02-理解Hooks：React为什么要发明Hooks？

你好，我是王沛。

React 作为目前最为主流的前端框架，自2013年诞生至今已经有近8年的时间了。这8年来，在其他前端框架发生翻天覆地变化的同时，React 的 API 则非常稳定，几乎从来没有出现过重大的向后兼容的问题，而且每一次版本的升级也都非常顺滑。

这不仅说明React 在 API 的设计上经受住了考验，同时也可以看到 React 团队在稳定 API 上所做的努力。

但是即使在这样稳定的背景下，React 还是在两年前的 16.8 版本中推出了一套全新的 Hooks 机制。新版本的发布，在让人耳目一新的同时，也让广大早就恐惧于各种新轮子的前端同学心头一紧，产生本能的抗拒：这是什么东西？有什么用？难道现在的 API 不好吗？

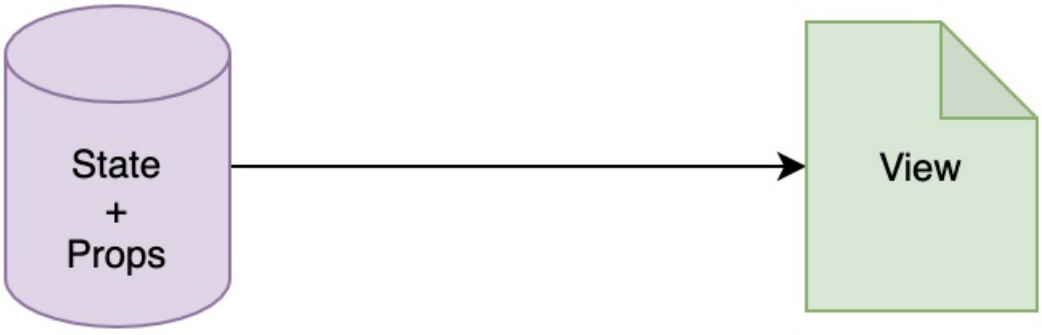
如果你也有这样的疑问，那我要先给你吃一颗定心丸，这样的担忧是没有必要的。

其实对于 React 开发而言，这只是多了一个选择。因为**原来的基于Class的组件完全可以继续使用**，这意味着这两种开发方式可以并存，已有代码不需要做任何改动，而新的代码可以根据具体情况采用 Hooks 的方式来实现就行了。

那么，既然在追求极致向后兼容性的原则下，React 还是推出了新的 Hooks API，一定有其过人的一面。所以今天这节课，我就跟你一起探讨它的过人之处究竟是什么；为什么要发明 Hooks；以及它所解决的问题是什么。

重新思考 React 组件的本质

React 组件的模型其实很直观，就是从 Model 到 View 的映射，这里的 Model 对应到 React 中就是 state 和 props。如下图所示：



在过去，我们需要处理当 Model 变化时，DOM 节点应该如何变化的细节问题。而现在，我们只需要通过 JSX，根据 Model 的数据用声明的方式去描述 UI 的最终展现就可以了，因为React 会帮助你处理所有 DOM 变化的细节。而且，当 Model 中的状态发生变化时，UI 会自动变化，即**所谓的数据绑定**。

所以呢，我们可以把UI 的展现看成一个函数的执行过程。其中，Model 是输入参数，函数的执行结果是 DOM 树，也就是 View。而React 要保证的，就是每当 Model 发生变化时，函数会重新执行，并且生成新的 DOM 树，然后React 再把新的 DOM 树以最优的方式更新到浏览器。

www.ixuexi.cf 分享站 用户标记 group_share

既然如此，使用 Class 作为组件是否真的合适呢？Class 在作为 React 组件的载体时，是否用了它所有的功能呢？如果你仔细思考，会发现使用 Class 其实是有点牵强的，主要有两方面的原因。

一方面，React 组件之间是不会互相继承的。比如说，你不会创建一个 Button 组件，然后再创建一个 DropdownButton 来继承 Button。所以说，React 中其实是没有利用到 Class 的继承特性的。

另一方面，因为所有 UI 都是由状态驱动的，因此很少会在外部去调用一个类实例（即组件）的方法。要知道，组件的所有方法都是在内部调用，或者作为生命周期方法被自动调用的。

所以你看，这两个 Class 最重要的特性其实都没有用到。而在 React 出现之时，主流的方式还是基于对象去考虑问题。例如获得一个对话框的实例，然后通过 dialog.show(), dialog.hide() 这样的方式细粒度地去控制 UI 的变化。

而这在 React 中其实是没有必要的，因为所有的 UI 都是声明出来的，不用处理细节的变化过程。因此，通过函数去描述一个组件才是最为自然的方式。这也是为什么 React 很早就提供了函数组件的机制。

只是当时有一个局限是，函数组件无法存在内部状态，必须是纯函数，而且也无法提供完整的生命周期机制。这就极大限制了函数组件的大规模使用。

那么我们自然就知道了，Class 作为 React 组件的载体，也许并不是最适合，反而函数是更适合去描述 State => View 这样的一个映射，但是函数组件又没有 State ，也没有生命周期方法。以此来看，我们应该如何去改进呢？

Hooks 是如何诞生的？

其实顺着函数组件的思路继续思考，就会发现，如果我们想要让函数组件更有用，目标就是给函数组件加上状态。这看上去似乎并不是难事。

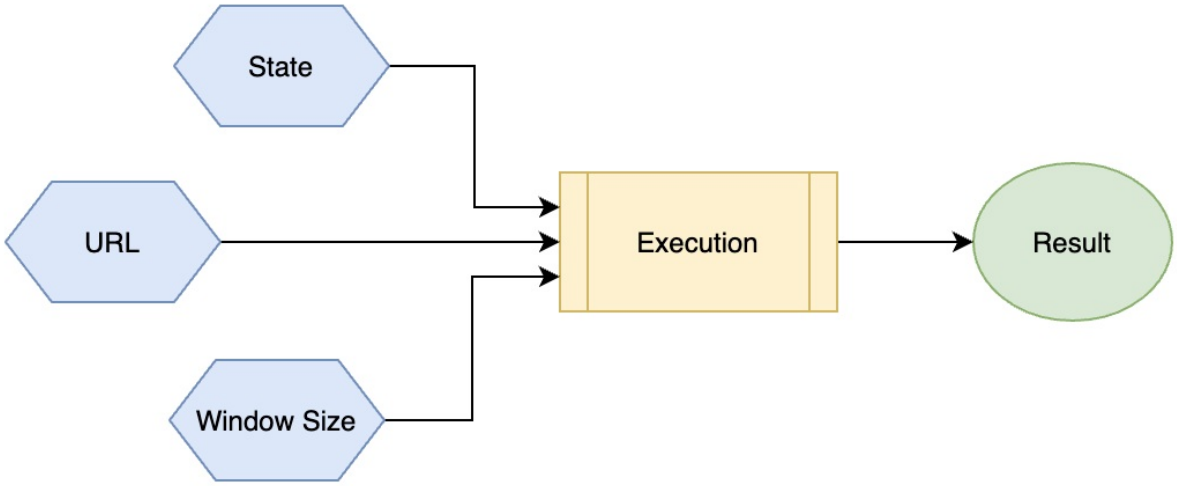
简单想一下，函数和对象不同，并没有一个实例的对象能够在多次执行之间保存状态，那势必需要一个函数之外的空间来保存这个状态，而且要能够检测其变化，从而能够触发函数组件的重新渲染。

再进一步想，那我们是不是就是需要这样一个机制，能够把一个外部的数据绑定到函数的执行。当数据变化时，函数能够自动重新执行。这样的话，任何会影响 UI 展现的外部数据，都可以通过这个机制绑定到 React 的函数组件。

在 React 中，这个机制就是 Hooks。

所以我们现在也能够理解这个机制为什么叫 Hooks了。顾名思义，Hook 就是“钩子”的意思。在 React 中，Hooks 就是**把某个目标结果钩到某个可能会变化的数据源或者事件源上，那么当被钩到的数据或事件发生变化时，产生这个目标结果的代码会重新执行，产生更新后的结果**。

对于函数组件，这个结果是最终的 DOM 树；对于 useCallback、useMemo 这样与缓存相关的组件，则是在依赖项发生变化时去更新缓存。所以 Hooks 的结构可以如下图所示：



从图中可以看到，一个执行过程（Execution），例如是函数组件本身，可以绑定在（钩在）传统意义的 State，或者 URL，甚至可以是窗口的大小。这样当State、URL、窗口大小发生变化时，都会重新执行某个函数，产生更新后的结果。

当然，既然我们的初衷是为了实现 UI 组件的渲染，那么在 React 中，其实所有的 Hooks 的最终结果都是导致 UI 的变化。但是正如 React 官方曾经提到过的，**Hooks 的思想其实不仅可以用在 React，在其它一些场景也可以被利用**。

通过这样的思考，你应该能够理解 Hooks 诞生的来龙去脉了。比起 Class 组件，函数组件是更适合去表达 React 组件的执行的，因为它更符合 State => View 这样的一个逻辑关系。但是因为缺少状态、生命周期等机制，让它一直功能受限。而现在有了 Hooks，函数组件的力量终于能真正发挥出来了！

不过这里有一点需要特别注意，Hooks 中被钩的对象，不仅可以是某个独立的数据源，也可以是另一个 Hook 执行的结果，这就带来了 Hooks 的最大好处：**逻辑的复用**。

Hooks 带来的最大好处：逻辑复用

在之前的 React 使用中，有一点经常被大家诟病，就是非常难以实现逻辑的复用，必须借助于高阶组件等复杂的设计模式。但是高阶组件会产生冗余的组件节点，让调试变得困难。不过这些问题可以通过 Hooks 得到很好的解决。所以如果有人问你 Hooks 有什么好处，那么最关键的答案就是**简化了逻辑复用**。

就以刚才我们提到的绑定窗口大小的场景为例。如果有多个组件需要在用户调整浏览器窗口大小时，重新调整布局，那么我们需要把这样的逻辑提取成一个公共的模块供多个组件使用。以 React 思想，在 JSX 中我们会根据 Size 大小来渲染不同的组件，例如：

```
function render() {
  if (size === "small") return <SmallComponent />;
  else return <LargeComponent />;
}
```

www.ixuexi.cf 分享站 用户标记 group_share

这段代码看上去简单，但如果在过去的 Class 组件中，我们甚至需要一个比较复杂的设计模式来解决，这就是高阶组件。所以接下来我们不妨通过实际的代码来进行一下比较，如果之前没有用过 React 也没有关系，你只需根据字面含义进行大概的理解即可。

在 Class 组件的场景下，我们首先需要定义一个高阶组件，负责监听窗口大小变化，并将变化后的值作为 props 传给下一个组件。

```
const withWindowSize = Component => {
  // 产生一个高阶组件 WrappedComponent，只包含监听窗口大小的逻辑
  class WrappedComponent extends React.PureComponent {
    constructor(props) {
      super(props);
      this.state = {
        size: this.getSize()
      };
    }
    componentDidMount() {
      window.addEventListener("resize", this.handleResize);
    }
    componentWillUnmount() {
      window.removeEventListener("resize", this.handleResize);
    }
    getSize() {
      return window.innerWidth > 1000 ? "large" : "small";
    }
    handleResize = ()=> {
      const currentSize = this.findSize();
      this.setState({
        size: this.findSize()
      });
    }
    render() {
      // 将窗口大小传递给真正的业务逻辑组件
      return <Component size={this.state.size} />;
    }
  }
  return WrappedComponent;
};
```

这样，在你的自定义组件中可以调用 withWindowSize 这样的函数来产生一个新组件，并自带 size 属性，例如：

```
class MyComponent extends React.Component({ size }){
  render() {
    if (size === "small") return <SmallComponent />;
    else return <LargeComponent />;
  }
}
// 使用 withWindowSize 产生高阶组件，用于产生 size 属性传递给真正的业务组件
export default withWindowSize(MyComponent);
```

在这里，我们可以看到，为了传递一个外部的状态，我们不得不定义一个没有UI 的外层组件，而这个组件

只是为了封装一段可重用的逻辑。更为糟糕的是，高阶组件几乎是 Class 组件中实现代码逻辑复用的唯一方式，其缺点其实比较显然：

1. 代码难理解，不直观，很多人甚至宁愿重复代码，也不愿用高阶组件；
2. 会增加很多额外的组件节点。每一个高阶组件都会多一层节点，这就会给调试带来很大的负担。

可以说，正因为这些缺点被抱怨已久，React 团队才终于提出了 Hooks 这样一个全新的方案。那么现在我们不妨看一下，同样的逻辑如果用 Hooks 和函数组件该如何实现。首先我们需要实现一个 Hooks：

```
const getSize = () => {
  return window.innerWidth > 1000 ? "large" : "small";
}
const useWindowSize = () => {
  const [size, setSize] = useState(getSize());
  useEffect(() => {
 const handler = () => {
      setState(getSize())
    };
    window.addEventListener('resize', handler);
    return () => {
      window.removeEventListener('resize', handler);
    };
  }, []);

  return size;
};
```

这样，我们在组件中使用窗口大小就会非常简单：

```
const Demo = () => {
  const size = useWindowSize();
  if (size === "small") return <SmallComponent />;
  else return <LargeComponent />;
};
```

可以看到，窗口大小是一个外部的一个数据状态，我们通过 Hooks 的方式对其进行了封装，从而将其变成一个可绑定的数据源。这样当窗口大小发生变化时，使用这个 Hook 的组件就都会重新渲染。而且代码也更加简洁和直观，不会产生额外的组件节点。

Hooks 的另一大好处：有助于关注分离

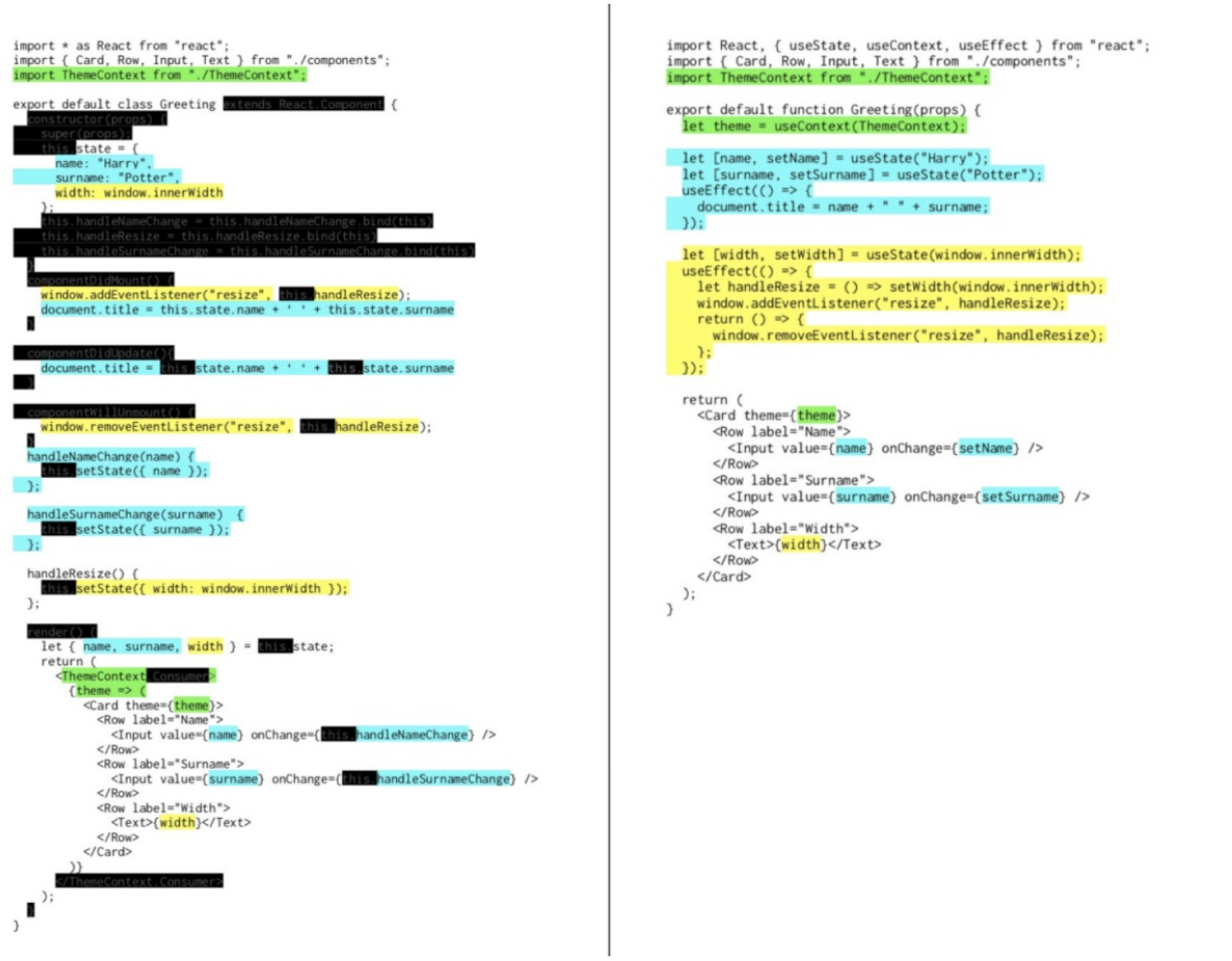
除了逻辑复用之外，Hooks 能够带来的另外一大好处就是**有助于关注分离**，意思是说Hooks 能够让针对同一个业务逻辑的代码尽可能聚合在一块儿。这是过去在 Class 组件中很难做到的。因为在 Class 组件中，你不得不把同一个业务逻辑的代码分散在类组件的不同生命周期的方法中。

所以通过 Hooks 的方式，把业务逻辑清晰地隔离开，能够让代码更加容易理解和维护。

www.ixuexi.cf 分享站 用户标记 group_share

仍然以上面监听浏览器窗口大小的变化为例，我们来看 Hooks 是如何做到关注分离的。在过去的 Class 组件中，我们需要在 componentDidMount 中监听事件，在 componentWillUnmount 中去解绑事件。而在函数组件中，我们可以把所有逻辑写在一起。

React 社区曾用一张图直观地展现了对比结果：



```
import * as React from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Harry",
      surname: "Potter",
      width: window.innerWidth
    };
    this.handleNameChange = this.handleNameChange.bind(this);
    this.handleResize = this.handleResize.bind(this);
    this.handleSurnameChange = this.handleSurnameChange.bind(this);
  }
  componentDidMount() {
    window.addEventListener("resize", this.handleResize);
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentDidUpdate() {
    document.title = this.state.name + ' ' + this.state.surname
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }
  handleNameChange(name) {
    this.setState({ name });
  };

  handleSurnameChange(surname) {
    this.setState({ surname });
  };

  handleResize() {
    this.setState({ width: window.innerWidth });
  };

  render() {
    let { name, surname, width } = this.state;
    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleNameChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

```
import React, { useState, useContext, useEffect } from "react";
import { Card, Row, Input, Text } from "./components";
import ThemeContext from "./ThemeContext";

export default function Greeting(props) {
  let theme = useContext(ThemeContext);

  let [name, setName] = useState("Harry");
  let [surname, setSurname] = useState("Potter");
  useEffect(() => {
    document.title = name + " " + surname;
  });

  let [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    let handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  });

  return (
    <Card theme={theme}>
      <Row label="Name">
        <Input value={name} onChange={setName} />
      </Row>
      <Row label="Surname">
        <Input value={surname} onChange={setSurname} />
      </Row>
      <Row label="Width">
        <Text>{width}</Text>
      </Row>
    </Card>
  );
}
```

图的左侧是 Class 组件，右侧是函数组件结合 Hooks。蓝色和黄色代表不同的业务功能。可以看到，在 Class 组件中，代码是从技术角度组织在一起的，例如在 componentDidMount 中都去做一些初始化的事情。而在函数组件中，代码是从业务角度组织在一起的，相关代码能够出现在集中的地方，从而更容易理解和维护。

小结

Hooks 作为 React 自诞生以来最大的一个新功能，可以说得到了普遍好评，成为了 React 开发的主流方式。而它也在一定程度上**更好地体现了 React 的开发思想，即从 State => View 的函数式映射**。

此外，**Hooks 也解决了 Class 组件存在的一些代码冗余、难以逻辑复用的问题**。但是正如我在这节课开头提到的，两种方式在 React 开发中几乎是完全等价的，没有绝对的优劣。如果你决定开始使用 Hooks，那么对于已有的 Class 组件，其实是完全没必要进行立刻重构的。只要在新的功能上，再来使用函数组件和 Hooks 就可以了。

思考题

文中举了窗口大小作为 Hooks 的数据源的例子，你还能想到哪些可能需要绑定到 React 组件的数据源？

欢迎在评论区写下你的想法和思考，我们一起交流讨论。如果今天这节课让你有所收获的话，也欢迎你把课

程分享给你的同事、朋友，我们一起共同进步。

精选留言：

- 凡凡 2021-05-27 11:40:52
  路由变化可以作为数据源吗 [2赞]

  作者回复2021-05-27 20:29:49
  可以的，像 react router 就提供了这个 Hook，也有一些第三方 library 也提供了，比如：https://github.com/streamich/react-use/blob/master/src/useLocation.ts 就是一个实现。

- Forest 2021-05-27 22:23:22
  逻辑复用应该是hook的最大优点吧

  作者回复2021-05-28 16:08:49
  没错！

- 独白 2021-05-27 11:21:01
  思考题：首先想到的是官方的一个api useReducer。还有的话，就是监听滚动条向上向下呀。
  看了老师这节课，才弄清除hooks真正重要的东西是啥。

  作者回复2021-05-27 20:29:03
  说的很好啊~ useReducer 是个比较通用的，类似于 useState 但是按照 Redux 的模式提供了一定的统一 state 管理机制。滚动条位置确实是个很好的例子，开发中也经常用到，在第6课就会讲到这个例子。

- 刘洋 2021-05-27 09:45:19
  多久更新完啊

  编辑回复2021-05-27 11:26:14
  每周二四六更新，预计7月13日更新完毕

- Geeker 2021-05-27 08:08:30
  加深了我对 hooks 的理解

  作者回复2021-05-27 20:30:00
  不错哈~

www.ixuexi.cf 分享站 用户标记 group_share