useMemo

`useMemo` 는 재렌더링 사이에 계산 결과를 캐싱할 수 있게 해주는 React Hook 입니다.

```
const cachedValue = useMemo(calculateValue, dependencies)
```

중요합니다!

React 컴파일러는 값과 함수를 자동으로 메모이제이션하므로 `useMemo` 를 수동으로 사용할 일이 줄어듭니다. 컴파일러를 사용해 메모이제이션을 자동으로 처리할 수 있습니다.

레퍼런스

useMemo(calculateValue, dependencies)

컴포넌트의 최상위 레벨에 있는 'useMemo'를 호출하여 재렌더링 사이의 계산을 캐싱합니다.

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

function TodoList({ todos, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  // ...
}
```

아래로 이동하여 더 많은 예시를 확인하세요.

매개변수

- `calculateValue` : 캐싱하려는 값을 계산하는 함수입니다. 순수해야 하며 인자를 받지 않고, 모든 타입의 값을 반환할 수 있어야 합니다. React는 초기 렌더링 중에 함수를 호출합니다. 다음 렌더링에서, React는 마지막 렌더링 이후 `dependencies` 가 변경되지 않았을 때 동일한 값을 다시 반환합니다. 그렇지 않다면 `calculateValue` 를 호출하고 결과를 반환하며, 나중에 재사용할 수 있도록 저장합니다.

- `dependencies` : `calculateValue` 코드 내에서 참조된 모든 반응형 값들의 목록입니다. 반응형 값에는 props, state와 컴포넌트 바디에 직접 선언된 모든 변수와 함수가 포함됩니다. 만약 linter가 React용으로 설정된 경우 모든 반응형 값이 의존성으로 올바르게 설정되었는지 확인할 수 있습니다. 의존성 목록은 일정한 수의 항목을 가져야 하며, `[dep1, dep2, dep3]` 와 같이 인라인 형태로 작성돼야 합니다. React는 `Object.is` 비교를 통해 각 의존성 들을 이전 값과 비교합니다.

반환값

초기 렌더링에서 `useMemo` 는 인자 없이 `calculateValue` 를 호출한 결과를 반환합니다.

다음 렌더링에서, 마지막 렌더링에서 저장된 값을 반환하거나(종속성이 변경되지 않은 경우), `calculateValue`를 다시 호출하고 반환된 값을 저장합니다.

주의 사항

- `useMemo`는 Hook이므로 **컴포넌트의 최상위 레벨** 또는 자체 Hook에서만 호출할 수 있습니다. 반복문이나 조건문 내부에서는 호출할 수 없습니다. 만일 호출이 필요하다면 새 컴포넌트를 추출하고 상태를 그 안으로 옮겨야 합니다.
- Strict Mode에서는 , React는 실수로 발생한 오류를 찾기 위해 **계산 함수를 두 번 호출합니다.** 이는 개발 환경에서만 동작하는 방식이며, 실제 프로덕션 환경에는 영향을 미치지 않습니다. (원래 그래야 하는 것처럼) 연산 함수가 순수하다면, 로직에는 영향을 미치지 않습니다. 호출 결과 중 하나는 무시됩니다.
- React는 **캐싱 된 값을 버려야 할 특별한 이유가 없는 한 버리지 않습니다.** 예를 들어, 개발 단계에서 컴포넌트 파일을 편집할 때 React는 캐시를 버립니다. 개발과 프로덕션 환경 모두에서는 컴포넌트가 초기 마운트 중에 일시 중단되면 React는 캐시를 버립니다. 앞으로 React는 캐시를 버리는 것을 활용하는 더 많은 기능을 추가할 수 있습니다. 예를 들어, 앞으로 React에 가상화된 목록에 대한 기본적인 지원이 추가된다면 가상화된 테이블 뷰포트에서 스크롤 되는 항목에 대한 캐시를 버리는 것이 합리적일 것입니다. 이는 성능 최적화를 위해 `useMemo`에만 의존한다면 괜찮을 것입니다. 그러나 이는 상태 변수나 ref를 사용하는 것이 더 적합할 수 있습니다.

중요합니다!

이와 같이 반환값을 캐싱하는 것을 *memoization*라고 하며, 이 훅을 `useMemo`라고 부르는 이유입니다.

사용법

비용이 높은 로직의 재계산 생략하기

재렌더링 사이에 계산을 캐싱하려면 컴포넌트의 최상위 레벨에서 `useMemo`를 호출하여 계산을 감싸면 됩니다.

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

function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

useMemo 에 두 가지를 전달해야 합니다.

1. () => 와 같이 인수를 받지 않고 계산하려는 값을 반환하는 계산 함수 입니다.
2. 계산 내부에서 사용되는 컴포넌트 내의 모든 값을 포함하는 종속성 목록 입니다.

초기 렌더링에서 useMemo 에서 얻을 수 있는 값 은 계산 함수 를 호출한 결과값 입니다.

이후 모든 렌더링에서 React는 종속성 목록을 마지막 렌더링 중에 전달한 종속성 목록과 비교합니다. 만일 (Object.is 와 비교했을 때) 종속성이 변경되지 않았다면, useMemo 는 이전에 이미 계산해둔 값을 반환합니다. 그렇지 않다면 React는 계산을 다시 실행하고 새로운 값을 반환합니다.

즉, useMemo 는 종속성이 변경되기 전까지 재렌더링 사이의 계산 결과를 캐싱합니다.

이 기능이 언제 유용한지 예시를 통해 살펴보겠습니다.

기본적으로 React는 컴포넌트를 다시 렌더링할 때마다 컴포넌트의 전체 본문을 다시 실행합니다. 예를 들어, TodoList 가 상태를 업데이트하거나 부모로부터 새로운 props를 받으면 filterTodos 함수가 다시 실행됩니다.

```
function TodoList({ todos, tab, theme }) {
  const visibleTodos = filterTodos(todos, tab);
  // ...
}
```

일반적으로 대부분의 계산을 매우 빠르기 때문에 문제가 되지 않습니다. 그러나 큰 배열을 필터링 혹은 변환하거나 비용이 많이 드는 계산을 수행하는 경우, 데이터가 변경되지 않았다면 계산을 생략하는 것이 좋습니다. 만약 todos 과 tab 이 마지막 렌더링 때와 동일한 경우, 앞서 언급한 것처럼 useMemo 로 계산을 감싸면 이전에 계산된 visibleTodos 를 재사용할 수 있습니다.

이러한 유형의 캐싱을 *메모이제이션* 라고 합니다.

중요합니다!

성능 최적화를 위해서만 `useMemo` 를 사용해야 합니다. 이 기능이 없어서 코드가 작동하지 않는다면 근본적인 문제를 먼저 찾아서 수정하세요. 그 후 `useMemo` 를 사용하여 성능을 개선해야 합니다.

자세히 살펴보기

비싼 연산인지 어떻게 알 수 있나요?

자세히 보기

자세히 살펴보기

모든 곳에 useMemo를 추가해야 하나요?

자세히 보기

useMemo와 값을 직접 계산하는 것의 차이점

1. `useMemo` 로 재계산 건너뛰기
2. 항상 값을 재계산하기

예시 1 of 2: `useMemo` 로 재계산 건너뛰기

이 예시에서는 렌더링 중에 호출하는 자바스크립트 함수가 실제로 느릴 때 어떤 일이 발생하는지 확인할 수 있도록 `filterTodos` 을 **인위적으로 느리게** 만들었습니다. 탭을 전환하고 테마를 토글해 보세요.

탭을 전환하면 느려진 `filterTodos` 가 다시 실행되므로 느리게 느껴집니다. 이는 `tab` 이 변경되었으므로 전체 계산이 *필수적으로* 다시 실행되기 때문에 나타나는 현상입니다. (왜 두 번 실행되는지 궁금하다면 여기를 클릭해서 설명을 확인하세요.)

테마를 전환합니다. **인위적인 속도 저하에도 불구하고 빠른 이유는 `useMemo` 덕분입니다!** 느린 속도의 `filterTodos` 는 마지막 렌더링 이후 (`useMemo` 에 종속성으로 전달한) `todos` 와 `tab` 이 모두 변경되지 않았기 때문에 호출을 건너뛰었습니다.

App.js TodoList.js utils.js 새로고침 Clear 포크

```
import { useMemo } from 'react';
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  return (
    <div className={theme}>
      <p><b>Note: <code>filterTodos</code> is artificially slowed down!</b>
      <ul>
```

자세히 보기

다음 예시

컴포넌트 재렌더링 건너뛰기

경우에 따라 `useMemo` 는 하위 컴포넌트 재렌더링 성능을 최적화하는데 도움이 될 수도 있습니다. 이를 설명하기 위해 `TodoList` 컴포넌트가 자식 컴포넌트인 `List` 에 `visibleTodos` 를 prop로 전달한다고 가정하겠습니다.

```
export default function TodoList({ todos, tab, theme }) {
  // ...
  return (
    <div className={theme}>
      <List items={visibleTodos} />
    </div>
  );
}
```

`theme` prop를 토글하면 앱이 잠시 멈추는 것을 확인할 수 있습니다. 그러나 JSX에서 `<List />` 를 제거하면 빠르게 느껴집니다. 이는 `List` 컴포넌트를 최적화할 가치가 있다는 것을 알려줍니다.

기본적으로 React는 컴포넌트가 다시 렌더링 될 때, 모든 자식 컴포넌트를 재귀적으로 다시 렌더링합니다. 그러므로 `TodoList` 가 다른 `theme` 로 다시 렌더링 되면 `List` 컴포넌트 *또한* 다시 렌더링 됩니다. 다시 렌더링하는 데 많은 계산이 필요하지 않는 컴포넌트는 괜찮지만, 다시 렌더링하는 것이 느리다는 것을 확인했다면 `List` 를 `memo` 를 통해 감싸서 props가 마지막 렌더링 시점과 동일 할 때 다시 렌더링하는 것을 생략할 수 있습니다.

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

const List = memo(function List({ items }) {
  // ...
});
```

이 변경으로 `List` 는 모든 props가 마지막 렌더링 때와 *동일*한 경우 다시 렌더링하지 않습니다. 여기서 계산을 캐싱하는 것이 중요합니다! `useMemo` 없이 `visibleTodos` 를 계산한다고 가정해 봅시다.

```
export default function TodoList({ todos, tab, theme }) {
  // 테마가 변경될 때 마다 다른 배열이 표시됩니다.
  const visibleTodos = filterTodos(todos, tab);
  return (
    <div className={theme}>
      {/* ... List의 props는 동일하지 않으며 매번 다시 렌더링 됩니다. */}
      <List items={visibleTodos} />
    </div>
  );
}
```

위의 예시에서 `filterTodos` 함수는 항상 *다른* 배열을 생성합니다. 이는 `{}` 객체 리터럴이 항상 새 객체를 생성하는 것과 유사합니다. 일반적으로 이는 문제가 되지 않지만 `List` 의 props는 동일하지 않으며 `memo` 를 사용한 최적화가 작동하지 않는다는 것을 의미합니다. 이러한 경우 `useMemo` 가 유용합니다.

```
export default function TodoList({ todos, tab, theme }) {
  // 재렌더링 사이에 계산을 캐싱하도록 React에 지시합니다...
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab] // ...따라서 해당 종속성이 변경되지 않는 한...
  );
  return (
    <div className={theme}>
      {/* ...List에 동일한 props가 전달되어 재렌더링을 생략할 수 있습니다. */}
      <List items={visibleTodos} />
    </div>
  );
}
```

`visibleTodos` 연산을 `useMemo` 로 감싸면 다시 렌더링 될 때마다 *같은* 값을 갖게 할 수 있습니다 (종속성이 변경되기 전까지). *특별한 이유가 없는 한* 연산을 `useMemo` 로 감싸지 않아도 됩니다. 이

예시에서는 `memo` 로 감싸진 컴포넌트에 전달하면 재렌더링을 건너뛸 수 있기 때문입니다. 이 페이지에서 자세히 설명하는 `useMemo` 를 추가해야 하는 몇 가지 다른 이유가 있습니다.

자세히 살펴보기

개별 JSX 노드 메모화

자세히 보기

재렌더링을 건너뛰는 것과 항상 재렌더링을 하는 것의 차이점

1. `useMemo` 및 `memo` 로 재렌더링 건너뛰기
2. 항상 컴포넌트 재렌더링 하기

예시 1 of 2: `useMemo` 및 `memo` 로 재렌더링 건너뛰기

이 예시에서는 `List` 컴포넌트를 **인위적으로 느리게 만들어** 렌더링 중인 React 컴포넌트가 실제로 느려질 때 어떤 일이 발생하는 지를 확인할 수 있습니다. 탭을 전환하고 테마를 토글해 보세요.

탭을 전환하면 느려진 `List` 가 다시 렌더링 되기 때문에 느리게 느껴집니다. 이는 `tab` 이 변경되었으므로 사용자의 새로운 선택 사항을 화면에 반영해야 하기 때문에 예상되는 현상입니다.

다음으로 테마를 토글해 보겠습니다. **인위적인 속도 저하에도 불구하고 `memo` 와 함께 사용된 `useMemo` 덕분에 빠릅니다!** `List` 는 마지막 렌더링 이후 `visibleItems` 배열이 변경되지 않았기 때문에 재렌더링을 생략했습니다. (`useMemo` 에 종속성으로 전달된) `todos` 와 `tab` 이 모두 마지막 렌더링 이후 변경되지 않았으므로 `visibleItems` 배열이 변경되지 않았습니다.

App.js TodoList.js List.js utils.js 새로고침 Clear 포크

```
import { useMemo } from 'react';
import List from './List.js';
```

```
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  return (
    <div className={theme}>
```

자세히 보기

다음 예시

Effect가 자주 실행되지 않도록 하기

때때로, Effect 안에 값을 사용하고 싶을 것입니다.

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');
```

```
  const options = {
    serverUrl: 'https://localhost:1234',
    roomId: roomId
  }

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    // ...
```

이것은 문제를 일으킵니다. 모든 반응형 값은 Effect의 종속성으로 선언되어야 합니다. 그러나 만약 `options` 을 종속성으로 선언한다면, 이것은 Effect가 chat room과 계속해서 재연결되도록 할 것입니다.

```
  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // 🔴 문제: 이 종속성은 렌더링마다 변경됩니다.
  // ...
```

이 문제를 해결하기 위해서는, Effect 안에서 사용되는 객체를 `useMemo` 로 감싸면 됩니다.

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = useMemo(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ roomId가 변경될때만 실행됩니다.

  useEffect(() => {
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // ✅ options가 변경될때만 실행됩니다.
```

```
  // ...
```

이것은 만약 `useMemo` 가 캐시된 객체를 반환할 경우, 재렌더링시 `options` 객체가 동일하다는 것을 보장합니다.

그러나 `useMemo` 는 성능 최적화를 위한 것이지 의미론적 보장은 아니기 때문에 React는 특별한 이유가 있는 경우 캐시된 값을 버릴 수 있습니다. 이로 인해 Effect가 다시 실행될 수 있습니다. 따라서 객체를 Effect *안으로* 이동시켜 **함수 종속성의 필요성을 제거하는 것이 더 좋습니다.**

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = { // ✅ 더이상 useMemo나 object dependencies가 필요없습니다!
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    }

    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ roomId가 변경될때에만 실행됩니다.
  // ...
```

이제 코드는 더 간단해지고 `useMemo` 가 필요하지 않습니다. Effect 종속성 제거에 대해 더 알아보세요.

다른 Hook의 종속성 메모화

컴포넌트 본문에서 직접 생성된 객체에 의존하는 연산이 있다고 가정하겠습니다.

```
function Dropdown({ allItems, text }) {
  const searchOptions = { matchMode: 'whole-word', text };

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // 🚩 주의: 컴포넌트 본문에서 생성된 객체에 대한 종속성
```

```
  // ...
```

이렇게 객체에 의존하는 것은 메모이제이션의 목적을 무색하게 합니다. 컴포넌트가 다시 렌더링 되면 컴포넌트 본문 내부의 모든 코드가 다시 실행되기 때문입니다. **`searchOptions` 객체를 생성하는 코드도 다시 렌더링 될 때마다 실행됩니다.** `searchOptions` 은 `useMemo` 호출의 종속성이고 매번 다르기 때문에, React는 종속성이 다른 것을 알고 `searchItems` 을 매번 다시 계산합니다.

이 문제를 해결하기 위해 `searchOptions` 객체 *자체를* 종속성으로 전달하기 전에 메모해두면 됩니다.

```
function Dropdown({ allItems, text }) {
  const searchOptions = useMemo(() => {
    return { matchMode: 'whole-word', text };
  }, [text]); // ✅ text가 변경될 때만 변경

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // ✅ allItems이나 searchOptions이 변경될 때만 변경
  // ...
```

위의 예시에서 `text` 가 변경되지 않았다면 `searchOptions` 객체도 변경되지 않습니다. 그러나 이보다 더 나은 방법은 `searchOptions` 를 `useMemo` 계산 함수의 *내부에* 선언하는 것입니다.

```
function Dropdown({ allItems, text }) {
  const visibleItems = useMemo(() => {
    const searchOptions = { matchMode: 'whole-word', text };
    return searchItems(allItems, searchOptions);
  }, [allItems, text]); // ✅ allItems이나 text가 변경될 때만 변경
  // ...
```

이제 연산은 `text` 에 직접적으로 의존합니다 (문자열이므로 “실수로” 달라질 수 없음).

함수 메모화

`Form` 컴포넌트가 `memo` 로 감싸져 있고 여기에 prop로 함수를 전달하고 싶다고 가정해봅시다.

```
export default function ProductPage({ productId, referrer }) {
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }

  return <Form onSubmit={handleSubmit} />;
}
```

`{}` 가 다른 객체를 생성하는 것처럼 `function() {}` 와 같은 함수 선언과 `() => {}` 같은 표현식은 다시 렌더링 될 때마다 *다른* 함수를 생성합니다. 새로운 함수를 만드는 것 자체는 문제가 되지 않으며 피해야 할 일이 아닙니다! 그러나 `Form` 컴포넌트가 메모화되어 있다면 props가 변경되지 않았을 때 다시 렌더링하는 것을 생략하고 싶을 것입니다. *항상* 다른 prop은 메모이제이션의 목적을 무색하게 만들 수 있습니다.

`useMemo` 로 함수를 메모하려면 계산 함수에서 다른 함수를 반환해야 합니다.

```
export default function Page({ productId, referrer }) {
  const handleSubmit = useMemo(() => {
    return (orderDetails) => {
      post('/product/' + productId + '/buy', {
        referrer,
        orderDetails
      });
    };
  }, [productId, referrer]);

  return <Form onSubmit={handleSubmit} />;
}
```

위 예시는 투박해 보입니다! **함수를 메모하는 것은 충분히 일반적이기 때문에 React에는 이를 위한 Hook이 내장되어 있습니다.** **`useMemo` 대신 `useCallback` 으로 함수를 감싸서** 중첩된 함수를 추가로 작성하지 않도록 하세요.

```
export default function Page({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }, [productId, referrer]);

  return <Form onSubmit={handleSubmit} />;
}
```

위 두 예시는 완전히 동일하게 동작합니다. `useCallback` 의 유일한 장점은 내부에 중첩된 함수를 추가로 작성하지 않아도 된다는 것입니다. 그 외에는 아무것도 하지 않습니다. `useCallback` 에 대해 더 읽어보세요.

문제 해결하기

렌더링할 때마다 계산이 두 번 실행됩니다

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

```
function TodoList({ todos, tab }) {
  // 이 컴포넌트 함수는 렌더링할 때마다 두 번 실행됩니다.

  const visibleTodos = useMemo(() => {
    // 종속성 중 하나라도 변경되면 이 계산은 두 번 실행됩니다.
    return filterTodos(todos, tab);
  }, [todos, tab]);

  // ...
```

이는 예상되는 현상이며 코드를 손상시키지 않습니다.

이 **개발 전용** 동작은 컴포넌트가 순수하게 유지될 수 있도록 도와줍니다. React는 호출 결과 중 하나를 사용하고 다른 호출 결과는 무시합니다. 컴포넌트와 계산 함수가 순수하다면 로직에 영향을

미치지 않을 것입니다. 그러나 실수로 발생하는 불순한 경우에 발생하는 실수를 발견하고 수정하는 데 도움을 줍니다.

예를 들어 아래의 불순한 계산 함수는 prop으로 받은 배열을 변경합니다.

```
const visibleTodos = useMemo(() => {
  // 🚩 Mistake: mutating a prop
  todos.push({ id: 'last', text: 'Go for a walk!' });
  const filtered = filterTodos(todos, tab);
  return filtered;
}, [todos, tab]);
```

React가 함수를 두 번 호출하므로 todo가 두 번 추가됩니다. 계산이 기존의 객체를 변경해서는 안 되지만 계산 중에 생성된 *새로운* 객체를 변경하는 것은 괜찮습니다. 예를 들어 `filterTodos` 함수가 항상 *다른* 배열을 반환하는 경우 대신 *해당 배열*을 변경할 수 있습니다.

```
const visibleTodos = useMemo(() => {
  const filtered = filterTodos(todos, tab);
  // ✅ 정답: 계산 중에 생성한 객체를 변경합니다.
  filtered.push({ id: 'last', text: 'Go for a walk!' });
  return filtered;
}, [todos, tab]);
```

순수성에 대해 자세히 알아보려면 컴포넌트 순수하게 유지하기를 읽어보세요.

또한 변경사항이 없는 객체 업데이트 및 배열 업데이트에 대한 가이드도 확인해보세요.

`useMemo`가 객체를 반환해야 하는데 undefined를 반환합니다.

이 코드는 작동하지 않습니다.

```
// 🔴 () => { 와 같은 화살표 함수는 객체를 반환하지 않습니다.
const searchOptions = useMemo(() => {
  matchMode: 'whole-word',
  text: text
```

```
}, [text]);
```

자바스크립트의 `() => {` 는 화살표 함수의 본문의 시작이므로 `{` 중괄호는 객체의 일부가 아닙니다. 이것이 객체를 반환하지 않고 실수하는 지점입니다. `({` 과 `})` 같은 괄호를 추가하여 이 문제를 해결할 수 있습니다.

```
// T이것은 작동하지만 누군가가 다시 위반하기 쉽습니다.
const searchOptions = useMemo(() => ({
  matchMode: 'whole-word',
  text: text
}), [text]);
```

하지만 해당 방식은 여전히 혼란을 주고, 괄호를 제거하면서 누군가 쉽게 위반할 수 있습니다.

이 실수를 방지하기 위해 `return` 문을 명시적으로 작성하세요.

```
// ✅ 이것은 작동하며 명확합니다.
const searchOptions = useMemo(() => {
  return {
    matchMode: 'whole-word',
    text: text
  };
}, [text]);
```

컴포넌트가 렌더링 될 때마다 `useMemo` 의 계산이 다시 실행됩니다.

두 번째 인수로 종속성 배열을 지정했는지 확인하세요!

종속성 배열을 지정하지 않았을 경우 `useMemo` 는 매번 다시 계산을 실행합니다.

```
function TodoList({ todos, tab }) {
  // 🔴 종속성 배열이 없어 매번 재계산 됨.
  const visibleTodos = useMemo(() => filterTodos(todos, tab));
```

```
  // ...
```

이것은 두 번째 인수로 종속성 배열을 전달하는 수정된 예시입니다.

```
function TodoList({ todos, tab }) {
  // ✅ 불필요한 재계산을 하지 않음.
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
```

만일 위의 예시가 도움이 되지 않았다면, 종속성 중 하나 이상이 이전 렌더링과 달라졌다는 문제일 수 있습니다. 종속성 들을 콘솔에 수동으로 로깅하여 이 문제를 디버그할 수 있습니다.

```
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  console.log([todos, tab]);
```

그런 다음 콘솔에서 서로 다른 리렌더의 배열을 마우스 오른쪽 버튼으로 클릭하고 두 배열 모두에 대해 "전역 변수로 저장"을 선택합니다. 첫 번째 배열은 `temp1`, 두 번째 배열이 `temp2` 로 저장되었다고 가정하면 브라우저 콘솔에서 두 배열의 각 종속성이 동일한지에 대해 확인할 수 있습니다.

```
Object.is(temp1[0], temp2[0]); // 배열 간의 첫 번째 종속성이 동일합니까?
Object.is(temp1[1], temp2[1]); // 배열 간의 두 번째 종속성이 동일합니까?
Object.is(temp1[2], temp2[2]); // ... 그리고 기타 모든 종속성들이 동일합니까? ...
```

메모를 방해하는 종속성을 발견하면 제거할 방법을 찾거나 메모할 방법을 찾으세요.

반복문에서 각 목록 항목에 대해 `useMemo` 를 호출해야 하는데 허용되지 않습니다.

`Chart` 컴포넌트가 `memo` 로 감싸져 있다고 가정해보겠습니다. `ReportList` 컴포넌트가 다시 렌더링 될 때 목록의 모든 `Chart` 를 다시 렌더링하는 것을 생략하고 싶을 것입니다. 그러나 반복문에서 `useMemo` 를 호출할 수 없습니다.

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 반복문에서는 useMemo를 호출할 수 없습니다.
        const data = useMemo(() => calculateReport(item), [item]);
        return (
          <figure key={item.id}>
            <Chart data={data} />
          </figure>
        );
      })}
    </article>
  );
}
```

대신 각 항목에 대한 컴포넌트를 추출하고 개별 항목에 대한 데이터를 메모하세요.

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ 최상위 수준에서 useMemo를 호출합니다.
  const data = useMemo(() => calculateReport(item), [item]);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
}
```

또는 `useMemo`를 제거하고 `Report` 자체를 `memo`로 감싸는 방법도 있습니다. `item` prop가 변경되지 않으면 `Report`는 재렌더링을 건너뛰므로 `Chart` 역시 재렌더링을 건너뛰게 됩니다.

```
function ReportList({ items }) {
  // ...
}

const Report = memo(function Report({ item }) {
  const data = calculateReport(item);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
});
```


이전
useLayoutEffect

다음
useOptimistic

커뮤니티

행동 강령

팀 소개

문서 기여자

감사의 말

더 보기

블로그

React Native

개인 정보 보호

약관