



리액트 useState는 어떤 모습일까?

2022년 04월 11일

함수는 값을 반환한다.

리액트 엘리먼트를 반환하는 함수를 리액트에서는 함수 컴포넌트(Function Component)라고 부른다. 클래스에 비해 함수가 간편한 것 처럼 함수 컴포넌트는 비교적 단순하다.

그만큼 제약도 있다.

함수 컴포넌트는 컴포넌트 생애 주기라는 것이 없다. 물론 클래스 컴포넌트가 제공하는 메소드도 없다. 컴포넌트가 생성되고 마운트 되고 업데이트 되는 시점에 관여할 수 없다는 것이다.

상태도 없다. 클래스 컴포넌트는 `this.state` 에 컴포넌트 상태를 저장한다. 반면 함수 컴포넌트는 함수를 한 번 실행한 뒤 값을 반환하면 끝이다. 변화를 관찰할 상태랄 것이 없다.

그럼에도 불구하고 함수 컴포넌트는 리액트에서 추천하는 컴포넌트다. 앞으로 컴포넌트를 새로 만들 때는 함수 컴포넌트를 이용하라고 권장한다 (Hook이나 class 또는 두 가지를 모두 사용해야 합니다). 물론 훅스(hooks)와 함께 말이다.

훅스는 함수 컴포넌트가 더 잘 일할 수 있도록 돕는다. `useState`는 컴포넌트가 상태를 사용할 수 있도록 기능을 제공한다. `useEffect`는 컴포넌트가 사이드 이펙트에 반응하도록 한다. 함수 컴포넌트가 **에스프레소**라면 훅스는 **우유 거품이나 모카 크림**이다. 훅스와 함수 컴포넌트라면 여러가지 컴포넌트를 만들어 낼 수 있다.



다양한 커피 메뉴(출처: unsplash.com)

이 글에서는 함수 컴포넌트가 어떻게 상태를 다룰 수 있는지 알아보겠다. `useState` 함수를 만들어 보면서 리액트가 함수 컴포넌트에서 상태를 관리하는 방식을 추측해 볼 것이다. 단 여기 나온 코드는 리액트의 것과 같다고 말할 수 없는 점은 유의하자.

문제 제기

`NameField` 컴포넌트를 만들텐데 이름을 입력할 수 있는 컴포넌트다.

```
function NameField {  
  const name = "정환";
```

```
const handleChange = event => {  
  // 변경한 값을 어디에 저장하지?  
}  
  
return <input value={name} onChange={handleChange} />;  
}
```

name 변수에 할당한 문자열을 input 요소의 value로 갖는다. 사용자가 필드에 값을 입력하면 handleChange 함수가 동작할 것이다.

NameField의 UI

인풋 필드에 값을 바인딩하려면 handleChange 함수에서 value 값을 바꿔 주어야 한다. 하지만 name은 함수 본문 선언한 상수일 뿐이다. 함수를 실행할 때는 문자열 값이기 때문에 변경할 방법이 없다.

let name 으로 선언하면 다를까?

```
function NameField {  
  let name = "정환";  
  
  const handleChange = event => {  
    name = event.target.value  
  }  
  
  return <input value={name} onChange={handleChange} />;  
}
```

함수가 반환한 엘리먼트에는 "정환"이 담겨 있다. 사용자가 필드에 입력하면 handleChange 함수가 움직이고 name 값을 바꿀 것이다.

```
name = "정환1"
```

하지만 여기까지다. 함수 컴포넌트가 다시 호출되기 전까지 돔에 보이는 것은 처음 반환한 엘리먼트 값이다.

```
<input value="정환">
```

만약 어디선가 함수 컴포넌트를 다시 호출한다고 해서 문제가 해결되지는 않을 것 같다. 다시 호출하더라도 함수 상단에 표현식 때문에 value는 여전히 "정환"이기 때문이다.

```
let name = "정환"
```

두 가지 문제

NameField가 사용자 입력을 제대로 처리하려면 두 가지 숙제를 풀어야 한다.

첫째, 입력값을 **어딘가에 저장해야** 한다.

이 값은 사용자 입력에 따라 변하니깐 "상태(state)"라고 부르자. 클래스 컴포넌트라면 this.state라는 멤버 변수에 상태를 관리했을 것이다. 함수 컴포넌트 자체는 상태가 없다. 함수 외부에 저장할 만한 곳을 마련해야 한다.

상태를 가지고 있다 하더라도 남은 문제가 더 있다. 변경된 상태를 가지고 컴포넌트를 다시 그려야 하는데 한 번 호출한 함수를 다시 호출할 방법이 없다.

둘째, 상태가 바뀌면 **함수를 다시 호출해야** 한다.

리액트의 `useState`를 사용하면 함수 컴포넌트가 상태에 따라 반응한다. 컴포넌트 안에서 `useState`를 사용했을 뿐인데 이런 효과가 나온다면 이것이 바로 리액트의 역할이 아닐까 가늠해 볼 수 있다.

코드를 보고 싶었지만 함수 정의 부분조차 찾지 못했다.

“ 만들면서 상상해 보자!

비슷한 기능을 직접 구현해 보면 구조를 추측할 수 있지 않을까? 함수형 컴포넌트가 상태를 구독하고 리액티브하게 반응할 수 있도록 만들어 보자.

1차 구현

리액트에서 처리하는 이 두 가지 기능을 MyReact로 만들어 보겠다.

```
function MyReact() {
```

```
// 이름
let firstname
let isInitialized = false

function useName(initialValue = "") {
  if (!isInitialized) {
    firstname = initialValue
    isInitialized = true
  }

  // 이름 변경
  const setFirstname = value => {
    firstname = value
  }

  return [firstname, setFirstname]
}

return {
  useName,
}
```

혹스처럼 사용하기 위해 useName이란 이름의 함수를 정의했다. 상태의 초기 값 initialValue를 받았다.

혹이 처음 실행될 때만 이 값을 설정하려고 isInitialized 플래그를 활용했다. 초기값을 firstname 이란 변수에 저장하는데 이것이 바로 상태이다.

이 상태를 변경하는 세터도 만들자. setFirstname이란 함수인데 값을 받아 firstname에 설정하는 역할을 한다.

마지막에 상태값과 세터를 배열 형식으로 반환한다.

MyReact는 방금 만든 useName을 담은 객체를 반환한다.

상태를 관리하라는 첫 번째 숙제는 일단 해결했다. 두 번째 숙제를 해결할 차례다. 상태가 변하면 함수 컴포넌트를 다시 호출해 새로운 엘리먼트가 돔에 반영되도록 해야한다.

간단히 세터 함수 안에서 다음 코드를 추가해보자.

```
// 클래스 컴포넌트의 forceUpdate를 흉내낸다.
```

```
// 리렌더링을 유발하는 역할이다.  
const { forceUpdate } = useForceUpdate()  
  
const setFirstname = value => {  
  firstname = value  
  
  // 상태를 변경하고 리렌더링을 유발한다.  
  forceUpdate()  
}
```

forceUpdate는 클래스 컴포넌트에서 사용한 함수 이름이다. 물론 함수 컴포넌트에는 없지만 리렌더링을 유발하기 위해 임시로 만들었다.

```
// 구현을 위해 이곳만 리액트 훅을 사용한다.  
function useForceUpdate {  
  const [value, setValue] = React.useState(1);  
  
  const forceUpdate = () => setValue(value + 1);  
  
  return {  
    forceUpdate,  
  };  
};
```

구현 상세는 글의 방향과 무관하므로 그냥 속 읽고 지나치자.

이렇게 해서 두 가지 숙제를 모두 해결했다.

리액트 모듈에서 api를 가져 오듯이 방금 만든 useName 훅을 사용해 보자.

```
const { userName } = (function MyReact(){/* ... */})();

function NameField {
  // userName을 이용해 name과 setName을 얻는다.
  const [firstname, setFirstname] = userName("정환"); // firstname = "정환"

  const handleChange = event => {
    setFirstname(event.target.value) // firstname = 사용자가 인풋 필드에 입력한 값
  }

  return <input value={firstname} onChange={handleChange} />;
}
```

MyReact를 즉시 실행하고 반환값에서 userName 함수를 가져왔다.


NameField 본문에서 이 함수를 사용한다.

초기값 "정환"을 전달하고 반환된 배열에서 상태 값과 세터를 가져와서 각 각 firstname, setFirstname 란 이름으로 저장했다.

firstname은 인풋의 value로 전달했다.

이벤트 핸들러에서는 setFieldname의 인자로 입력한 값을 전달했다. 이 세터는 상태를 변경하고 나서 곧장 화면을 다시 그리도록 리액트에 요청할 것이다.

NameField는 다시 호출되는데 이때 firstname은 방금 변경한 상태값이 될 것이다. 이 값을 이용해 리액트 엘리먼트를 반환할 것이고 리액트는 이 걸 돔에 그리게 될 것이다.



정환입니다

NameField가 상태를 사용할 수 있다

이제 NameField 컴포넌트는 상태를 가지게 되었다.

다중 상태 관리

NameField에 이름(firstname)뿐만 아니라 성(lastname)도 입력 받게 하겠다. 필드 갯수만큼 상태도 추가해야 할텐데 지금 구조로 가능할까?

먼저 MyReact 함수 본문을 범용적인 이름으로 바꿔보자.

```
function MyReact() {
  // 상태 값
  let value

  // 상태를 사용하는 함수
  function useState(initilaValue = "") {
    const { forceUpdate } = useForceUpdate()

    if (!isInitialized) {
      value = initilaValue
      isInitialized = true
    }

    // 상태 변경
    const setValue = nextValue => {
      value = nextValue
      forceUpdate()
    }

    return [value, setValue]
  }

  return {
    useState,
  }
}
```

name이란 용어를 쏙 빼고 범용적인 state란 용어로 바꿨다.

이 useState를 이용해 상태를 하나 더 추가하자.

```
const { useState } = (function MyReact(){/* ... */})();

function NameField {
  const [firstname, setFirstname] = useState("정환"); // firstname = "정환"
  // lastname 상태를 추가한다
  const [lastname, setLastname] = useState("김"); // lastname = "김" ?
  // ...
  return (
    <>
      <input value={firstname}/>
    </>
  )
}
```



```
        <input value={lastname}/>
      </>;
    )
  }
```

useState를 두 번 호출해서 firstname 상태와 lastname 상태 두 개를 두었다. 그리고 각 input의 value 속성에 값을 바인딩했다.

어떻게 될까?

정환 정환

두 필드가 같은 상태를 바라본다

이런..... 두 필드가 같은 값이다.

왜 이렇게 되었을까? 원인은 MyReact에 선언한 value 변수다. useState를 여러번 호출하더라도 항상 value에 값을 할당하기 때문이다. 게다가 처음 호출되었을 경우만 초기값을 할당하기 때문에 가장 먼저 전달한 "정환"이 지금의 상태값이다.

2차 구현

다중 상태를 구현하려면 value 구조를 좀 바꿔야겠다.

```
function MyReact() {
  // 상태 목록
  const values = []
  // 각 상태의 초기화 여부 목록
  const isInitialized = []

  // 상태 목록에서 사용할 cursor를 받는다
  function useState(cursor: number, initilaValue = "") {
    const { forceUpdate } = useForceUpdate()

    // 지정한 상태의 초기화 여부 값을 조회한다
    if (!isInitialized[cursor]) {
      // 지정한 상태의 값을 설정한다
      values[cursor] = initilaValue
      isInitialized[cursor] = true
    }
  }
}
```

```
// 지정한 상태를 조회한다
const value = values[cursor]
const setValue = (value: any) => {
  // 지정한 상태의 값을 변경한다
  values[cursor] = value
  forceUpdate()
}

return [value, setValue]
}

return {
  useState,
}
}
```

value를 values 배열로 바꿨다. useState를 호출할 때마다 이 배열의 적당한 위치에 초기값을 설정 하겠다.

초기값은 처음 호출될 때 설정하는데 initialized 플래그를 보고 판단한다. 이것도 불리언 타입에서 배열로 변경했다.

이제 useState는 초기값 뿐만아니라 위치정보 cursor도 받아야 한다. 상태를 목록으로 만들었기 때문에 어느 위치에 있는 상태를 조회할 것인지 찾기 위한 포인터다.

이 값은 isInitialized 배열에서 최초 호출 여부를 찾기위에서도 사용된다. 물론 세터 안에서 값을 세팅 하기 위해서도 사용된다.

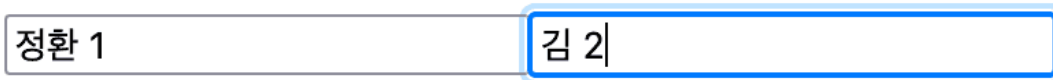
다시 상태를 만들어 보겠다.

```
function NameField3() {
  const [firstname, setFirstname] = useState(0, "정환")
  const [lastname, setLastname] = useState(1, "감")

  const handleChangeFirstname = event => {
    setFirstname(event.target.value)
  }
  const handleChangeLastname = event => {
    setLastname(event.target.value)
  }
}
```

```
return (  
  <>  
    <input value={firstname} onChange={handleChangeFirstname} />  
    <input value={lastname} onChange={handleChangeLastname} />  
  </>  
)  
}
```

useState를 호출할 때 호출 순서에 맞게 cursor 값으로 0, 1을 전달했다.



다중 상태를 사용할 수 있다

이제 두 필드가 각 각 다른 상태를 구독할 수 있게 되었다.

더 단순하게

useState를 호출할때 어느 상태를 구독할지 커서를 넘기는 것은 불편하다. 실수하기 쉬운 api다.

useState 함수의 커서를 관리하는 역할은 사용하는 측이 아니라 제공하는 측이다. MyReact가 스스로 커서를 관리하도록 개선해 보겠다.

```
function MyReact() {  
  // 상태를 가리킨다  
  let cursor = 0  
  
  function useState() {  
    // 커서를 지정한 세터를 반환한다  
    const setValueAt = cursor => value => {  
      values[cursor] = value  
      forceUpdate()  
    }  
  
    // 커서를 지정한 세터를 만든다  
    const setValue = setValueAt(cursor)  
  
    // 커서를 1 증가한다  
    cursor++  
  
    return [value, setValue]  
  }  
}
```

```
}

// 커서 초기화 함수
function resetCursor() {
  cursor = 0
}

return {
  useState,
  resetCursor,
}
}
```

MyReact 함수 본문에 cursor 변수를 선언했다.

useState가 호출될 시점에는 cursor가 특정 상태를 가리키고 있을 것이다. 이 값을 세터에 잡아 두기 위해 setValueAt 함수를 만들었다. 이것으로 특정 커서의 setValue 함수를 만든다.

그리고 나서 cursor를 올려 다음 useState 호출에서 사용하도록 준비해 둔다.

함수 컴포넌트가 호출될 때 커서를 리셋하기 위해 resetCursor 함수도 제공한다.

이번에는 이렇게 사용해 보겠다.

```
function NameField() {
  // 커서를 다시 설정한다
  resetCursor()

  const [firstname, setFirstname] = useState("정환") // cursor 0
  const [lastname, setLastname] = useState("김") // cursor 1

  // ...
}
```

좀 더 리액트의 useState와 비슷해졌다. 컴포넌트 시작부에 resetCursor를 호출하는데 이것은 MyReact의 한계다.

결론

혹스에는 사용 규칙이 있다.

“ 최상위에서만 Hook을 호출해야 합니다. 반복문, 조건문 혹은 중첩된 함수 내에서 Hook을 호출하지 마세요.

MyReact로 구현했던 것을 떠올려 보면 순서에 따라 cursor 값이 1씩 증가 한다. 만약 컴포넌트의 조건문이나 반복문 안에서 훅스를 사용한다면 함수 호출마다 커서가 동일하다는 것을 보장하지 못한다. 컴포넌트에서 useState를 이용해 상태를 가져와 firstname으로 사용하는데 cursor가 달라진다면 동일한 상태를 얻지 못한다.

“ 오직 React 함수 내에서 Hook을 호출해야 합니다.

useState가 제공하는 세터에서 다시 함수 컴포넌트가 리렌더링 되도록 했다. 훅스를 설명하는 이 글에서 훅스를 이용해 렌더 트리거를 만들었지만 의도만 보자면 상태가 변하면 컴포넌트를 다시 그리는 것이다.

일반 함수 안에서 사용한다면 어떻게 될까? 리액트가 이 함수를 다시 실행하고 리액트 엘리먼트를 반환하길 기대할 것이다. 하지만 일반 함수는 그렇지 않기 때문에 리액트가 렌더하는데 문제가 생길 것 같다.

« 2022-03 스크랩

리액트 useEffect는 어떤 모습일까? »

useState 훅의 동작 방식을 알아보기 위해 직접 만들어 보았다. 서두에서도 언급했지만 실제 리액트 코드의 이 훅의 코드는 다를 것이다.

shmoon2917 commented 2 weeks ago

잘 읽었습니다 감사합니다 :)

jeonghwan-kim commented 2 weeks ago

Owner

@shmoon2917 방문해 주셔서 고맙습니다.

#react

joohaem commented 2 weeks ago

알기 쉽게 작성해주셔서 많이 배웠습니다
감사합니다☺☺

jeonghwan-kim commented a week ago

Owner

@joohaem 피드백 주셔서 고맙습니다.

© 김정환 2022