useTransition

`useTransition` is a React Hook that lets you update the state without blocking the UI.

```
const [isPending, startTransition] = useTransition()
```

Reference

useTransition()

Call `useTransition` at the top level of your component to mark some state updates as transitions.

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

function TabContainer() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

See more examples below.

Parameters

`useTransition` does not take any parameters.

Returns

`useTransition` returns an array with exactly two items:

1. The `isPending` flag that tells you whether there is a pending transition.
2. The `startTransition` function that lets you mark a state update as a transition.

`startTransition` function

The `startTransition` function returned by `useTransition` lets you mark a state update as a transition.

```
function TabContainer() {
  const [isPending, startTransition] = useTransition();
```

```
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

Parameters

- `scope`: A function that updates some state by calling one or more `set` functions. React immediately calls `scope` with no parameters and marks all state updates scheduled synchronously during the `scope` function call as transitions. They will be non-blocking and will not display unwanted loading indicators.

Returns

`startTransition` does not return anything.

Caveats

- `useTransition` is a Hook, so it can only be called inside components or custom Hooks. If you need to start a transition somewhere else (for example, from a data library), call the standalone `startTransition` instead.
- You can wrap an update into a transition only if you have access to the `set` function of that state. If you want to start a transition in response to some prop or a custom Hook value, try `useDeferredValue` instead.
- The function you pass to `startTransition` must be synchronous. React immediately executes this function, marking all state updates that happen while it executes as transitions. If you try to perform more state updates later (for example, in a timeout), they won't be marked as transitions.

- A state update marked as a transition will be interrupted by other state updates. For example, if you update a chart component inside a transition, but then start typing into an input while the chart is in the middle of a re-render, React will restart the rendering work on the chart component after handling the input update.
- Transition updates can't be used to control text inputs.
- If there are multiple ongoing transitions, React currently batches them together. This is a limitation that will likely be removed in a future release.

Usage

Marking a state update as a non-blocking transition

Call `useTransition` at the top level of your component to mark state updates as non-blocking *transitions*.

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

function TabContainer() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

`useTransition` returns an array with exactly two items:

1. The `isPending` flag that tells you whether there is a pending transition.
2. The `startTransition` function that lets you mark a state update as a transition.

You can then mark a state update as a transition like this:

```
function TabContainer() {
  const [isPending, startTransition] = useTransition();
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

Transitions let you keep the user interface updates responsive even on slow devices.

With a transition, your UI stays responsive in the middle of a re-render. For example, if the user clicks a tab but then change their mind and click another tab, they can do that without waiting for the first re-render to finish.

The difference between useTransition and regular state updates

1. Updating the current tab in a transition
2. Updating the current tab

Example 1 of 2:

Updating the current tab in a transition

In this example, the “Posts” tab is **artificially slowed down** so that it takes at least a second to render.

Click “Posts” and then immediately click “Contact”. Notice that this interrupts the slow render of “Posts”. The “Contact” tab shows

immediately. Because this state update is marked as a transition, a slow re-render did not freeze the user interface.

App.js TabButton.js AboutTab.js PostsTab.js Con Reset

```
import { useState, useTransition } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
  const [isPending, startTransition] = useTransition();
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
```

Show more

Next Example

Updating the parent component in a transition

You can update a parent component's state from the `useTransition` call, too. For example, this `TabButton` component wraps its `onClick` logic in a transition:

```
export default function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  return (
    <button onClick={() => {
      startTransition(() => {
        onClick();
      });
    }}>
      {children}
    </button>
  );
}
```

Because the parent component updates its state inside the `onClick` event handler, that state update gets marked as a transition. This is why, like in the earlier example, you can click on "Posts" and then immediately click "Contact". Updating the selected tab is marked as a transition, so it does not block user interactions.

App.js TabButton.js AboutTab.js PostsTab.js ContactTab. Reset

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

export default function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
```

```
      if (isActive) {
        return <b>{children}</b>
      }
      return (
        <button onClick={() => {
          startTransition(() => {
            onClick();
```

Show more

Displaying a pending visual state during the transition

You can use the `isPending` boolean value returned by `useTransition` to indicate to the user that a transition is in progress. For example, the tab button can have a special “pending” visual state:

```
function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  // ...
  if (isPending) {
```

```
    return <b className="pending">{children}</b>;
  }
  // ...
```

Notice how clicking “Posts” now feels more responsive because the tab button itself updates right away:

App.js TabButton.js AboutTab.js PostsTab.js ContactTab. Reset

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

export default function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  return (
    <button onClick={() => {
```

Show more

Preventing unwanted loading indicators

In this example, the `PostsTab` component fetches some data using a Suspense-enabled data source. When you click the “Posts” tab, the `PostsTab` component *suspends*, causing the closest loading fallback to appear:

App.js TabButton.js Reset

```
import { Suspense, useState } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
  const [tab, setTab] = useState('about');
  return (
    <Suspense fallback={<h1>🌀 Loading...</h1>}>
      <TabButton
        isActive={tab === 'about'}
```

Show more

Hiding the entire tab container to show a loading indicator leads to a jarring user experience. If you add `useTransition` to `TabButton`, you can instead indicate display the pending state in the tab button instead.

Notice that clicking “Posts” no longer replaces the entire tab container with a spinner:

App.js TabButton.js Reset

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

export default function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  return (
    <button onClick={() => {
```

Show more

Read more about using transitions with Suspense.

Note

Transitions will only “wait” long enough to avoid hiding *already revealed* content (like the tab container). If the Posts tab had a nested `<Suspense>` boundary, the transition would not “wait” for it.

Building a Suspense-enabled router

If you’re building a React framework or a router, we recommend marking page navigations as transitions.

```
function Router() {
  const [page, setPage] = useState('/');
  const [isPending, startTransition] = useTransition();

  function navigate(url) {
    startTransition(() => {
      setPage(url);
    });
  }
  // ...
```

This is recommended for two reasons:

- Transitions are interruptible, which lets the user click away without waiting for the re-render to complete.
- Transitions prevent unwanted loading indicators, which lets the user avoid jarring jumps on navigation.

Here is a tiny simplified router example using transitions for navigations.

App.js Layout.js IndexPage.js ArtistPage.js Reset

```
import { Suspense, useState, useTransition } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
import Layout from './Layout.js';

export default function App() {
  return (
    <Suspense fallback={<BigSpinner />}>
      <Router />
    </Suspense>
  );
}
```

Show more

Note

Suspense-enabled routers are expected to wrap the navigation updates into transitions by default.

Troubleshooting

Updating an input in a transition doesn't work

You can't use a transition for a state variable that controls an input:

```
const [text, setText] = useState('');
// ...
function handleChange(e) {
  // ❌ Can't use transitions for controlled input state
  startTransition(() => {
    setText(e.target.value);
  });
}
// ...
return <input value={text} onChange={handleChange} />;
```

This is because transitions are non-blocking, but updating an input in response to the change event should happen synchronously. If you want to run a transition in response to typing, you have two options:

1. You can declare two separate state variables: one for the input state (which always updates synchronously), and one that you will update in a transition. This lets you control the input using the synchronous state, and pass the

transition state variable (which will “lag behind” the input) to the rest of your rendering logic.

2. Alternatively, you can have one state variable, and add `useDeferredValue` which will “lag behind” the real value. It will trigger non-blocking re-renders to “catch up” with the new value automatically.

React doesn’t treat my state update as a transition

When you wrap a state update in a transition, make sure that it happens *during* the `startTransition` call:

```
startTransition(() => {
  // ✅ Setting state *during* startTransition call
  setPage('/about');
});
```

The function you pass to `startTransition` must be synchronous.

You can’t mark an update as a transition like this:

```
startTransition(() => {
  // ❌ Setting state *after* startTransition call
  setTimeout(() => {
    setPage('/about');
  }, 1000);
});
```

Instead, you could do this:

```
setTimeout(() => {
  startTransition(() => {
```

```
    // ✅ Setting state *during* startTransition call
    setPage('/about');
  });
}, 1000);
```

Similarly, you can't mark an update as a transition like this:

```
startTransition(async () => {
  await someAsyncFunction();
  // ❌ Setting state *after* startTransition call
  setPage('/about');
});
```

However, this works instead:

```
await someAsyncFunction();
startTransition(() => {
  // ✅ Setting state *during* startTransition call
  setPage('/about');
});
```

I want to call `useTransition` from outside a component

You can't call `useTransition` outside a component because it's a Hook. In this case, use the standalone `startTransition` method instead. It works the same way, but it doesn't provide the `isPending` indicator.

The function I pass to `startTransition` executes immediately

If you run this code, it will print 1, 2, 3:

```
console.log(1);
startTransition(() => {
  console.log(2);
  setPage('/about');
});
console.log(3);
```

It is expected to print 1, 2, 3. The function you pass to `startTransition` does not get delayed. Unlike with the browser `setTimeout`, it does not run the callback later. React executes your function immediately, but any state updates scheduled *while it is running* are marked as transitions. You can imagine that it works like this:

```
// A simplified version of how React works

let isInsideTransition = false;

function startTransition(scope) {
  isInsideTransition = true;
  scope();
  isInsideTransition = false;
}

function setState() {
  if (isInsideTransition) {
    // ... schedule a transition state update ...
  } else {
    // ... schedule an urgent state update ...
  }
}
```

PREVIOUS

useSyncExternalStore

NEXT
Components

How do you like these docs?

Take our survey!

Meta Open Source

Acknowledgements

Terms