컴포넌트와 Hook은 순수해야 합니다

순수 함수는 오직 계산만 수행하고 그 외의 작업은 하지 않습니다. 이는 코드의 이해와 디버깅을 더 쉽게 만들어 주며 React가 컴포넌트와 Hook을 자동으로 최적화할 수 있게 해줍니다.

중요합니다!

이 페이지는 고급 주제를 다루며 컴포넌트 순수하게 유지하기에서 다룬 개념에 대한 이해가 필요합니다.

순수성이 중요한 이유

React를 만드는 핵심 개념 중 하나는 *순수성* 입니다. 순수한 컴포넌트나 Hook은 다음과 같습니다.

- **멱등성** – 컴포넌트의 Props, State, Context 혹은 Hook의 인수에 대한 동일한 입력으로 실행할 때마다 항상 같은 결과를 얻습니다.
- **렌더링 시 사이드 이펙트가 없어야 합니다** – 사이드 이펙트가 있는 코드는 **렌더링과 별도로 실행**해야 합니다. 예를 들어, 이벤트 핸들러를 통해 사용자가 UI와 상호작용하여 UI를 업데이트하는 경우나, 렌더링 후에 실행되는 Effect를 사용할 수 있습니다.
- **지역 변수가 아닌 값을 변경하지 마세요** – 컴포넌트와 Hook은 렌더링 시 지역에서 생성되지 않은 값을 수정하지 않아야 합니다.

렌더링을 순수하게 유지하면, React는 어떤 업데이트가 사용자에게 먼저 보이는 것이 중요한지를 이해할 수 있습니다. 이는 렌더링의 순수성 덕분에 가능합니다. 컴포넌트가 렌더링하는 동안 사이드 이펙트가 없기 때문에 React는 중요하지 않은 컴포넌트의 렌더링을 일시 중지하고 나중에 필요할 때 다시 돌아올 수 있습니다.

구체적으로, 이는 렌더링 로직을 여러 번 실행할 수 있어 React가 사용자에게 쾌적한 사용자 경험을 제공할 수 있음을 의미합니다. 그러나 렌더링 중에 전역 변수 값을 수정하는 것처럼 컴포넌트에 추적되지 않은 사이드 이펙트가 있는 경우, React가 렌더링 코드를 다시 실행할 때 사이드 이펙트가 원하지 않는 방식으로 트리거될 수 있습니다. 이는 종종 예상치 못한 버그로 이어져 사용자의 앱 경험을 저하할 수 있습니다. 컴포넌트를 순수하게 유지하기에서 이 예시를 볼 수 있습니다.

React가 코드를 실행하는 방식

React는 선언적입니다. React에 *무엇을* 렌더링할지 말해주면, React는 사용자에게 *어떻게* 최적으로 표시할지 알아냅니다. 이를 위해 React는 코드를 실행하는 몇 가지 단계를 거칩니다. React를 잘 사용하기 위해 이 모든 단계를 알 필요는 없습니다. 하지만 고수준에서 *렌더링* 중에 실행되는 코드와 그 외부에서 실행되는 코드에 대해 알아야 합니다.

*렌더링*은 UI의 다음 버전이 어떻게 보일지 계산하는 것을 의미합니다. 렌더링 후에 Effect가 *flushed* 됩니다. (더 이상 남아 있지 않을 때까지 실행됨.) 그리고 Effect가 레이아웃에 영향을 미치는 경우, 계산을 업데이트할 수 있습니다. React는 새로운 계산을 이전 UI 버전을 만드는 데 사용된 계산과 비교한 다음, 최신 버전과 일치시키기 위해 DOM(사용자가 실제로 보는 것)에 필요한 최소한의 변경만을 *커밋* 합니다.

자세히 살펴보기

코드가 렌더링 중에 실행되는지 확인하는 방법

자세히 보기

컴포넌트와 Hook은 멱등해야 합니다

컴포넌트는 항상 Props, State, Context와 같은 입력에 대해 동일한 출력을 반환해야 합니다. 이를 *멱등성*이라고 합니다. 멱등성은 함수형 프로그래밍에서 널리 사용되는 용어입니다. 이는 동일한 입력으로 해당 코드를 실행할 때마다 항상 동일한 결과를 얻는 것을 의미합니다.

이는 *모든* 코드가 렌더링 중에도 멱등성을 유지해야 한다는 것을 의미합니다. 예를 들어 다음 코드 라인은 멱등하지 않습니다. (따라서 컴포넌트도 멱등하지 않습니다.)

```
function Clock() {
  const time = new Date(); // 🔴 Bad: 항상 다른 결과를 반환합니다!
  return <span>{time.toLocaleString()}</span>
}
```

`new Date()` 함수는 항상 현재 날짜를 반환하고 호출할 때마다 결과가 변경되기 때문에 멱등하지 않습니다. 위 컴포넌트를 렌더링하면 화면에 표시되는 시간이 컴포넌트가 렌더링된 시간에 고정됩니다. 마찬가지로 `Math.random()` 과 같은 함수도 멱등하지 않습니다. 입력이 동일해도 호출할 때마다 다른 결과를 반환하기 때문입니다.

이는 `new Date()` 와 같은 비멱등 함수를 *전혀* 사용하지 말아야 한다는 뜻이 아닙니다. 단지 이를 렌더링 중에 사용하지 않아야 한다는 것입니다. 이 경우 Effect를 사용하여 최신 날짜를 컴포넌트에 *동기화* 할 수 있습니다.

App.js 다운로드 새로고침 Clear 포크

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

function useTime() {
  // 1. 현재 날짜의 State를 추적합니다. `useState`는 초기 State로 초기화 함수를 받습니다.
  //    이 함수는 Hook이 호출될 때 한 번만 실행되므로, Hook이 호출되는 시점의 현재 날짜가
  //    처음으로 설정됩니다.
```

```
  const [time, setTime] = useState(() => new Date());

  useEffect(() => {
    // 2. `setInterval`을 사용하여 현재 날짜를 매초마다 업데이트합니다.
    const id = setInterval(() => {
```

자세히 보기

멱등하지 않은 `new Date()` 호출을 Effect로 감싸면, 그 계산을 렌더링 외부로 이동시킵니다.

React와 외부 상태를 동기화할 필요가 없다면, 사용자 상호작용에 대한 응답으로만 업데이트할 경우 이벤트 핸들러를 사용하는 것도 고려해 볼 수 있습니다.

사이드 이펙트는 렌더링 외부에서 실행되어야 합니다

사이드 이펙트는 렌더링 중에 실행되어서는 안 됩니다. 이는, React가 최상의 사용자 경험을 제공하기 위해 컴포넌트를 여러 번 렌더링할 수 있기 때문입니다!

중요합니다!

사이드 이펙트는 Effect보다 더 넓은 개념입니다. Effect는 특히 `useEffect` 안에 감싸진 코드를 지칭하는 반면, 사이드 이펙트는 호출자에게 값을 반환하는 기본 결과 외에도 어떤 관찰 가능한 영향을 미치는 코드를 지칭하는 일반적인 용어입니다.

사이드 이펙트는 일반적으로 이벤트 핸들러나 Effect 안에서 작성합니다. 하지만 렌더링 중에는 절대로 작성해서는 안 됩니다.

렌더링은 순수하게 유지되어야 하지만, 화면에 무언가를 보여주는 것처럼 앱이 흥미로운 동작을 하려면 어느 시점에서는 사이드 이펙트가 필요합니다! 이 규칙의 핵심은 React가 컴포넌트를 여러 번 렌더링할 수 있기 때문에, 사이드 이펙트가 렌더링 중에 실행되어서는 안 된다는 것입니다. 대부분의 경우 이벤트 핸들러를 사용하여 사이드 이펙트를 처리합니다. 이벤트 핸들러를 사용하면 이 코드가 렌더링 중에 실행될 필요가 없다는 것을 React에 명시적으로 알려주어 렌더링을 순수하게 유지합니다. 모든 옵션을 다 시도해 보고 나서도 해결되지 않는 경우, 정말 최후의 수단으로 `useEffect` 를 사용하여 사이드 이펙트를 처리할 수도 있습니다.

변경이 허용되는 경우

지역 변경

사이드 이펙트의 일반적인 예로 변경Mutation이 있습니다. 자바스크립트에서 변경은 원시 값이 아닌 값을 변경하는 것을 의미합니다. 흔히 React에서는 변경을 권장하지 않지만, *지역* 변경은 전혀 문제가 되지 않습니다.

```
function FriendList({ friends }) {
  const items = []; // ✅ Good: 지역 변수로 생성됨.
  for (let i = 0; i < friends.length; i++) {
    const friend = friends[i];
    items.push(
      <Friend key={friend.id} friend={friend} />
    ); // ✅ Good: 지역 변경은 괜찮습니다.
  }
  return <section>{items}</section>;
}
```

지역 변경을 피하고자 코드를 비틀 필요는 없습니다. `Array.map` 을 사용하여 간결하게 만들 수도 있지만, 지역 배열을 생성한 다음 렌더링 중에 항목을 추가하는 것은 전혀 문제가 되지 않습니다.

비록 `items` 를 변경시키는 것처럼 보이지만, 이 코드를 *지역에서만* 변경한다는 점이 중요합니다. 즉 이 변경은 컴포넌트가 다시 렌더링될 때 "기억되지" 않습니다. `items` 는 컴포넌트가 존재하는 동안에만 유지됩니다. `<FriendList />` 가 렌더링 될 때마다 `items` 는 항상 *재생성* 되므로, 컴포넌트는 항상 동일한 결과를 반환합니다.

반면 `items` 가 컴포넌트 외부에서 생성되면 이전 값을 유지하고 변경 사항을 기억합니다.

```
const items = []; // 🔴 Bad: 컴포넌트 외부에서 생성됨.
function FriendList({ friends }) {
  for (let i = 0; i < friends.length; i++) {
    const friend = friends[i];
    items.push(
      <Friend key={friend.id} friend={friend} />
    ); // 🔴 Bad: 렌더링 외부에서 생성된 값을 변경합니다.
  }
  return <section>{items}</section>;
}
```

`<FriendList />` 가 다시 실행될 때마다 `items` 에 `friends` 를 계속 추가하여 중복된 결과가 여러 번 나타나게 됩니다. 이 버전의 `<FriendList />` 는 렌더링 중에 관찰 가능한 사이드 이펙트가 발생하며 **규칙을 위반**합니다.

Lazy 초기화

Lazy 초기화는 완전히 "순수"하지 않더라도 괜찮습니다.

```
function ExpenseForm() {
  SuperCalculator.initializeIfNotReady(); // ✅ Good: 다른 컴포넌트에 영향을 미치지 않는 경
  // 계속 렌더링...
}
```

DOM 변경

사용자에게 직접적으로 보이는 사이드 이펙트는 React 컴포넌트의 렌더링 로직에서 허용되지 않습니다. 즉, 단순히 컴포넌트 함수를 호출하는 것만으로 화면에 변화를 일으켜서는 안 됩니다.

```
function ProductDetailPage({ product }) {
  document.title = product.title; // 🔴 Bad: DOM을 변경함
}
```

원하는 결과를 얻기 위해 렌더링 외부에서 `document.title` 을 업데이트하는 한 가지 방법은 컴포넌트를 `document` 와 동기화하는 것입니다.

컴포넌트를 여러 번 호출해도 안전하고 다른 컴포넌트의 렌더링에 영향을 주지 않는 한, React는 엄격한 함수형 프로그래밍의 의미에서 100% 순수하지 않아도 상관하지 않습니다. 더 중요한 것은 컴포넌트가 반드시 멱등해야 한다는 것입니다.

Props와 State는 불변입니다

컴포넌트의 Props와 State는 불변의 스냅샷입니다. 직접적으로 변경하면 안 됩니다. 대신 새로운 Props를 전달하거나 `useState` 에서 제공하는 Setter 함수를 사용하세요.

Props와 State 값을 렌더링 후에 업데이트되는 스냅샷으로 생각할 수 있습니다. 이러한 이유로 Props나 State 변수를 직접 수정하지 않아야 합니다. 새로운 Props를 전달하거나 제공된 Setter 함수를 사용하여 컴포넌트가 다음 번에 렌더링 될 때 State가 업데이트되어야 함을 React에 알려줍니다.

Props를 변경하지 마세요

Props는 불변입니다. Props를 변경한다면 애플리케이션이 일관성 없는 출력을 생성하게 되며, 상황에 따라 작동할 수도 있고 안 할 수도 있기 때문에 디버깅이 어려워질 수 있습니다.

```
function Post({ item }) {
  item.url = new Url(item.url, base); // 🔴 Bad: 절대 Props를 직접 변경하지 마세요.
  return <Link url={item.url}>{item.title}</Link>;
}
```

```
function Post({ item }) {
  const url = new Url(item.url, base); // ✅ Good: 대신, 복사본을 만드세요.
  return <Link url={url}>{item.title}</Link>;
}
```

State를 변경하지 마세요

`useState` 는 State 변수와 그 State를 변경하기 위한 Setter 함수를 반환합니다.

```
const [stateVariable, setter] = useState(0);
```

State 변수를 직접 변경하는 대신, `useState` 에 의해 반환된 Setter 함수를 사용하여 변경해야 합니다. State 변수의 값을 변경하면 컴포넌트가 변경되지 않아 사용자는 이전 UI를 보게 됩니다. Setter 함수를 사용하면 React에 State가 변경되었으며 UI를 변경하기 위해 리렌더링을 대기열에 추가해야 한다는 것을 알립니다.

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

  function handleClick() {
    count = count + 1; // 🔴 Bad: 절대 State를 직접 변경하지 마세요.
  }

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

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

  function handleClick() {
    setCount(count + 1); // ✅ Good: useState에서 반환된 setter 함수를 사용하세요
```

```
  }

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

Hook의 반환값과 인수는 불변입니다

값이 Hook에 전달되면, 이를 수정해서는 안 됩니다. JSX의 Props와 마찬가지로 Hook에 전달된 값은 불변입니다.

```
function useIconStyle(icon) {
  const theme = useContext(ThemeContext);
  if (icon.enabled) {
    icon.className = computeStyle(icon, theme); // 🔴 Bad: 절대 Hook 인수를 직접 변경하지
  }
  return icon;
}
```

```
function useIconStyle(icon) {
  const theme = useContext(ThemeContext);
  const newIcon = { ...icon }; // ✅ Good: 대신 복사본을 만드세요.
  if (icon.enabled) {
    newIcon.className = computeStyle(icon, theme);
  }
  return newIcon;
}
```

React의 중요한 원칙 중 하나는 *지역 추론* 입니다. 이는 코드를 독립적으로 살펴봄으로써 컴포넌트나 Hook이 하는 일을 이해할 수 있는 능력입니다. Hook은 호출될 때 “블랙박스”처럼 다뤄져야 합니다. 예를 들어 커스텀 Hook은 내부에서 값을 메모이제이션 하기 위해 인수를 의존성으로 사용할 수 있습니다.

```
function useIconStyle(icon) {
  const theme = useContext(ThemeContext);

  return useMemo(() => {
    const newIcon = { ...icon };
    if (icon.enabled) {
      newIcon.className = computeStyle(icon, theme);
    }
    return newIcon;
  }, [icon, theme]);
}
```

Hook 인수를 변경하면 커스텀 Hook의 메모이제이션이 잘못되므로, 이를 피하는 것이 중요합니다.

```
style = useIconStyle(icon);          // `style`은 `icon`을 기준으로 메모이제이션됨.
icon.enabled = false;                // Bad: 🔴 절대 Hook 인수를 직접 변경하지 마세요.
style = useIconStyle(icon);          // 이전에 메모이제이션된 결과가 반환됨.
```

```
style = useIconStyle(icon);          // `style`은 `icon`을 기준으로 메모이제이션됨.
icon = { ...icon, enabled: false }; // Good: ✅ 대신 복사본을 만드세요.
style = useIconStyle(icon);          // `style`의 새로운 값이 계산됨.
```

마찬가지로 Hook의 반환 값을 수정하지 않는 것이 중요합니다. 반환 값이 메모이제이션되어 있을 수 있기 때문입니다.

JSX로 전달된 값은 불변입니다

JSX에 사용된 후에는 값을 변경하지 마세요. JSX가 생성되기 전에 변경을 수행하세요.

JSX를 표현식에서 사용할 때, React는 컴포넌트의 렌더링이 끝나기 전에 JSX를 성급하게 평가할 수 있습니다. 이는 JSX에 전달된 후에 값을 변경하면 React가 컴포넌트의 출력을 업데이트할지 여부를 알지 못하므로 오래된 UI로 이어질 수 있음을 의미합니다.

```jsx
function Page({ colour }) {
  const styles = { colour, size: "large" };
  const header = <Header styles={styles} />;
  styles.size = "small"; // 🔴 Bad: `styles`는 이미 위의 JSX에서 사용되었습니다.
  const footer = <Footer styles={styles} />;
  return (
    <>
      {header}
      <Content />
      {footer}
    </>
  );
}
```

```jsx
function Page({ colour }) {
  const headerStyles = { colour, size: "large" };
  const header = <Header styles={headerStyles} />;
  const footerStyles = { colour, size: "small" }; // ✅ Good: 새로운 값을 생성했습니다.
  const footer = <Footer styles={footerStyles} />;
  return (
    <>
      {header}
      <Content />
      {footer}
    </>
  );
}
```


이전
개요

다음
React가 컴포넌트와 Hook을 호출하는 방식

Meta Open Source

uwu?

React 학습하기

- 빠르게 시작하기
- 설치하기
- UI 표현하기
- 상호작용성 더하기
- State 관리하기
- 탈출구

API 참고서

- React APIs
- React DOM APIs

커뮤니티

- 행동 강령
- 팀 소개
- 문서 기여자
- 감사의 말

더 보기

- 블로그
- React Native
- 개인 정보 보호
- 약관