memo

`memo`를 사용하면 컴포넌트의 Props가 변경되지 않은 경우 리렌더링을 건너뛸 수 있습니다.

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

중요합니다!

React 컴파일러는 모든 컴포넌트에 `memo`와 동일한 최적화를 자동으로 적용하므로 수동으로 메모이제이션을 할 필요가 줄어듭니다. 컴파일러를 사용해 컴포넌트 메모이제이션을 자동으로 처리할 수 있습니다.

레퍼런스

`memo(Component, arePropsEqual?)`

컴포넌트를 `memo` 로 감싸면 해당 컴포넌트의 메모된Memoized 버전을 얻을 수 있습니다. 메모된 버전의 컴포넌트는 일반적으로 부모 컴포넌트가 리렌더링 되어도 Props가 변경되지 않았다면 리렌더링되지 않습니다. 그러나 메모이제이션은 성능을 최적화하는 것이지, 보장하는 것은 아니기 때문에 React는 여전히 다시 렌더링될 수도 있습니다.

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

const SomeComponent = memo(function SomeComponent(props) {
  // ...
});
```

아래 예시를 참고하세요.

매개변수

- `Component` : 메모Memoize하려는 컴포넌트입니다. `memo` 는 이 컴포넌트를 수정하지 않고 대신 새로운 메모된 컴포넌트를 반환합니다. 함수와 `forwardRef` 컴포넌트를 포함한 모든 유효한 React 컴포넌트가 허용됩니다.
- **optional** `arePropsEqual` : 컴포넌트의 이전 Props와 새로운 Props의 두 가지 인수를 받는 함수입니다. 이전 Props와 새로운 Props가 동일한 경우, 컴포넌트가 이전 Props와 동일한 결과를 렌더링하고 새로운 Props에서도 이전 Props와 동일한 방식으로 동작하는 경우 `true` 를 반환해야 합니다. 그렇지 않으면 `false` 를 반환해야 합니다. 일반적으로 이 함수를 지정하지 않습니다. React는 기본적으로 `Object.is` 로 각 Props를 비교합니다.

반환값

`memo` 는 새로운 React 컴포넌트를 반환합니다. `memo` 에 제공한 컴포넌트와 동일하게 동작하지만, 부모가 리렌더링되더라도 Props가 변경되지 않는 한 React는 이를 리렌더링하지 않습니다.

사용법

Props가 변경되지 않았을 때 리렌더링 건너뛰기

React는 일반적으로 부모가 리렌더링될 때마다 컴포넌트를 리렌더링합니다. `memo` 를 사용하면, 새로운 Props가 이전 Props와 같으면 부모 컴포넌트가 다시 렌더링되더라도 React가 해당 컴포넌트를 다시 렌더링하지 않도록 만들 수 있습니다. 이러한 컴포넌트를 메모된Memoized 상태라고 합니다.

컴포넌트를 메모하려면 `memo` 로 감싸고 기존 컴포넌트 대신에 반환된 값을 사용하세요.

```
const Greeting = memo(function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
});

export default Greeting;
```

React 컴포넌트는 항상 순수한 렌더링 로직을 가져야 합니다. 이는 Props, State 그리고 Context가 변경되지 않으면 항상 동일한 결과를 반환해야 함을 의미합니다. `memo` 를 사용하면 컴포넌트가 이 요구 사항을 준수한다고 알리므로, Props가 변경되지 않는 한 React는 리렌더링 될 필요가 없습니다. `memo` 를 사용하더라도 컴포넌트의 State가 변경되거나 사용 중인 Context가 변경되면 리렌더링 됩니다.

아래 예시에서 `Greeting` 컴포넌트는 `name` 이 Props 중 하나이기 때문에 `name` 이 변경될 때마다 리렌더링 됩니다. 하지만 `address` 는 `Greeting` 의 Props가 아니기 때문에 `address` 가 변경될 때는 리렌더링되지 않습니다.

App.js 다운로드 새로고침 Clear 포크

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

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
        Name{': '}
        <input value={name} onChange={e => setName(e.target.value)} />
```

```
        </label>
```

자세히 보기

중요합니다!

`memo` 는 성능 최적화를 위해서 사용해야 합니다. `memo` 없이 코드가 작동하지 않는다면, 먼저 근본적인 문제를 찾아서 해결하세요. 이후에 `memo` 를 추가하여 성능을 개선할 수 있습니다.

자세히 살펴보기

모든 곳에 `memo` 를 추가해야할까요?

자세히 보기

State를 사용해 메모이제이션된 컴포넌트 업데이트하기

컴포넌트가 메모이제이션된 경우에도, 컴포넌트의 State가 변경되면 리렌더링됩니다. 메모이제이션은 부모에서 컴포넌트로 전달되는 Props에만 적용됩니다.

App.js 다운로드 새로고침 Clear 포크

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

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
        Name{': '}
        <input value={name} onChange={e => setName(e.target.value)} />
      </label>
      <label>
```

자세히 보기

State 변수를 현재 값으로 설정하면 React는 `memo` 없이도 컴포넌트 리렌더링을 건너뜁니다. 컴포넌트가 한 번 더 호출될 수 있지만, 결과는 무시됩니다.

Context를 사용하여 메모화된 컴포넌트 업데이트하기

컴포넌트가 메모되었더라도, 사용 중인 Context가 변경될 때 컴포넌트는 리렌더링됩니다. 메모는 부모로부터 전달되는 Props에만 적용됩니다.

App.js 다운로드 새로고침 Clear 포크

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

const ThemeContext = createContext(null);

export default function MyApp() {
  const [theme, setTheme] = useState('dark');

  function handleClick() {
    setTheme(theme === 'dark' ? 'light' : 'dark');
  }

  return (
```

자세히 보기

일부 Context의 *일정 부분*이 변경될 때만 컴포넌트가 리렌더링되도록 하려면 컴포넌트를 두 개로 나눠야 합니다. 외부 컴포넌트의 Context에서 필요한 내용을 읽고, 메모화된 자식에게 Prop으로 전달하세요.

Props 변경 최소화하기

`memo` 를 사용할 때 어떤 Prop든 이전의 Prop과 *얕은 비교 결과*가 같지 않을 때마다 컴포넌트가 리렌더링 됩니다. 즉 React는 `Object.is` 비교를 사용하여 컴포넌트의 모든 Prop을 이전 값과 비교합니다. `Object.is(3, 3)` 는 `true` 이지만 `Object.is({}, {})` 는 `false` 입니다.

`memo` 를 최대한 활용하려면, Props가 변경되는 횟수를 최소화해야 합니다. 예를 들어 Prop이 객체인 경우, `useMemo` 를 사용하여 부모 컴포넌트가 해당 객체를 매번 다시 만드는 것을 방지하세요.

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

  const person = useMemo(
    () => ({ name, age }),
    [name, age]
  );

  return <Profile person={person} />;
}

const Profile = memo(function Profile({ person }) {
  // ...
});
```

Props의 변경을 최소화하는 더 좋은 방법은 컴포넌트가 Props에 필요한 최소한의 정보만 받도록 하는 것입니다. 예를 들어, 전체 객체 대신 개별 값을 받을 수 있습니다.

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

```
  return <Profile name={name} age={age} />;
}

const Profile = memo(function Profile({ name, age }) {
  // ...
});
```

때로는 개별 값도 자주 변경되지 않는 값으로 사용할 수 있습니다. 예를 들어 다음 컴포넌트는 값 자체가 아니라 값의 존재를 나타내는 불리언 값을 받습니다.

```
function GroupsLanding({ person }) {
  const hasGroups = person.groups !== null;
  return <CallToAction hasGroups={hasGroups} />;
}

const CallToAction = memo(function CallToAction({ hasGroups }) {
  // ...
});
```

메모화된 컴포넌트에 함수를 전달해야 하는 경우, 컴포넌트 외부에 함수를 선언하여 변경되지 않도록 하거나, `useCallback` 을 사용하여 리렌더링 사이에 함수의 선언을 캐시합니다.

사용자 정의 비교 함수 지정하기

드물지만 메모화된 컴포넌트의 Props 변경을 최소화하는 것이 불가능할 수 있습니다. 이 경우 사용자 정의 비교 함수를 제공하여 React가 얕은 비교를 사용하는 대신에 이전 Props와 새로운 Props를 비교할 수 있습니다. 이 함수는 `memo` 의 두 번째 인수로 전달됩니다. 새로운 Props가 이전 Props와 동일한 결과를 생성하는 경우에만 `true` 를 반환해야 합니다. 그렇지 않으면 `false` 를 반환해야 합니다.

```
const Chart = memo(function Chart({ dataPoints }) {
  // ...
}, arePropsEqual);

function arePropsEqual(oldProps, newProps) {
  return (
```

```
    oldProps.dataPoints.length === newProps.dataPoints.length &&
    oldProps.dataPoints.every((oldPoint, index) => {
      const newPoint = newProps.dataPoints[index];
      return oldPoint.x === newPoint.x && oldPoint.y === newPoint.y;
    })
  );
}
```

이 경우 브라우저 개발자 도구의 성능 패널을 사용하여 비교 기능이 실제로 컴포넌트를 다시 렌더링하는 것보다 빠른지 확인하세요. 놀랄 수도 있습니다.

성능 측정을 할 때, React가 프로덕션 환경에서 실행되고 있는지 확인하세요.

주의하세요!

`arePropsEqual`를 구현하는 경우 **함수를 포함하여 모든 Prop를 비교해야 합니다.** 함수는 종종 부모 컴포넌트의 Props와 State를 클로저Closure로 다룹니다. `oldProps.onClick !== newProps.onClick`일 때 `true`를 반환하면 컴포넌트가 `onClick` 핸들러 내에서 이전 렌더링의 Props와 State를 계속 "인식"하여 매우 혼란스러운 버그가 발생할 수 있습니다.

작업 중인 데이터 구조가 알려진 제한된 깊이를 가지고 있다고 100% 확신하지 않는 한, `arePropsEqual` 내에서 깊은 비교를 수행하지 마세요. **깊은 비교는 매우 느려질 수 있으며** 나중에 누군가 데이터 구조를 변경하면 앱이 잠깐 정지될 수 있습니다.

Do I still need React.memo if I use React Compiler?

When you enable React Compiler, you typically don't need `React.memo` anymore. The compiler automatically optimizes component re-rendering for you.

Here's how it works:

Without React Compiler, you need `React.memo` to prevent unnecessary re-renders:

```
// Parent re-renders every second
function Parent() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(s => s + 1);
    }, 1000);
    return () => clearInterval(interval);
  }, []);

  return (
    <>
      <h1>Seconds: {seconds}</h1>
      <ExpensiveChild name="John" />
    </>
  );
}

// Without memo, this re-renders every second even though props don't change
const ExpensiveChild = memo(function ExpensiveChild({ name }) {
  console.log('ExpensiveChild rendered');
  return <div>Hello, {name}!</div>;
});
```

With React Compiler enabled, the same optimization happens automatically:

```
// No memo needed - compiler prevents re-renders automatically
function ExpensiveChild({ name }) {
  console.log('ExpensiveChild rendered');
  return <div>Hello, {name}!</div>;
}
```

Here's the key part of what the React Compiler generates:

```
function Parent() {
  const $ = _c(7);
  const [seconds, setSeconds] = useState(0);
```

```
  // ... other code ...

  let t3;
  if ($[4] === Symbol.for("react.memo_cache_sentinel")) {
    t3 = <ExpensiveChild name="John" />;
    $[4] = t3;
  } else {
    t3 = $[4];
  }
  // ... return statement ...
}
```

Notice the highlighted lines: The compiler wraps `<ExpensiveChild name="John" />` in a cache check. Since the `name` prop is always `"John"`, this JSX is created once and reused on every parent re-render. This is exactly what `React.memo` does - it prevents the child from re-rendering when its props haven't changed.

The React Compiler automatically:

1. Tracks that the `name` prop passed to `ExpensiveChild` hasn't changed
2. Reuses the previously created JSX for `<ExpensiveChild name="John" />`
3. Skips re-rendering `ExpensiveChild` entirely

This means **you can safely remove `React.memo` from your components when using React Compiler**. The compiler provides the same optimization automatically, making your code cleaner and easier to maintain.

중요합니다!

The compiler's optimization is actually more comprehensive than `React.memo`. It also memoizes intermediate values and expensive computations within your components, similar to combining `React.memo` with `useMemo` throughout your component tree.

Troubleshooting

My component re-renders when a prop is an object, array, or function

React는 얕은 비교를 기준으로 이전 Props와 새로운 Props를 비교합니다. 즉, 각각의 새로운 Prop가 이전 Prop와 참조가 동일한지 여부를 고려합니다. 부모가 리렌더링 될 때마다 새로운 객체나 배열을 생성하면, 개별 요소들이 모두 동일하더라도 React는 여전히 변경된 것으로 간주합니다. 마찬가지로 부모 컴포넌트를 렌더링할 때 새로운 함수를 만들면 React는 함수의 정의가 동일하더라도 변경된 것으로 간주합니다. 이를 방지하려면 부모 컴포넌트에서 Props를 단순화하거나 메모화 하세요.

이전
lazy

다음
startTransition