useMemo

`useMemo` is a React Hook that lets you cache the result of a calculation between re-renders.

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


- Reference
  - `useMemo(calculateValue, dependencies)`
- Usage
  - Skipping expensive recalculations
  - Skipping re-rendering of components
  - Memoizing a dependency of another Hook
  - Memoizing a function
- Troubleshooting
  - My calculation runs twice on every re-render
  - My `useMemo` call is supposed to return an object, but returns undefined
  - Every time my component renders, the calculation in `useMemo` re-runs
  - I need to call `useMemo` for each list item in a loop, but it's not allowed

---

Reference

`useMemo(calculateValue, dependencies)`

Call `useMemo` at the top level of your component to cache a calculation between re-renders:

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

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

See more examples below.

Parameters

- `calculateValue` : The function calculating the value that you want to cache. It should be pure, should take no arguments, and should return a value of any type. React will call your function during the initial render. On next renders, React will return the same value again if the `dependencies` have not changed since the last render. Otherwise, it will call `calculateValue` , return its result, and store it so it can be reused later.

- `dependencies` : The list of all reactive values referenced inside of the `calculateValue` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]` . React will compare each dependency with its previous value using the `Object.is` comparison.

Returns

On the initial render, `useMemo` returns the result of calling `calculateValue` with no arguments.

During next renders, it will either return an already stored value from the last render (if the dependencies haven't changed), or call `calculateValue` again, and return the result that `calculateValue` has returned.

Caveats

- `useMemo` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- In Strict Mode, React will **call your calculation function twice** in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your calculation function is pure (as it should be), this should not affect your logic. The result from one of the calls will be ignored.
- React **will not throw away the cached value unless there is a specific reason to do that.** For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should be fine if you rely on `useMemo` solely as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Note

Caching return values like this is also known as *memoization*, which is why this Hook is called `useMemo`.

Usage

Skipping expensive recalculations

To cache a calculation between re-renders, wrap it in a `useMemo` call at the top level of your component:

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

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

You need to pass two things to `useMemo`:

1. A calculation function that takes no arguments, like `() =>`, and returns what you wanted to calculate.
2. A list of dependencies including every value within your component that's used inside your calculation.

On the initial render, the value you'll get from `useMemo` will be the result of calling your calculation.

On every subsequent render, React will compare the dependencies with the dependencies you passed during the last render. If none of the dependencies have changed (compared with `Object.is`), `useMemo` will return the value you already calculated before. Otherwise, React will re-run your calculation and return the new value.

In other words, `useMemo` caches a calculation result between re-renders until its dependencies change.

Let's walk through an example to see when this is useful.

By default, React will re-run the entire body of your component every time that it re-renders. For example, if this `TodoList` updates its state or receives new props from its parent, the `filterTodos` function will re-run:

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

Usually, this isn't a problem because most calculations are very fast. However, if you're filtering or transforming a large array, or doing some expensive computation, you might want to skip doing it again if data hasn't changed. If both `todos` and `tab` are the same as they were during the last render, wrapping the calculation in `useMemo` like earlier lets you reuse `visibleTodos` you've already calculated before.

This type of caching is called *memoization.*

Note

You should only rely on `useMemo` as a performance optimization. If your code doesn't work without it, find the underlying problem and fix it first. Then you may add `useMemo` to improve performance.

DEEP DIVE

How to tell if a calculation is expensive?

Show Details

DEEP DIVE

Should you add useMemo everywhere?

Show Details

The difference between useMemo and calculating a value directly

1. Skipping recalculation with `useMemo` 2. Always recalculating a valu

Example 1 of 2:

Skipping recalculation with `useMemo`

In this example, the `filterTodos` implementation is **artificially slowed down** so that you can see what happens when some JavaScript function you're calling during rendering is genuinely slow. Try switching the tabs and toggling the theme.

Switching the tabs feels slow because it forces the slowed down `filterTodos` to re-execute. That's expected because the `tab` has changed, and so the entire calculation *needs* to re-run. (If you're curious why it runs twice, it's explained here.)

Toggle the theme. **Thanks to `useMemo`, it's fast despite the artificial slowdown!** The slow `filterTodos` call was skipped because both `todos` and `tab` (which you pass as dependencies to `useMemo`) haven't changed since the last render.

App.js TodoList.js utils.js Reset

```
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 slowe
```

Show more

Next Example

Skipping re-rendering of components

In some cases, `useMemo` can also help you optimize performance of re-rendering child components. To illustrate this, let's say this `TodoList` component passes the `visibleTodos` as a prop to the child `List` component:

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

You've noticed that toggling the `theme` prop freezes the app for a moment, but if you remove `<List />` from your JSX, it feels fast. This tells you that it's worth trying to optimize the `List` component.

By default, when a component re-renders, React re-renders all of its children recursively. This is why, when `TodoList` re-renders with a different `theme`, the `List` component *also* re-renders. This is fine for components that don't require much calculation to re-render. But if you've verified that a re-render is slow, you can tell `List` to skip re-rendering when its props are the same as on last render by wrapping it in `memo`:

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

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

With this change, `List` will skip re-rendering if all of its props are the *same* as on the last render. This is where caching the calculation becomes important! Imagine that you calculated `visibleTodos` without `useMemo`:

```
export default function TodoList({ todos, tab, theme }) {
  // Every time the theme changes, this will be a different array...
  const visibleTodos = filterTodos(todos, tab);
  return (
    <div className={theme}>
      {/* ... so List's props will never be the same, and it will re-render e
      <List items={visibleTodos} />
    </div>
  );
}
```

In the above example, the `filterTodos` function always creates a *different* array, similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `List` props will never be the same, and your `memo` optimization won't work. This is where `useMemo` comes in handy:

```
export default function TodoList({ todos, tab, theme }) {
  // Tell React to cache your calculation between re-renders...
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab] // ...so as long as these dependencies don't change...
  );
  return (
    <div className={theme}>
      {/* ...List will receive the same props and can skip re-rendering */}
```

```
      <List items={visibleTodos} />
    </div>
  );
}
```

By wrapping the `visibleTodos` calculation in `useMemo`, you ensure that it has the *same* value between the re-renders (until dependencies change). You don't *have to* wrap a calculation in `useMemo` unless you do it for some specific reason. In this example, the reason is that you pass it to a component wrapped in `memo`, and this lets it skip re-rendering. There are a few other reasons to add `useMemo` which are described further on this page.

DEEP DIVE

Memoizing individual JSX nodes

Show Details

The difference between skipping re-renders and always re-rendering

1. Skipping re-rendering with `useMemo` and `memo` 2. Always re-render

Example 1 of 2:

Skipping re-rendering with `useMemo` and `memo`

In this example, the `List` component is **artificially slowed down** so that you can see what happens when a React component you're rendering is genuinely slow. Try switching the tabs and toggling the theme.

Switching the tabs feels slow because it forces the slowed down `List` to re-render. That's expected because the `tab` has changed, and so you need to reflect the user's new choice on the screen.

Next, try toggling the theme. **Thanks to `useMemo` together with `memo`, it's fast despite the artificial slowdown!** The `List` skipped re-rendering because the `visibleTodos` array has not changed since the last render. The `visibleTodos` array has not changed because both `todos` and `tab` (which you pass as dependencies to `useMemo`) haven't changed since the last render.

App.js TodoList.js List.js utils.js Reset

```
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}>
```

Show more

Next Example

Memoizing a dependency of another Hook

Suppose you have a calculation that depends on an object created directly in the component body:

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

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // 🚩 Caution: Dependency on an object crea
  // ...
```

Depending on an object like this defeats the point of memoization. When a component re-renders, all of the code directly inside the component body runs again. **The lines of code creating the `searchOptions` object will also run on every re-render.** Since `searchOptions` is a dependency of your `useMemo` call, and it's different every time, React knows the dependencies are different, and recalculate `searchItems` every time.

To fix this, you could memoize the `searchOptions` object *itself* before passing it as a dependency:

```
function Dropdown({ allItems, text }) {
  const searchOptions = useMemo(() => {
    return { matchMode: 'whole-word', text };
  }, [text]); // ✅ Only changes when text changes

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // ✅ Only changes when allItems or searchO
  // ...
```

In the example above, if the `text` did not change, the `searchOptions` object also won't change. However, an even better fix is to move the `searchOptions` object declaration *inside* of the `useMemo` calculation function:

```
function Dropdown({ allItems, text }) {
  const visibleItems = useMemo(() => {
    const searchOptions = { matchMode: 'whole-word', text };
    return searchItems(allItems, searchOptions);
  }, [allItems, text]); // ✅ Only changes when allItems or text changes
  // ...
```

Now your calculation depends on `text` directly (which is a string and can't "accidentally" become different).

Memoizing a function

Suppose the `Form` component is wrapped in `memo`. You want to pass a function to it as a prop:

```
export default function ProductPage({ productId, referrer }) {
  function handleSubmit(orderDetails) {
```

```
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }

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

Just as `{}` creates a different object, function declarations like `function() {}` and expressions like `() => {}` produce a *different* function on every re-render. By itself, creating a new function is not a problem. This is not something to avoid! However, if the `Form` component is memoized, presumably you want to skip re-rendering it when no props have changed. A prop that is *always* different would defeat the point of memoization.

To memoize a function with `useMemo`, your calculation function would have to return another function:

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

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

This looks clunky! **Memoizing functions is common enough that React has a built-in Hook specifically for that. Wrap your functions into `useCallback` instead of `useMemo` to avoid having to write an extra nested function:**

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

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

The two examples above are completely equivalent. The only benefit to `useCallback` is that it lets you avoid writing an extra nested function inside. It doesn't do anything else. Read more about `useCallback`.

Troubleshooting

My calculation runs twice on every re-render

In Strict Mode, React will call some of your functions twice instead of once:

```
function TodoList({ todos, tab }) {
  // This component function will run twice for every render.

  const visibleTodos = useMemo(() => {
    // This calculation will run twice if any of the dependencies change.
    return filterTodos(todos, tab);
  }, [todos, tab]);

  // ...
```

This is expected and shouldn't break your code.

This **development-only** behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component and calculation functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice and fix the mistake.

For example, this impure calculation function mutates an array you received as a prop:

```
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]);
```

React calls your function twice, so you'd notice the todo is added twice. Your calculation shouldn't change any existing objects, but it's okay to change any *new* objects you created during the calculation. For example, if the `filterTodos` function always returns a *different* array, you can mutate *that* array instead:

```
const visibleTodos = useMemo(() => {
  const filtered = filterTodos(todos, tab);
  // ✅ Correct: mutating an object you created during the calculation
  filtered.push({ id: 'last', text: 'Go for a walk!' });
  return filtered;
}, [todos, tab]);
```

Read keeping components pure to learn more about purity.

Also, check out the guides on updating objects and updating arrays without mutation.

My `useMemo` call is supposed to return an object, but returns undefined

This code doesn't work:

```
// 🔴 You can't return an object from an arrow function with () => {
const searchOptions = useMemo(() => {
  matchMode: 'whole-word',
  text: text
}, [text]);
```

In JavaScript, `() => {` starts the arrow function body, so the `{` brace is not a part of your object. This is why it doesn't return an object, and leads to mistakes. You could fix it by adding parentheses like `({` and `})`:

```
// This works, but is easy for someone to break again
const searchOptions = useMemo(() => ({
  matchMode: 'whole-word',
  text: text
}), [text]);
```

However, this is still confusing and too easy for someone to break by removing the parentheses.

To avoid this mistake, write a `return` statement explicitly:

```
// ✅ This works and is explicit
const searchOptions = useMemo(() => {
  return {
    matchMode: 'whole-word',
    text: text
```

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

Every time my component renders, the calculation in `useMemo` re-runs

Make sure you've specified the dependency array as a second argument!

If you forget the dependency array, `useMemo` will re-run the calculation every time:

```
function TodoList({ todos, tab }) {
  // 🔴 Recalculates every time: no dependency array
  const visibleTodos = useMemo(() => filterTodos(todos, tab));
  // ...
```

This is the corrected version passing the dependency array as a second argument:

```
function TodoList({ todos, tab }) {
  // ✅ Does not recalculate unnecessarily
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
```

If this doesn't help, then the problem is that at least one of your dependencies is different from the previous render. You can debug this problem by manually logging your dependencies to the console:

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

You can then right-click on the arrays from different re-renders in the console and select “Store as a global variable” for both of them. Assuming the first one got saved as `temp1` and the second one got saved as `temp2`, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between th
Object.is(temp1[1], temp2[1]); // Is the second dependency the same between t
Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

When you find which dependency breaks memoization, either find a way to remove it, or memoize it as well.

I need to call `useMemo` for each list item in a loop, but it’s not allowed

Suppose the `Chart` component is wrapped in `memo`. You want to skip re-rendering every `Chart` in the list when the `ReportList` component re-renders. However, you can’t call `useMemo` in a loop:

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 You can't call useMemo in a loop like this:
        const data = useMemo(() => calculateReport(item), [item]);
        return (
          <figure key={item.id}>
            <Chart data={data} />
          </figure>
        );
      })}
```

```
    </article>
  );
}
```

Instead, extract a component for each item and memoize data for individual items:

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

function Report({ item }) {
  // ✅ Call useMemo at the top level:
  const data = useMemo(() => calculateReport(item), [item]);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
}
```

Alternatively, you could remove `useMemo` and instead wrap `Report` itself in `memo`. If the `item` prop does not change, `Report` will skip re-rendering, so `Chart` will skip re-rendering too:

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

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

PREVIOUS
useLayoutEffect

NEXT
useReducer

How do you like these docs?

Take our survey!

Meta Open Source

Learn React

Quick Start

Installation

Describing the UI

API Reference

React APIs

React DOM APIs

Adding Interactivity

Managing State

Escape Hatches

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

More

Blog

React Native

Privacy

Terms