

转

30 分钟精通 React 新特性React Hooks

2018年11月14日 15:57:11

liuyingv8

阅读数 : 1144

你还在为该使用无状态组件（Function）还是有状态组件（Class）而烦恼吗？——拥有了hooks，你再也不需要写Class了，你的所有组件都将是Function。

你还在为搞不清使用哪个生命周期钩子函数而日夜难眠吗？——拥有了Hooks，生命周期钩子函数可以先丢一边了。

你在还在为组件中的this指向而晕头转向吗？——既然Class都丢掉了，哪里还有this？你的人生第一次不再需要面对this。

这样看来，说React Hooks是今年最劲爆的新特性真的毫不夸张。如果你也对react感兴趣，或者正在使用react进行项目开发，答应我，请一定抽出至少30分钟的时间来阅读本文好吗？所有你需要了解的React Hooks的知识点，本文都涉及到了，相信完整读完后你一定会有所收获。

一个最简单的Hooks

首先让我们看一下一个简单的有状态组件：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

我们再来看一下使用hooks后的版本：

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

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

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

是不是简单多了！可以看到，Example变成了一个函数，但这个函数却有自己的状态（count），同时它还可以更新自己的状态（setCount）。这个函数之所以这么了不得，就是因为它注入了一个hook-- useState，就是这个hook让我们的函数变成了一个有状态的函数。

除了useState这个hook外，还有很多别的hook，比如 useEffect提供了类似于 componentDidMount等生命周期钩子的功能，useContext提供了上下文（context）的功能等等。

Hooks本质上就是一类特殊的函数，它们可以为你的函数型组件（function component）注入一些特殊的功能。噢？这听起来有点像被诟病的Mixins啊？难道是Mixins要在react中死灰复燃了吗？当然不会了，等会我们再来谈两者的区别。总而言之，这些hooks的目标就是让你不再写class，让function一统江湖。

React为什么要搞一个Hooks？

想要复用 一个有状态的组件太麻烦了！

我们都知道react都核心思想就是，将一个页面拆成一堆独立的，可复用的组件，并且用自上而下的单向数据流的形式将这些组件串联起来。但假如你在大型的工作项目中用react，你会发现你的项目中实际上很多react组件冗长且难以复用。尤其是那些写成class的组件，它们本身包含了状态（state），所以复用这类组件就变得很麻烦。

那之前，官方推荐怎么解决这个问题呢？答案是：渲染属性（Render Props）和高阶组件（Higher-Order Components）。我们可以稍微跑下题简单看一下这两种模式。

渲染属性指的是使用一个值为函数的prop来传递需要动态渲染的nodes或组件。如下面的代码可以看到我们的DataProvider组件包含了所有跟状态相关的代码，而Cat组件则可以是一个单纯的展示型组件，这样一来DataProvider就可以单独复用了。

```
import Cat from 'components/cat'
class DataProvider extends React.Component {
  constructor(props) {
    super(props);
    this.state = { target: 'Zac' };
  }

  render() {
    return (
      <div>
        {this.props.render(this.state)}
      </div>
    )
  }
}

<DataProvider render={data => (
  <Cat target={data.target} />
)}>/>
```

👍
0

💬

🔖

📱

<

>

虽然这个模式叫Render Props，但不是说非用一个叫render的props不可，习惯上大家更常写成下面这种：

```
...
<DataProvider>
  {data => (
    <Cat target={data.target} />
  )}
</DataProvider>
```

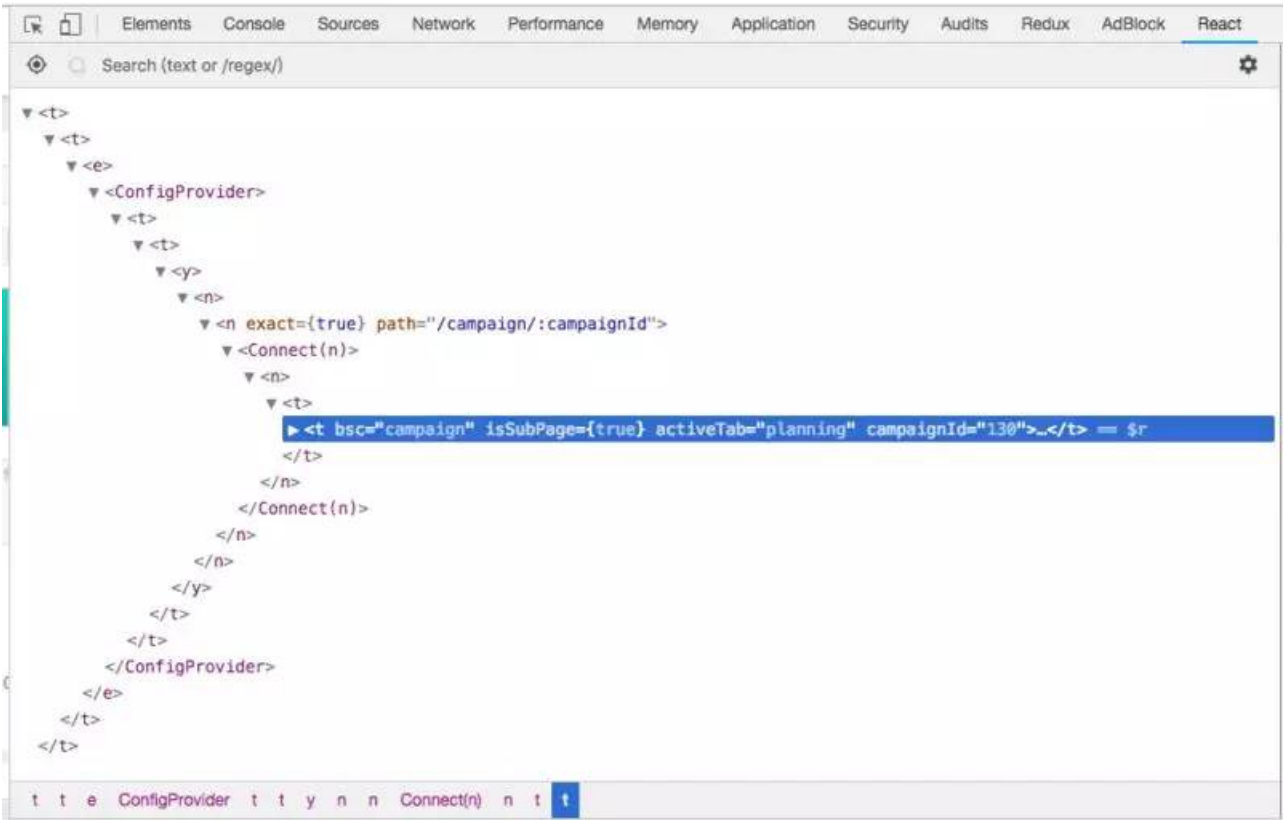
高阶组件这个概念就更好理解了，说白了就是一个函数接受一个组件作为参数，经过一系列加工后，最后返回一个新的组件。看下面的代码示例， withUser函数就是一个高阶组件，它返回了一个新的组件，这个组件具有了它提供的获取用户信息的功能。

```
const withUser = WrappedComponent => {
  const user = sessionStorage.getItem("user");
  return props => <WrappedComponent user={user} {...props} />;
};

const UserPage = props => (
  <div class="user-container">
    <p>My name is {props.user}!</p>
  </div>
);

export default withUser(UserPage);
```

以上这两种模式看上去都挺不错的，很多库也运用了这种模式，比如我们常用的React Router。但我们仔细看这两种模式，会发现它们会增加我们代码的层级关系。最直观的体现，打开devtool看看你的组件层级嵌套是不是很夸张吧。这时候再回过头看我们上一节给出的hooks例子，是不是简洁多了，没有多余的层级嵌套。把各种想要的功能写成一个一个可复用的自定义hook，当你的组件想用什么功能时，直接在组件里调用这个hook即可。



生命周期钩子函数里的逻辑太乱了吧！

我们通常希望一个函数只做一件事情，但我们的生命周期钩子函数里通常同时做了很多事情。比如我们需要在 componentDidMount中发起ajax请求获取数据，绑定一些事件监听等等。同时，有时候我们还需要在 componentDidMount做一遍同样的事情。当项目变复杂后，这一块的代码也变得不那么直观。

classes真的太让人困惑了！

我们用class来创建react组件时，还有一件很麻烦的事情，就是this的指向问题。为了保证this的指向正确，我们要经常写这样的代码：this.handleClick=this.handleClick.bind(this)，或者是这样的代码： <buttononClick={()=>this.handleClick(e)}>。一旦我们不小心忘了绑定this，各种bug就随之而来，很麻烦。

还有一件让我很苦恼的事情。我在之前的react系列文章当中曾经说过，尽可能把你的组件写成无状态组件的形式，因为它们更方便复用，可独立测试。然而很多时候，我们用function写了一个简洁完美的无状态组件，后来因为需求变动这个组件必须得有自己的state，我们又得很麻烦的把function改成class。

在这样的背景下，Hooks便横空出世了！

什么是State Hooks？

回到一开始我们用的例子，我们分解来看到底state hooks做了什么：

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

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

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

声明一个状态变量

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

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

useState是react自带的一个hook函数，它的作用就是用来声明状态变量。 useState这个函数接收的参数是我们的状态初始值（initial state），它返回了一个数组，这个数组的第 [0]项是当前当前的状态值，第 [1]项是可以改变状态值的方法函数。

所以我们做的事情其实就是，声明了一个状态变量count，把它的初始值设为0，同时提供了一个可以更改count的函数setCount。

上面这种表达形式，是借用了es6的数组解构（array destructuring），它可以让我们的代码看起来更简洁。不清楚这种用法的可以先去看下我的这篇文章：30分钟掌握ES6/ES2015核心内容（上）。

如果不用数组解构的话，可以写成下面这样。实际上数组解构是一件开销很大的事情，用下面这种写法，或者改用对象解构，性能会有很大的提升。具体可以去这篇文章的分析：Array destructuring for multi-value returns (in light of React hooks)，这里不详细展开，我们就按照官方推荐使用数组解构就好。

```
let _useState = useState(0);
let count = _useState[0];
let setCount = _useState[1];
```

读取状态值

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

是不是超简单？因为我们的状态count就是一个单纯的变量而已，我们再也不需要写成 {this.state.count}这样了。

更新状态

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

当用户点击按钮时，我们调用setCount函数，这个函数接收的参数是修改过的新状态值。接下来的事情就交给react了，react将会重新渲染我们的Example组件，并且使用的是更新后的新的状态，即count=1。这里我们要停下来思考一下，Example本质上也是一个普通的函数，为什么它可以记住之前的状态？

一个至关重要的问题

这里我们就发现了问题，通常来说我们在一个函数中声明的变量，当函数运行完成后，这个变量也就销毁了（这里我们先不考虑闭包等情况），比如考虑下面的例子：

```
function add(n) {
  const result = 0;
  return result + 1;
}

add(1); //1
add(1); //1
```

不管我们反复调用add函数多少次，结果都是1。因为每一次我们调用add时，result变量都是从初始值0开始的。那为什么上面的Example函数每次执行的时候，都是拿的上一次执行完的状态值作为初始值？答案是：是react帮我们记住的。至于react是用什么机制记住的，我们可以再思考一下。

假如一个组件有多个状态值怎么办？

首先，useState是可以多次调用的，所以我们完全可以这样写：

👍
0

💬

🔖

📱

<

>

```
function ExampleWithManyStates() {
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

👍
0

💬

🔖

📱

<

>

其次，useState接收的初始值没有规定一定要是string/number/boolean这种简单数据类型，它完全可以接收对象或者数组作为参数。唯一需要注意的点是，之前我们说的 this.setState做的是合并状态后返回一个新状态，而 useState是直接替换老状态后返回新状态。最后，react也给我们提供了一个useReducer的hook，如果你喜欢redux式的状态管理方案的话。

从ExampleWithManyStates函数我们可以看到，useState无论调用多少次，相互之间是独立的。这一点至关重要。为什么这么说呢？

其实我们看hook的“形态”，有点类似之前被官方否定掉的Mixins这种方案，都是提供一种“插拔式的功能注入”的能力。而mixins之所以被否定，是因为Mixins强制是让多个Mixins共享一个对象的数据空间，这样就很难确保不同Mixins依赖的状态不发生冲突。

而现在的hook，一方面它是直接用在function当中，而不是class；另一方面每一个hook都是相互独立的，不同组件调用同一个hook也能保证各自状态的独立性。这就是两者的本质区别了。

react是怎么保证多个useState的相互独立的？

还是看上面给出的ExampleWithManyStates例子，我们调用了三次useState，每次我们传的参数只是一个值（如42，‘banana’），我们根本没有告诉react这些值对应的key是哪个，那react是怎么保证这三个useState找到它对应的state呢？

答案是，react是根据useState出现的顺序来定的。我们具体来看一下：

```
//第一次渲染
useState(42); //将age初始化为42
useState('banana'); //将fruit初始化为banana
useState([{ text: 'Learn Hooks' }]); //...

//第二次渲染
useState(42); //读取状态变量age的值（这时候传的参数42直接被忽略）
useState('banana'); //读取状态变量fruit的值（这时候传的参数banana直接被忽略）
useState([{ text: 'Learn Hooks' }]); //...
```

假如我们改一下代码：

```
let showFruit = true;
function ExampleWithManyStates() {
  const [age, setAge] = useState(42);

  if(showFruit) {
    const [fruit, setFruit] = useState('banana');
    showFruit = false;
  }

  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

这样一来，

```
//第一次渲染
useState(42); //将age初始化为42
useState('banana'); //将fruit初始化为banana
useState([{ text: 'Learn Hooks' }]); //...

//第二次渲染
useState(42); //读取状态变量age的值（这时候传的参数42直接被忽略）
// useState('banana');
useState([{ text: 'Learn Hooks' }]); //读取到的却是状态变量fruit的值，导致报错
```

鉴于此，react规定我们必须把hooks写在函数的最外层，不能写在ifelse等条件语句当中，来确保hooks的执行顺序一致。

什么是Effect Hooks?

我们在上一节的例子中增加一个新功能：

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

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

  // 类似于componentDidMount 和 componentDidUpdate:
  useEffect(() => {
    // 更新文档的标题
    document.title = `You clicked ${count} times`;
  });

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

我们对比着看一下，如果没有hooks，我们会怎么写？


```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

👍
0

💬

🔖

📱

<

>

我们写的有状态组件，通常会产生很多的副作用（ side effect ），比如发起ajax请求获取数据，添加一些监听的注册和取消注册，手动修改dom等等。我们之前都把这些副作用的函数写在生命周期函数钩子里，比如componentDidMount，componentDidUpdate和componentWillUnmount。而现在的useEffect就相当与这些声明周期函数钩子的集合体。它以一抵三。

同时，由于前文所说hooks可以反复多次使用，相互独立。所以我们合理的做法是，给每一个副作用一个单独的useEffect钩子。这样一来，这些副作用不再一股脑堆在生命周期钩子里，代码变得更加清晰。

useEffect做了什么？

我们再梳理一遍下面代码的逻辑：

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

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

首先，我们声明了一个状态变量 count，将它的初始值设为0。然后我们告诉react，我们的这个组件有一个副作用。我们给 useEffecthook传了一个匿名函数，这个匿名函数就是我们的副作用。在这个例子里，我们的副作用是调用browser API来修改文档标题。当react要渲染我们的组件时，它会先记住我们用到的副作用。等react更新了DOM之后，它再依次执行我们定义的副作用函数。

这里要注意几点：

第一，react首次渲染和之后的每次渲染都会调用一遍传给useEffect的函数。而之前我们要用两个声明周期函数来分别表示首次渲染（ componentDidMount ），和之后的更新导致的重新渲染（ componentDidUpdate ）。

第二，useEffect中定义的副作用函数的执行不会阻碍浏览器更新视图，也就是说这些函数是异步执行的，而之前的componentDidMount或componentDidUpdate中的代码则是同步执行的。这种安排对大多数副作用说都是合理的，但有的情况除外，比如我们有时候需要先根据DOM计算出某个元素的尺寸再重新渲染，这时候我们希望这次重新渲染是同步发生的，也就是说它会在浏览器真的去绘制这个页面前发生。

useEffect怎么解绑一些副作用

这种场景很常见，当我们在componentDidMount里添加了一个注册，我们得马上在componentWillUnmount中，也就是组件被注销之前清除掉我们添加的注册，否则内存泄漏的问题就出现了。

怎么清除呢？让我们传给useEffect的副作用函数返回一个新的函数即可。这个新的函数将会在组件下一次重新渲染之后执行。这种模式在一些pubsub模式的实现中很常见。看下面的例子：

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // 一定注意下这个顺序：告诉react在下次重新渲染组件之后，同时是下次调用ChatAPI.subscribeTo
    FriendStatus之前执行cleanup
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

这里有一个点需要重视！这种解绑的模式跟componentWillUnmount不一样。componentWillUnmount只会在组件被销毁前执行一次而已，而useEffect里的函数，每次组件渲染后都会执行一遍，包括副作用函数返回的这个清理函数也会重新执行一遍。所以我们一起来看一下下面这个问题。

为什么要让副作用函数每次组件更新都执行一遍？

我们先看以前的模式：

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

很清除，我们在componentDidMount注册，再在componentWillUnmount清除注册。但假如这时候 props.friend.id变了怎么办？我们不得不再添加一个componentDidUpdate来处理这种情况：

```
...
componentDidUpdate(prevProps) {
  // 先把上一个friend.id解绑
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // 再重新注册新但friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
...
```

看到了吗？很繁琐，而我们但useEffect则没这个问题，因为它在每次组件更新后都会重新执行一遍。所以代码的执行顺序是这样的：

- 页面首次渲染
- 替friend.id=1的朋友注册
- 突然friend.id变成了2
- 页面重新渲染
- 清除friend.id=1的绑定
- 替friend.id=2的朋友注册
- ...

怎么跳过一些不必要的副作用函数

按照上一节的思路，每次重新渲染都要执行一遍这些副作用函数，显然是不经济的。怎么跳过一些不必要的计算呢？我们只需要给useEffect传第二个参数即可。用第二个参数来告诉react只有当这个参数的值发生改变时，才执行我们传的副作用函数（第一个参数）。


```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // 只有当count的值发生变化时，才会重新执行`document.title`这一句
```


当我们第二个参数传一个空数组[]时，其实就相当于只在首次渲染的时候执行。也就是componentDidMount加componentWillUnmount的模式。不过这种用法可能带来bug，少用。

还有哪些自带的Effect Hooks?


除了上文重点介绍的useState和useEffect，react还给我们提供来很多有用的hooks：


- useContext
- useReducer
- useCallback
- useMemo
- useRef

0











- useImperativeMethods
- useMutationEffect
- useLayoutEffect

我不再一一介绍，大家自行去查阅官方文档。

怎么写自定义的Effect Hooks?

为什么要自己去写一个Effect Hooks? 这样我们才能把可以复用的逻辑抽离出来，变成一个个可以随意插拔的“插销”，哪个组件要用来，我就插进哪个组件里，easy！看一个完整的例子，你就明白了。

比如我们可以把上面写的FriendStatus组件中判断朋友是否在线的功能抽出来，新建一个useFriendStatus的hook专门用来判断某个id是否在线。

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

这时候FriendStatus组件就可以简写为：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

简直Perfect！假如这个时候我们又有一个朋友列表也需要显示是否在线的信息：

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);


  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```


简直Fabulous!


编辑：千锋HTML5


作者：zach5078

segmentfault.com/a/1190000016950339


0













想对作者说点什么