useCallback

`useCallback` is a React Hook that lets you cache a function definition between re-renders.

```
const cachedFn = useCallback(fn, dependencies)
```

Reference

`useCallback(fn, dependencies)`

Call `useCallback` at the top level of your component to cache a function definition between re-renders:

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

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

See more examples below.

Parameters

- `fn` : The function value that you want to cache. It can take any arguments and return any values. React will return (not call!) your function back to you during the initial render. On next renders, React will give you the same function again if the `dependencies` have not changed since the last render. Otherwise, it will give you the function that you have passed during the current render, and store it in case it can be reused later. React will not call your function. The function is returned to you so you can decide when and whether to call it.

- `dependencies` : The list of all reactive values referenced inside of the `fn` 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 algorithm.

Returns

On the initial render, `useCallback` returns the `fn` function you have passed.

During subsequent renders, it will either return an already stored `fn` function from the last render (if the dependencies haven't changed), or return the `fn` function you have passed during this render.

Caveats

- `useCallback` 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.
- React **will not throw away the cached function 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 match your expectations if you rely on `useCallback` as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Usage

Skipping re-rendering of components

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components. Let's first look at the syntax for how to do this, and then see in which cases it's useful.

To cache a function between re-renders of your component, wrap its definition into the `useCallback` Hook:

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

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

You need to pass two things to `useCallback`:

1. A function definition that you want to cache between re-renders.
2. A list of dependencies including every value within your component that's used inside your function.

On the initial render, the returned function you'll get from `useCallback` will be the function you passed.

On the following renders, React will compare the dependencies with the dependencies you passed during the previous render. If none of the dependencies have changed (compared with `Object.is`), `useCallback` will return the same function as before. Otherwise, `useCallback` will return the function you passed on *this* render.

In other words, `useCallback` caches a function between re-renders until its dependencies change.

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

Say you're passing a `handleSubmit` function down from the `ProductPage` to the `ShippingForm` component:

```
function ProductPage({ productId, referrer, theme }) {
  // ...
  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
```

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

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

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

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
  // ...
});
```

With this change, `ShippingForm` will skip re-rendering if all of its props are the *same* as on the last render. This is when caching a function becomes important! Let's say you defined `handleSubmit` without `useCallback`:

```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
```

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

  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will r
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

In JavaScript, a `function () {}` or `() => {}` always creates a *different* function, similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `ShippingForm` props will never be the same, and your `memo` optimization won't work. This is where `useCallback` comes in handy:

```
function ProductPage({ productId, referrer, theme }) {
  // Tell React to cache your function between re-renders...
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ...so as long as these dependencies don't cha

  return (
    <div className={theme}>
      {/* ...ShippingForm will receive the same props and can skip re-renderi
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

By wrapping `handleSubmit` in `useCallback`, you ensure that it's the *same* function between the re-renders (until dependencies change). You don't *have to* wrap a function in `useCallback` 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 other reasons you might need `useCallback` which are described further on this page.

Note

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

DEEP DIVE

How is useCallback related to useMemo?

Show Details

DEEP DIVE

Should you add useCallback everywhere?

Show Details

The difference between useCallback and declaring a function directly

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

Example 1 of 2:

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

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

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

Next, try toggling the theme. **Thanks to `useCallback` together with `memo`, it's fast despite the artificial slowdown!** `ShippingForm` skipped re-rendering because the `handleSubmit` function has not changed. The `handleSubmit` function has not changed because both `productId` and `referrer` (your `useCallback` dependencies) haven't changed since last render.

App.js ProductPage.js ShippingForm.js Reset

```
import { useCallback } from 'react';
import ShippingForm from './ShippingForm.js';

export default function ProductPage({ productId, referrer, theme
  const handleSubmit = useCallback((orderDetails) => {
```

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

Show more

Next Example

Updating state from a memoized callback

Sometimes, you might need to update state based on previous state from a memoized callback.

This `handleAddTodo` function specifies `todos` as a dependency because it computes the next todos from it:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos([...todos, newTodo]);
  }, [todos]);
  // ...
```

You'll usually want memoized functions to have as few dependencies as possible. When you read some state only to calculate the next state, you can remove that dependency by passing an updater function instead:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos(todos => [...todos, newTodo]);
  }, []); // ✅ No need for the todos dependency
  // ...
```

Here, instead of making `todos` a dependency and reading it inside, you pass an instruction about *how* to update the state (`todos => [...todos, newTodo]`) to React. Read more about updater functions.

Preventing an Effect from firing too often

Sometimes, you might want to call a function from inside an Effect:

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

```
  function createOptions() {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }

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

This creates a problem. Every reactive value must be declared as a dependency of your Effect. However, if you declare `createOptions` as a dependency, it will cause your Effect to constantly reconnect to the chat room:

```
  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🔴 Problem: This dependency changes on every render
  // ...
```

To solve this, you can wrap the function you need to call from an Effect into `useCallback`:

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

  const createOptions = useCallback(() => {
    return {
      serverUrl: 'https://localhost:1234',
```

```
      roomId: roomId
    };
  }, [roomId]); // ✅ Only changes when roomId changes

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // ✅ Only changes when createOptions changes
  // ...
```

This ensures that the `createOptions` function is the same between re-renders if the `roomId` is the same. **However, it's even better to remove the need for a function dependency.** Move your function *inside* the Effect:

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

  useEffect(() => {
    function createOptions() { // ✅ No need for useCallback or function depe
      return {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ Only changes when roomId changes
  // ...
```

Now your code is simpler and doesn't need `useCallback`. Learn more about removing Effect dependencies.

Optimizing a custom Hook

If you're writing a custom Hook, it's recommended to wrap any functions that it returns into `useCallback`:

```
function useRouter() {
  const { dispatch } = useContext(RouterStateContext);

  const navigate = useCallback((url) => {
    dispatch({ type: 'navigate', url });
  }, [dispatch]);

  const goBack = useCallback(() => {
    dispatch({ type: 'back' });
  }, [dispatch]);

  return {
    navigate,
    goBack,
  };
}
```

This ensures that the consumers of your Hook can optimize their own code when needed.

Troubleshooting

Every time my component renders, `useCallback` returns a different function

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

If you forget the dependency array, `useCallback` will return a new function every time:

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }); // 🔴 Returns a new function every time: no dependency array
  // ...
```

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

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ✅ Does not return a new function unnecessar
  // ...
```

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 handleSubmit = useCallback((orderDetails) => {
    // ..
  }, [productId, referrer]);
```

```
  console.log([productId, referrer]);
```

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 is breaking memoization, either find a way to remove it, or memoize it as well.

I need to call `useCallback` 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 `useCallback` in a loop:

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 You can't call useCallback in a loop like this:
        const handleClick = useCallback(() => {
          sendReport(item)
        }, [item]);
```

```
        return (
          <figure key={item.id}>
            <Chart onClick={handleClick} />
          </figure>
        );
      })}
    </article>
  );
}
```

Instead, extract a component for an individual item, and put `useCallback` there:

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

function Report({ item }) {
  // ✅ Call useCallback at the top level:
  const handleClick = useCallback(() => {
    sendReport(item)
  }, [item]);

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
}
```

Alternatively, you could remove `useCallback` in the last snippet 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 }) {
  function handleClick() {
    sendReport(item);
  }

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
});
```

PREVIOUS
use

NEXT
useContext

Meta Open Source

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Meet the Team

Docs Contributors

Acknowledgements

More

Blog

React Native

Privacy

Terms