<StrictMode>

`<StrictMode>`를 통해 개발 중에 컴포넌트에서 일반적인 버그를 빠르게 찾을 수 있도록 합니다.

```
<StrictMode>
  <App />
</StrictMode>
```

레퍼런스

`<StrictMode>`

컴포넌트 트리 내부에서 추가적인 개발 동작 및 경고를 활성화하기 위해 `StrictMode`를 사용하세요.

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

```
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

아래 예시를 참고하세요.

Strict Mode는 다음과 같은 개발 전용 동작을 활성화합니다.

- 컴포넌트가 순수하지 않은 렌더링으로 인한 버그를 찾기 위해 추가로 다시 렌더링합니다.
- 컴포넌트가 Effect 클린업이 누락되어 발생한 버그를 찾기 위해 Effect를 다시 실행합니다.
- 컴포넌트가 Ref 클린업이 누락되어 발생한 버그를 찾기 위해 Ref 콜백을 다시 실행합니다.
- 컴포넌트가 더 이상 사용되지 않는 API를 사용하는지 확인합니다.

Props

`StrictMode` 는 Props를 받지 않습니다.

주의 사항

- `<StrictMode>` 로 래핑된 트리 내에서 Strict Mode를 해제할 수 있는 방법은 없습니다. 이를 통해 `<StrictMode>` 내부의 모든 컴포넌트가 검사되었음을 확신할 수 있습니다. 제품을 개발하는 두 팀이 검사가 가치 있는지에 대해 의견이 갈리는 경우, 합의에 도달하거나 `<StrictMode>` 를 트리에서 하단으로 옮겨야 합니다.

사용법

전체 앱에 대해 Strict Mode 활성화

Strict Mode는 `<StrictMode>` 컴포넌트 내부의 모든 컴포넌트 트리에 대해 추가적인 개발 전용 검사를 활성화합니다. 이러한 검사는 개발 프로세스 초기에 컴포넌트에서 일반적인 버그를 찾는 데 도움이 됩니다.

전체 앱에 대한 Strict Mode를 활성화하려면 렌더링할 때 루트 컴포넌트를 `<StrictMode>` 로 래핑하세요.

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

전체 앱을 (특히 새로 생성된 앱의 경우) Strict Mode로 래핑하는 것을 권장합니다. `createRoot` 를 호출하는 프레임워크를 사용하는 경우, Strict Mode를 활성화하는 방법에 대한 문서를 확인하세요.

Strict Mode 검사는 **개발 환경에서만 실행되지만**, 이미 코드에 존재하는 버그를 찾아내는 데 도움을 줍니다. 이러한 버그는 실제 운영 환경에서 재현하기 까다로울 수 있습니다. Strict Mode를 사용하면 사용자가 보고하기 전에 버그를 수정할 수 있습니다.

중요합니다!

Strict Mode에서는 개발 시 다음과 같은 검사를 가능하게 합니다.

- 컴포넌트가 순수하지 않은 렌더링으로 인한 버그를 찾기 위해 추가로 다시 렌더링합니다.
- 컴포넌트가 Effect 클린업이 누락되어 발생한 버그를 찾기 위해 Effect를 다시 실행합니다.
- 컴포넌트가 Ref 클린업이 누락되어 발생한 버그를 찾기 위해 Ref 콜백을 다시 실행합니다.
- 컴포넌트가 더 이상 사용되지 않는 API를 사용하는지 확인합니다.

이러한 모든 검사는 개발 환경 전용이며 프로덕션 빌드에는 영향을 미치지 않습니다.

앱의 일부분에서 Strict Mode 활성화

애플리케이션의 어떤 부분에서라도 Strict Mode를 활성화할 수 있습니다.

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

function App() {
  return (
    <>
      <Header />
      <StrictMode>
        <main>
          <Sidebar />
          <Content />
        </main>
      </StrictMode>
      <Footer />
    </>
  );
}
```

이 예시에서 `Header` 와 `Footer` 컴포넌트에서는 Strict Mode 검사가 실행되지 않습니다. 그러나 `Sidebar` 와 `Content` , 그리고 그 자손 컴포넌트는 깊이에 상관없이 검사가 실행됩니다.

중요합니다!

앱의 일부에서 `StrictMode` 가 활성화되면 React는 실제 운영 환경에서만 가능한 동작만을 허용합니다. 예를 들어, 앱의 루트에서 `<StrictMode>` 가 활성화되지 않으면 초기 마운트 시 Effect를 다시 실행하지 않습니다. 이는 부모 Effect 없이 자식 Effect가 두 번 실행되는 상황을 방지하기 위함이며, 이러한 상황은 실제 운영 환경에서는 발생하지 않습니다.

개발 중 이중 렌더링으로 발견한 버그 수정

React는 작성하는 모든 컴포넌트가 순수 함수라고 가정합니다. 이것은 React 컴포넌트는 항상 동일한 입력(Props, State, Context)에 대해 동일한 JSX를 반환해야 한다는 것을 의미합니다.

이 규칙을 위반하는 컴포넌트는 예기치 않게 동작하며 버그를 일으킵니다. Strict Mode는 실수로 작성된 순수하지 않은 코드를 찾아내기 위해 몇 가지 함수(순수 함수여야 하는 것만)를 **개발 환경에서 두 번 호출**합니다. 이에는 다음이 포함됩니다.

- 컴포넌트 함수 본문. (단, 최상위 로직만 해당하며, 이벤트 핸들러 내부의 코드는 포함하지 않음.)
- `useState`, `set` 함수, `useMemo`, 또는 `useReducer` 에 전달한 함수.
- `constructor`, `render`, `shouldComponentUpdate` 와 같은 일부 클래스 컴포넌트 메소드. (전체 목록 보기)

함수가 순수한 경우 두 번 실행하여도 동작이 변경되지 않습니다. 순수 함수는 항상 같은 결과를 생성하기 때문입니다. 그러나 함수가 순수하지 않다면 (예: 받은 데이터를 변경하는 함수) 두 번 실행하면 보통 알아챌 수 있습니다. (이것이 바로 함수가 순수하지 않다는 것을 의미합니다!) 이는 버그를 조기에 발견하고 수정하는 데 도움이 됩니다.

다음은 Strict Mode의 이중 렌더링이 어떻게 버그를 조기에 발견하는 데 도움이 되는지 보여주는 예시입니다.

`StoryTray` 컴포넌트는 `stories` 배열을 받아 마지막에 “이야기 만들기” 항목을 추가합니다.

index.js App.js StoryTray.js 새로고침 Clear 포크

```
export default function StoryTray({ stories }) {
  const items = stories;
  items.push({ id: 'create', label: '이야기 만들기' });
  return (
    <ul>
      {items.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
```

위 코드에는 실수가 있습니다. 하지만 초기 출력이 올바르게 나타나기 때문에 놓치기 쉽습니다.

This mistake will become more noticeable if the `StoryTray` component re-renders multiple times. For example, let's make the `StoryTray` re-render with a different background color whenever you hover over it:

index.js App.js StoryTray.js 새로고침 Clear 포크

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

export default function StoryTray({ stories }) {
  const [isHover, setIsHover] = useState(false);
  const items = stories;
  items.push({ id: 'create', label: '이야기 만들기' });
  return (
    <ul
      onPointerEnter={() => setIsHover(true)}
      onPointerLeave={() => setIsHover(false)}
      style={{
        backgroundColor: isHover ? '#ddd' : '#fff'
```

자세히 보기

`StoryTray` 컴포넌트 위로 마우스를 가져갈 때마다 "이야기 만들기"가 목록에 다시 추가되는 것을 확인할 수 있습니다. 이 코드의 의도는 마지막에 한 번 추가하는 것이었습니다. 하지만 `StoryTray` 는 소품의 `stories` 배열을 직접 수정합니다. `StoryTray` 는 렌더링할 때마다 같은 배열의 끝에 "이야기 만들기"를 다시 추가합니다. 즉, `StoryTray` 는 순수 함수가 아니므로 여러 번 실행하면 다른 결과가 생성됩니다.

이 문제를 해결하기 위해 배열의 사본을 만든 다음 원본이 아닌 사본을 수정할 수 있습니다.

```
export default function StoryTray({ stories }) {
  const items = stories.slice(); // 배열 복제
  // ✅ Good: 새로운 배열에 추가
  items.push({ id: 'create', label: '이야기 만들기' });
```

이렇게 하면 `StoryTray` 함수를 순수하게 만들 수 있습니다. 함수가 호출될 때마다 배열의 사본만 수정하고, 외부 객체나 변수에는 영향을 미치지 않습니다. 이렇게 하면 버그를 해결할 수 있지만, 컴포넌트를 여러번 다시 렌더링하도록 만들어야 비로소 컴포넌트의 동작에 문제가 있다는 것이 명확해졌습니다.

원래 예시에서는 버그가 명확하지 않았습니다. 이제 원래 (버그가 있는) 코드를 `<StrictMode>` 로 래핑해 보겠습니다.

index.js App.js StoryTray.js 새로고침 Clear 포크

```
export default function StoryTray({ stories }) {
  const items = stories;
  items.push({ id: 'create', label: '이야기 만들기' });
  return (
    <ul>
      {items.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
```

Strict Mode에서는 *항상* 렌더링 함수를 두 번 호출하므로 실수를 바로 확인할 수 있습니다. ("이야기 만들기"가 두 번 나타남.) 따라서 프로세스 초기에 이러한 실수를 발견할 수 있습니다. 컴포넌트가 Strict Mode에서 렌더링되도록 수정하면 이전의 호버 기능과 같이 향후 발생할 수 있는 많은 프로덕션 버그도 *수정*할 수 있습니다.

index.js App.js StoryTray.js 새로고침 Clear 포크

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

export default function StoryTray({ stories }) {
```

```
  const [isHover, setIsHover] = useState(false);
  const items = stories.slice(); // 배열 복제
  items.push({ id: 'create', label: '이야기 만들기' });
  return (
    <ul
      onPointerEnter={() => setIsHover(true)}
      onPointerLeave={() => setIsHover(false)}
      style={{
```

자세히 보기

Strict Mode가 없으면 리렌더링을 더 추가하기 전까지는 버그를 놓치기 쉬웠습니다. Strict Mode를 사용하면 동일한 버그가 즉시 나타납니다. Strict Mode는 버그를 팀이나 사용자에게 푸시하기 전에 발견할 수 있도록 도와줍니다.

컴포넌트를 순수하게 유지하는 방법에 대해 자세히 알아보세요.

중요합니다!

React 개발자 도구가 설치되어 있다면, 두 번째 렌더링 호출 중 `console.log` 호출이 약간 흐리게 표시됩니다. React 개발자 도구는 이를 완전히 억제하는 설정(기본값은 꺼짐)

도 제공합니다.

개발 환경에서 Effect를 다시 실행하여 발견된 버그 수정

Strict Mode는 Effect의 버그를 찾는 데도 도움이 될 수 있습니다.

모든 Effect에는 몇 가지 셋업 코드가 있고 어쩌면 클린업 코드가 있을 수 있습니다. 일반적으로 React는 컴포넌트가 *마운트*(화면에 추가)될 때 셋업을 호출하고 컴포넌트가 *마운트 해제*(화면에서 제거)될 때 클린업을 호출합니다. 그런 다음 React는 마지막 렌더링 이후로부터 의존성이 변경된 경우 클린업과 셋업을 다시 호출합니다.

Strict Mode가 켜져 있으면 React는 **모든 Effect에 대해 개발 환경에서 한 번 더 셋업+클린업 사이클을 실행합니다.** 의외로 느껴질 수도 있지만 수동으로 파악하기 어려운 미묘한 버그를 드러내는 데 도움이 됩니다.

다음은 Strict Mode에서 Effect를 다시 실행하는 것이 버그를 조기에 발견하는 데 어떻게 도움이 되는지 보여주는 예시입니다.

컴포넌트를 채팅에 연결하는 이 예시를 살펴봅시다.

index.js App.js chat.js 새로고침 Clear 포크

```
import { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App';

const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

이 코드에는 문제가 있지만 즉시 파악하기 어려울 수 있습니다.

문제를 더 명확하게 드러내기 위해 기능을 구현해 보겠습니다. 아래 예시에서는 `roomId` 가 하드코딩되어 있지 않습니다. 대신 사용자가 연결하려는 `roomId` 를 드롭다운에서 선택할 수 있습니다. “대화 열기”을 클릭한 다음 다른 대화방을 하나씩 선택합니다. 콘솔에서 활성화된 연결 수를 추적합니다.

index.js App.js chat.js 새로고침 Clear 포크

```
import { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App';

const root = createRoot(document.getElementById("root"));
root.render(<App />);
```

열린 연결 수가 항상 계속 증가하는 것을 알 수 있습니다. 실제 앱에서는 성능 및 네트워크 문제가 발생할 수 있습니다. 문제는 Effect에 클린업 함수가 누락되었다는 것입니다.

```
useEffect(() => {
  const connection = createConnection(serverUrl, roomId);
  connection.connect();
  return () => connection.disconnect();
}, [roomId]);
```

이제 Effect가 자체적으로 "클린업"하고 오래된 연결을 파괴하므로 누수가 해결되었습니다. 그러나 더 많은 기능(선택 상자)을 추가하기 전까지는 문제가 드러나지 않았음을 알 수 있습니다.

원래 예시에서는 버그가 명확하지 않았습니다. 이제 원래 (버그가 있는) 코드를 `<StrictMode>`로 래핑해 보겠습니다.

index.js App.js chat.js 새로고침 Clear 포크

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App';

const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App />
```

```
  </StrictMode>
```

Strict Mode를 사용하면 문제가 있음을 즉시 알 수 있습니다(활성화된 연결 수가 2개로 증가함). Strict Mode는 모든 Effect에 대해 추가 셋업+클린업 사이클을 실행합니다. 이 Effect에는 클린업 로직이 없으므로 추가 연결을 생성하지만 파괴하지는 않습니다. 이것은 클린업 함수가 누락되었다는 힌트입니다.

Strict Mode를 사용하면 이러한 실수를 프로세스 초기에 발견할 수 있습니다. Strict Mode에서 클린업 함수를 추가하여 Effect를 수정하면 이전의 선택 상자와 같이 향후 프로덕션에서 발생할 수 있는 많은 버그 *또한* 수정합니다.

index.js App.js chat.js 새로고침 Clear 포크

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './styles.css';

import App from './App';

const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App />
```

```
    </StrictMode>
```

콘솔의 활성화된 연결 수가 더 이상 증가하지 않는 것을 확인할 수 있습니다.

Strict Mode가 없으면 Effect를 클린업해야 한다는 사실을 놓치기 쉬웠습니다. 개발 환경에서 Effect에 대해 *셋업* 대신 *셋업 → 클린업 → 셋업*을 실행하면 Strict Mode에서 누락된 클린업 로직이 더 눈에 띄게 됩니다.

Effect 클린업을 구현하는 방법에 대해 자세히 알아보세요.

개발 환경에서 ref 콜백을 다시 실행하여 발견된 버그 수정

Strict Mode는 callbacks refs의 버그를 찾는 데도 도움이 됩니다.

모든 콜백 `ref` 에는 몇 가지 셋업 코드가 있고 어쩌면 클린업 코드가 있을 수 있습니다. 일반적으로 React는 엘리먼트가 생성(DOM에 추가)될 때 셋업 코드를 실행하고, 엘리먼트가 제거(DOM에서 삭제)될 때 셋업 코드를 실행합니다.

Strict Mode가 켜져 있으면 React는 **모든 콜백 `ref` 에 대해 개발 환경에서 한 번 더 셋업+클린업 사이클을 실행합니다.** 이외로 느껴질 수도 있지만, 수동으로 파악하기 어려운 미묘한 버그를 드러내는 데 도움이 됩니다.

다음 예시를 살펴봅시다. 이 예시는 동물을 선택한 후 목록 중 하나로 스크롤 할 수 있게 해줍니다. “cats”에서 “dogs”로 전환할 때 콘솔 로그를 보면 목록에 있는 동물의 수가 계속 증가하고, “Scroll to” 버튼이 동작하지 않게 되는 점을 확인할 수 있습니다.

index.js App.js 새로고침 Clear 포크

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

export default function CatFriends() {
  const itemsRef = useRef([]);
  const [catList, setCatList] = useState(setupCatList);
  const [cat, setCat] = useState('neo');

  function scrollToCat(index) {
    const list = itemsRef.current;
    const {node} = list[index];
    node.scrollIntoView({
      behavior: "smooth",
```

자세히 보기

이것은 프로덕션 버그입니다! ref 콜백이 클린업 과정에서 동물 목록을 제거하지 않으므로 동물 목록이 계속 증가합니다. 이는 메모리 누수를 일으켜 실제 앱에서 성능 문제를 유발할 수 있으며, 앱의 동작을 망가뜨립니다.

문제는 ref 콜백이 스스로 클린업을 하지 않는 점입니다.

```
<li
  ref={node => {
    const list = itemsRef.current;
    const item = {animal, node};
    list.push(item);
    return () => {
      // 🚩 No cleanup, this is a bug!
    }
  }}
</li>
```

이제 원본 (버그가 있는) 코드를 `<StrictMode>` 로 감싸봅시다.

index.js App.js 새로고침 Clear 포크

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

export default function CatFriends() {
  const itemsRef = useRef([]);
  const [catList, setCatList] = useState(setupCatList);
  const [cat, setCat] = useState('neo');

  function scrollToCat(index) {
    const list = itemsRef.current;
    const {node} = list[index];
    node.scrollIntoView({
      behavior: "smooth",
```

자세히 보기

Strict Mode를 사용하면 문제를 즉시 찾을 수 있습니다. Strict Mode는 모든 콜백 ref에 대해 추가적인 셋업+클린업 사이클을 실행 실행합니다. 이 콜백 ref에는 클린업 로직이 없기 때문에 ref를 추가만 하고 제거하지 않습니다. 이는 클린업 함수가 누락되었다는 힌트입니다.

Strict Mode를 통해 콜백 ref에서 발생하는 실수를 조기에 발견할 수 있습니다. Strict Mode에서 클린업 함수를 추가해 콜백을 수정하면, 이전에 발생했던 “Scroll to” 버그와 같은 많은 잠재적인 프로덕션 버그도 함께 해결할 수 있습니다.

index.js App.js 새로고침 Clear 포크

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

export default function CatFriends() {
  const itemsRef = useRef([]);
  const [catList, setCatList] = useState(setupCatList);
  const [cat, setCat] = useState('neo');

  function scrollToCat(index) {
    const list = itemsRef.current;
    const {node} = list[index];
    node.scrollIntoView({
      behavior: "smooth",
```

자세히 보기

이제 StrictMode에서 초기 마운트 시, ref 콜백이 모두 셋업되고, 클린업 후, 다시 셋업 됩니다.

```
...
✅ 동물을 목록에 추가하는 중. 총 동물 수: 10
...
❌ 목록에서 동물을 제거합니다. 총 동물 수: 0
...
✅ 동물을 목록에 추가하는 중. 총 동물 수: 10
```

이것이 예상된 결과입니다. Strict Mode는 ref 콜백이 올바르게 클린업 되었는지 확인해 주기 때문에 크기가 예상된 양을 초과하지 않습니다. 수정 후에는 메모리 누수가 발생하지 않으며, 모든 기능이 예상대로 작동합니다.

Strict Mode 없이는 고장 난 기능을 알아차릴 때까지 여기저기 클릭해야 하므로 버그를 놓치기 쉽습니다. Strict Mode는 버그를 즉시 드러나도록 하여 프로덕션에 배포하기 전에 문제를 발견할 수 있습니다.

Fixing deprecation warnings enabled by Strict Mode

React는 `<StrictMode>` 트리 내부의 컴포넌트가 더 이상 사용되지 않는 다음 API 중 하나를 사용하는 경우 경고를 표시합니다.

- `UNSAFE_componentWillMount` 와 같은 `UNSAFE_` 클래스 생명주기 메서드. 대안 확인하기.

이러한 API는 주로 이전 클래스 컴포넌트에서 사용되므로 최신 앱에서는 거의 나타나지 않습니다.

이전
<Profiler>

다음
<Suspense>

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