useEffect 完整指南

2019年3月9日 • 🍱🍱🍱🍱 52 min read

你用Hooks写了一些组件，甚或写了一个小型应用。你可能很满意，使用它的API很舒服并且在这个过程中获得了一些小技巧。你甚至可能写了一些 custom Hooks去抽离重复的逻辑（精简掉了300行代码），并且得意地展示给你的同事看，“干得漂亮”，他们如是说。

但有时候当你使用`useEffect`你总觉得哪儿有点不对劲。你会嘀咕你可能遗漏了什么。它看起来像class的生命周期...但真的是这样吗？你发觉自己在问类似下面的这些问题：

- 🤔 如何用`useEffect`模拟`componentDidMount`生命周期？
- 🤔 如何正确地在`useEffect`里请求数据？`[]`又是什么？
- 🤔 我应该把函数当做effect的依赖吗？
- 🤔 为什么有时候会出现无限重复请求的问题？
- 🤔 为什么有时候在effect里拿到的是旧的state或prop？

当我刚开始使用Hooks的时候，我也同样被上面这些问题所困扰。甚至当我写最初的文档时，我也并没有扎实地掌握某些细节。我经历了一些“啊哈”的开窍时刻，我想把这些分享给你。这篇文章会深入讲解帮你明白上面问题的答案。

在看答案之前，我们需要先往后退一步。这篇文章的目的不是给你一个要点清单，而是想帮你真正地领会`useEffect`。其实我们并没有太多需要学习的，事实上，我们会花很多时间试图忘记某些已经习得的概念（unlearning）。

当我不再透过熟悉的**class**生命周期方法去窥视`useEffect`这个**Hook**的时候，我才得以融会贯通。

> “忘记你已经学到的。” — *Yoda*

这篇文章会假设你对`useEffect` API有一定程度的了解。

这篇文章真的很长。它更像一本**mini**书，这也是我更喜欢的形式。如果你很匆忙或者并不是太关心本文主题的话，你也可以直接看下面的摘要。

如果你对于深入研究感觉不是很适应的话，你或许可以等下面这些解释出现在其他文章中再去了解也行。就像**2013**年**React**刚出世的时候，大家需要时间去理解消化一种不同的心智模型。知识也需要时间去普及。

摘要

如果你不想阅读整篇文章，可以快速浏览这份摘要。要是某些部分不容易理解，你可以往下滚动寻找相关的内容去阅读。

如果你打算阅读整篇文章，你完全可以跳过这部分。我会在文章末尾带上摘要的链接。

🤔 **Question:** 如何用`useEffect`模拟`componentDidMount`生命周期？

虽然可以使用`useEffect(fn, [])`，但它们并不完全相等。和`componentDidMount`不一样，`useEffect`会捕获 props和state。所以即便在回调函数里，你拿到的还是初始的props和state。如果你想得到“最新”的值，你可以使用ref。不过，通常会有更简单的实现方式，所以你并不一定要用ref。记住，effects的心智模型和`componentDidMount`以及其

他生命周期是不同的，试图找到它们之间完全一致的表达反而更容易使你混淆。想要更有效，你需要“think in effects”，它的心智模型更接近于实现状态同步，而不是响应生命周期事件。

🤔 **Question:** 如何正确地在`useEffect`里请求数据？`[]`又是什么？

这篇文章是很好的入门，介绍了如何在`useEffect`里做数据请求。请务必读完它！它没有我的这篇这么长。`[]`表示effect没有使用任何React数据流里的值，因此该effect仅被调用一次是安全的。`[]`同样也是一类常见问题的来源，也即你以为没使用数据流里的值但其实使用了。你需要学习一些策略（主要是`useReducer`和`useCallback`）来移除这些effect依赖，而不是错误地忽略它们。

🤔 **Question:** 我应该把函数当做**effect**的依赖吗？

一般建议把不依赖props和state的函数提到你的组件外面，并且把那些仅被effect使用的函数放到effect里面。如果这样做了以后，你的effect还是需要用到组件内的函数（包括通过props传进来的函数），可以在定义它们的地方用`useCallback`包一层。为什么要这样做呢？因为这些函数可以访问到props和state，因此它们会参与到数据流中。我们官网的FAQ有更详细的答案。

🤔 **Question:** 为什么有时候会出现无限重复请求的问题？

这个通常发生于你在effect里做数据请求并且没有设置effect依赖参数的情况。没有设置依赖，effect会在每次渲染后执行一次，然后在effect中更新了状态引起渲染并再次触发effect。无限循环的发生也可能是因为你设置的依赖总是会改变。你可以通过一个一个移除的方式排查出哪个依赖导致了问题。但是，移除你使用的依赖（或者盲目地使用`[]`）通常是一种错误的解决方式。你应该做的是解决问题的根源。举个例子，函数可能会导致这个问题，你可以把它们放到effect里，或者提到组件外面，或者用`useCallback`包一层。`useMemo`可以做类似的事情以避免重复生成对象。

🤔 为什么有时候在**effect**里拿到的是旧的**state**或**prop**呢？

Effect拿到的总是定义它的那次渲染中的props和state。这能够避免一些bugs，但在一些场景中又会有些讨人嫌。对于这些场景，你可以明确地使用可变的ref保存一些值（上面文章的末尾解释了这一点）。如果你觉得在渲染中拿到了一些旧的props和state，且不是你想要的，你很可能遗漏了一些依赖。可以尝试使用这个lint 规则来训练你发现这些依赖。可

能没过几天，这种能力会变得像是你的第二天性。同样可以看我们官网FAQ中的这个回答。

我希望这个摘要对你有所帮助！要不，我们开始正文。

每一次渲染都有它自己的 Props and State

在我们讨论effects之前，我们需要先讨论一下渲染（rendering）。

我们来看一个计数器组件Counter，注意高亮的那一行：

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

高亮的代码究竟是什么意思呢？`count` 会“监听”状态的变化并自动更新吗？这么想可能是学习React的时候有用的第一直觉，但它并不是精确的心智模型。

上面例子中，`count` 仅是一个数字而已。它不是神奇的“data binding”, “watcher”, “proxy”，或者其他任何东西。它就是一个普通的数字像下面这个一样：

```
const count = 42;
// ...
<p>You clicked {count} times</p>
// ...
```

我们的组件第一次渲染的时候，从`useState()`拿到`count`的初始值`0`。当我们调用`setCount(1)`，React会再次渲染组件，这一次`count`是`1`。如此等等：

```
// During first render
function Counter() {
  const count = 0; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>
  // ...
}

// After a click, our function is called again
function Counter() {
  const count = 1; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>
  // ...
}

// After another click, our function is called again
function Counter() {
  const count = 2; // Returned by useState()
  // ...
  <p>You clicked {count} times</p>
  // ...
}
```

当我们更新状态的时候，**React**会重新渲染组件。每一次渲染都能拿到独立的`count`状态，这个状态值是函数中的一个常量。

所以下面的这行代码没有做任何特殊的数据绑定：

```
<p>You clicked {count} times</p>
```

它仅仅只是在渲染输出中插入了**count**这个数字。这个数字由React提供。当`setCount`的时候，React会带着一个不同的`count`值再次调用组件。然后，React会更新DOM以保持和渲染输出一致。

这里关键的点在于任意一次渲染中的`count`常量都不会随着时间改变。渲染输出会变是因为我们的组件被一次次调用，而每一次调用引起的渲染中，它包含的`count`值独立于其他

渲染。

（关于这个过程更深入的探讨可以查看我的另一篇文章 *React as a UI Runtime*。）

每一次渲染都有它自己的事件处理函数

到目前为止一切都还好。那么事件处理函数呢？

看下面的这个例子。它在三秒后会alert点击次数`count`：

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

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={handleAlertClick}>
        Show alert
      </button>
    </div>
  );
}
```

如果我按照下面的步骤去操作：

- 点击增加counter到3
- 点击一下 “Show alert”
- 点击增加 counter到5并且在定时器回调触发前完成

你猜alert会弹出什么呢？会是5吗？— 这个值是alert的时候counter的实时状态。或者会是3吗？— 这个值是我点击时候的状态。

剧透预警

来自己试试吧！

如果结果和你预料不一样，你可以想象一个更实际的例子：一个聊天应用在state中保存了当前接收者的ID，以及一个发送按钮。这篇文章深入探索了个中缘由。正确的答案就是3。

alert会“捕获”我点击按钮时候的状态。

（虽然有其他办法可以实现不同的行为，但现在我会专注于这个默认的场景。当我们在构建一种心智模型的时候，在可选的策略中分辨出“最小阻力路径”是非常重要的。）

但它究竟是如何工作的呢？

我们发现`count`在每一次函数调用中都是一个常量值。值得强调的是 — 我们的组件函数每次渲染都会被调用，但是每一次调用中`count`值都是常量，并且它被赋予了当前渲染中的状态值。

这并不是React特有的，普通的函数也有类似的行为：

```
function sayHi(person) {
  const name = person.name;
```

```
  setTimeout(() => {
    alert('Hello, ' + name);
  }, 3000);
}

let someone = {name: 'Dan'};
sayHi(someone);

someone = {name: 'Yuzhi'};
sayHi(someone);

someone = {name: 'Dominic'};
sayHi(someone);
```

在这个例子中，外层的`someone`会被赋值很多次（就像在React中，当前的组件状态会改变一样）。然后，在`sayHi`函数中，局部常量`name`会和某次调用中的`person`关联。因为这个常量是局部的，所以每一次调用都是相互独立的。结果就是，当定时器回调触发的时候，每一个alert都会弹出它拥有的`name`。

这就解释了我们的事件处理函数如何捕获了点击时候的`count`值。如果我们应用相同的替换原理，每一次渲染“看到”的是它自己的`count`：

```
// During first render
function Counter() {
  const count = 0; // Returned by useState()
  // ...
  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }
  // ...
}

// After a click, our function is called again
function Counter() {
  const count = 1; // Returned by useState()
  // ...
  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }
  // ...
}
```

```
// After another click, our function is called again
function Counter() {
  const count = 2; // Returned by useState()
  // ...
  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }
  // ...
}
```

所以实际上，每一次渲染都有一个“新版本”的`handleAlertClick`。每一个版本的`handleAlertClick`“记住”了它自己的`count`：

```
// During first render
function Counter() {
  // ...
  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + 0);
    }, 3000);
  }
  // ...
  <button onClick={handleAlertClick} /> // The one with 0 inside
  // ...
}

// After a click, our function is called again
function Counter() {
  // ...
  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + 1);
    }, 3000);
  }
  // ...
  <button onClick={handleAlertClick} /> // The one with 1 inside
  // ...
}

// After another click, our function is called again
function Counter() {
  // ...
  function handleAlertClick() {
    setTimeout(() => {
```

```
      alert('You clicked on: ' + 2);
    }, 3000);
  }
  // ...
  <button onClick={handleAlertClick} /> // The one with 2 inside
  // ...
}
```

这就是为什么在这个demo中中，事件处理函数“属于”某一次特定的渲染，当你点击的时候，它会使用那次渲染中`counter`的状态值。

在任意一次渲染中，**props**和**state**是始终保持不变的。如果props和state在不同的渲染中是相互独立的，那么使用到它们的任何值也是独立的（包括事件处理函数）。它们都“属于”一次特定的渲染。即便是事件处理中的异步函数调用“看到”的也是这次渲染中的`count`值。

备注：上面我将具体的`count`值直接内联到了`handleAlertClick`函数中。这种心智上的替换是安全的因为`count`值在某次特定渲染中不可能被改变。它被声明成了一个常量并且是一个数字。这样去思考其他类型的值比如对象也同样是安全的，当然需要在我们都同意应该避免直接修改*state*这个前提下。通过调用`setSomething(newObj)`的方式去生成一个新的对象而不是直接修改它是更好的选择，因为这样能保证之前渲染中的*state*不会被污染。

每次渲染都有它自己的Effects

这篇文章是关于effects的，但目前我们居然还没有讨论effects！ 言归正传，由上面的分析得出一个结果，effects其实并没有什么两样。

让我们回到官网文档中的这个例子：

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
```

```
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

抛一个问题给你：**effect**是如何读取到最新的`count`状态值的呢？

也许，是某种“data binding”或“watching”机制使得`count`能够在effect函数内更新？也或许`count`是一个可变的值，React会在我们组件内部修改它以使我们的effect函数总能拿到最新的值？

都不是。

我们已经知道`count`是某个特定渲染中的常量。事件处理函数“看到”的是属于它那次特定渲染中的`count`状态值。对于effects也同样如此：

并不是`count`的值在“不变”的**effect**中发生了改变，而是***effect***函数本身在每一次渲染中都不相同。

每一个effect版本“看到”的`count`值都来自于它属于的那次渲染：

```
// During first render
function Counter() {
  // ...
  useEffect(
    // Effect function from first render
    () => {
      document.title = `You clicked ${0} times`;
    }
  );
  // ...
}

// After a click, our function is called again
function Counter() {
  // ...
  useEffect(
    // Effect function from second render
    () => {
```

```
      document.title = `You clicked ${1} times`;
    }
  );
  // ...
}

// After another click, our function is called again
function Counter() {
  // ...
  useEffect(
    // Effect function from third render
    () => {
      document.title = `You clicked ${2} times`;
    }
  );
  // ..
}
```

React会记住你提供的effect函数，并且会在每次更改作用于DOM并让浏览器绘制屏幕后去调用它。

所以虽然我们说的是一个 *effect*（这里指更新document的title），但其实每次渲染都是一个不同的函数 — 并且每个effect函数“看到”的props和state都来自于它属于的那次特定渲染。

概念上，你可以想象**effects**是渲染结果的一部分。

严格地说，它们并不是（为了允许Hook的组合并且不引入笨拙的语法或者运行时）。但是在我们构建的心智模型上，effect函数属于某个特定的渲染，就像事件处理函数一样。

为了确保我们已经有了扎实的理解，我们再回顾一下第一次的渲染过程：

- **React:** 给我状态为 `0` 时候的UI。
- 你的组件：
  - 给你需要渲染的内容：`<p>You clicked 0 times</p>`。
  - 记得在渲染完了之后调用这个effect：`() => { document.title = 'You clicked 0 times' }`。
- **React:** 没问题。开始更新UI，喂浏览器，我要给DOM添加一些东西。

- 浏览器：酷，我已经把它绘制到屏幕上了。
- **React:** 好的， 我现在开始运行给我的effect
  - 运行 `() => { document.title = 'You clicked 0 times' }`。

现在我们回顾一下我们点击之后发生了什么：

- 你的组件：喂 React, 把我的状态设置为`1`。
- **React:** 给我状态为 `1`时候的UI。
- 你的组件：
  - 给你需要渲染的内容: `<p>You clicked 1 times</p>`。
  - 记得在渲染完了之后调用这个effect： `() => { document.title = 'You clicked 1 times' }`。
- **React:** 没问题。开始更新UI，喂浏览器，我修改了DOM。
- **Browser:** 酷，我已经将更改绘制到屏幕上了。
- **React:** 好的， 我现在开始运行属于这次渲染的effect
  - 运行 `() => { document.title = 'You clicked 1 times' }`。

每一次渲染都有它自己的...所有

我们现在知道**effects**会在每次渲染后运行，并且概念上它是组件输出的一部分，可以“看到”属于某次特定渲染的**props**和**state**。

我们来做一个思想实验，思考下面的代码：

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

  useEffect(() => {
    setTimeout(() => {
      console.log(`You clicked ${count} times`);
```

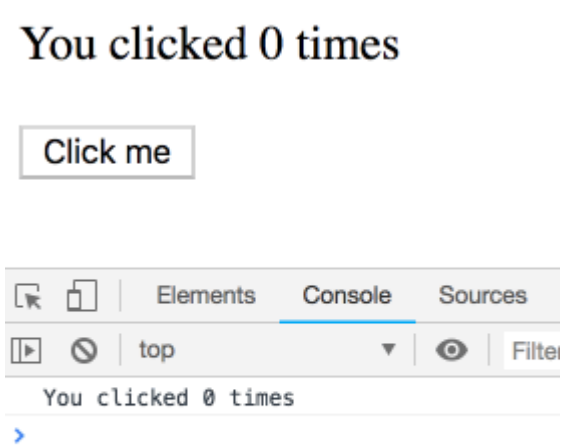
```
    }, 3000);
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

如果我点击了很多次并且在effect里设置了延时，打印出来的结果会是什么呢？

剧透预警

你可能会认为这是一个很绕的题并且结果是反直觉的。完全错了！我们看到的就是顺序的打印输出 — 每一个都属于某次特定的渲染，因此有它该有的`count`值。你可以自己试一试：

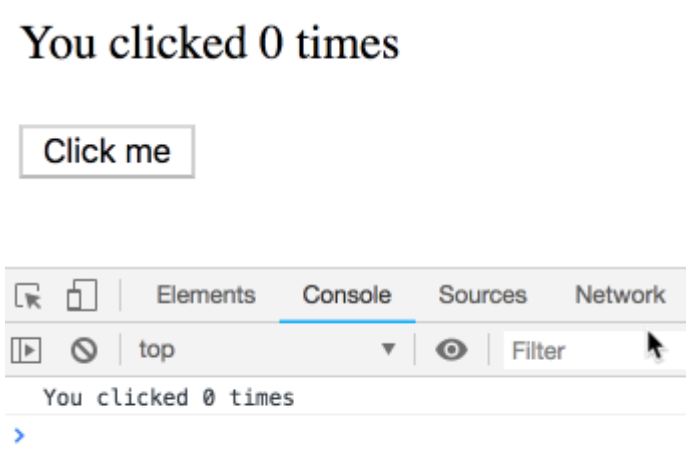


你可能会想：“它当然应该是这样的。否则还会怎么样呢？”

不过，class中的`this.state`并不是这样运作的。你可能会想当然以为下面的class 实现和上面是相等的：

```
componentDidUpdate() {
  setTimeout(() => {
    console.log(`You clicked ${this.state.count} times`);
  }, 3000);
}
```

然而，`this.state.count`总是指向最新的count值，而不是属于某次特定渲染的值。所以你会看到每次打印输出都是`5`：



我觉得Hooks这么依赖Javascript闭包是挺讽刺的一件事。有时候组件的class实现方式会受闭包相关的苦（the canonical wrong-value-in-a-timeout confusion），但其实这个例子中真正的混乱来源是可变数据（React 修改了class中的`this.state`使其指向最新状态），并不是闭包本身的错。

当封闭的值始终不会变的情况下闭包是非常棒的。这使它们非常容易思考因为你本质上在引用常量。正如我们所讨论的，props和state在某个特定渲染中是不会改变的。顺便说一下，我们可以使用闭包修复上面的class版本...

逆潮而动

到目前为止，我们可以明确地喊出下面重要的事实：每一个组件内的函数（包括事件处理函数，effects，定时器或者API调用等等）会捕获某次渲染中定义的props和state。

所以下面的两个例子是相等的：

```
function Example(props) {
  useEffect(() => {
    setTimeout(() => {
      console.log(props.counter);
    }, 1000);
  });
  // ...
}
```

```
function Example(props) {
  const counter = props.counter;
  useEffect(() => {
    setTimeout(() => {
      console.log(counter);
    }, 1000);
  });
  // ...
}
```

在组件内什么时候去读取**props**或者**state**是无关紧要的。因为它们不会改变。在单次渲染的范围内，props和state始终保持不变。（解构赋值的props使得这一点更明显。）

当然，有时候你可能想在effect的回调函数里读取最新的值而不是捕获的值。最简单的实现方法是使用refs，这篇文章的最后一部分介绍了相关内容。

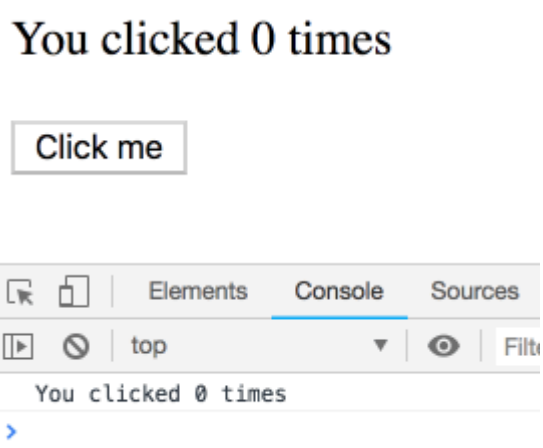
需要注意的是当你想要从过去渲染中的函数里读取未来的props和state，你是在逆潮而动。虽然它并没有错（有时候可能也需要这样做），但它因为打破了默认范式会使代码显得不够“干净”。这是我们有意为之的，因为它能帮助突出哪些代码是脆弱的，是需要依赖时间次序的。在class中，如果发生这种情况就没那么显而易见了。

下面这个计数器版本 模拟了class中的行为：

```
```

```
function Example() {
  const [count, setCount] = useState(0);
  const latestCount = useRef(count);

  useEffect(() => {
    // Set the mutable latest value
    latestCount.current = count;
    setTimeout(() => {
      // Read the mutable latest value
      console.log(`You clicked ${latestCount.current} times`);
    }, 3000);
  });
  // ...
```



在React中去直接修改值看上去有点怪异。然而，在class组件中React正是这样去修改`this.state`的。不像捕获的props和state，你没法保证在任意一个回调函数中读取的`latestCount.current`是不变的。根据定义，你可以随时修改它。这就是为什么它不是默认行为，而是需要你主动选择这样做。

那Effect中的清理又是怎样的呢？

像 文档中解释的, 有些 effects 可能需要有一个清理步骤。本质上，它的目的是消除副作用（effect)，比如取消订阅。

思考下面的代码:

```
useEffect(() => {
  ChatAPI.subscribeToFriendStatus(props.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.id, handleStatusChange);
  };
});
```

假设第一次渲染的时候`props`是`{id: 10}`，第二次渲染的时候是`{id: 20}`。你可能会认为发生了下面的这些事：

- React 清除了`{id: 10}`的effect。
- React 渲染`{id: 20}`的UI。
- React 运行`{id: 20}`的effect。

(事实并不是这样。)

如果依赖这种心智模型，你可能会认为清除过程“看到”的是旧的props因为它是在重新渲染之前运行的，新的effect“看到”的是新的props因为它是在重新渲染之后运行的。这种心智模型直接来源于class组件的生命周期。不过它并不精确。让我们来一探究竟。

React只会在浏览器绘制后运行effects。这使得你的应用更流畅因为大多数effects并不会阻塞屏幕的更新。Effect的清除同样被延迟了。上一次的**effect**会在重新渲染后被清除：

- **React** 渲染`{id: 20}`的**UI**。
- 浏览器绘制。我们在屏幕上看到`{id: 20}`的UI。
- **React** 清除`{id: 10}`的**effect**。
- React 运行`{id: 20}`的effect。

你可能会好奇：如果清除上一次的effect发生在props变成`{id: 20}`之后，那它为什么还能“看到”旧的`{id: 10}`？

你曾经来过这里... 😕

引用上半部分得到的结论:

> 组件内的每一个函数（包括事件处理函数，*effects*，定时器或者*API*调用等等）会捕获定义它们的那次渲染中的*props*和*state*。

现在答案显而易见。effect的清除并不会读取“最新”的props。它只能读取到定义它的那次渲染中的props值：

```
// First render, props are {id: 10}
function Example() {
  // ...
  useEffect(
    // Effect from first render
    () => {
      ChatAPI.subscribeToFriendStatus(10, handleStatusChange);
      // Cleanup for effect from first render
      return () => {
        ChatAPI.unsubscribeFromFriendStatus(10, handleStatusChange);
      };
    }
  );
  // ...
}

// Next render, props are {id: 20}
function Example() {
  // ...
  useEffect(
    // Effect from second render
    () => {
      ChatAPI.subscribeToFriendStatus(20, handleStatusChange);
      // Cleanup for effect from second render
      return () => {
        ChatAPI.unsubscribeFromFriendStatus(20, handleStatusChange);
      };
    }
  );
```

```
  // ...
}
```

王国会崛起转而复归尘土，太阳会脱落外层变为白矮星，最后的文明也迟早会结束。但是第一次渲染中effect的清除函数只能看到`{id: 10}`这个props。

这正是为什么React能做到在绘制后立即处理effects — 并且默认情况下使你的应用运行更流畅。如果你的代码需要依然可以访问到老的props。

同步， 而非生命周期

我最喜欢React的一点是它统一描述了初始渲染和之后的更新。这降低了你程序的熵。

比如我有个组件像下面这样：

```
function Greeting({ name }) {
  return (
    <h1 className="Greeting">
      Hello, {name}
    </h1>
  );
}
```

我先渲染`<Greeting name="Dan" />`然后渲染`<Greeting name="Yuzhi" />`，和我直接渲染`<Greeting name="Yuzhi" />`并没有什么区别。在这两种情况中，我最后看到的都是“Hello, Yuzhi”。

人们总是说：“重要的是旅行过程，而不是目的地”。在React世界中，恰好相反。重要的是目的，而不是过程。这就是JQuery代码中`$.addClass`或`$.removeClass`这样的调用（过程）和React代码中声明CSS类名应该是什么（目的）之间的区别。

React会根据我们当前的**props**和**state**同步到**DOM**。“mount”和“update”之于渲染并没有什么区别。

你应该以相同的方式去思考effects。`useEffect`使你能够根据**props**和**state**同步**React tree**之外的东西。

```
function Greeting({ name }) {
  useEffect(() => {
    document.title = 'Hello, ' + name;
  });
  return (
    <h1 className="Greeting">
      Hello, {name}
    </h1>
  );
}
```

这就是和大家熟知的*mount/update/unmount*心智模型之间细微的区别。理解和内化这种区别是非常重要的。如果你试图写一个**effect**会根据是否第一次渲染而表现不一致，你正在逆潮而动。如果我们的结果依赖于过程而不是目的，我们会在同步中犯错。

先渲染属性A，B再渲染C，和立即渲染C并没有什么区别。虽然他们可能短暂地会有点不同（比如请求数据时），但最终的结果是一样的。

不过话说回来，在每一次渲染后都去运行所有的effects可能并不高效。（并且在某些场景下，它可能会导致无限循环。）

所以我们该怎么解决这个问题？

告诉React去比对你的Effects

其实我们已经从React处理DOM的方式中学习到了解决办法。React只会更新DOM真正发生改变的部分，而不是每次渲染都大动干戈。

当你把

```
<h1 className="Greeting">
  Hello, Dan
</h1>
```

更新到

```
```

```
<h1 className="Greeting">
  Hello, Yuzhi
</h1>
```

React 能够看到两个对象：

```
const oldProps = {className: 'Greeting', children: 'Hello, Dan'};
const newProps = {className: 'Greeting', children: 'Hello, Yuzhi'};
```

它会检测每一个props，并且发现`children`发生改变需要更新DOM，但`className`并没有。所以它只需要这样做：

```
domNode.innerText = 'Hello, Yuzhi';
// No need to touch domNode.className
```

我们也可以用类似的方式处理**effects**吗？如果能够在不需要的时候避免调用**effect**就太好了。

举个例子，我们的组件可能因为状态变更而重新渲染：

```
function Greeting({ name }) {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    document.title = 'Hello, ' + name;
  });

  return (
    <h1 className="Greeting">
      Hello, {name}
      <button onClick={() => setCounter(counter + 1)}>
        Increment
      </button>
    </h1>
  );
}
```

但是我们的effect并没有使用`counter`这个状态。我们的**effect**只会同步`name`属性给`document.title`，但`name`并没有变。在每一次counter改变后重新给`document.title`赋值并不是理想的做法。

好了，那React可以...区分effects的不同吗？

```
let oldEffect = () => { document.title = 'Hello, Dan'; };
let newEffect = () => { document.title = 'Hello, Dan'; };
// Can React see these functions do the same thing?
```

并不能。React并不能猜测到函数做了什么如果不先调用的话。（源码中并没有包含特殊的值，它仅仅是引用了`name`属性。）

这是为什么你如果想要避免effects不必要的重复调用，你可以提供给`useEffect`一个依赖数组参数(deps)：

```
  useEffect(() => {
    document.title = 'Hello, ' + name;
  }, [name]); // Our deps
```

这好比你告诉**React**：“**Hey**，我知道你看不到这个函数里的东西，但我可以保证只使用了渲染中的`name`，别无其他。”

如果当前渲染中的这些依赖项和上一次运行这个effect的时候值一样，因为没有什么需要同步React会自动跳过这次effect：

```
const oldEffect = () => { document.title = 'Hello, Dan'; };
const oldDeps = ['Dan'];

const newEffect = () => { document.title = 'Hello, Dan'; };
const newDeps = ['Dan'];

// React can't peek inside of functions, but it can compare deps.
// Since all deps are the same, it doesn’t need to run the new effect.
```

即使依赖数组中只有一个值在两次渲染中不一样，我们也不能跳过effect的运行。要同步所有！

关于依赖项不要对React撒谎

关于依赖项对React撒谎会有不好的结果。直觉上，这很好理解，但我曾看到几乎所有依赖class心智模型使用`useEffect`的人都试图违反这个规则。（我刚开始也这么干了！）

```
function SearchResults() {
  async function fetchData() {
    // ...
  }

  useEffect(() => {
    fetchData();
  }, []); // Is this okay? Not always -- and there's a better way to write it.

  // ...
}
```

(官网的 *Hooks FAQ* 解释了应该怎么做。 我们在下面会重新回顾这个例子。)

“但我只是想在挂载的时候运行它！”，你可能会说。现在只需要记住：如果你设置了依赖项，**effect**中用到的所有组件内的值都要包含在依赖中。这包括props，state，函数 — 组件内的任何东西。

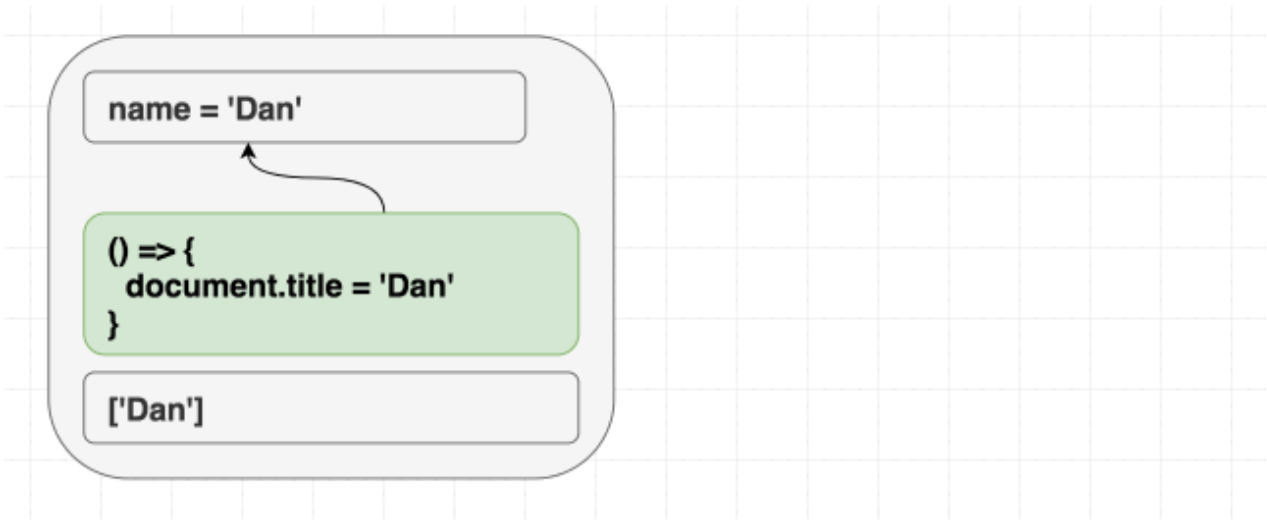
有时候你是这样做了，但可能会引起一个问题。比如，你可能会遇到无限请求的问题，或者socket被频繁创建的问题。解决问题的方法不是移除依赖项。我们会很快了解具体的解决方案。

不过在我们深入解决方案之前，我们先尝试更好地理解问题。

如果设置了错误的依赖会怎么样呢？

如果依赖项包含了所有effect中使用到的值，React就能知道何时需要运行它：

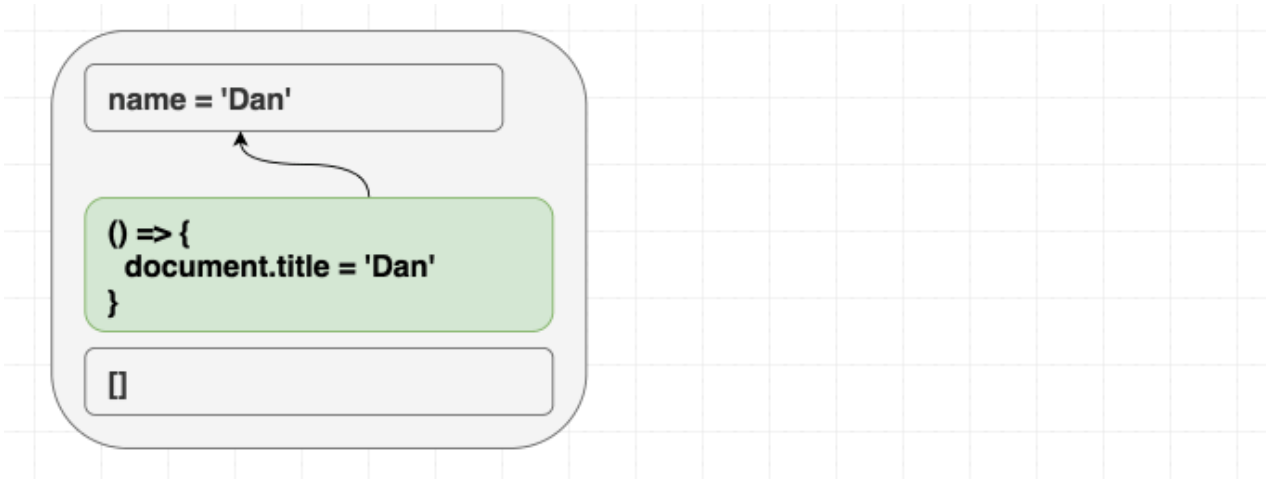
```
useEffect(() => {
  document.title = 'Hello, ' + name;
}, [name]);
```



(依赖发生了变更，所以会重新运行effect。)

但是如果我们将 `[]` 设为effect的依赖，新的effect函数不会运行：

```
useEffect(() => {
  document.title = 'Hello, ' + name;
}, []); // Wrong: name is missing in deps
```



(依赖没有变，所以不会再次运行effect。)

在这个例子中，问题看起来显而易见。但在某些情况下如果你脑子里“跳出”class组件的解决办法，你的直觉很可能会欺骗你。

举个例子，我们来写一个每秒递增的计数器。在Class组件中，我们的直觉是：“开启一次定时器，清除也是一次”。这里有一个例子说明怎么实现它。当我们理所当然地把它用`useEffect`的方式翻译，直觉上我们会设置依赖为`[]`。“我只想运行一次effect”，对吗？

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);

  return <h1>{count}</h1>;
}
```

然而，这个例子只会递增一次。天了噜。

如果你的心智模型是“只有当我想重新触发effect的时候才需要去设置依赖”，这个例子可能会让你产生存在危机。你想要触发一次因为它是定时器 — 但为什么会有问题？

如果你知道依赖是我们给React的暗示，告诉它effect所有需要使用的渲染中的值，你就不会吃惊了。effect中使用了`count`但我们撒谎说它没有依赖。如果我们这样做迟早会出幺蛾子。

在第一次渲染中，`count`是`0`。因此，`setCount(count + 1)`在第一次渲染中等价于`setCount(0 + 1)`。既然我们设置了`[]`依赖，**effect**不会再重新运行，它后面每一秒都会调用`setCount(0 + 1)`：

```
// First render, state is 0
function Counter() {
  // ...
  useEffect(
    // Effect from first render
    () => {
      const id = setInterval(() => {
        setCount(0 + 1); // Always setCount(1)
      }, 1000);
      return () => clearInterval(id);
    },
```

```
    [] // Never re-runs
  );
  // ...
}

// Every next render, state is 1
function Counter() {
  // ...
  useEffect(
    // This effect is always ignored because
    // we lied to React about empty deps.
    () => {
      const id = setInterval(() => {
        setCount(1 + 1);
      }, 1000);
      return () => clearInterval(id);
    },
    []
  );
  // ...
}
```

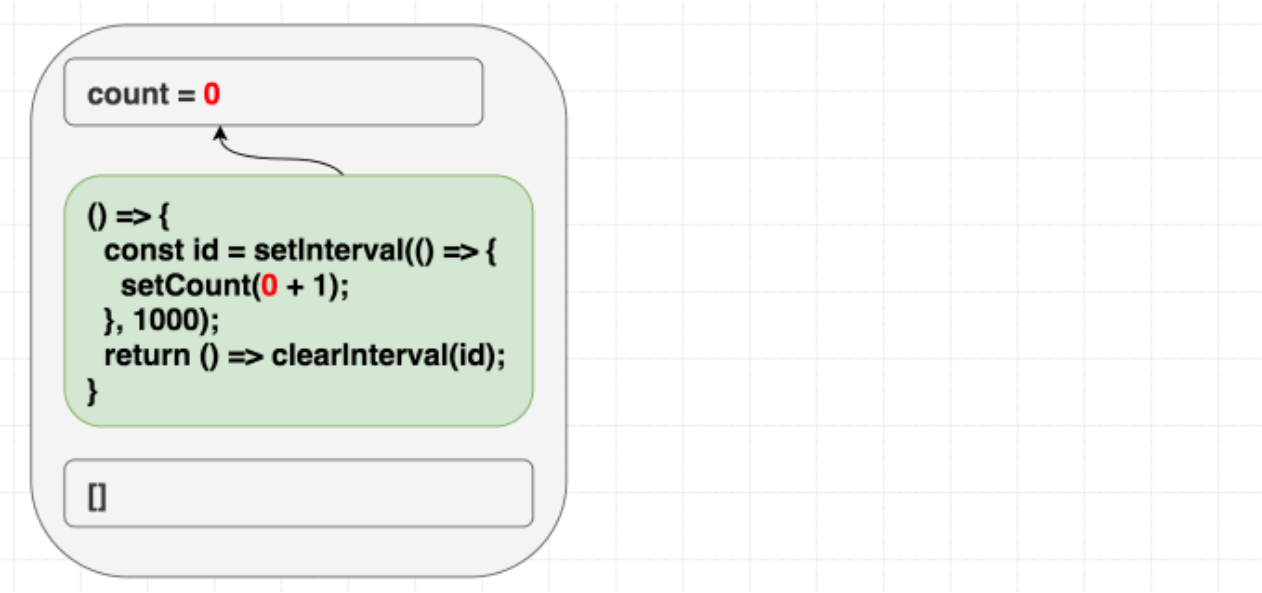
我们对React撒谎说我们的effect不依赖组件内的任何值，可实际上我们的effect有依赖！

我们的effect依赖`count` - 它是组件内的值（不过在effect外面定义）：

```
  const count = // ...

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
```

因此，设置`[]`为依赖会引入一个bug。React会对比依赖，并且跳过后面的effect：



(依赖没有变，所以不会再次运行effect。)

类似于这样的问题是很难被想到的。因此，我鼓励你将诚实地告知effect依赖作为一条硬性规则，并且要列出所以依赖。（我们提供了一个lint规则如果你想在你的团队内做硬性规定。）

两种诚实告知依赖的方法

有两种诚实告知依赖的策略。你应该从第一种开始，然后在需要的时候应用第二种。

第一种策略是在依赖中包含所有**effect**中用到的组件内的值。让我们在依赖中包含`count`：

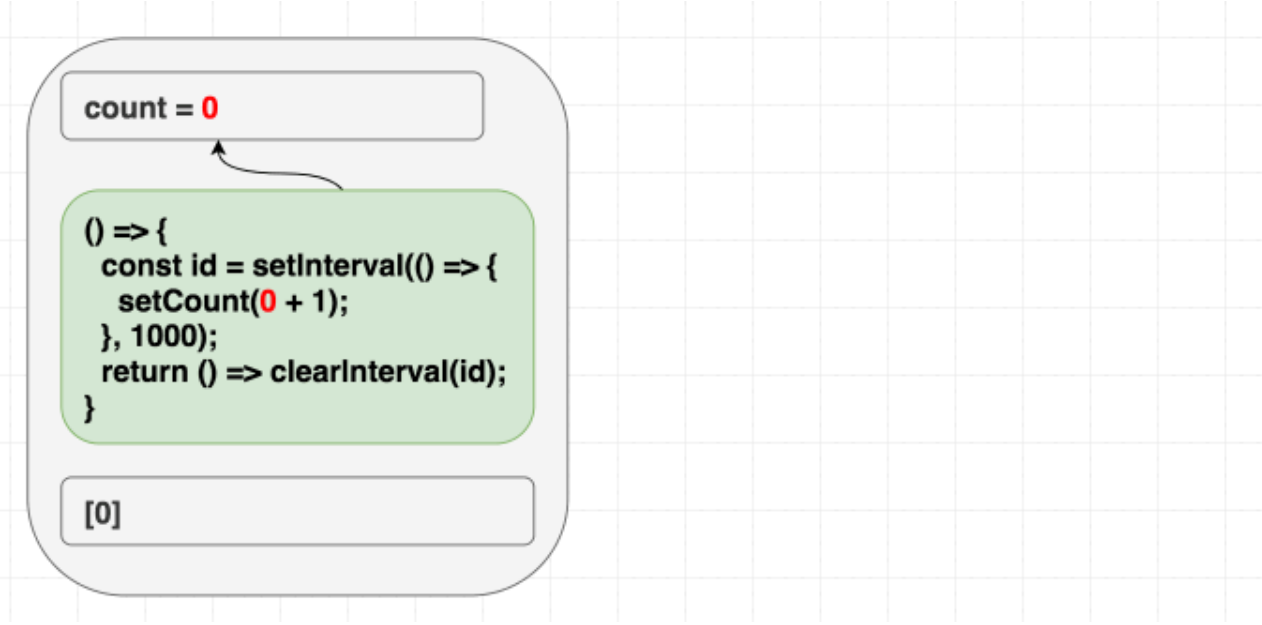
```
useEffect(() => {
  const id = setInterval(() => {
    setCount(count + 1);
  }, 1000);
  return () => clearInterval(id);
}, [count]);
```

现在依赖数组正确了。虽然它可能不是太理想但确实解决了上面的问题。现在，每次`count`修改都会重新运行effect，并且定时器中的`setCount(count + 1)`会正确引用某次渲染中的`count`值：

```
// First render, state is 0
function Counter() {
  // ...
  useEffect(
    // Effect from first render
    () => {
      const id = setInterval(() => {
        setCount(0 + 1); // setCount(count + 1)
      }, 1000);
      return () => clearInterval(id);
    },
    [0] // [count]
  );
  // ...
}

// Second render, state is 1
function Counter() {
  // ...
  useEffect(
    // Effect from second render
    () => {
      const id = setInterval(() => {
        setCount(1 + 1); // setCount(count + 1)
      }, 1000);
      return () => clearInterval(id);
    },
    [1] // [count]
  );
  // ...
}
```

这能解决问题但是我们的定时器会在每一次`count`改变后清除和重新设定。这应该不是我们想要的结果：



(依赖发生了变更，所以会重新运行effect。)

第二种策略是修改**effect**内部的代码以确保它包含的值只会在需要的时候发生变更。我们不想告知错误的依赖 - 我们只是修改effect使得依赖更少。

让我们来看一些移除依赖的常用技巧。

让Effects自给自足

我们想去掉effect的`count`依赖。

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(count + 1);
  }, 1000);
  return () => clearInterval(id);
}, [count]);
```

为了实现这个目的，我们需要问自己一个问题：我们为什么要用`count`？可以看到我们只在`setCount`调用中用到了`count`。在这个场景中，我们其实并不需要在effect中使用

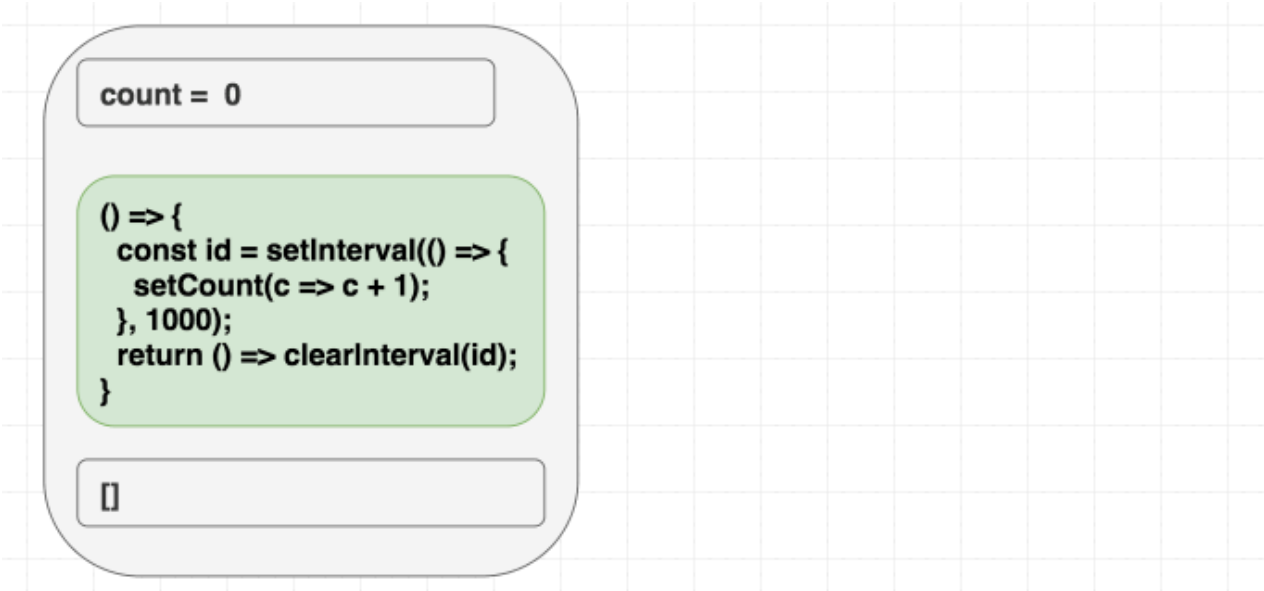
`count`。当我们想要根据前一个状态更新状态的时候，我们可以使用`setState`的函数形式：

```
useEffect(() => {
  const id = setInterval(() => {
    setCount(c => c + 1);
  }, 1000);
  return () => clearInterval(id);
}, []);
```

我喜欢把类似这种情况称为“错误的依赖”。是的，因为我们在effect中写了`setCount(count + 1)`所以`count`是一个必需的依赖。但是，我们真正想要的是把`count`转换为`count+1`，然后返回给React。可是React其实已经知道当前的`count`。我们需要告知**React**的仅仅是去递增状态 - 不管它现在具体是什么值。

这正是`setCount(c => c + 1)`做的事情。你可以认为它是在给React“发送指令”告知如何更新状态。这种“更新形式”在其他情况下也有帮助，比如你需要批量更新。

注意我们做到了移除依赖，并且没有撒谎。我们的**effect**不再读取渲染中的`count`值。



(依赖没有变，所以不会再次运行effect。)

你可以自己试试。

尽管effect只运行了一次，第一次渲染中的定时器回调函数可以完美地在每次触发的时候给React发送`c => c + 1`更新指令。它不再需要知道当前的`count`值。因为React已经知道了。

函数式更新 和 Google Docs

还记得我们说过同步才是理解effects的心智模型吗？同步的一个有趣地方在于你通常想要把同步的“信息”和状态解耦。举个例子，当你在Google Docs编辑文档的时候，Google并不会把整篇文章发送给服务器。那样做会非常低效。相反的，它只是把你的修改以一种形式发送给服务端。

虽然我们effect的情况不尽相同，但可以应用类似的思想。只在**effects**中传递最小的信息会很有帮助。类似于`setCount(c => c + 1)`这样的更新形式比`setCount(count + 1)`传递了更少的信息，因为它不再被当前的count值“污染”。它只是表达了一种行为（“递增”）。“Thinking in React”也讨论了如何找到最小状态。原则是类似的，只不过现在关注的是如何更新。

表达意图（而不是结果）和Google Docs如何处理共同编辑异曲同工。虽然这个类比略微延伸了一点，函数式更新在React中扮演了类似的角色。它们确保能以批量地和可预测的方式来处理各种源头（事件处理函数，effect中的订阅，等等）的状态更新。

然而，即使是`setCount(c => c + 1)`也并不完美。它看起来有点怪，并且非常受限于它能做的事。举个例子，如果我们有两个互相依赖的状态，或者我们想基于一个prop来计算下一次的state，它并不能做到。幸运的是，`setCount(c => c + 1)`有一个更强大的姐妹模式，它的名字叫`useReducer`。

解耦来自Actions的更新

我们来修改上面的例子让它包含两个状态：`count` 和 `step`。我们的定时器会每次在count上增加一个`step`值：

```
function Counter() {
  const [count, setCount] = useState(0);
  const [step, setStep] = useState(1);
```

```
  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + step);
    }, 1000);
    return () => clearInterval(id);
  }, [step]);

  return (
    <>
      <h1>{count}</h1>
      <input value={step} onChange={e => setStep(Number(e.target.value))} />
    </>
  );
}
```

(这里是demo.)

注意我们没有撒谎。既然我们在effect里使用了`step`，我们就把它加到依赖里。所以这也是为什么代码能运行正确。

这个例子目前的行为是修改`step`会重启定时器 - 因为它是依赖项之一。在大多数场景下，这正是你所需要的。清除上一次的effect然后重新运行新的effect并没有任何错。除非我们有很好的理由，我们不应该改变这个默认行为。

不过，假如我们不想在`step`改变后重启定时器，我们该如何从effect中移除对`step`的依赖呢？

当你想更新一个状态，并且这个状态更新依赖于另一个状态的值时，你可能需要用`useReducer`去替换它们。

当你写类似`setSomething(something => ...)`这种代码的时候，也许就是考虑使用reducer的契机。reducer可以让你把组件内发生了什么**(actions)**和状态如何响应并更新分开表述。

我们用一个`dispatch`依赖去替换effect的`step`依赖：

```
const [state, dispatch] = useReducer(reducer, initialState);
const { count, step } = state;

useEffect(() => {
  const id = setInterval(() => {
```

```
      dispatch({ type: 'tick' }); // Instead of setCount(c => c + step);
    }, 1000);
    return () => clearInterval(id);
  }, [dispatch]);
```

(查看 demo。)

你可能会问："这怎么就更好了？"答案是**React**会保证`dispatch`在组件的声明周期内保持不变。所以上面例子中不再需要重新订阅定时器。

我们解决了问题!

（你可以从依赖中去除`dispatch`，`setState`，和`useRef`包裹的值因为*React*会确保它们是静态的。不过你设置了它们作为依赖也没什么问题。）

相比于直接在effect里面读取状态，它dispatch了一个*action*来描述发生了什么。这使得我们的effect和`step`状态解耦。我们的effect不再关心怎么更新状态，它只负责告诉我们发生了什么。更新的逻辑全都交由reducer去统一处理:

```
const initialState = {
  count: 0,
  step: 1,
};

function reducer(state, action) {
  const { count, step } = state;
  if (action.type === 'tick') {
    return { count: count + step, step };
  } else if (action.type === 'step') {
    return { count, step: action.step };
  } else {
    throw new Error();
  }
}
```

(这里是demo 如果你之前错过了。)

为什么**useReducer**是**Hooks**的作弊模式

我们已经学习到如何移除effect的依赖，不管状态更新是依赖上一个状态还是依赖另一个状态。但假如我们需要依赖***props***去计算下一个状态呢？举个例子，也许我们的API是`<Counter step={1} />`。确定的是，在这种情况下，我们没法避免依赖`props.step`。是吗？

实际上， 我们可以避免！我们可以把*reducer*函数放到组件内去读取props：

```
function Counter({ step }) {
  const [count, dispatch] = useReducer(reducer, 0);

  function reducer(state, action) {
    if (action.type === 'tick') {
      return state + step;
    } else {
      throw new Error();
    }
  }

  useEffect(() => {
    const id = setInterval(() => {
      dispatch({ type: 'tick' });
    }, 1000);
    return () => clearInterval(id);
  }, [dispatch]);

  return <h1>{count}</h1>;
}
```

这种模式会使一些优化失效，所以你应该避免滥用它，不过如果你需要你完全可以在reducer里面访问props。（这里是demo。）

即使是在这个例子中，**React**也保证`dispatch`在每次渲染中都是一样的。 所以你可以在依赖中去掉它。它不会引起effect不必要的重复执行。

你可能会疑惑：这怎么可能？在之前渲染中调用的reducer怎么“知道”新的props？答案是当你`dispatch`的时候，React只是记住了action - 它会在下一次渲染中再次调用reducer。在那个时候，新的props就可以被访问到，而且reducer调用也不是在effect里。

这就是为什么我倾向认为`useReducer`是**Hooks**的“作弊模式”。它可以把更新逻辑和描述发生了什么分开。结果是，这可以帮助我移除不必需的依赖，避免不必要的**effect**调用。

把函数移到Effects里

一个典型的误解是认为函数不应该成为依赖。举个例子，下面的代码看上去可以运行正常：

```
function SearchResults() {
  const [data, setData] = useState({ hits: [] });

  async function fetchData() {
    const result = await axios(
      'https://hn.algolia.com/api/v1/search?query=react',
    );
    setData(result.data);
  }

  useEffect(() => {
    fetchData();
  }, []); // Is this okay?

  // ...
```

(这个例子改编自*Robin Wieruch*这篇很棒的文章 — 点击查看！)

需要明确的是，上面的代码可以正常工作。但这样做在组件日渐复杂的迭代过程中我们很难确保它在各种情况下还能正常运行。

想象一下我们的代码做下面这样的分离，并且每一个函数的体量是现在的五倍：

```
function SearchResults() {
  // Imagine this function is long
  function getFetchUrl() {
    return 'https://hn.algolia.com/api/v1/search?query=react';
  }

  // Imagine this function is also long
  async function fetchData() {
    const result = await axios(getFetchUrl());
    setData(result.data);
  }

  useEffect(() => {
    fetchData();
  }, []);
```

```
  // ...
}
```

然后我们在某些函数内使用了某些state或者prop：

```
function SearchResults() {
  const [query, setQuery] = useState('react');

  // Imagine this function is also long
  function getFetchUrl() {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }

  // Imagine this function is also long
  async function fetchData() {
    const result = await axios(getFetchUrl());
    setData(result.data);
  }

  useEffect(() => {
    fetchData();
  }, []);

  // ...
}
```

如果我们忘记去更新使用这些函数（很可能通过其他函数调用）的effects的依赖，我们的effects就不会同步props和state带来的变更。这当然不是我们想要的。

幸运的是，对于这个问题有一个简单的解决方案。如果某些函数仅在**effect**中调用，你可以把它们的定义移到**effect**中：

```
function SearchResults() {
  // ...
  useEffect(() => {
    // We moved these functions inside!
    function getFetchUrl() {
      return 'https://hn.algolia.com/api/v1/search?query=react';
    }
    async function fetchData() {
      const result = await axios(getFetchUrl());
      setData(result.data);
```

```
    }

    fetchData();
  }, []); // ✅ Deps are OK
  // ...
}
```

(这里是demo.)

这么做有什么好处呢？我们不再需要去考虑这些“间接依赖”。我们的依赖数组也不再撒谎：在我们的**effect**中确实没有再使用组件范围内的任何东西。

如果我们后面修改 `getFetchUrl` 去使用 `query` 状态，我们更可能会意识到我们正在effect里面编辑它 - 因此，我们需要把 `query` 添加到effect的依赖里：

```
function SearchResults() {
  const [query, setQuery] = useState('react');

  useEffect(() => {
    function getFetchUrl() {
      return 'https://hn.algolia.com/api/v1/search?query=' + query;
    }

    async function fetchData() {
      const result = await axios(getFetchUrl());
      setData(result.data);
    }

    fetchData();
  }, [query]); // ✅ Deps are OK

  // ...
}
```

(这里是demo.)

添加这个依赖，我们不仅仅是在“取悦React”。在query改变后去重新请求数据是合理的。`useEffect` 的设计意图就是要强迫你关注数据流的改变，然后决定我们的effects该如何和它同步 - 而不是忽视它直到我们的用户遇到了bug。

感谢`eslint-plugin-react-hooks`插件的`exhaustive-deps`lint规则，它会在你编码的时候就分析effects并且提供可能遗漏依赖的建议。换句话说，机器会告诉你组件中哪些数据流变更没有被正确地处理。

```
function SearchResults() {
  const [query, setQuery] = useState('react');

  function getFetchUrl() {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }

  async function fetchData() {
    const result = await axios(getFetchUrl());
    setData(result.data);
  }

  useEffect(() => {
    fetchData();
  }, []);

  // ...
}
```

非常棒。

但我不能把这个函数放到**Effect**里

有时候你可能不想把函数移入effect里。比如，组件内有几个effect使用了相同的函数，你不想在每个effect里复制黏贴一遍这个逻辑。也或许这个函数是一个prop。

在这种情况下你应该忽略对函数的依赖吗？我不这么认为。再次强调，**effects**不应该对它的依赖撒谎。通常我们还有更好的解决办法。一个常见的误解是，“函数从来不会改变”。但是这篇文章你读到现在，你知道这显然不是事实。实际上，在组件内定义的函数每一次渲染都在变。

函数每次渲染都会改变这个事实本身就是个问题。 比如有两个effects会调用`getFetchUrl`:

```
function SearchResults() {
  function getFetchUrl(query) {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }

  useEffect(() => {
    const url = getFetchUrl('react');
    // ... Fetch data and do something ...
  }, []); // 🔴 Missing dep: getFetchUrl

  useEffect(() => {
    const url = getFetchUrl('redux');
    // ... Fetch data and do something ...
  }, []); // 🔴 Missing dep: getFetchUrl

  // ...
}
```

在这个例子中，你可能不想把`getFetchUrl`移到effects中，因为你想复用逻辑。

另一方面，如果你对依赖很“诚实”，你可能会掉到陷阱里。我们的两个effects都依赖`getFetchUrl`，而它每次渲染都不同，所以我们的依赖数组会变得无用：

```
function SearchResults() {
  // 🔴 Re-triggers all effects on every render
  function getFetchUrl(query) {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }

  useEffect(() => {
    const url = getFetchUrl('react');
    // ... Fetch data and do something ...
  }, [getFetchUrl]); // 🚧 Deps are correct but they change too often

  useEffect(() => {
    const url = getFetchUrl('redux');
    // ... Fetch data and do something ...
  }, [getFetchUrl]); // 🚧 Deps are correct but they change too often

  // ...
}
```

一个可能的解决办法是把`getFetchUrl`从依赖中去掉。但是，我不认为这是好的解决方式。这会使我们后面对数据流的改变很难被发现从而忘记去处理。这会导致类似于上面“定时器不更新值”的问题。

相反的，我们有两个更简单的解决办法。

第一个， 如果一个函数没有使用组件内的任何值，你应该把它提到组件外面去定义，然后就可以自由地在**effects**中使用：

```
// ✅ Not affected by the data flow
function getFetchUrl(query) {
  return 'https://hn.algolia.com/api/v1/search?query=' + query;
}

function SearchResults() {
  useEffect(() => {
    const url = getFetchUrl('react');
    // ... Fetch data and do something ...
  }, []); // ✅ Deps are OK

  useEffect(() => {
    const url = getFetchUrl('redux');
    // ... Fetch data and do something ...
  }, []); // ✅ Deps are OK

  // ...
}
```

你不再需要把它设为依赖，因为它们不在渲染范围内，因此不会被数据流影响。它不可能突然意外地依赖于props或state。

或者， 你也可以把它包装成`useCallback` Hook:

```
function SearchResults() {
  // ✅ Preserves identity when its own deps are the same
  const getFetchUrl = useCallback((query) => {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }, []);  // ✅ Callback deps are OK

  useEffect(() => {
    const url = getFetchUrl('react');
    // ... Fetch data and do something ...
```

```
  }, [getFetchUrl]); // ✅ Effect deps are OK

  useEffect(() => {
    const url = getFetchUrl('redux');
    // ... Fetch data and do something ...
  }, [getFetchUrl]); // ✅ Effect deps are OK

  // ...
}
```

`useCallback`本质上是添加了一层依赖检查。它以另一种方式解决了问题 - 我们使函数本身只在需要的时候才改变，而不是去掉对函数的依赖。

我们来看看为什么这种方式是有用的。之前，我们的例子中展示了两种搜索结果（查询条件分别为`'react'`和`'redux'`）。但如果我们想添加一个输入框允许你输入任意的查询条件(`query`)。不同于传递`query`参数的方式，现在`getFetchUrl`会从状态中读取。

我们很快发现它遗漏了`query`依赖：

```
function SearchResults() {
  const [query, setQuery] = useState('react');
  const getFetchUrl = useCallback(() => { // No query argument
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }, []); // 🔴 Missing dep: query
  // ...
}
```

如果我把`query`添加到`useCallback`的依赖中，任何调用了`getFetchUrl`的effect在`query`改变后都会重新运行：

```
function SearchResults() {
  const [query, setQuery] = useState('react');

  // ✅ Preserves identity until query changes
  const getFetchUrl = useCallback(() => {
    return 'https://hn.algolia.com/api/v1/search?query=' + query;
  }, [query]);  // ✅ Callback deps are OK

  useEffect(() => {
    const url = getFetchUrl();
    // ... Fetch data and do something ...
```

```
  }, [getFetchUrl]); // ✅ Effect deps are OK

  // ...
}
```

我们要感谢`useCallback`，因为如果`query`保持不变，`getFetchUrl`也会保持不变，我们的effect也不会重新运行。但是如果`query`修改了，`getFetchUrl`也会随之改变，因此会重新请求数据。这就像你在Excel里修改了一个单元格的值，另一个使用它的单元格会自动重新计算一样。

这正是拥抱数据流和同步思维的结果。对于通过属性从父组件传入的函数这个方法也适用：

```
function Parent() {
  const [query, setQuery] = useState('react');

  // ✅ Preserves identity until query changes
  const fetchData = useCallback(() => {
    const url = 'https://hn.algolia.com/api/v1/search?query=' + query;
    // ... Fetch data and return it ...
  }, [query]);  // ✅ Callback deps are OK

  return <Child fetchData={fetchData} />
}

function Child({ fetchData }) {
  let [data, setData] = useState(null);

  useEffect(() => {
    fetchData().then(setData);
  }, [fetchData]); // ✅ Effect deps are OK

  // ...
}
```

因为`fetchData`只有在`Parent`的`query`状态变更时才会改变，所以我们的`Child`只会在需要的时候才去重新请求数据。

函数是数据流的一部分吗？

有趣的是，这种模式在class组件中行不通，并且这种行不通恰到好处地揭示了effect和生命周期范式之间的区别。考虑下面的转换：

```
class Parent extends Component {
  state = {
    query: 'react'
  };
  fetchData = () => {
    const url = 'https://hn.algolia.com/api/v1/search?query=' + this.state.query;
    // ... Fetch data and do something ...
  };
  render() {
    return <Child fetchData={this.fetchData} />;
  }
}

class Child extends Component {
  state = {
    data: null
  };
  componentDidMount() {
    this.props.fetchData();
  }
  render() {
    // ...
  }
}
```

你可能会想：“少来了Dan，我们都知道`useEffect`就像`componentDidMount`和`componentDidUpdate`的结合，你不能老是破坏这一条！”好吧，就算加了**`componentDidUpdate`**照样无用：

```
class Child extends Component {
  state = {
    data: null
  };
  componentDidMount() {
    this.props.fetchData();
  }
  componentDidUpdate(prevProps) {
    // 🔴 This condition will never be true
    if (this.props.fetchData !== prevProps.fetchData) {
      this.props.fetchData();
    }
  }
```

```
  render() {
    // ...
  }
}
```

当然如此，`fetchData`是一个class方法！（或者你也可以说是class属性 - 但这不能改变什么。）它不会因为状态的改变而不同，所以`this.props.fetchData`和`prevProps.fetchData`始终相等，因此不会重新请求。那我们删掉条件判断怎么样？

```
  componentDidUpdate(prevProps) {
    this.props.fetchData();
  }
```

等等，这样会在每次渲染后都去请求。(添加一个加载动画可能是一种有趣的发现这种情况的方式。)也许我们可以绑定一个特定的query?

```
  render() {
    return <Child fetchData={this.fetchData.bind(this, this.state.query)} />;
  }
```

但这样一来，`this.props.fetchData !== prevProps.fetchData`表达式永远是`true`，即使`query`并未改变。这会导致我们总是去请求。

想要解决这个class组件中的难题，唯一现实可行的办法是硬着头皮把`query`本身传入`Child`组件。`Child`虽然实际并没有直接使用这个`query`的值，但能在它改变的时候触发一次重新请求：

```
class Parent extends Component {
  state = {
    query: 'react'
  };
  fetchData = () => {
    const url = 'https://hn.algolia.com/api/v1/search?query=' + this.state.query;
    // ... Fetch data and do something ...
  };
  render() {
    return <Child fetchData={this.fetchData} query={this.state.query} />;
```

```
  }
}

class Child extends Component {
  state = {
    data: null
  };
  componentDidMount() {
    this.props.fetchData();
  }
  componentDidUpdate(prevProps) {
    if (this.props.query !== prevProps.query) {
      this.props.fetchData();
    }
  }
  render() {
    // ...
  }
}
```

在使用React的class组件这么多年后，我已经如此习惯于把不必要的props传递下去并且破坏父组件的封装以至于我在一周之前才意识到我为什么一定要这样做。

在**class**组件中，函数属性本身并不是数据流的一部分。组件的方法中包含了可变的`this`变量导致我们不能确定无疑地认为它是不变的。因此，即使我们只需要一个函数，我们也必须把一堆数据传递下去仅仅是为了做“diff”。我们无法知道传入的`this.props.fetchData`是否依赖状态，并且不知道它依赖的状态是否改变了。

使用`useCallback`，函数完全可以参与到数据流中。我们可以说如果一个函数的输入改变了，这个函数就改变了。如果没有，函数也不会改变。感谢周到的`useCallback`，属性比如`props.fetchData`的改变也会自动传递下去。

类似的，`useMemo`可以让我们对复杂对象做类似的事情。

```
function ColorPicker() {
  // Doesn't break Child's shallow equality prop check
  // unless the color actually changes.
  const [color, setColor] = useState('pink');
  const style = useMemo(() => ({ color }), [color]);
  return <Child style={style} />;
}
```

我想强调的是，到处使用`useCallback`是件挺笨拙的事。当我们需要将函数传递下去并且函数会在子组件的effect中被调用的时候，`useCallback`是很好的技巧且非常有用。或者你想试图减少对子组件的记忆负担，也不妨一试。但总的来说Hooks本身能更好地避免传递回调函数。

在上面的例子中，我更倾向于把`fetchData`放在我的effect里（它可以抽离成一个自定义Hook）或者是从顶层引入。我想让effects保持简单，而在里面调用回调会让事情变得复杂。（“如果某个`props.onComplete`回调改变了而请求还在进行中会怎么样？”）你可以模拟class的行为但那样并不能解决竞态的问题。

说说竞态

下面是一个典型的在class组件里发请求的例子：

```
class Article extends Component {
  state = {
    article: null
  };
  componentDidMount() {
    this.fetchData(this.props.id);
  }
  async fetchData(id) {
    const article = await API.fetchArticle(id);
    this.setState({ article });
  }
  // ...
}
```

你很可能已经知道，上面的代码埋伏了一些问题。它并没有处理更新的情况。所以第二个你能够在网上找到的经典例子是下面这样的：

```
class Article extends Component {
  state = {
    article: null
  };
  componentDidMount() {
    this.fetchData(this.props.id);
  }
  componentDidUpdate(prevProps) {
```

```
    if (prevProps.id !== this.props.id) {
      this.fetchData(this.props.id);
    }
  }
  async fetchData(id) {
    const article = await API.fetchArticle(id);
    this.setState({ article });
  }
  // ...
}
```

这显然好多了！但依旧有问题。有问题的原因是请求结果返回的顺序不能保证一致。比如我先请求 `{id: 10}`，然后更新到 `{id: 20}`，但 `{id: 20}` 的请求更先返回。请求更早但返回更晚的情况会错误地覆盖状态值。

这被叫做竞态，这在混合了 `async` / `await`（假设在等待结果返回）和自顶向下数据流的代码中非常典型（props和state可能会在async函数调用过程中发生改变）。

Effects并没有神奇地解决这个问题，尽管它会警告你如果你直接传了一个 `async` 函数给effect。（我们会改善这个警告来更好地解释你可能会遇到的这些问题。）

如果你使用的异步方式支持取消，那太棒了。你可以直接在清除函数中取消异步请求。

或者，最简单的权宜之计是用一个布尔值来跟踪它：

```
function Article({ id }) {
  const [article, setArticle] = useState(null);

  useEffect(() => {
    let didCancel = false;

    async function fetchData() {
      const article = await API.fetchArticle(id);
      if (!didCancel) {
        setArticle(article);
      }
    }

    fetchData();

    return () => {
      didCancel = true;
    };
```

```
  }, [id]);

  // ...
}
```

这篇文章讨论了更多关于如何处理错误和加载状态，以及抽离逻辑到自定义的Hook。我推荐你认真阅读一下如果你想学习更多关于如何在Hooks里请求数据的内容。

提高水准

在class组件生命周期的思维模型中，副作用的行为和渲染输出是不同的。UI渲染是被props和state驱动的，并且能确保步调一致，但副作用并不是这样。这是一类常见问题的来源。

而在`useEffect`的思维模型中，默认都是同步的。副作用变成了React数据流的一部分。对于每一个`useEffect`调用，一旦你处理正确，你的组件能够更好地处理边缘情况。

然而，用好`useEffect`的前期学习成本更高。这可能让人气恼。用同步的代码去处理边缘情况天然就比触发一次不用和渲染结果步调一致的副作用更难。

这难免让人担忧如果`useEffect`是你现在使用最多的工具。不过，目前大抵还处理低水平使用阶段。因为Hooks太新了所以大家都还在低水平地使用它，尤其是在一些教程示例中。但在实践中，社区很可能即将开始高水平地使用Hooks，因为好的API会有更好的动量和冲劲。

我看到不同的应用在创造他们自己的Hooks，比如封装了应用鉴权逻辑的`useFetch`或者使用theme context的`useTheme`。你一旦有了包含这些的工具箱，你就不会那么频繁地直接使用`useEffect`。但每一个基于它的Hook都能从它的适应能力中得到益处。

目前为止，`useEffect`主要用于数据请求。但是数据请求准确说并不是一个同步问题。因为我们的依赖经常是`[]`所以这一点尤其明显。那我们究竟在同步什么？

长远来看，Suspense用于数据请求会允许第三方库通过第一等的途径告诉React暂停渲染直到某些异步事物（任何东西：代码，数据，图片）已经准备就绪。

当Suspense逐渐地覆盖到更多的数据请求使用场景，我预料`useEffect`会退居幕后作为一个强大的工具，用于同步props和state到某些副作用。不像数据请求，它可以很好地处理这些场景因为它就是为此而设计的。不过在那之前，自定义的Hooks比如这儿提到的是复用数据请求逻辑很好的方式。

在结束前

现在你差不多知道了我关于如何使用effects的所有知识，可以检查一下开头的TLDR。你现在觉得它说得有道理吗？我有遗漏什么吗？（我的纸还没有写完！）

我很想在Twitter上听听你的想法。谢谢阅读。