React

Hook

use

它可以让你读取类似于`Promise`或`context`的资源的值

```
const value = use(resource);
```

示例

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

function MessageComponent({ messagePromise }) {
  const message = use(messagePromise);
  const theme = use(ThemeContext);
  // ...
```

于其他React Hook不同的是，可以在循环和条件语句（如if）中调用use。但需要注意的是，调用`use`的函数仍然必须是一个组件或者Hook。

当使用Promise调用`use` Hook 时， 它会与`Suspense`的`错误边界`集成。当传递给`use`的Promise处于pending时，调用`use`的组件也会挂起。如果用use的组件被包装在Suspense边界内，将显示后备UI。一旦Promise被解决，Suspense后背方案将被使用`use`Hook返回的数据替换。如果传递给`use`Promise被拒绝，将显示最近错误边界的后背UI

useCallback

`useCallback`是一个允许你多次渲染中缓存函数的React Hook

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

用法

- 跳过组件的重新渲染
- 从记忆化回调中更新state
- 防止频繁触发Effect
- 优化自定义Hook

参数

- fn: 想要缓存的函数。此函数可以接受任何参数并且返回任何值。React将会在初次渲染而非调用时返回该函数。当进行下一次渲染时，如果`dependencies`相比于上一次渲染时没有改变，那么React将会返回相同的函数。否则，React将返回在最新一次渲染中传入的函数，并且将其缓存以便之后使用。React不会调用此函数，而是返回此函数。你可以自己决定何时调用以及是否调用。

- `dependencies`：有关是否更新 fn的所有响应式值的列表。响应式值包括props，state，和所有在你组件内部直接声明的变量和函数。如果你的代码检查工具配置了React，那么它将校验每一个正确指定为依赖的响应式值。依赖列表必须具有确切数量的项，并且必须像 `[dep1, dep2, dep3]` 这样编写。React 使用 `Object.is` 比较每一个依赖和它的之前的值。

返回值

在初次渲染时，`useCallback` 返回你已经传入的 `fn` 函数

在之后的渲染中, 如果依赖没有改变，`useCallback` 返回上一次渲染中缓存的 `fn` 函数；否则返回这一次渲染传入的 `fn`。

注意
`useCallback` **只应作用于性能优化。**如果代码在没有它的情况下无法运行，请找到根本问题并首先修复它，然后再使用 `useCallback`。

`useCallback` 与 `useMemo` 有何关系?

`useMemo` 经常与 `useCallback` 一同出现。当尝试优化子组件时，它们都很有用。他们会 记住（或者说，缓存）正在传递的东西：

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

function ProductPage({ productId, referrer }) {
  const product = useData('/product/' + productId);

  const requirements = useMemo(() => { //调用函数并缓存结果
    return computeRequirements(product);
  }, [product]);

  const handleSubmit = useCallback((orderDetails) => { // 缓存函数本身
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  return (
    <div className={theme}>
      <ShippingForm requirements={requirements} onSubmit={handleSubmit} />
    </div>
  );
}
```

区别在于你需要缓存 **什么**:

- `useMemo` **缓存函数调用的结果。**在这里，它缓存了调用 `computeRequirements(product)` 的结果。除非 `product` 发生改变，否则它将不会发生变化。这让你向下传递 `requirements` 时而无需不必要地重新渲染 `ShippingForm`。必要时，React 将会调用传入的函数重新计算结果。

- `useCallback` **缓存函数本身**。不像 `useMemo`，它不会调用你传入的函数。相反，它缓存此函数。从而除非 `productId` 或 `referrer` 发生改变，`handleSubmit` 自己将不会发生改变。这让你向下传递 `handleSubmit` 函数而无需不必要地重新渲染 `ShippingForm`。直至用户提交表单，你的代码都将不会运行。

是否应该在任何地方添加 `useCallback`？

如果你的应用程序与本网站类似，并且大多数交互都很粗糙（例如替换页面或整个部分），则通常不需要缓存。另一方面，如果你的应用更像是一个绘图编辑器，并且大多数交互都是精细的（如移动形状），那么你可能会发现缓存非常有用。

使用 `useCallback` 缓存函数仅在少数情况下有意义：

- 将其作为 props 传递给包装在 [`memo`] 中的组件。如果 props 未更改，则希望跳过重新渲染。缓存允许组件仅在依赖项更改时重新渲染。
- 传递的函数可能作为某些 Hook 的依赖。比如，另一个包裹在 `useCallback` 中的函数依赖于它，或者依赖于 `useEffect` 中的函数。

useContext

`useContext` 可以让你读取和订阅组件中的context

```
const value = useContext(someContext)
```

参数

- someContext：先前用 `createContext` 创建的context。context本身不包含信息，它只代表你可以提供或者从组件中读取的信息类型。

返回值

`useContext` 为调用组件返回的context的值。它被确定为传递给树中调用组件上方最近的SomeContext.provider的value。如果没有这样的provider，那么返回值将会是为创建该context传递给 `createContext` 的defaultValue。返回的值始终是最新的。如果Context发生变化，React会自动重新渲染读取context的组件。

注意事项

- 组件中的 `useContext()` 调用不受 **同一** 组件返回的 provider 的影响。相应的 `<Context.Provider>` 需要位于调用 `useContext()` 的组件 **之上**。
- 从 provider 接收到不同的 `value` 开始，React 自动重新渲染使用了该特定 context 的所有子级。先前的值和新的值会使用 `Object.is` 来做比较。使用 `memo` 来跳过重新渲染并不妨碍子级接收到新的 context 值。
- 如果您的构建系统在输出中产生重复的模块（可能发生在符号链接中），这可能会破坏 context。通过 context 传递数据只有在用于传递 context 的 `SomeContext` 和用于读取数据的 `SomeContext` 是完全相同的对象时才有效，这是由 `===` 比较决定的。

useReducer

它允许你向组件里面添加一个`reducer`

```
const [ state, dispatch ] = useReducer(reducer, initialArg, init?)
```

用法

- 向组件添加reducer
- 实现reducer函数
- 避免重新创建初始值

参数

- `reducer` 用于更新state的纯函数。参数为state和action，返回值是更新后的state。state与action可以是任意合法值
- `inittialArg` 用于初始化state的任意值，初始值的计算逻辑取决于接下来的init参数
- 可选参数`init`：用于计算初始值的函数，如果存在，使用`init(initialArg)`的执行结果作为初始值，否则使用initialArg。

返回值

`useReducer` 返回一个两个值组成的数组：

1. 当前的state。初次渲染时，它是`init(initialArg)`或`initialArg`(如果没有init函数)。
2. `dispatch函数`。用于更新state并触发组件的重新渲染。

注意事项

- `useReducer` 是一个hook， 所以只能在组件的顶层作用域或自定义Hook中调用，而不能在循环或者条件语句中调用，如果你有这种需求，可以创建一个新的组件，并把state移入其中。
- 严格模式下React会调用两次reducer和初始化函数，这可以帮助你发现意外的副作用，这只是开发模式下的行为，并不会影响生成环境，只要reducer和初始化函数是纯函数就不会改变你的逻辑，其中一个调用结果会被忽略。

dispatch函数

`useReducer` 返回的 `dispatch` 函数允许你更新 state 并触发组件的重新渲染。它需要传入一个 action 作为参数：

```
const [state, dispatch] = useReducer(reducer, { age: 42 });

function handleClick() {
  dispatch({ type: 'incremented_age' });
  // ...
```

React 会调用 `reducer` 函数以更新 state， `reducer` 函数的参数为当前的 state 与传递的 action。

参数

- `action`：用户执行的操作。可以是任意类型的值。通常来说 action 是一个对象，其中 `type` 属性标识类型，其它属性携带额外信息。

返回值

`dispatch` 函数没有返回值。

注意

- `dispatch` 函数 **是为下一次渲染而更新 state**。因此在调用 `dispatch` 函数后读取 state 并不会拿到更新后的值，也就是说只能获取到调用前的值。
- 如果你提供的新值与当前的 `state` 相同（使用 `Object.is` 比较），React 会 **跳过组件和子组件的重新渲染**，这是一种优化手段。虽然在跳过重新渲染前 React 可能会调用你的组件，但是这不应该影响你的代码。
- React 会批量更新 state。state 会在 **所有事件函数执行完毕** 并且已经调用过它的 `set` 函数后进行更新，这可以防止在一个事件中多次进行重新渲染。如果在访问 DOM 等极少数情况下需要强制 React 提前更新，可以使用 `flushSync`。

使用 Immer 编写简洁的更新逻辑

如果使用复制方法更新数组和对象让你不厌其烦，那么可以使用 Immer 这样的库来减少一些重复的样板代码。Immer 让你可以专注于逻辑，因为它在内部均使用复制方法来完成更新：

```
import { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

function tasksReducer(draft, action) {
  switch (action.type) {
    case 'added': {
      draft.push({
        id: action.id,
        text: action.text,
        done: false
      });
      break;
    }
    case 'changed': {
      const index = draft.findIndex(t =>
        t.id === action.task.id
      );
      draft[index] = action.task;
      break;
    }
    case 'deleted': {
      return draft.filter(t => t.id !== action.id);
    }
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
```

```
}

export default function TaskApp() {
  const [tasks, dispatch] = useImmerReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task
    });
  }

  function handleDeleteTask(taskId) {
    dispatch({
      type: 'deleted',
      id: taskId
    });
  }

  return (
    <>
      <h1>Prague itinerary</h1>
      <AddTask
        onAddTask={handleAddTask}
      />
      <TaskList
        tasks={tasks}
        onChangeTask={handleChangeTask}
        onDeleteTask={handleDeleteTask}
      />
    </>
  );
}

let nextId = 3;
const initialTasks = [
  { id: 0, text: 'Visit Kafka Museum', done: true },
  { id: 1, text: 'Watch a puppet show', done: false },
  { id: 2, text: 'Lennon Wall pic', done: false },
];
```

避免重新创建初始值

React 会保存 state 的初始值并在下一次渲染时忽略它。

```
function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, createInitialState(username));
  // ...
```

虽然 `createInitialState(username)` 的返回值只用于初次渲染，但是在每一次渲染的时候都会被调用。如果它创建了比较大的数组或者执行了昂贵的计算就会浪费性能。

你可以通过给 `useReducer` 的第三个参数传入 **初始化函数** 来解决这个问题：

```
function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, username, createInitialState);
  // ...
```

需要注意的是你传入的参数是 `createInitialState` 这个 **函数自身**，而不是执行 `createInitialState()` 后的返回值。这样传参就可以保证初始化函数不会再次运行。

在上面这个例子中，`createInitialState` 有一个 `username` 参数。如果初始化函数不需要参数就可以计算出初始值，可以把 `useReducer` 的第二个参数改为 `null`。

问题?

我已经 dispatch 了一个 action，但是打印出来仍然还是旧的 state

调用 `dispatch` 函数 **不会改变当前渲染的 state**：

```
function handleClick() {
  console.log(state.age);  // 42

  dispatch({ type: 'incremented_age' }); // 用 43 进行重新渲染
  console.log(state.age);  // 还是 42!

  setTimeout(() => {
    console.log(state.age); // 一样是 42!
  }, 5000);
}
```

这是因为 state 的行为和快照一样。更新 state 会使用新的值来对组件进行重新渲染，但是不会改变当前执行的事件处理函数里面 `state` 的值。

如果你需要获取更新后的 state，可以手动调用 reducer 来得到结果：

```
const action = { type: 'incremented_age' };
dispatch(action);

const nextState = reducer(state, action);
console.log(state);     // { age: 42 }
console.log(nextState); // { age: 43 }
```

我已经 dispatch 了一个 action，但是屏幕并没有更新

React 使用 `Object.is` 比较更新前后的 state，如果 **它们相等就会跳过这次更新。**这通常是因为你直接修改了对象或数组：

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // 🚩 错误行为：直接修改对象
      state.age++;
      return state;
    }
    case 'changed_name': {
      // 🚩 错误行为：直接修改对象
      state.name = action.nextName;
      return state;
    }
    // ...
  }
}
```

你直接修改并返回了一个 `state` 对象，所以 React 会跳过这次更新。为了修复这个错误，你应该确保总是使用正确的方式更新对象 和 使用正确的方式更新数组：

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ✅ 修复：创建一个新的对象
      return {
        ...state,
        age: state.age + 1
      };
    }
    case 'changed_name': {
      // ✅ 修复：创建一个新的对象
      return {
        ...state,
        name: action.nextName
      };
    }
    // ...
```

```
  }
}
```

在 dispatch 后 state 的某些属性变为了 `undefined`

请确保每个 `case` 语句中所返回的新的 state **都复制了当前的属性**：

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      return {
        ...state, // 不要忘记复制之前的属性！
        age: state.age + 1
      };
    }
    // ...
```

在 dispatch 后整个 state 都变为了 `undefined`

如果你的 state 错误地变成了 `undefined`，可能是因为你忘记在某个分支返回 state，或者是你遗漏了某些 `case` 分支。可以通过在 `switch` 语句之后抛出一个错误来查找原因：

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ...
    }
    case 'edited_name': {
      // ...
    }
  }
  throw Error('Unknown action: ' + action.type);
}
```

我收到了一个报错：“Too many re-renders”

你可能会收到这样一条报错信息：`Too many re-renders. React limits the number of renders to prevent an infinite loop.`。这通常是在 **渲染期间** dispatch 了 action 而导致组件进入了无限循环：dispatch（会导致一次重新渲染）、渲染、dispatch（再次导致重新渲染），然后无限循环。大多数这样的错误是由于事件处理函数中存在错误的逻辑：

```
// 🚩 错误：渲染期间调用了处理函数
return <button onClick={handleClick()}>Click me</button>

// ✅ 修复：传递一个处理函数，而不是调用
return <button onClick={handleClick}>Click me</button>

// ✅ 修复：传递一个内联的箭头函数
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

我的 reducer 和初始化函数运行了两次

严格模式 下 React 会调用两次 reducer 和初始化函数，但是这不应该会破坏你的代码逻辑。

这个 **仅限于开发模式** 的行为可以帮助你 保持组件纯粹：React 会使用其中一次调用结果并忽略另一个结果。如果你的组件、初始化函数以及 reducer 函数都是纯函数，这并不会影响你的逻辑。不过一旦它们存在副作用，这个额外的行为就可以帮助你发现它。

比如下面这个 reducer 函数直接修改了数组类型的 state

```
function reducer(state, action) {
  switch (action.type) {
    case 'added_todo': {
      // 🚩 错误：直接修改 state
      state.todos.push({ id: nextId++, text: action.text });
      return state;
    }
    // ...
  }
}
```

因为 React 会调用 reducer 函数两次，导致你看到添加了两条代办事项，于是你就发现了这个错误行为。在这个示例中，你可以通过 返回新的数组而不是修改数组 来修复它：

```
function reducer(state, action) {
  switch (action.type) {
    case 'added_todo': {
      // ✅ 修复：返回一个新的 state 数组
      return {
        ...state,
        todos: [
          ...state.todos,
          { id: nextId++, text: action.text }
        ]
      };
    }
    // ...
  }
}
```

useEffect

它允许你将组件与外部系统同步

用法

- 连接到外部系统
- 在自定义Hook中封装Effect
- 控制非React小部件
- 使用Effect请求数据

- 指定响应式依赖
- 在Effect中根据先前state更新state
- 删除不必要的对象依赖项
- 删除不必要的函数依赖项
- 从Effect读取最新的props和state
- 在服务器可客户端上显示不同的内容

```
useEffect(setup, dependencies?)
```

在组件的顶层调用 `useEffect` 来声明一个 Effect：

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

参数

- setup： 处理Effect的函数。setup函数选择性返回一个清理函数。当组件被添加到DOM的时候，React将运行setup函数。在每一次依赖项变更重新渲染后，React将首先使用旧值运行清理函数（如果你提供了该函数），然后使用新值运行setup函数。在组件从Dom中移除后，React将最后一次运行清理函数。
- **可选** `dependencies`：`setup` 代码中引用的所有响应式值的列表。响应式值包括 props、state 以及所有直接在组件内部声明的变量和函数。如果你的代码检查工具 配置了 React，那么它将验证是否每个响应式值都被正确地指定为一个依赖项。依赖项列表的元素数量必须是固定的，并且必须像 `[dep1, dep2, dep3]` 这样内联编写。React 将使用 `Object.is` 来比较每个依赖项和它先前的值。如果省略此参数，则在每次重新渲染组件之后，将重新运行 Effect 函数。如果你想了解更多，请参见 传递依赖数组、空数组和不传递依赖项之间的区别。

返回值

`useEffect` 返回 `undefined`。

注意

- `useEffect` 是一个 Hook，因此只能在 **组件的顶层** 或自己的 Hook 中调用它，而不能在循环或者条件内部调用。如果需要，抽离出一个新组件并将 state 移入其中。

- 如果你 **没有打算与某个外部系统同步**，那么你可能不需要 Effect。
- 当严格模式启动时，React 将在真正的 setup 函数首次运行前，**运行一个开发模式下专有的额外 setup + cleanup 周期。**这是一个压力测试，用于确保 cleanup 逻辑"映射"到了 setup 逻辑，并停止或撤消 setup 函数正在做的任何事情。如果这会导致一些问题，
- 如果你的一些依赖项是组件内部定义的对象或函数，则存在这样的风险，即它们将 **导致 Effect 过多地重新运行。**要解决这个问题，请删除不必要的 对象 和 函数 依赖项。你还可以 抽离状态更新 和 非响应式的逻辑 到 Effect 之外。
- 如果你的 Effect 不是由交互（比如点击）引起的，那么 React 会让浏览器 **在运行 Effect 前先绘制出更新后的屏幕。**如果你的 Effect 正在做一些视觉相关的事情（例如，定位一个 tooltip），并且有显著的延迟（例如，它会闪烁），那么将 `useEffect` 替换为 `useLayoutEffect`。
- 即使你的 Effect 是由一个交互（比如点击）引起的，**浏览器也可能在处理 Effect 内部的状态更新之前重新绘制屏幕。**通常，这就是你想要的。但是，如果你一定要阻止浏览器重新绘制屏幕，则需要用 `useLayoutEffect` 替换 `useEffect`。
- Effect **只在客户端上运行**，在服务端渲染中不会运行。

连接到外部系统

有些组件需要与网络、某些浏览器 API 或第三方库保持连接，当它们显示在页面上时。这些系统不受 React 控制，所以称为外部系统。

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

在自定义 Hook 中封装 Effect

Effect 是一个 "逃生出口"：当你需要"走出 React 之外"或者当你的使用场景没有更好的内置解决方案时，你可以使用它们。如果你发现自己经常需要手动编写 Effect，那么这通常表明你需要为组件所依赖的通用行为提取一些 自定义 Hook。

```
function useChatRoom({ serverUrl, roomId }) {
  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId, serverUrl]);
}
```

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useChatRoom({
    roomId: roomId,
    serverUrl: serverUrl
  });
  // ...
```

控制非 React 小部件

有时，你希望外部系统与你组件的某些 props 或 state 保持同步。

例如，如果你有一个没有使用 React 编写的第三方地图小部件或视频播放器组件，你可以使用 Effect 调用该组件上的方法，使其状态与 React 组件的当前状态相匹配。此 Effect 创建了在 `map-widget.js` 中定义的 `MapWidget` 类的实例。当你更改 `Map` 组件的 `zoomLevel` prop 时，Effect 调用类实例上的 `setZoom()` 来保持同步

使用 Effect 请求数据

```
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
  const [person, setPerson] = useState('Alice');
  const [bio, setBio] = useState(null);

  useEffect(() => {
    let ignore = false;
    setBio(null);
    fetchBio(person).then(result => {
      if (!ignore) {
        setBio(result);
      }
    });
    return () => {
      ignore = true;
    };
```

```
  }, [person]);

  // ...
```

指定响应式依赖项

你无法“选择” Effect 的依赖项。Effect 代码中使用的每个 响应式值 都必须声明为依赖项。你的 Effect 依赖列表是由周围代码决定的：

```
function ChatRoom({ roomId }) { // 这是一个响应式值
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // 这也是一个响应式值

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId); // 这个 Effect 读取这些响应式值
    connection.connect();
    return () => connection.disconnect();
  }, [serverUrl, roomId]); // ✅ 因此你必须指定它们作为 Effect 的依赖项
  // ...
}
```

在 Effect 中根据先前 state 更新 state

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(count + 1); // 你想要每秒递增该计数器...
    }, 1000)
    return () => clearInterval(intervalId);
  }, [count]); // 🚩 ... 但是指定 `count` 作为依赖项总是重置间隔定时器。
  // ...
}
```

删除不必要的对象依赖项

```
const serverUrl = 'https://localhost:1234';

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

  const options = { // 🚩 这个对象在每次渲染时都是从头创建的
    serverUrl: serverUrl,
    roomId: roomId
  };

  useEffect(() => {
    const connection = createConnection(options); // 它在 Effect 内部使用
```

```
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // 🚩 因此，这些依赖在重新渲染时总是不同的
  // ...
```

删除不必要的函数依赖项

如果你的 Effect 依赖于在渲染期间创建的对象或函数，则它可能会频繁运行。例如，此 Effect 在每次渲染后重新连接，因为 `createOptions` 函数

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

  function createOptions() { // 🚩 此函数在每次重新渲染都从头开始创建
    return {
      serverUrl: serverUrl,
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions(); // 它在 Effect 中被使用
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🚩 因此，此依赖项在每次重新渲染都是不同的
  // ...
```

从 Effect 读取最新的 props 和 state

```
function Page({ url, shoppingCart }) {
  useEffect(() => {
    logVisit(url, shoppingCart.length);
  }, [url, shoppingCart]); // ✅ 所有声明的依赖项
  // ...
}
```

在服务器和客户端上显示不同的内容

如果你的应用程序使用服务端（直接 或通过 框架）渲染，你的组件将会在两个不同的环境中渲染。在服务器上，它将渲染生成初始 HTML。在客户端，React 将再次运行渲染代码，以便将事件处理附加到该 HTML 上。这就是为什么要让 hydration 发挥作用，你的初始渲染输出必须在客户端和服务器上完全相同的原因。

在极少数情况下，你可能需要在客户端上显示不同的内容。例如，如果你的应用从 `localStorage` 中读取某些数据，服务器上肯定不可能做到这一点。以下是这如何实现的：

```
function MyComponent() {
  const [didMount, setDidMount] = useState(false);

  useEffect(() => {
    setDidMount(true);
  }, []);

  if (didMount) {
    // ... 返回仅客户端的 JSX ...
  }  else {
    // ... 返回初始 JSX ...
  }
}
```

问题

Effect 在组件挂载时运行了两次

在开发环境下，如果开启严格模式，React 会在实际运行 setup 之前额外运行一次 setup 和 cleanup。

这是一个压力测试，用于验证 Effect 的逻辑是否正确实现。如果出现可见问题，则 cleanup 函数缺少某些逻辑。cleanup 函数应该停止或撤消 setup 函数所做的任何操作。一般来说，用户不应该能够区分 setup 被调用一次（如在生产环境中）和调用 setup → cleanup → setup 序列（如在开发环境中）

Effect 在每次重新渲染后都运行

首先，请检查是否忘记指定依赖项数组：

```
useEffect(() => {
  // ...
}); // 🚩 没有依赖项数组：每次重新渲染后重新运行！
```

Effect 函数一直在无限循环中运行

如果你的 Effect 函数一直在无限循环中运行，那么必须满足以下两个条件：

- 你的 Effect 函数更新了一些状态。
- 这些状态的改变导致了重新渲染，从而导致 Effect 函数依赖的状态发生改变。

即使组件没有卸载，cleanup 逻辑也会运行

cleanup 函数不仅在卸载期间运行，也在每个依赖项变更的重新渲染前运行。此外，在开发环境中

```
useEffect(() => {
  // 🔴 避免：cleanup 逻辑没有相应的 setup 逻辑
  return () => {
    doSomething();
  };
}, []);
```

useInsertionEffect

注意

`useInsertionEffect` 是为 CSS-in-JS 库的作者特意打造的。除非你正在使用 CSS-in-JS 库并且需要注入样式，否则你应该使用 `useEffect` 或者 `useLayoutEffect`。

`useInsertionEffect` 可以在布局副作用触发之前将元素插入到 DOM 中。

```
useInsertionEffect(setup, dependencies?)
```

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

// 在你的 CSS-in-JS 库中
function useCSS(rule) {
  useInsertionEffect(() => {
    // ... 在此注入 <style> 标签 ...
  });
  return rule;
}
```

```
// 在你的 CSS-in-JS 库中
let isInserted = new Set();
function useCSS(rule) {
  useInsertionEffect(() => {
    // 同前所述，我们不建议在运行时注入 <style> 标签。
    // 如果你必须这样做，那么应当在 useInsertionEffect 中进行。
    if (!isInserted.has(rule)) {
      isInserted.add(rule);
      document.head.appendChild(getStyleForRule(rule));
    }
  });
  return rule;
}

function Button() {
  const className = useCSS('...');
  return <div className={className} />;
}
```

useLayoutEffect

注意

`useLayoutEffect` 可能会影响性能。尽可能使用 `useEffect`。

`useLayoutEffect` 是 `useEffect` 的一个版本，在浏览器重新绘制屏幕之前触发。

```
useLayoutEffect(setup, dependencies?)
```

调用 `useLayoutEffect` 在浏览器重新绘制屏幕之前进行布局测量

```
import { useState, useRef, useLayoutEffect } from 'react';

function Tooltip() {
  const ref = useRef(null);
  const [tooltipHeight, setTooltipHeight] = useState(0);

  useLayoutEffect(() => {
    const { height } = ref.current.getBoundingClientRect();
    setTooltipHeight(height);
  }, []);
  // ...
```

参数

- `setup`：处理副作用的函数。setup 函数选择性返回一个*清理*（cleanup）函数。在将组件首次添加到 DOM 之前，React 将运行 setup 函数。在每次因为依赖项变更而重新渲染后，React 将首先使用旧值运行 cleanup 函数（如果你提供了该函数），然后使用新值运行 setup 函数。在组件从 DOM 中移除之前，React 将最后一次运行 cleanup 函数。
- **可选** `dependencies`：`setup` 代码中引用的所有响应式值的列表。响应式值包括 props、state 以及所有直接在组件内部声明的变量和函数。如果你的代码检查工具 配置了 React，那么它将验证每个响应式值都被正确地指定为一个依赖项。依赖项列表必须具有固定数量的项，并且必须像 `[dep1, dep2, dep3]` 这样内联编写。React 将使用 `Object.is` 来比较每个依赖项和它先前的值。如果省略此参数，则在每次重新渲染组件之后，将重新运行副作用函数。

useRef

它能帮助引用一个不需要渲染的值

```
const ref = useRef(initialValue)
```

用法

- 使用用ref引用一个值
- 通过ref操作Dom
- 避免重复创建ref的内容

在组件顶层调用 `useRef` 以声明一个 ref。

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

function MyComponent() {
  const intervalRef = useRef(0);
  const inputRef = useRef(null);
  // ...
```

返回值

`useRef` 返回一个只有一个属性的对象:

- `current`：初始值为传递的 `initialValue`。之后可以将其设置为其他值。如果将 ref 对象作为一个 JSX 节点的 `ref` 属性传递给 React，React 将为它设置 `current` 属性。

注意

- 可以修改 `ref.current` 属性。与 state 不同，它是可变的。然而，如果它持有一个用于渲染的对象（例如 state 的一部分），那么就不应该修改这个对象。
- 改变 `ref.current` 属性时，React 不会重新渲染组件。React 不知道它何时会发生改变，因为 ref 是一个普通的 JavaScript 对象。
- 除了 初始化 外不要在渲染期间写入或者读取 `ref.current`，否则会使组件行为变得不可预测。
- 在严格模式下，React 将会 **调用两次组件方法**，这是为了 帮助发现意外问题。但这只是开发模式下的行为，不会影响生产模式。每个 ref 对象都将会创建两次，但是其中一个版本将被丢弃。如果使用的是组件纯函数（也应当如此），那么这不会影响其行为。

使用用 ref 引用一个值

在组件顶层调用 `useRef` 声明一个或多个 ref。

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

function Stopwatch() {
  const intervalRef = useRef(0);
  // ...
```

`useRef` 返回一个具有单个 `current` 属性 的 ref 对象，并初始化为你提供的 初始值。

在后续的渲染中，`useRef` 将返回相同的对象。你可以改变它的 `current` 属性来存储信息，并在之后读取它。这会让人联想到 state，但是有一个重要的区别。

改变 ref 不会触发重新渲染。这意味着 ref 是存储一些不影响组件视图输出信息的完美选择。例如，如果需要存储一个 interval ID 并在以后检索它，那么可以将它存储在 ref 中。只需要手动改变它的 `current` 属性即可修改 ref 的值:

```
function handleStartClick() {

  const intervalId = setInterval(() => {

    // ...

  }, 1000);

  intervalRef.current = intervalId;

}
```

在之后，从 ref 中读取 interval ID 便可以 清除定时器:

```
function handleStopClick() {

  const intervalId = intervalRef.current;

  clearInterval(intervalId);

}
```

使用 ref 可以确保：

- 可以在重新渲染之间 **存储信息**（普通对象存储的值每次渲染都会重置）。
- 改变它 **不会触发重新渲染**（状态变量会触发重新渲染）。
- 对于组件的每个副本而言，**这些信息都是本地的**（外部变量则是共享的）。

useState

它允许你向组件添加一个 状态变量

```
const [state, setState] = useState(initialState);
```

用法

- 为组件添加状态
- 根据先前的state更新state
- 更新状态中的对象和数组
- 避免重复创建初始状态
- 使用key重置状态
- 存储前一次渲染信息

在组件的顶层调用 `useState` 来声明一个 状态变量。

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

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

参数

- `initialState`：你希望 state 初始化的值。它可以是任何类型的值，但对于函数有特殊的行为。在初始渲染后，此参数将被忽略。
- 如果传递函数作为 `initialState`，则它将被视为 **初始化函数**。它应该是纯函数，不应该接受任何参数，并且应该返回一个任何类型的值。当初始化组件时，React 将调用你的初始化函数，并将其返回值存储为初始状态。

返回

`useState` 返回一个由两个值组成的数组：

1. 当前的 state。在首次渲染时，它将与你传递的 `initialState` 相匹配。
2. `set` 函数，它可以让你将 state 更新为不同的值并触发重新渲染。

useMemo

它在每次重新渲染的时候能够缓存计算的结果。

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

用法

- 跳过代价昂贵的重新计算
- 跳过组件重新渲染
- 记忆另一个Hook依赖
- 记忆一个函数

在组件的顶层调用 `useMemo` 来缓存每次重新渲染都需要计算的结果。

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

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

参数

- `calculateValue`：要缓存计算值的函数。它应该是一个没有任何参数的纯函数，并且可以返回任意类型。React 将会在首次渲染时调用该函数；在之后的渲染中，如果 `dependencies` 没有发生变化，React 将直接返回相同值。否则，将会再次调用 `calculateValue` 并返回最新结果，然后缓存该结果以便下次重复使用。
- `dependencies`：所有在 `calculateValue` 函数中使用的响应式变量组成的数组。响应式变量包括 props、state 和所有你直接在组件中定义的变量和函数。如果你在代码检查工具中 配置了 React，它将会确保每一个响应式数据都被正确地定义为依赖项。依赖项数组的长度必须是固定的并且必须写成 `[dep1, dep2, dep3]` 这种形式。React 使用 `Object.is` 将每个依赖项与其之前的值进行比较。

返回值

在初次渲染时，`useMemo` 返回不带参数调用 `calculateValue` 的结果。

在接下来的渲染中，如果依赖项没有发生改变，它将返回上次缓存的值；否则将再次调用 `calculateValue`，并返回最新结果。

useDebugValue

可以让你在 React 开发工具 中为自定义 Hook 添加标签。

```
useDebugValue(value, format?)
```

参数

- `value`：你想在 React 开发工具中显示的值。可以是任何类型。
- **可选** `format`：它接受一个格式化函数。当组件被检查时，React 开发工具将用 `value` 作为参数来调用格式化函数，然后显示返回的格式化值（可以是任何类型）。如果不指定格式化函数，则会显示 `value`。

useDeferredValue

可以让你延迟更新 UI 的某些部分。

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

用法

- 在新内容加载期间显示旧内容
- 表明内容以过时
- 延迟渲染Ui的某些部分

在组件的顶层调用 `useDeferredValue` 来获取该值的延迟版本。

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

function SearchPage() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  // ...
}
```

参数

- `value`：你想延迟的值，可以是任何类型。

返回值

在组件的初始渲染期间，返回的延迟值将与你提供的值相同。但是在组件更新时，React 将会先尝试使用旧值进行重新渲染（因此它将返回旧值），然后再在后台使用新值进行另一个重新渲染（这时它将返回更新后的值）。

useId

生成传递给无障碍属性的唯一 ID。

```
const id = useId()
```

用法

- 为无障碍属性生成唯一ID
- 为多个相关元素生成ID
- 为所有生成的ID指定共享前缀

useImpertiveHandle

它能让你自定义由ref暴露出来的句柄

```
useImperativeHandle(ref, createHandle, dependencies?)
```

用法

- 向父组件暴露一个自定义的ref句柄
- 暴露你自己的命令式方法

在组件顶层通过调用 `useImperativeHandle` 来自定义 ref 暴露出来的句柄：

```
import { forwardRef, useImperativeHandle } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
  useImperativeHandle(ref, () => {
    return {
      // ... 你的方法 ...
    };
  }, []);
```

参数

- ref:该ref是你从 `forwardRef` 渲染函数中获得的第二个参数
- `createHandle`：该函数无需参数，它返回你想要暴露的 ref 的句柄。该句柄可以包含任何类型。通常，你会返回一个包含你想暴露的方法的对象。
- **可选的** `dependencies`：函数 `createHandle` 代码中所用到的所有反应式的值的列表。反应式的值包含 props、状态和其他所有直接在你组件体内声明的变量和函数。倘若你的代码检查器已 为 React 配置好，它会验证每一个反应式的值是否被正确指定为依赖项。该列表的长度必须是一个常数项，并且必须按照 `[dep1, dep2, dep3]` 的形式罗列各依赖项。React 会使用 `Object.is` 来比较每一个依赖项与其对应的之前值。如果一次重新渲染导致某些依赖项发生了改变，或你没有提供这个参数列表，你的函数 `createHandle` 将会被重新执行，而新生成的句柄则会被分配给 ref。

返回值

`useImperativeHandle` 返回 `undefined`。

向父组件暴露一个自定义的 ref 句柄

默认情况下，组件不会将它们的 DOM 节点暴露给父组件。举例来说，如果你想要 `MyInput` 的父组件 能访问到 `<input>` DOM 节点，你必须选择使用 `forwardRef`。

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

const MyInput = forwardRef(function MyInput(props, ref) {
  return <input {...props} ref={ref} />;
});
```

在上方的代码中，MyInput 的 ref 会接收到 DOM 节点然而，你可以选择暴露一个自定义的值。为了修改被暴露的句柄，在你的顶层组件调用 `useImperativeHandle`：

```
import { forwardRef, useImperativeHandle } from 'react';
const MyInput = forwardRef(function MyInput(props, ref) {
  useImperativeHandle(ref, () => {
    return {
      // ... 你的方法 ...
    };
  }, []);

  return <input {...props} />;
});
```

注意在上述代码中，该 `ref` 已不再被转发到 `<input>` 中。

举例来说，假设你不想暴露出整个 `<input>` DOM 节点，但你想要它其中两个方法：`focus` 和 `scrollIntoView`。为此，用单独额外的 ref 来指向真实的浏览器 DOM。然后使用 `useImperativeHandle` 来暴露一个句柄，它只返回你想要父组件去调用的方法：

```
import { forwardRef, useRef, useImperativeHandle } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => {
    return {
      focus() {
        inputRef.current.focus();
      },
      scrollIntoView() {
        inputRef.current.scrollIntoView();
      },
    };
  }, []);

  return <input {...props} ref={inputRef} />;
});
```

现在，如果你的父组件获得了 `MyInput` 的 ref，就能通过该 ref 来调用 `focus` 和 `scrollIntoView` 方法。然而，它的访问是受限的，无法读取或调用下方 `<input>` DOM 节点的其他所有属性和方法。

useSyncExternalStore

`useSuncExternalStore` 是一个让你订阅外部stroe的React Hook

```
const snapshot = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)
```

用法

- 订阅外部store
- 订阅浏览器API
- 把逻辑抽取到自定义Hook
- 添加服务端渲染支持

在组件顶层调用 `useSyncExternalStore` 以从外部 store 读取值。

```
import { useSyncExternalStore } from 'react';
import { todosStore } from './todoStore.js';

function TodosApp() {
  const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);
  // ...
}
```

它返回store中数据的快照。你需要传两个函数作为参数

1. `subscribe` 函数应当订阅store并返回一个取消订阅的函数。
2. getSnapshot 函数应当从该store读取快照

参数

- `subscribe`：一个函数，接收一个单独的 `callback` 参数并把它订阅到 store 上。当 store 发生改变，它应当调用被提供的 `callback`。这会导致组件重新渲染。`subscribe` 函数会返回清除订阅的函数。
- `getSnapshot`：一个函数，返回组件需要的 store 中的数据快照。在 store 不变的情况下，重复调用 `getSnapshot` 必须返回同一个值。如果 store 改变，并且返回值也不同了（用 `Object.is` 比较），React 就会重新渲染组件。
- **可选** `getServerSnapshot`：一个函数，返回 store 中数据的初始快照。它只会在服务端渲染时，以及在客户端进行服务端渲染内容的 hydration 时被用到。快照在服务端与客户端之间必须相同，它通常是从服务端序列化并传到客户端的。如果你忽略此参数，在服务端渲染这个组件会抛出一个错误。

返回值

该 store 的当前快照，可以在你的渲染逻辑中使用。

useTransition

`useTransition` 是一个帮助你在不不阻塞 UI 的情况下更新状态的 React Hook。

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

用法

- 将状态更新标记为非阻塞的transition
- 在transition中更新父组件
- 在transition期间显示待处理的视觉状态
- 避免不必要的加载指示器
- 构建一个Suspense-enabled的路由

在组件顶层调用 `useTransition`，将某些状态更新标记为 transition。

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

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

返回值

`useTransition` 返回一个由两个元素组成的数组：

1. `isPending`，告诉你是否存在待处理的 transition。
2. `startTransition函数`,你可以使用此方法将状态更新为transition

`startTransition` 函数

`useTransition` 返回的 `startTransition` 函数允许你将状态更新标记为 transition。

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

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

参数

- 作用域（scope）：一个通过调用一个或多个 `set` 函数 更新状态的函数。React 会立即不带参数地调用此函数，并将在 `scope` 调用期间将所有同步安排的状态更新标记为 transition。它们将是非阻塞的，