useState

useState 는 컴포넌트에 state 변수를 추가할 수 있는 React Hook입니다.

```
const [state, setState] = useState(initialState)
```

레퍼런스

useState(initialState)

컴포넌트의 최상위 레벨에서 `useState`를 호출하여 state 변수를 선언합니다.

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

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

배열 구조 분해를 사용하여 `[something, setSomething]` 과 같은 state 변수의 이름을 지정하는 것이 규칙입니다.

아래에서 더 많은 예시를 확인하세요.

매개변수

- `initialState` : state의 초기 설정값입니다. 어떤 유형의 값이든 지정할 수 있지만 함수에 대해서는 특별한 동작이 있습니다. 이 인수는 초기 렌더링 이후에는 무시됩니다.
  - 함수를 `initialState`로 전달하면 이를 *초기화 함수*로 취급합니다. 이 함수는 순수해야 하고 인수를 받지 않아야 하며 반드시 어떤 값을 반환해야 합니다. React는 컴포넌트를 초기화할 때 초기화 함수를 호출하고, 그 반환값을 초기 state로 저장합니다. 아래 예시를 참고하세요.

반환값

`useState`는 정확히 두 개의 값을 가진 배열을 반환합니다.

1. 현재 state입니다. 첫 번째 렌더링 중에는 전달한 `initialState`와 일치합니다.
2. state를 다른 값으로 업데이트하고 리렌더링을 촉발할 수 있는 `set` 함수입니다.

주의 사항

- `useState`는 Hook이므로 **컴포넌트의 최상위 레벨**이나 직접 만든 Hook에서만 호출할 수 있습니다. 반복문이나 조건문 안에서는 호출할 수 없습니다. 필요한 경우 새 컴포넌트를 추출하고 state를 그 안으로 옮기세요.
- Strict Mode에서 React는 의도치 않은 불순물을 찾기 위해 **초기화 함수를 두 번 호출합니다.** 이는 개발 환경 전용 동작이며 프로덕션 환경에는 영향을 미치지 않습니다. 초기화 함수가 순

수하다면(그래야 합니다) 동작에 영향을 미치지 않습니다. 호출 중 하나의 결과는 무시됩니다.

`setSomething(nextState)` 과 같은 `set` 함수

`useState` 가 반환하는 `set` 함수를 사용하면 state를 다른 값으로 업데이트하고 리렌더링을 촉발할 수 있습니다. 여기에는 다음 state를 직접 전달하거나, 이전 state로부터 계산한 함수를 전달할 수도 있습니다.

```
const [name, setName] = useState('Edward');

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
  // ...
```

매개변수

- `nextState` : state가 될 값입니다. 값은 모든 데이터 타입이 허용되지만, 함수에 대해서는 특별한 동작이 있습니다.
  - 함수를 `nextState` 로 전달하면 *업데이터 함수*로 취급합니다. 이 함수는 순수해야 하고, 대기 중인 state를 유일한 인수로 사용해야 하며, 다음 state를 반환해야 합니다. React는 업데이터 함수를 대기열에 넣고 컴포넌트를 리렌더링 합니다. 다음 렌더링 중에 React는 대기열에 있는 모든 업데이터를 이전 state에 적용하여 다음 state를 계산합니다. 아래 예시를 참고하세요.

반환값

`set` 함수는 반환값이 없습니다.

주의 사항

- `set` 함수는 ***다음* 렌더링에 대한 state 변수만 업데이트합니다.** `set` 함수를 호출한 후에도 state 변수에는 여전히 호출 전 화면에 있던 이전 값이 담겨 있습니다.
- 사용자가 제공한 새로운 값이 `Object.is` 에 의해 현재 `state` 와 동일하다고 판정되면, React는 **컴포넌트와 그 자식들을 리렌더링하지 않습니다.** 이것이 바로 최적화입니다. 경우에 따라

React가 자식을 건너뛰기 전에 컴포넌트를 호출해야 할 수도 있지만, 코드에 영향을 미치지는 않습니다.

- React는 state 업데이트를 batch 합니다. **모든 이벤트 핸들러가 실행되고** `set` 함수를 호출한 후에 화면을 업데이트합니다. 이렇게 하면 단일 이벤트 중에 여러 번 리렌더링 하는 것을 방지할 수 있습니다. 드물지만 DOM에 접근하기 위해 React가 화면을 더 일찍 업데이트하도록 강제해야 하는 경우, `flushSync` 를 사용할 수 있습니다.

- `set` 함수는 항상 동일한 식별자를 가지기 때문에 Effect 의존성 목록에서 자주 생략됩니다. 하지만 의존성에 포함하더라도 Effect가 다시 실행되지는 않습니다. 린터가 오류 없이 생략을 허용한다면, 그대로 생략해도 안전합니다. Effect 의존성 제거 방법에 대해 더 알아보세요.

- *렌더링 도중* `set` 함수를 호출하는 것은 현재 렌더링 중인 컴포넌트 내에서만 허용됩니다. React는 해당 출력을 버리고 즉시 새로운 state로 다시 렌더링을 시도합니다. 이 패턴은 거의 필요하지 않지만 **이전 렌더링의 정보를 저장하는 데 사용할 수 있습니다**. 아래 예시를 참고하세요.

- Strict Mode에서 React는 의도치않은 불순물을 찾기 위해 **업데이터 함수를 두 번 호출합니다**. 이는 개발 환경 전용 동작이며 프로덕션 환경에는 영향을 미치지 않습니다. 만약 업데이터 함수가 순수하다면(그래야 합니다) 동작에 영향을 미치지 않습니다. 호출 중 하나의 결과는 무시됩니다.

사용법

컴포넌트에 state 추가하기

컴포넌트의 최상위 레벨에서 `useState` 를 호출하여 하나 이상의 state 변수를 선언하세요.

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

function MyComponent() {
  const [age, setAge] = useState(42);
  const [name, setName] = useState('Taylor');
  // ...
```

배열 구조 분해를 사용하여 `[something, setSomething]` 과 같은 state 변수의 이름을 지정하는 것이 관례입니다.

`useState` 는 정확히 두 개의 항목이 있는 배열을 반환합니다.

1. 이 state 변수의 현재 state 로, 처음에 제공한 초기 state 로 설정됩니다.
2. 상호작용에 반응하여 다른 값으로 변경할 수 있는 `set` 함수 입니다.

화면의 내용을 업데이트하려면 다음 state로 `set` 함수를 호출합니다.

```
function handleClick() {
  setName('Robin');
}
```

React는 다음 state를 저장하고 새로운 값으로 컴포넌트를 다시 렌더링한 후 UI를 업데이트합니다.

주의하세요!

`set` 함수를 호출해도 이미 실행 중인 코드의 현재 state는 변경되지 **않습니다**.

```
function handleClick() {
  setName('Robin');
  console.log(name); // 아직 "Taylor"입니다!
}
```

`set` 함수는 **다음** 렌더링에서 반환할 `useState` 에만 영향을 줍니다.

useState 기본 예시

1. 카운터 (숫자) 2. 텍스트 필드 (문자열) 3. 체크박스 (불리언) 4. 폼 (두 개의 변수)

예시 1 of 4: 카운터 (숫자)

예시에서 `count` state 변수는 숫자를 받습니다. 버튼을 클릭하면 숫자가 증가합니다.

App.js

다운로드 새로고침 Clear 포크

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

export default function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
```

다음 예시

이전 state를 기반으로 state 업데이트하기

age 가 42 라고 가정합니다. 이 핸들러는 setAge(age + 1) 를 세 번 호출합니다.

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

하지만 클릭해보면 age 는 45 가 아니라 43 이 됩니다! 이는 set 함수를 호출해도 이미 실행 중인 코드에서 age state 변수가 업데이트되지 않기 때문입니다. 따라서 각 setAge(age + 1) 호출은 setAge(43) 이 됩니다.

이 문제를 해결하려면 다음 state 대신 setAge 에 ***업데이터 함수*를 전달할 수 있습니다.**

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

여기서 a => a + 1 은 업데이터 함수입니다. 이 함수는 대기 중인 state 를 가져와서 다음 state 를 계산합니다.

React는 업데이터 함수를 큐에 넣습니다. 그러면 다음 렌더링 중에 동일한 순서로 호출합니다.

1. a => a + 1 은 대기 중인 state로 42 를 받고 다음 state로 43 을 반환합니다.
2. a => a + 1 은 대기 중인 state로 43 을 받고 다음 state로 44 를 반환합니다.
3. a => a + 1 은 대기 중인 state로 44 를 받고 다음 state로 45 를 반환합니다.

대기 중인 다른 업데이트가 없으므로, React는 결국 45 를 현재 state로 저장합니다.

규칙상 대기 중인 state 인수의 이름을 age 의 a 와 같이 state 변수 이름의 첫 글자로 지정하는 것이 일반적입니다. 그러나 prevAge 또는 더 명확하다고 생각하는 다른 이름으로 지정해도 됩니다.

React는 개발 환경에서 순수한지 확인하기 위해 업데이터를 두 번 호출할 수 있습니다.

자세히 살펴보기

항상 업데이터를 사용하는 것이 더 좋은가요?

자세히 보기

업데이터를 전달하는 것과 다음 state를 직접 전달하는 것의 차이점

1. 업데이터 함수 전달하기
2. 다음 state 바로 전달하기

예시 1 of 2: 업데이터 함수 전달하기

이 예시는 업데이터 함수를 전달하므로 “+3” 버튼이 작동합니다.

App.js

다운로드 새로고침 Clear 포크

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

export default function Counter() {
  const [age, setAge] = useState(42);

  function increment() {
    setAge(a => a + 1);
  }

  return (
    <>
      <h1>Your age: {age}</h1>
```

자세히 보기

다음 예시

객체 및 배열 state 업데이트하기

state에는 객체와 배열도 넣을 수 있습니다. React에서 state는 읽기 전용으로 간주되므로 **기존 객체를 *변경*하지 않고, *교체* 해야 합니다**. 예를 들어, state에 `form` 객체가 있는 경우 변경하지 마세요.

```
// 🚩 state 안에 있는 객체를 다음과 같이 변경하지 마세요.
form.firstName = 'Taylor';
```

대신 새로운 객체를 생성하여 전체 객체를 교체하세요.

```
// ✅ 새로운 객체로 state를 교체합니다.
setForm({
  ...form,
  firstName: 'Taylor'
});
```

자세한 내용은 객체 state 업데이트하기 및 배열 state 업데이트하기에서 확인하세요.

객체 및 배열 state 예시

1. 폼 (객체) 2. 폼 (중첩 객체) 3. 리스트 (배열) 4. Immer로 간결한 업데이트 로직 직

예시 1 of 4: 폼 (객체)

이 예시에서 `form` state 변수는 객체를 받습니다. 각 input에는 전체 `form` 의 다음 state로 `setForm` 을 호출하는 change 핸들러가 있습니다. 전개 구문인 `{ ...form }` 은 state 객체를 변경하지 않고 교체합니다.

App.js 다운로드 새로고침 Clear 포크

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

export default function Form() {
  const [form, setForm] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com',
  });

  return (
    <>
      <label>
```

자세히 보기

다음 예시

초기 state 다시 생성하지 않기

React는 초기 state를 한 번 저장하고 다음 렌더링부터는 이를 무시합니다.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
  // ...
```

`createInitialTodos()` 의 결과는 초기 렌더링에만 사용되지만, 여전히 모든 렌더링에서 이 함수를 호출합니다. 이는 큰 배열을 생성하거나 값비싼 계산을 수행하는 경우 낭비일 수 있습니다.

이 문제를 해결하려면, `useState` 에 **초기화 함수로 전달하세요**.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  // ...
```

함수를 호출한 결과인 `createInitialTodos()` 가 아니라 함수 자체인 `createInitialTodos` 를 전달하고 있다는 것에 주목하세요. 함수를 `useState` 에 전달하면 React는 초기화 중에만 함수를 호출합니다.

개발 환경에서는 React가 초기화 함수가 순수한지 확인하기 위해 초기화 함수를 두 번 호출할 수 있습니다.

초기화 함수를 전달하는 것과 초기 state를 직접 전달하는 것의 차이점

예시 1 of 2: 초기화 함수 전달하기

이 예시에서는 초기화 함수를 전달하므로, `createInitialTodos` 함수는 초기화 중에만 실행됩니다. input에 타이핑할 때 같이 컴포넌트가 리렌더링할 때에는 실행되지 않습니다.

App.js 다운로드 새로고침 Clear 포크

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

function createInitialTodos() {
  const initialTodos = [];
  for (let i = 0; i < 50; i++) {
    initialTodos.push({
      id: i,
      text: 'Item ' + (i + 1)
    });
  }
  return initialTodos;
}
```

자세히 보기

다음 예시

key로 state 초기화하기

목록을 렌더링할 때 `key` 속성을 자주 접하게 됩니다. 하지만 `key` 속성은 다른 용도로도 사용됩니다.

컴포넌트에 다른 `key` 를 전달하여 컴포넌트의 state를 초기화할 수 있습니다. 이 예시에서는 Reset 버튼이 `version` state 변수를 변경하고, 이를 `Form` 에 `key` 로 전달합니다. `key` 가 변경되면 React는 `Form` 컴포넌트(및 그 모든 자식)를 처음부터 다시 생성하므로 state가 초기화됩니다.

자세히 알아보려면 State를 보존하고 초기화하기를 읽어보세요.

App.js

다운로드 새로고침 Clear 포크

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

export default function App() {
  const [version, setVersion] = useState(0);

  function handleReset() {
    setVersion(version + 1);
  }

  return (
    <>
      <button onClick={handleReset}>Reset</button>
```

자세히 보기

이전 렌더링에서 얻은 정보 저장하기

보통은 이벤트 핸들러에서 state를 업데이트합니다. 하지만 드물게 렌더링에 대한 응답으로 state를 조정해야 하는 경우도 있습니다. 예를 들어, props가 변경될 때 state 변수를 변경하고 싶을 수 있습니다.

대부분의 경우 이 기능은 필요하지 않습니다.

- **필요한 값을 현재 props나 다른 state에서 모두 계산할 수 있는 경우, 중복되는 state를 모두 제거하세요**. 너무 자주 재계산하는 것이 걱정된다면, `useMemo` Hook을 사용하면 도움이 될 수 있습니다.
- 전체 컴포넌트 트리의 state를 초기화하려면 컴포넌트에 다른 `key` 를 전달하세요.
- 가능하다면 이벤트 핸들러의 모든 관련 state를 업데이트하세요.

이 중 어느 것에도 해당하지 않는 희귀한 경우에는, 컴포넌트가 렌더링되는 동안 `set` 함수를 호출하여 지금까지 렌더링된 값을 바탕으로 state를 업데이트하는 데 사용할 수 있는 패턴이 있습니다.

다음은 그 예시입니다. `CountLabel` 컴포넌트는 전달된 `count` props를 표시합니다.

```
export default function CountLabel({ count }) {
  return <h1>{count}</h1>
}
```

카운터가 마지막 변경 이후 *증가 또는 감소했는지*를 표시하고 싶다고 가정해 보겠습니다. `count` prop는 이를 알려주지 않으므로 이전 값을 추적해야 합니다. 이를 추적하기 위해 `prevCount` state 변수를 추가합니다. `trend` 라는 또 다른 state 변수를 추가하여 count의 증가 또는 감소 여부를 추적합니다. `prevCount` 와 `count` 를 비교해서, 같지 않은 경우 `prevCount` 와 `trend` 를 모두 업데이트합니다. 이제 현재 count props와 마지막 렌더링 이후 count가 *어떻게* 변경되었는지 모두 표시할 수 있습니다.

App.js CountLabel.js 새로고침 Clear 포크

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

export default function CountLabel({ count }) {
  const [prevCount, setPrevCount] = useState(count);
  const [trend, setTrend] = useState(null);
  if (prevCount !== count) {
    setPrevCount(count);
    setTrend(count > prevCount ? 'increasing' : 'decreasing');
  }
  return (
    <>
      <h1>{count}</h1>
```

자세히 보기

렌더링하는 동안 `set` 함수를 호출하는 경우, 그 `set` 함수는 `prevCount !== count`와 같은 조건 안에 있어야 하며, 조건 내부에 `setPrevCount(count)`와 같은 호출이 있어야 한다는 점에 유의하세요. 그렇지 않으면 리렌더링을 반복하다가 결국 깨질 것입니다. 또한 이 방식은 오직 *현재* 렌더링 중인 컴포넌트의 state만을 업데이트할 수 있습니다. 렌더링 중에 다른 컴포넌트의 `set` 함수를 호출하는 것은 에러입니다. 마지막으로, 이 경우에도 `set` 함수 호출은 여전히 변경이 아닌 state 업데이트여야만 합니다. 순수 함수의 다른 규칙을 어겨도 된다는 의미가 아닙니다.

이 패턴은 이해하기 어려울 수 있으며 일반적으로 피하는 것이 가장 좋습니다. 하지만 Effect에서 state를 업데이트하는 것보다는 낫습니다. 렌더링 도중 `set` 함수를 호출하면 React는 컴포넌트가 `return` 문으로 종료된 직후, 자식을 렌더링하기 전에 해당 컴포넌트를 리렌더링 합니다. 이렇게 하면 자식 컴포넌트를 두 번 렌더링할 필요가 없습니다. 나머지 컴포넌트 함수는 계속 실행되고 결과는 버려집니다. 조건이 모든 Hook 호출보다 아래에 있으면 이른(early) `return;` 을 통해 렌더링을 더 일찍 다시 시작할 수 있습니다.

문제 해결

state를 업데이트했지만 로그에는 계속 이전 값이 표시됩니다

`set` 함수를 호출해도 **실행 중인 코드의 state는 변경되지 않습니다**.

```
function handleClick() {
  console.log(count);  // 0

  setCount(count + 1); // 1로 리렌더링 요청합니다.
  console.log(count);  // 아직 0입니다!

  setTimeout(() => {
    console.log(count); // 여기도 0이고요!
  }, 5000);
}
```

그 이유는 state가 스냅샷처럼 동작하기 때문입니다. state를 업데이트하면 새로운 state 값으로 다른 렌더링을 요청하지만 이미 실행 중인 이벤트 핸들러의 `count` 변수에는 영향을 미치지 않습니다.

다음 state를 사용해야 하는 경우에는, `set` 함수에 전달하기 전에 변수에 저장할 수 있습니다:

```
const nextCount = count + 1;
setCount(nextCount);

console.log(count);     // 0
console.log(nextCount); // 1
```

---

state를 업데이트해도 화면이 바뀌지 않습니다

React는 Object.is로 비교한 뒤 **다음 state가 이전 state와 같으면 업데이트를 무시**합니다. 이는 보통 객체나 배열의 state를 직접 변경할 때 발생합니다.

```
obj.x = 10;  // 🚩 잘못된 방법: 기존 객체를 변경
setObj(obj); // 🚩 아무것도 하지 않습니다.
```

기존 `obj` 객체를 변경한 후 다시 `setObj` 로 전달했기 때문에 React가 업데이트를 무시했습니다. 이 문제를 해결하려면 객체나 배열 state를 변경하는 대신 항상 교체해야 합니다.

```
// ✅ 올바른 방법: 새로운 객체 생성
setObj({
  ...obj,
  x: 10
});
```

에러가 발생했습니다: “리렌더링 횟수가 너무 많습니다”

다음과 같은 에러가 발생할 수 있습니다. `리렌더링 횟수가 너무 많습니다. React는 무한 루프를 방지하기 위해 렌더링 횟수를 제한합니다.` 전형적으로 이는 *렌더링 중*에 state를 무조건적으로 설정하고 있음을 의미 하기 때문에, 컴포넌트가 렌더링, state 설정(렌더링 유발), 렌더링, state

설정(렌더링 유발) 등의 루프에 들어가는 것입니다. 이 문제는 이벤트 핸들러를 지정하는 과정에서 실수로 발생하는 경우가 많습니다.

```
// 🚩 잘못된 방법: 렌더링 동안 핸들러 요청
return <button onClick={handleClick()}>Click me</button>

// ✅ 올바른 방법: 이벤트 핸들러로 전달
return <button onClick={handleClick}>Click me</button>

// ✅ 올바른 방법: 인라인 함수로 전달
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

이 에러의 원인을 찾을 수 없는 경우, 콘솔에서 에러 옆에 있는 화살표를 클릭한 뒤 JavaScript 스택에서 에러의 원인이 되는 특정 `set` 함수 호출을 찾아보세요.

초기화 함수 또는 업데이터 함수가 두 번 실행됩니다

Strict Mode에서 React는 일부 함수를 한 번이 아닌 두 번 호출합니다.

```
function TodoList() {
  // 해당 함수형 컴포넌트는 렌더링 될 때마다 두 번 호출됩니다.

  const [todos, setTodos] = useState(() => {
    // 해당 초기화 함수는 초기화 동안 두 번 호출됩니다.
    return createTodos();
  });

  function handleClick() {
    setTodos(prevTodos => {
      // 해당 업데이터 함수는 클릭할 때마다 두 번 호출됩니다.
      return [...prevTodos, createTodo()];
    });
  }
  // ...
```

이는 정상적인 현상이며, 이로 인해 코드가 손상되지 않아야 합니다.

이 **개발 환경 전용** 동작은 컴포넌트를 순수하게 유지하는 데 도움이 됩니다. React는 호출 중 하나의 결과를 사용하고 다른 호출의 결과는 무시합니다. 컴포넌트, 초기화 함수, 업데이터 함수가 순수하다면 이 동작이 로직에 영향을 미치지 않습니다. 반면 의도치 않게 순수하지 않을 경우에는 실수를 알아차리는 데 도움이 됩니다.

예를 들어, 순수하지 않은 업데이터 함수는 state의 배열을 다음과 같이 변경합니다.

```
setTodos(prevTodos => {
  // 🚩 실수: state 변경
  prevTodos.push(createTodo());
});
```

React는 업데이터 함수를 두 번 호출하기 때문에 할 일이 두 번 추가되었음을 알 수 있으므로, 실수가 있음을 파악할 수 있습니다. 이 예시에서는 배열을 변경하는 대신 교체하여 실수를 수정할 수 있습니다:

```
setTodos(prevTodos => {
  // ✅ 올바른 방법: 새로운 state로 교체
  return [...prevTodos, createTodo()];
});
```

이제 이 업데이터 함수는 순수하므로 한 번 더 호출해도 동작에 차이가 없습니다. 그렇기 때문에 React가 두 번 호출하는 것이 실수를 찾는 데 도움이 된다는 것입니다. **컴포넌트, 초기화 함수, 업데이터 함수는 순수해야 합니다**. 이벤트 핸들러는 순수할 필요가 없으므로 React는 이벤트 핸들러를 두 번 호출하지 않습니다.

자세히 알아보려면 컴포넌트 순수하게 유지하기를 읽어보세요.

---

state의 값으로 함수를 설정하려고 하면 설정 대신 호출됩니다

state에 함수를 넣을 수 없습니다.

```
const [fn, setFn] = useState(someFunction);
```

```
function handleClick() {
  setFn(someOtherFunction);
}
```

함수를 값으로 전달하면 React는 `someFunction`을 초기화 함수로 여기고, `someOtherFunction`은 업데이터 함수라고 여깁니다. 따라서 이들을 호출해서 그 결과를 저장하려고 시도합니다. 정말로 함수를 저장하길 원한다면, 두 경우 모두 함수 앞에 `() =>`를 넣어야 합니다. 그러면 React는 전달한 함수를 값으로 저장합니다.

```
const [fn, setFn] = useState(() => someFunction);

function handleClick() {
  setFn(() => someOtherFunction);
}
```


이전
useRef

다음
useSyncExternalStore

커뮤니티

행동 강령

팀 소개

문서 기여자

감사의 말

더 보기

블로그

React Native

개인 정보 보호

약관