HTMLInputElement> 타입을 정의  
  const onChange = useCallback((e: ChangeEvent<HTMLInputElement>  
    setValue(e.target.value);  
  }, []);  
  
  return { value, onChange };  
};  
  
export default useInput;
```