useContext

`useContext`는 컴포넌트에서 Context를 읽고 구독할 수 있는 React Hook입니다.

```
const value = useContext(SomeContext)
```

레퍼런스

`useContext(SomeContext)`

`useContext`를 컴포넌트의 최상위 수준에서 호출하여 Context를 읽고 구독합니다.

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

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

아래 예시를 참고하세요.

매개변수

- `SomeContext`: `createContext` 로 생성한 Context입니다. Context 자체는 정보를 담고 있지 않으며, 컴포넌트에서 제공하거나 읽을 수 있는 정보의 종류를 나타냅니다.

반환값

`useContext` 는 호출하는 컴포넌트에 대한 Context 값을 반환합니다. 이 값은 트리에서 호출하는 컴포넌트 상위의 가장 가까운 `SomeContext` 에 전달된 값으로 결정됩니다. Provider가 없으면 반환된 값은 해당 Context에 대해 `createContext` 에 전달한 `defaultValue` 가 됩니다. 반환된 값은 항상 최신 상태입니다. Context가 변경되면 React는 자동으로 해당 Context를 읽는 컴포넌트를 다시 렌더링합니다.

주의 사항

- 컴포넌트 내의 `useContext()` 호출은 **동일한** 컴포넌트에서 반환된 Provider에 영향을 받지 않습니다. 해당하는 `<Context>` 는 `useContext()` 호출을 하는 컴포넌트 ***상위에*** **배치되어야 합니다.**
- React는 다른 `value` 을 받는 Provider로부터 시작해서 특정 Context를 사용하는 모든 자식들을 **자동으로 리렌더링**합니다. 이전 값과 다음 값은 `Object.is` 를 통해 비교합니다. `memo` 로 리렌더링을 건너뛰어도 자식들이 새로운 Context 값을 받는 것을 막지는 못합니다.
- 빌드 시스템이 결과물에 중복 모듈을 생성하는 경우(심볼릭 링크에서 발생할 수 있음) Context가 손상될 수 있습니다. Context를 통해 무언가를 전달하는 것은 `===` 비교에 의해 결정되는 것처럼 Context를 제공하는 데 사용하는 `SomeContext` 와 Context를 읽는 데 사용하는 `SomeContext` 가 ***정확하게*** **동일한 객체**인 경우에만 작동합니다.

사용법

트리의 깊은 곳에 데이터 전달하기

컴포넌트의 최상위 수준에서 `useContext` 를 호출하여 Context를 읽고 구독합니다.

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

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

`useContext`는 전달한 `Context`에 대한 `Context Value`를 반환합니다. Context 값을 결정하기 위해 React는 컴포넌트 트리를 탐색하고 특정 Context에 대해 **상위에서 가장 가까운 Context Provider**를 찾습니다.

Context를 `Button`에 전달하려면 해당 버튼 또는 상위 컴포넌트 중 하나를 해당 Context Provider로 감쌉니다.

```
function MyPage() {
  return (
    <ThemeContext value="dark">
      <Form />
    </ThemeContext>
  );
}

function Form() {
  // ... 내부에서 버튼을 렌더링합니다. ...
}
```

Provider와 `Button` 사이에 얼마나 많은 컴포넌트 레이어가 있는지는 중요하지 않습니다. `Form` 내부의 `Button`이 *어디에서나* `useContext(ThemeContext)`를 호출하면, `"dark"`를 값으로 받습니다.

주의하세요!

`useContext()` 는 항상 호출하는 컴포넌트 *상위*에서 가장 가까운 Provider를 찾습니다. 위쪽 방향으로 찾고 `useContext()` 를 호출하는 컴포넌트 안의 Provider는 **고려하지 않습니다.**

App.js

다운로드 새로고침 Clear 포크

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext value="dark">
      <Form />
    </ThemeContext>
  )
}
```

자세히 보기

Context를 통해 전달된 데이터 업데이트하기

때때로 Context가 시간이 지남에 따라 변경되기를 원할 것입니다. Context를 업데이트 하려면 State와 결합하세요. 부모 컴포넌트에서 State변수를 선언하고 현재 State를 Context Value 로 Provider에 전달합니다.

```
function MyPage() {
  const [theme, setTheme] = useState('dark');
  return (
    <ThemeContext value={theme}>
      <Form />
      <Button onClick={() => {
        setTheme('light');
      }}>
        Switch to light theme
      </Button>
    </ThemeContext>
  );
}
```

이제 Provider 내부의 모든 `Button` 은 현재 `theme` 값을 받게 됩니다. Provider에 전달된 `theme` 값을 업데이트 하기 위해 `setTheme` 을 호출하면, 모든 `Button` 컴포넌트가 새로운 `'light'` 값으로 다시 렌더링됩니다.

Context 업데이트 예시

1. Context를 통해 값 업데이트
2. Context를 통해 객체 업데이트
3. 다양한 Context

예시 1 of 5: Context를 통해 값 업데이트

이 예시에서 `MyApp` 컴포넌트는 State 변수를 가지고 있고, 이 State 변수는 `ThemeContext` Provider로 전달됩니다. “Use dark mode” 체크박스를 체크하면 State가 업데이트 됩니다. 제공된 값을 변경하면 해당 Context를 사용하는 모든 컴포넌트가 다시 렌더링됩니다.

App.js 다운로드 새로고침 Clear 포크

```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext value={theme}>
      <Form />
      <label>
        <input
          type="checkbox"
```

자세히 보기

`value="dark"`는 `"dark"` 문자열을 전달하지만, `value={theme}`는 JSX 중괄호를 사용하여 자바스크립트 `theme` 변수 값을 전달합니다. 중괄호를 사용하면 문자열이 아닌 Context 값도 전달할 수 있습니다.

다음 예시

Fallback 기본값 지정

React가 부모 트리에서 특정 Context Provider를 찾을 수 없는 경우, `useContext()` 가 반환하는 Context 값은 해당 Context를 생성할 때 지정한 기본값과 동일합니다.

```
const ThemeContext = createContext(null);
```

기본값은 **변경되지 않습니다**. Context를 업데이트하려면 위에서 설명한 대로 State와 함께 사용하세요.

예를 들어 `null` 대신에 기본값으로 사용할 수 있는 더 의미 있는 값이 있는 경우가 많습니다.

```
const ThemeContext = createContext('light');
```

이렇게 하면 실수로 해당 Provider 없이 일부 컴포넌트를 렌더링해도 깨지지 않습니다. 또한 테스트 환경에서 많은 Provider를 설정하지 않고도 컴포넌트가 테스트 환경에서 잘 작동하는 데 도움이 됩니다.

아래 예시에서 “Toggle theme” 버튼은 **테마 Context Provider의 외부**에 있고 기본 컨텍스트 테마 값이 `'light'` 이기 때문에 항상 밝게 표시되어 있습니다. 기본 테마를 `'dark'` 로 변경해 보세요.

App.js 다운로드 새로고침 Clear 포크

```
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext('light');

export default function MyApp() {
  const [theme, setTheme] = useState('light');
  return (
    <>
      <ThemeContext value={theme}>
        <Form />
      </ThemeContext>
```

<Button onClick={() => {

자세히 보기

트리의 일부 Context 오버라이딩 하기

트리의 일부분을 다른 값의 Provider로 감싸서 해당 부분에 대한 Context를 오버라이딩 할 수 있습니다.

```
<ThemeContext value="dark">
  ...
  <ThemeContext value="light">
    <Footer />
  </ThemeContext>
  ...
</ThemeContext>
```

필요한 만큼 Provider를 중첩하고 오버라이딩 할 수 있습니다.

Context 오버라이딩 예시

예시 1 of 2: 테마 오버라이드

여기서 `Footer` *내부의* 버튼은 외부의 버튼(`"dark"`)과 다른 Context 값(`"light"`)을 받습니다.

App.js 다운로드 새로고침 Clear 포크

```
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
  return (
    <ThemeContext value="dark">
      <Form />
    </ThemeContext>
  )
}
```

자세히 보기

다음 예시

객체와 함수를 전달할 때 리렌더링 최적화하기

Context를 통해 객체와 함수를 포함한 모든 값을 전달할 수 있습니다.

```
function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  function login(response) {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }

  return (
    <AuthContext value={{ currentUser, login }}>
      <Page />
    </AuthContext>
  );
}
```

여기서 Context Value 는 두 개의 프로퍼티를 가진 자바스크립트 객체이며, 그 중 하나는 함수입니다. `MyApp` 이 다시 렌더링할 때마다(예를 들어 경로 업데이트 시) *다른* 함수를 가리키는 *다른* 객체가 될 것이므로 React는 `useContext(AuthContext)` 를 호출하는 트리 깊숙한 곳에 있는 모든 컴포넌트도 다시 렌더링해야 합니다.

작은 앱에서는 문제가 되지 않습니다. 그러나 `currentUser` 와 같은 기본적인 데이터가 변경되지 않았다면 다시 렌더링할 필요가 없습니다. React가 이 사실을 활용할 수 있도록 `login` 함수를 `useCallback` 으로 감싸고 객체 생성을 `useMemo` 로 감싸면 됩니다. 이것이 성능 최적화입니다.

```
import { useCallback, useMemo } from 'react';

function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);
```

```
  const login = useCallback((response) => {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }, []);

  const contextValue = useMemo(() => ({
    currentUser,
    login
  }), [currentUser, login]);

  return (
    <AuthContext value={contextValue}>
      <Page />
    </AuthContext>
  );
}
```

이 변경으로 인해 `MyApp` 이 다시 렌더링해야 하는 경우에도 `currentUser` 가 변경되지 않는 한 `useContext(AuthContext)` 를 호출하는 컴포넌트는 다시 렌더링할 필요가 없습니다.

`useMemo` 와 `useCallback` 에 대해 자세히 알아보세요.

문제 해결

컴포넌트가 Provider에서 값을 인식하지 못하고 있습니다.

이런 일이 발생하는 몇 가지 이유가 있습니다.

1. `useContext()` 를 호출하는 컴포넌트와 동일한 컴포넌트(또는 그 아래)에서 `<SomeContext>` 를 렌더링하는 경우, `<SomeContext>` 를 `useContext()` 를 호출하는 컴포넌트의 *위와 바깥*으로 이동하세요.
2. 컴포넌트를 `<SomeContext>` 로 감싸는 것을 잊었거나 생각했던 것과 다른 트리의 다른 부분에 배치했을 수 있습니다. React 개발자 도구를 사용하여 계층 구조가 올바른지 확인하세요.
3. 사용 중인 도구에서 발생하는 빌드 문제로 인해, 제공하는 컴포넌트에서의 `someContext` 와 값을 읽는 컴포넌트에서의 `someContext` 가 서로 다른 객체로 처리되는 문제가 발생할 수 있습니다. 예를 들어 심볼릭 링크를 사용하는 경우 이런 문제가 발생할 수 있습니다. 이를 확인하려면 `window.SomeContext1` 과 `window.SomeContext2` 를 전역에 할당하고 콘솔에서

`window.SomeContext1 === window.SomeContext2` 인지 확인하면 됩니다. 동일하지 않은 경우 빌드 도구 수준에서 해당 문제를 수정하세요.

기본값이 다른데도 Context가 `undefined`를 반환합니다.

트리에 `value`가 없는 Provider가 있을 수 있습니다.

```
// 🚩 Doesn't work: no value prop
<ThemeContext>
   <Button />
</ThemeContext>
```

`value`를 지정하는 것을 잊어버린 경우, `value={undefined}`를 전달하는 것과 같습니다.

실수로 다른 Prop의 이름을 실수로 사용했을 수도 있습니다.

```
// 🚩 Doesn't work: prop should be called "value"
<ThemeContext theme={theme}>
   <Button />
</ThemeContext>
```

두 가지 경우 모두 콘솔에 React에서 경고가 표시될 것입니다. 이를 수정하려면 Prop `value`를 호출하세요.

```
// ✅ Passing the value prop
<ThemeContext value={theme}>
   <Button />
</ThemeContext>
```

`createContext(defaultValue)` 호출의 기본값은 **위에 일치하는 Provider가 전혀 없는 경우**에만 사용된다는 점에 유의하세요. 부모 트리 어딘가에 `<SomeContext value={undefined}>` 컴포넌트가 있는 경우, `useContext(SomeContext)`를 호출하는 컴포넌트는 `undefined`를 Context 값으로 받습니다.

이전
useCallback

다음
useDebugValue