

我们最新的主要版本包括开箱即用的改进，如自动批处理，新的API，如startTransition，以及支持Suspense的流式服务器端渲染。

React 18很多新特性是在新的并发渲染器上构建的，这是一种幕后变化，可以解锁强大的新功能。React 并发是可选的，只有在使用并发特性时才启用，但我们认为它会对人们构建应用程序的方式产生重大影响。

## 什么是并发的React?

React 18中最重要的补充是，我们希望您永远不用考虑并发性。我们认为这在很大程度上对应用程序开发人员来说是正确的，尽管对于库维护人员来说可能会更复杂一些。

并发本身并不是一个特性。它是一种新的幕后机制，使React能够同时准备多个版本的UI。您可以将并发性看作是一个实现细节，它很有价值，因为它解锁了一些特性。React在其内部实现中使用了复杂的技术，如优先级队列和多重缓冲。但在我们的公共API中，您不会看到这些概念。

当我们设计API时，我们试图向开发人员隐藏实现细节。作为一名React开发人员，您关注的是您想要的用户体验，而React负责处理如何提供这种体验。因此，我们不希望React开发人员知道并发在幕后是如何工作的。

然而，Concurrent React比典型的实现细节更重要，它是React核心渲染模型的基础更新。因此，虽然了解并发是如何工作的并不十分重要，但在较高的级别上了解它是什么可能是值得的。

Concurrent React的一个关键特性是渲染是可中断的。当您第一次升级到React 18时，在添加任何并发功能之前，更新的呈现方式与以前版本的React相同-在单个、不间断的同步事务中。使用同步渲染，一旦更新开始渲染，在用户可以在屏幕上看到结果之前，没有任何东西可以中断它。

在并发渲染中，情况并不总是这样。React可能会开始渲染更新，在中间暂停，然后稍后继续。它甚至可能完全放弃正在进行的渲染。React保证即使渲染中断，UI也会保持一致。为此，一旦对整个树进行了计算，它将等待执行DOM突变直到结束。有了这个功能，React可以在后台准备新屏幕，而不会阻塞主线程。这意味着UI可以立即响应用户输入，即使它处于大型渲染任务的中间，也可以创建流畅的用户体验。

另一个例子是可重用状态。Concurrent React可以从屏幕上删除UI的各个部分，然后在重用之前的状态的同时将其添加回来。例如，当用户离开屏幕并返回时，React应该能够将前一个屏幕恢复到以前的状

态。在即将到来的一个小调中，我们计划添加一个名为<Offscreen>的新组件来实现此模式。类似地，您将能够使用Offscreen在后台准备新的UI，以便在用户显示之前做好准备

并发渲染是React中一个强大的新工具，我们的大多数新功能都是为了利用它而构建的，包括Suspense、转换和流式服务器渲染。但React 18只是我们在这个新基础上建立目标的开始。

## What's New in React 18

### 自动批处理

批处理是指React将多个状态更新分组为单个重新渲染以获得更好的性能。没有自动批处理，我们只在React事件处理程序中批处理更新。默认情况下，React不会批处理在promise、setTimeout、原生事件处理函数或者其他事件中的update。使用自动批处理，这些更新将自动批处理：

```
// Before: only React events were batched.
setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will render twice, once for each state update (no batching)
}, 1000);

// After: updates inside of timeouts, promises,
// native event handlers or any other event are batched.
setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

有关更多信息，请参阅此帖子，了解React 18中[更少渲染的自动批处理](#)。

### 概述

React 18通过在默认情况下进行更多的批处理，增加了开箱即用的性能提升，消除了在应用程序或库代码中手动批处理更新的需要。这篇文章将解释什么是批处理，它以前是如何工作的，以及发生了什么变化。

# 什么是批处理

批处理是指React将多个状态更新分组为单个重新渲染以获得更好的性能。

例如，如果你在同一个点击事件中有两个状态更新，React总是会把它们批处理成单次重新渲染。如果运行以下代码，您将看到每次单击时，React只执行一次渲染，尽管您设置了两次状态

```
function App() {
  const [count, setCount] = useState(0);
  const [flag, setFlag] = useState(false);

  function handleClick() {
    setCount(c => c + 1); // Does not re-render yet
    setFlag(f => !f); // Does not re-render yet
    // React will only re-render once at the end (that's batching!)
  }

  return (
    <div>
      <button onClick={handleClick}>Next</button>
      <h1 style={{ color: flag ? "blue" : "black" }}>{count}</h1>
    </div>
  );
}
```

这对性能非常有用，因为它避免了不必要的重新渲染。它还可以防止组件在只更新了一个状态变量的情况下呈现“半成品”状态，这可能会导致错误。这可能会提醒你，当你选择第一道菜时，餐厅服务员不会跑到厨房，而是等待你完成订单。

然而，React在何时批量更新方面并不一致。例如你想fetch数据，然后在上面的handleClick上更新数据，这个时候React不会批处理更新，而是会执行两次独立的更新。

这是因为React过去只在浏览器事件（如单击）期间进行批量更新，但在这里，我们在事件处理后更新状态（在fetch回调中）

```
function App() {
  const [count, setCount] = useState(0);
  const [flag, setFlag] = useState(false);

  function handleClick() {
    fetchSomething().then(() => {
      // React 17 and earlier does NOT batch these because
      // they run *after* the event in a callback, not *during* it
      setCount(c => c + 1); // Causes a re-render
      setFlag(f => !f); // Causes a re-render
    });
  }

  return (
    <div>
      <button onClick={handleClick}>Next</button>
      <h1 style={{ color: flag ? "blue" : "black" }}>{count}</h1>
    </div>
  );
}
```

React 17 或者更早的版本不会批处理因为他们会在事件之后才运行callback，而不是在事件过程中

在React18之前，我们只在React事件处理程序期间批量更新。默认情况下，在React中不会批处理promises、setTimeout、原生事件处理程序（addEventListener）或任何其他事件中的更新。

## 什么是自动批处理？

从使用createRoot的React 18开始，所有更新都将自动批处理，无论它们来自何处。

意味着对promises、setTimeout、原生事件处理程序（addEventListener）或任何其他事件中的更新将与React事件内部更新相同的方式进行批处理

因此：

```
function handleClick() {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}
```

和下面的行为是相同的：

```
setTimeout(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
}, 1000);
```

```
fetch(/*...*/).then(() => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
})
```

```
elm.addEventListener('click', () => {
  setCount(c => c + 1);
  setFlag(f => !f);
  // React will only re-render once at the end (that's batching!)
});
```

注意：React仅在通常安全的情况下进行批量更新。例如，React确保对于每个用户启动的事件（如单击或按键），DOM在下一个事件之前完全更新。例如，这可以确保在提交时禁用的表单不能提交两次。

## 万一我不想用批处理呢？

通常，批处理是安全的，但有些代码可能依赖于在状态更改后立即从DOM读取内容。对于这些用例，可以使用`ReactDOM.flushSync()`选择退出批处理：

```
import { flushSync } from 'react-dom'; // Note: react-dom, not react

function handleClick() {
  flushSync(() => {
    setCounter(c => c + 1);
  });
  // React has updated the DOM by now
  flushSync(() => {
    setFlag(f => !f);
  });
  // React has updated the DOM by now
}
```

```

flushSync(() => {
  setCount((c) => c + 1);
});

console.log(ref.current.style.color);
flushSync(() => {
  setFlag((f) => !f);
});
console.log(ref.current.style.color);

```

```

=== click ===
Render
Commit
black
Render
Commit
blue

```

## 这对Hooks有什么影响吗？

自动批处理在绝大多数情况下都能“正常工作”。也就是说hooks中也是批处理

## 对Class有影响吗？

请记住，React事件处理程序期间的更新始终是批处理的，因此这些更新没有任何更改。

类组件中存在边缘情况，这可能是一个问题。

类组件有一个实现怪癖，在那里可以同步读取事件内部的状态更新。这意味着你能在setState直接按读取this.state：



```

handleClick = () => {
  setTimeout(() => {
    this.setState(({ count }) => ({ count: count + 1 }));

    // { count: 1, flag: false }
    console.log(this.state);

    this.setState(({ flag }) => ({ flag: !flag }));
  });
};

```

```

setTimeout(() => {
  console.log(this.state);
  this.setState(({ count }) => ({ count: count + 1 }));

  // { count: 1, flag: false }
  console.log(this.state);

  this.setState(({ flag }) => ({ flag: !flag }));
});

```

```

▶ {flag: false, count: 0}
▶ {flag: false, count: 1}

```

在React 18中，这种情况不存在了

由于setTimeout中的所有更新都是批处理的，因此React不会同步渲染第一个setState的结果，渲染将在下一个浏览器计时期间进行。因此，渲染尚未发生：

```

handleClick = () => {
  setTimeout(() => {
    this.setState(({ count }) => ({ count: count + 1 }));

    // { count: 0, flag: false }
    console.log(this.state);

    this.setState(({ flag }) => ({ flag: !flag }));
  });
};

```

```

▶ {flag: false, count: 0}
▶ {flag: false, count: 0}

```

如果这个是升级到React 18的阻碍，那么你还是可以用ReactDOM.flushSync来强制更新：

```

handleClick = () => {
  setTimeout(() => {
    ReactDOM.flushSync(() => {
      this.setState(({ count }) => ({ count: count + 1 }));
    });

    // { count: 1, flag: false }
    console.log(this.state);

    this.setState(({ flag }) => ({ flag: !flag }));
  });
};

```

此问题不会影响带hooks的函数组件，因为设置状态不会更新useState中的现有变量：

```

function handleClick() {
  setTimeout(() => {
    console.log(count); // 0
    setCount(c => c + 1);
    setCount(c => c + 1);
    setCount(c => c + 1);
    console.log(count); // 0
  }, 1000)
}

```

## Transitions

过渡是一个新概念，用来区分紧急更新和非紧急更新

- 紧急更新反映了直接的交互，如键入、单击、按下等
- 过渡更新将UI从一个视图过渡到另一个视图

诸如键入、单击或按下等紧急更新需要立即响应，以匹配我们对物理对象行为的直觉。否则会感觉不对劲。然而，过渡是不同的，因为用户不希望看到每个中间的值显示在屏幕上

比如，当你在下拉列表中选择了一个filter，你期望这个filter按钮能够立即响应你的点击。然而，真实情况可能分开过渡。短暂的延迟是无法察觉的并且也是期望的。如果你在结果渲染完之前，再次改变过滤器，你只想看到最新的结果

通常，为了获得最佳用户体验，单个用户的输入应该导致紧急更新和非紧急更新。您可以在输入事件中使用startTransition API来通知React哪些更新是紧急的，哪些是“转换”：



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

// Urgent: Show what was typed
setInputValue(input);

// Mark any state updates inside as transitions
startTransition(() => {
  // Transition: Show the results
  setSearchQuery(input);
});
```

包裹在startTransition里面的更新会被处理为非紧急更新，并且会被更加紧急的更新打断，比如点击事件或者按键。如果转换被用户打断（例如，通过在一行中键入多个字符），React将丢弃尚未完成的旧的渲染工作，并只渲染最新的更新

- useTransition: 过渡的钩子，它包含一个值来跟踪挂起的状态
- startTransition: 一个用来开启过渡的方法，它在hook不能使用时使用（在class组件中）

过渡将选择**并发渲染**，从而允许中断更新。如果内容重新挂起，转换也会告诉React继续显示当前内容，同时在后台呈现转换内容

## React.startTransition

React.startTransition(callback)

React.startTransition 让你把提供的 fallback 里面的更新标记为 transitions。这个方法是为了在 [React.useTransition](#) 不可用时使用。

### 注意：

过渡期的更新会被更紧急的更新取代，如点击操作。

过渡期的更新不会显示重新挂起内容的 fallback，允许用户在渲染更新时继续进行交互。

React.startTransition 不提供 **isPending** 的标志。要跟踪过渡的待定状态，请参阅 [React.useTransition](#)。

# useTransition

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

返回一个状态值表示过渡任务的等待状态，以及一个启动该过渡任务的函数。

`startTransition` 允许你通过标记更新将提供的回调函数作为一个过渡任务：

```
startTransition(() => {  
  setCount(count + 1);  
})
```

`isPending` 指示过渡任务何时活跃以显示一个等待状态：

```
function App() {  
  const [isPending, startTransition] = useTransition();  
  const [count, setCount] = useState(0);  
  
  function handleClick() {  
    startTransition(() => {  
      setCount(c => c + 1);  
    })  
  }  
  
  return (  
    <div>  
      {isPending && <Spinner />}  
      <button onClick={handleClick}>{count}</button>  
    </div>  
  );  
}
```

## 注意：

过渡任务中触发的更新会让更紧急地更新先进行，比如点击。

过渡任务中的更新将不会展示由于再次挂起而导致降级的内容。这个机制允许用户在 React 渲染更新的时候继续与当前内容进行交互。

上一次的render还没渲染结束，block了下一次的渲染，导致input和其他的都没响应。

```

export const Transition = () => {
  const [isPending, startTransition] = useTransition();
  const [input, setInput] = useState('');
  const [list, setList] = useState([]);

  function handleChange(e) {
    setInput(e.target.value);

    // 设置完input之后，如果不用transition，那么会等到for循环执行完之后，setList，并把这两个更新合并成一次渲染，同时显示在页面上
    // 由于mapping和for循环化了太多时间，会导致页面卡顿
    startTransition(() => {
      // 我们优先想显示input，然后list的展示是低优先级的，可以被打断的，如果我在它计算过程中输入了新的值
      let l = [];
      for (let i = 0; i < LIST_SIZE; i++) {
        l.push(e.target.value);
      }

      setList(l);
    });

    // startTransition 将setList的update设置为低优先级，并放在后台进行计算。
    // 这个时候onChange时间的update会分开为2次render，第一次render会更新input，第二次就是下面的列表
  }

  return (
    <div>
      <input type='text' value={input} style={{ height: '50px', width: '500px', fontSize: '25px', margin: '30px' }} onChange={handleChange} />
      <br />
      {isPending ? (
        <div>'loading...'</div>
      ) : (
        list.map((value, index) => {
          return <div key={index}>{value}</div>;
        })
      )}
    </div>
  );
};

```

这里的isPending代表的是transition的更新开始到完全显示到页面之间的状态

# New Suspense Features

## New Client and Server Rendering APIs

在这个版本中，我们利用这个机会重新设计了在客户端和服务端上呈现的API。这些更改允许用户在React 17模式下继续使用旧API，同时升级到React 18中的新API。

## React DOM Client

这些新API现在从react-dom/client导出：

- createRoot：创建要渲染或卸载的根的新方法。使用它代替ReactDOM.render。React 18中的新功能没有它就无法工作。
- hydrateRoot：对服务器渲染的应用程序进行水合的新方法。使用它代替ReactDOM。与新的React DOM Server API结合使用。React 18中的新功能没有它就无法工作。

# 新 Strict Mode 行为

将来，我们希望添加一个功能，允许React在保留状态的同时添加和删除UI的部分。例如，当用户离开屏幕并返回时，React应该能够立即显示上一个屏幕。为此，React将使用与以前相同的组件状态卸载和重新装载树。

此功能将为React应用提供更好的开箱即用性能，但要求组件对多次装载和销毁的效果具有弹性。大多数效果将在没有任何更改的情况下工作，但有些效果假定它们只装载或销毁一次。

为了帮助解决这些问题，React 18引入了一个新的仅限开发的严格模式检查。每当组件首次装载时，此新检查将自动卸载并重新装载每个组件，从而恢复第二次装载时的前一状态。

在此更改之前，React将安装组件并创建effects：

```
* React mounts the component.  
  * Layout effects are created.  
  * Effects are created.
```

使用React 18中的Strict Mode，React将模拟在开发模式下卸载和重新安装组件：

```
* React mounts the component.  
  * Layout effects are created.  
  * Effects are created.  
* React simulates unmounting the component.  
  * Layout effects are destroyed.  
  * Effects are destroyed.  
* React simulates mounting the component with the previous state.  
  * Layout effects are created.  
  * Effects are created.
```

## New Hooks

### useId

### useTransition

# useDeferredValue

useDeferredValue 允许您延迟重新渲染树的非紧急部分。它类似于去抖，但与之相比有一些优点。没有固定的时间延迟，因此 React 将在第一次渲染反映在屏幕上后立即尝试延迟渲染。延迟渲染是可中断的，不会阻止用户输入

```
const deferredValue = useDeferredValue(value);
```

useDeferredValue 接受一个值，并返回该值的新副本，该副本将推迟到更紧急地更新之后。如果当前渲染是一个紧急更新的结果，比如用户输入，React 将返回之前的值，然后在紧急渲染完成后渲染新的值。

该 hook 与使用防抖和节流去延迟更新的用户空间 hooks 类似。使用 useDeferredValue 的好处是，React 将在其他工作完成（而不是等待任意时间）后立即进行更新，并且像 startTransition 一样，延迟值可以暂停，而不会触发现有内容的意外降级。

## Memoizing deferred children

useDeferredValue 仅延迟你传递给它的值。如果你想要在紧急更新期间防止子组件重新渲染，则还必须使用 React.memo 或 React.useMemo 记忆该子组件：

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing 告诉 React 仅当 deferredQuery 改变,
  // 而不是 query 改变的时候才重新渲染
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

记忆该子组件告诉 React 它仅当 deferredQuery 改变而不是 query 改变的时候才需要去重新渲染。这个限制不是 useDeferredValue 独有的，它和使用防抖或节流的 hooks 使用的相同模式。



```
const [list, setList] = useState([]);  
const deferredList = useDeferredValue(list);
```

产生和useTransition一样的效果



## 与useTransition区别

- 1 对同一个资源的优化，这两个接口的提供的优化效果是一样的，因此不需要同时使用，也就是说使用一个就行了，因为一旦使用这两个任何一个都会带来一定性能上的损耗。
- 2 建议只有数据量大的时候考虑使用这两个接口中的一个，平时的普通组件不需要使用，原因是使用这两个任何一个都会带来一定性能上的损耗。
- 3 既然使用哪个接口都一样，为啥做了两个接口？因为：useTransition是用来处理更新函数的，而useDeferredValue是用来处理更新函数执行后所更新的数据本身的。有些情况下，你并不能直接获得更新函数，比如你是用的是第三方的hooks库，你在使用的时候更新函数并不能直接对外暴露，这时候你就只能去优化数据，从而只能使用useDeferredValue，useTransition的好处是它可以一次性的处理好几个更新函数。



**useId**

**useSyncExternalStore**

**useInsertionEffect**