React 的未来（3）：函数化的 Hooks

这一节我们来介绍 Hooks，React v16.7.0-alpha 中第一次引入了 Hooks 的概念，因为这是一个 alpha 版本，不算正式发布，所以，将来正式发布时 API 可能会有变化。

Hooks 的目的，简而言之就是让开发者不需要再用 class 来实现组件。

还记得之前我们介绍的经典 Counter 组件吗？不考虑用 Redux 或者 Mobx 来管理状态的话，Counter 组件就需要把计数数据放在 state 里，要用 state，就意味着需要定义一个 class。

很多时候，一个简单组件也需要实现一个 class，的确是一件很烦的事，有了 Hooks 之后，事情就简单多了，我们用几个已经公开的 Hooks API 来看看如何避免写 class。

useState

Hooks 会提供一个叫 `useState` 的方法，它开启了一扇新的定义 state 的门，对应 Counter 的代码可以这么写：

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

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

  return (
    <div>
      <div>{count}</div>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

注意看，Counter 拥有自己的“状态”，但它只是一个函数，不是 class。

`useState` 只接受一个参数，也就是 state 的初始值，它返回一个只有两个元素的数组，第一个元素就是 state 的值，第二个元素是更新 state 的函数。

我们利用解构赋值（destructuring assignment）把两个元素分别赋值给 count 和 setCount，相当于这样的代码：

```
  // 下面代码等同于： const [count, setCount] = useState(0);
  const result = useState(0);
  const count = result[0];
  const setCount = result[1];
```

利用 `count` 可以读取到这个 state，利用 `setCount` 可以更新这个 state，而且我们完全可以控制这两个变量的命名，只要高兴，你完全可以这么写：

```
  const [theCount, updateCount] = useState(0);
```

因为 `useState` 在 `Counter` 这个函数体中，每次 Counter 被渲染的时候，这个 `useState` 调用都会被执行，`useState` 自己肯定不是一个纯函数，因为它要区分第一次调用（组件被 mount 时）和后续调用（重复渲染时），只有第一次才用得上参数的初始值，而后续的调用就返回“记住”的 state 值。

读者看到这里，心里可能会有这样的疑问：如果组件中多次使用 `useState` 怎么办？React 如何“记住”哪个状态对应哪个变量？

React 是完全根据 `useState` 的调用顺序来“记住”状态归属的，假设组件代码如下：

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const [foo, updateFoo] = useState('foo');

  ...
}
```

每一次 Counter 被渲染，都是第一次 `useState` 调用获得 `count` 和 `setCount`，第二次 `useState` 调用获得 `foo` 和 `updateFoo`（这里我故意让命名不用 `set` 前缀，可见函数名可以随意）。React 是渲染过程中的“上帝”，每一次渲染 Counter 都要由 React 发起，所以它有机会准备好一个内存记录，当开始执行的时候，每一次 useState 调用对应内存记录上一个位置，而且是按照顺序来记录的。React 不知道你把 `useState` 等 Hooks API 返回的结果赋值给什么变量，但是它也不需要知道，它只需要按照 `useState` 调用顺序记录就好了。

正因为这个原因，**Hooks，千万不要在 if 语句或者 for 循环语句中使用！**

像下面的代码，肯定会出乱子的：

```
const Counter = () => {
    const [count, setCount] = useState(0);
    if (count % 2 === 0) {
        const [foo, updateFoo] = useState('foo');
    }
    const [bar, updateBar] = useState('bar');
  ...
}
```

因为条件判断，让每次渲染中 `useState` 的调用次序不一致了，于是 React 就错乱了。

useEffect

除了 `useState`，React 还提供 `useEffect`，用于支持组件中增加副作用的支持。

在 React 组件生命周期中如果要做有副作用的操作，代码放在哪里？

当然是放在 componentDidMount 或者 componentDidUpdate 里，但是这意味着组件必须是一个 class。

在 Counter 组件，如果我们想要在用户点击“+”或者“-”按钮之后把计数值体现在网页标题上，这就是一个修改 DOM 的副作用操作，所以必须把 Counter 写成 class，而且添加下面的代码：

```
componentDidMount() {
  document.title = `Count: ${this.state.count}`;
}

componentDidUpdate() {
  document.title = `Count: ${this.state.count}`;
}
```

而有了 `useEffect`，我们就不用写一个 class 了，对应代码如下：

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

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

  useEffect(() => {
    document.title = `Count: ${count}`;
  });

  return (
    <div>
      <div>{count}</div>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

`useEffect` 的参数是一个函数，组件每次渲染之后，都会调用这个函数参数，这样就达到了 componentDidMount 和 componentDidUpdate 一样的效果。

虽然本质上，依然是 componentDidMount 和 componentDidUpdate 两个生命周期被调用，但是现在我们关心的不是 mount 或者 update 过程，而是“after render”事件，`useEffect` 就是告诉组件在“渲染完”之后做点什么事。

读者可能会问，现在把 componentDidMount 和 componentDidUpdate 混在了一起，那假如某个场景下我只在 mount 时做事但 update 不做事，用 useEffect 不就不行了吗？

其实，用一点小技巧就可以解决。`useEffect` 还支持第二个可选参数，只有同一 `useEffect` 的两次调用第二个参数不同时，第一个函数参数才会被调用，所以，如果想模拟 `componentDidMount`，只需要这样写：

```
  useEffect(() => {
    // 这里只有mount时才被调用，相当于componentDidMount
  }, [123]);
```

在上面的代码中，`useEffect` 的第二个参数是 `[123]`，其实也可以是任何一个常数，因为它永远不变，所以 `useEffect` 只在 mount 时调用第一个函数参数一次，达到了 componentDidMount 一样的效果。

useContext

在前面介绍“提供者模式”章节我们介绍过 React 新的 Context API，这个 API 不是完美的，在多个 Context 嵌套的时候尤其麻烦。

比如，一段 JSX 如果既依赖于 ThemeContext 又依赖于 LanguageContext，那么按照 React Context API 应该这么写：

```
<ThemeContext.Consumer>
    {
        theme => (
            <LanguageContext.Cosumer>
                language => {
                    //可以使用theme和lanugage了
                }
            </LanguageContext.Cosumer>
        )
    }
</ThemeContext.Consumer>
```

因为 Context API 要用 render props，所以用两个 Context 就要用两次 render props，也就用了两个函数嵌套，这样的缩格看起来也的确过分了一点点。

使用 Hooks 的 `useContext`，上面的代码可以缩略为下面这样：

```
const theme = useContext(ThemeContext);
const language = useContext(LanguageContext);
// 这里就可以用theme和language了
```

这个 `useContext` 把一个需要很费劲才能理解的 Context API 使用大大简化，不需要理解render props，直接一个函数调用就搞定。

但是，`useContext` 也并不是完美的，它会造成意想不到的重新渲染，我们看一个完整的使用 `useContext` 的组件。

```
const ThemedPage = () => {
    const theme = useContext(ThemeContext);

    return (
       <div>
            <Header color={theme.color} />
            <Content color={theme.color}/>
            <Footer color={theme.color}/>
       </div>
    );
};
```

因为这个组件 `ThemedPage` 使用了 `useContext`，它很自然成为了Context的一个消费者，所以，只要Context的值发生了变化，`ThemedPage` 就会被重新渲染，这很自然，因为不重新渲染也就没办法重新获得 `theme` 值，但现在有一个大问题，对于ThemedPage来说，实际上只依赖于 `theme` 中的 `color` 属性，如果只是 `theme` 中的 `size` 发生了变化但是 `color` 属性没有变化，`ThemedPage` 依然会被重新渲染，当然，我们通过给 `Header`、`Content` 和 `Footer` 这些组件添加 `shouldComponentUpdate` 实现可以减少没有必要的重新渲染，但是上一层的 `ThemedPage` 中的JSX重新渲染是躲不过去了。

说到底，`useContext` 需要一种表达方式告诉React：“我没有改变，重用上次内容好了。”

希望Hooks正式发布的时候能够弥补这一缺陷。

Hooks 带来的代码模式改变

上面我们介绍了 `useState`、`useEffect` 和 `useContext` 三个最基本的 Hooks，可以感受到，Hooks 将大大简化使用 React 的代码。

首先我们可能不再需要 class了，虽然 React 官方表示 class 类型的组件将继续支持，但是，业界已经普遍表示会迁移到 Hooks 写法上，也就是放弃 class，只用函数形式来编写组件。

对于 `useContext`，它并没有为消除 class 做贡献，却为消除 render props 模式做了贡献。很长一段时间，高阶组件和 render props 是组件之间共享逻辑的两个武器，但如同我前面章节介绍的那样，这两个武器都不是十全十美的，现在 Hooks 的出现，也预示着高阶组件和 render props 可能要被逐步取代。

但读者朋友，不要觉得之前学习高阶组件和 render props 是浪费时间，相反，你只有明白 React 的使用历史，才能更好地理解 Hooks 的意义。

可以预测，在 Hooks 兴起之后，共享代码之间逻辑会用函数形式，而且这些函数会以 `use-` 前缀为约定，重用这些逻辑的方式，就是在函数形式组件中调用这些 `useXXX` 函数。

例如，我们可以写这样一个共享 Hook `useMountLog`，用于在 mount 时记录一个日志，代码如下：

```
const useMountLog = (name) => {
    useEffect(() => {
        console.log(`${name} mounted`);
    }, [123]);
}
```

任何一个函数形式组件都可以直接调用这个 `useMountLog` 获得这个功能，如下：

```
const Counter = () => {
    useMountLog('Counter');

    ...
}
```

对了，所有的 Hooks API 都只能在函数类型组件中调用，class 类型的组件不能用，从这点看，很显然，class 类型组件将会走向消亡。

小结

这一节我们介绍了 React Hooks，读者应该能够理解：

1. Hooks 的意义就是可以淘汰 class 类型的组件；
2. Hooks 将改变重用组件逻辑的模式；
3. 在未来，Hooks 将是 React 使用的主流。

留言

评论将在后台进行审核，审核通过后对所有人可见

Chimingphang Front-End Developer @ 无业游民
为什么那么抗拒class？
▲ 0 评论 18天前

轩邈同学
环境：ReactNative
问题：
在用hooks去替换类的过程中发现一个问题，以前在类中父组件用ref调用子组件方法，hooks改造后，使用useImperativeMethods，但useImperativeMethods应与 forwardRef 一起使用，forwardRef返回值为 React 节点，这造成无法再次对该组件进行增强。
十分困扰，不知道有人碰到这种情况吗，或者有什么解决方案不，十分感谢🙏。
▲ 0 评论 28天前

KK_FE FE @ Shopee
函数类型组件 有没有类似PureComponent 的api 来处理性能问题呢
▲ 0 收起评论 1月前

程墨 Hulu
PureCompnent可以部分提高性能，但是PureComponent本质上只是实现了一个缺省的shouldComponentUpdate函数，只能提高重复渲染的性能，提高不了第一次mount的性能。
1月前

xiari 高级前端工程师 @ 拼多多
React.memo
19天前

评论审核通过后显示 评论

陈小俊 前端开发
有个疑问就是高阶组件是在组件外层复用组件逻辑，而hooks的几个api是在组件内部复用组件逻辑，一个在外，一个在内。对文章里“现在 Hooks 的出现，也预示着高阶组件和 render props 可能要被逐步取代。”这个取代不甚理解。
▲ 0 收起评论 1月前

程墨 Hulu
customized hooks也可以用来重用，所以功能上可以取代
1月前

评论审核通过后显示 评论

红谷滩陈冠希 前端 @ 无限996公司
如果只是theme中的不相关的size发生了变化但是color属性发生了变化， 这句话有点问题吧。
▲ 0 评论 1月前