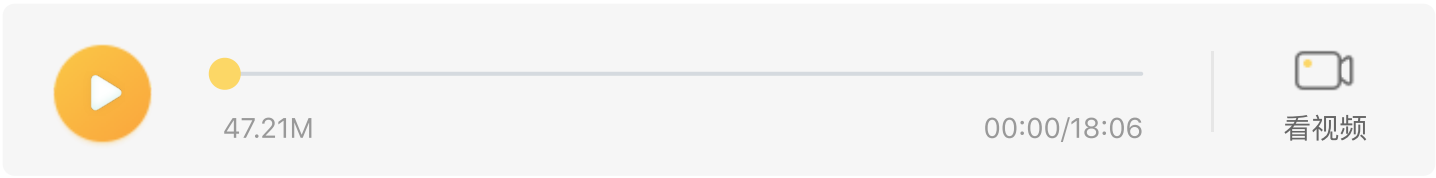
07 | React-Hooks 设计动机与工作模式（下）

2020/11/02 修言



经过第 6 课时的学习，相信你已经清楚了 React-Hooks 的来头，并理解了其背后的“设计动机”。本课时我们的任务是构建对 React-Hooks 的整体认知。

在本课时的主体部分，我将通过一系列的编码实例来帮助你认识 useState、useEffect 这两个有代表性的 Hook，这一步意在帮助初学者对 React-Hooks 可以快速上手。在此基础上，我们将重新理解“Why React-Hooks”这个问题。在课时的最后，我将结合自身的开发体验，和你分享当下这个阶段，我所认识到的 Hooks 的局限性。

> 注：在学习本课时的过程中，请你摒弃“认识的 API 名字越多就越牛”这种错误的学习理念。如果你希望掌握尽可能多的 Hook 的用法，点击这里可以一键进入 React-Hooks API 文档的海洋。对本课时来说，所有涉及对 API 用法的介绍都是 **“教具”**，仅仅是为后续更深层次的知识讲解作铺垫。

先导知识：从核心 API 看 Hooks 的基本形态

useState()：为函数组件引入状态

早期的函数组件相比于类组件，其一大劣势是缺乏定义和维护 state 的能力，而 state（状态）作为 React 组件的灵魂，必然是不可省略的。因此 React-Hooks 在诞生之初，就优先考虑了对 state 的支持。useState 正是这样一个能够为函数组件引入状态的 API。

函数组件，真的很轻

在过去，你可能会为了使用 state，不得不去编写一个类组件（这里我给出一个 Demo，编码如下所示）：

复制代码

```
import React, { Component } from "react";
export default class TextButton extends Component {

  constructor() {
    super();
    this.state = {
      text: "初始文本"
    };
  }
```

```
10.
11.   changeText = () => {
12.     this.setState(() => {
13.       return {
14.         text: "修改后的文本"
15.       };
16.     });
17.   };
18.   render() {
19.     const { text } = this.state;
20.     return (
21.       <div className="textButton">
22.         <p>{text}</p>
23.         <button onClick={this.changeText}>点击修改文本</button>
24.       </div>
25.     );
26.   }
27. }
```

有了 useState 后，我们就可以直接在函数组件里引入 state。以下是使用 useState 改造过后的 TextButton 组件：

复制代码

```
1. import React, { useState } from "react";
2. export default function Button() {
3.   const [text, setText] = useState("初始文本");
4.   function changeText() {
5.     return setText("修改后的文本");
6.   }
7.   return (
8.     <div className="textButton">
9.       <p>{text}</p>
10.      <button onClick={changeText}>点击修改文本</button>
11.    </div>
12.  );
13. }
```

上面两套代码实现的界面交互效果完全一样，而函数组件的代码量几乎是类组件代码量的一半！

如果你在第 06 课时曾或多或少地对“类组件太重了”这个观点感到茫然，那么相信眼前这个 Demo 足以让你真真切切地感受到两类组件在复杂度上的差异——**同样逻辑的函数组件相比类组件而言，复杂度要低得多得多**。

useState 快速上手

从用法上看，useState 返回的是一个数组，数组的第一个元素对应的是我们想要的那个 state 变量，第二个元素对应的是能够修改这个变量的 API。我们可以通过数组解构的语法，将这两个元素取出来，并且按照我们自己的想法命名。一个典型的调用示例如下：

复制代码

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

在这个示例中，我们给自己期望的那个状态变量命名为 state，给修改 state 的 API 命名为 setState。useState 中传入的 initialState 正是 state 的初始值。后续我们可以通过调用 setState，来修改 state 的值，像这样：

复制代码

```
setState(newState)
```

状态更新后会触发渲染层面的更新，这点和类组件是一致的。

这里需要向初学者强调的一点是：**状态和修改状态的 API 名都是可以自定义的**。比如在上文的 Demo 中，就分别将其自定义为 text 和 setText：

复制代码

```
const [text, setText] = useState("初始文本");
```

“set + 具体变量名”这种命名形式，可以帮助我们快速地将 API 和它对应的状态建立逻辑联系。

当我们在函数组件中调用 React.useState 的时候，实际上是给这个组件关联了一个状态——注意，是“一个状态”而不是“一批状态”。这一点是相对于类组件中的 state 来说的。在类组件中，我们定义的 state 通常是一个这样的对象，如下所示：

复制代码

```
this.state {
  text: "初始文本",
  length: 10000,
  author: ["xiuyan", "cuicui", "yisi"]
}
```

这个对象是“包容万物”的：整个组件的状态都在 state 对象内部做收敛，当我们需要某个具体状态的时候，会通过 this.state.xxx 这样的访问对象属性的形式来读取它。

而在 useState 这个钩子的使用背景下，state 就是单独的一个状态，它可以是任何你需要的 JS 类型。像这样：

复制代码

```
// 定义为数组
const [author, setAuthor] = useState(["xiuyan", "cuicui", "yisi"]);

// 定义为数值
const [length, setLength] = useState(100);
// 定义为字符串
```

```
const [text, setText] = useState("初始文本")
```

你还可以定义为布尔值、对象等，都是没问题的。**它就像类组件中 state 对象的某一个属性一样，对应着一个单独的状态**，允许你存储任意类型的值。

useEffect()：允许函数组件执行副作用操作

函数组件相比于类组件来说，最显著的差异就是 state 和生命周期的缺失。useState 为函数组件引入了 state，而 useEffect 则在一定程度上弥补了生命周期的缺席。

useEffect 能够为函数组件引入副作用。过去我们习惯放在 componentDidMount、componentDidUpdate 和 componentWillUnmount 三个生命周期里来做的事，现在可以放在 useEffect 里来做，比如操作 DOM、订阅事件、调用外部 API 获取数据等。

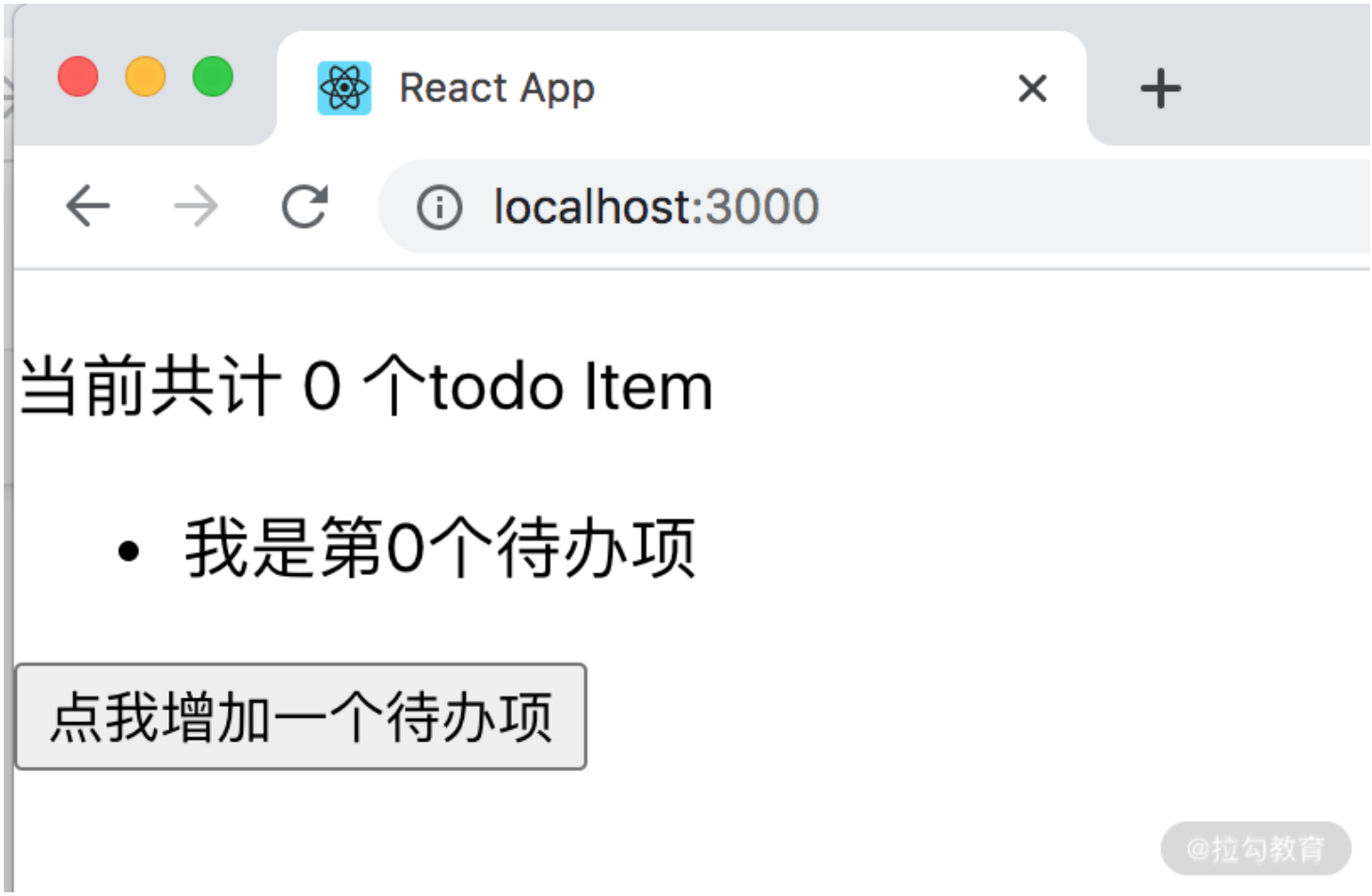
useEffect 和生命周期函数之间的"替换"关系

我们可以通过下面这个例子来理解 useEffect 和生命周期函数之间的替换关系。这里我先给到你一个用 useEffect 编写的函数组件示例：

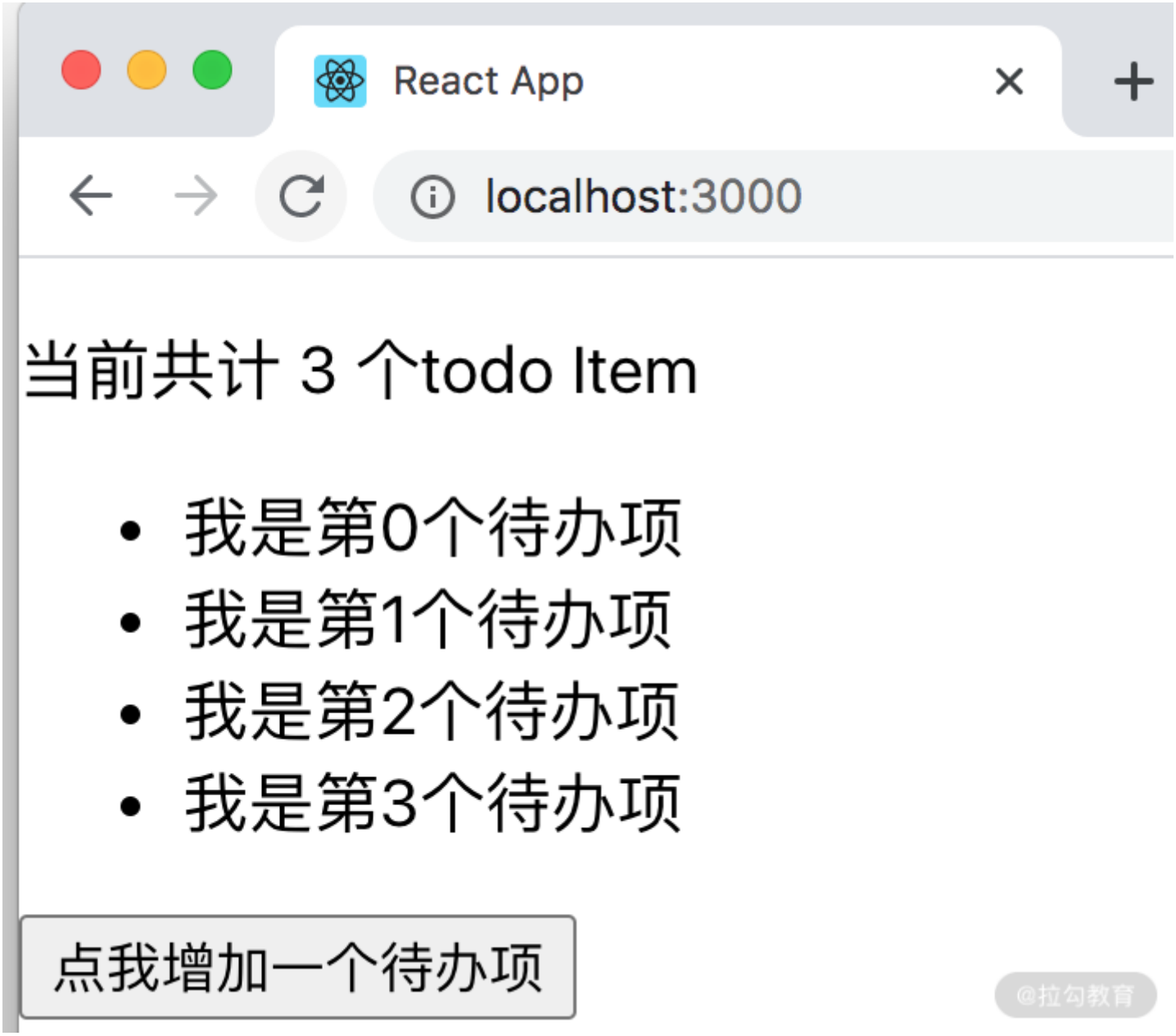
复制代码

```
// 注意 hook 在使用之前需要引入
import React, { useState, useEffect } from 'react';
// 定义函数组件
function IncreasingTodoList() {
  // 创建 count 状态及其对应的状态修改函数
  const [count, setCount] = useState(0);
  // 此处的定位与 componentDidMount 和 componentDidUpdate 相似
  useEffect(() => {
    // 每次 count 增加时，都增加对应的待办项
    const todoList = document.getElementById("todoList");
    const newItem = document.createElement("li");
    newItem.innerHTML = `我是第${count}个待办项`;
    todoList.append(newItem);
  });
  // 编写 UI 逻辑
  return (
    <div>
      <p>当前共计 {count} 个todo Item</p>
      <ul id="todoList"></ul>
      <button onClick={() => setCount(count + 1)}>点我增加一个待办项</button>
    </div>
  );
}
```

通过上面这段代码构造出来的界面在刚刚挂载完毕时，就是如下图所示的样子：



IncreasingTodoList 是一个只允许增加 item 的 ToDoList（待办事项列表）。按照 useEffect 的设定，每当我们点击“点我增加一个待办项”这个按钮，驱动 count+1 的同时，DOM 结构里也会被追加一个 li 元素。以下是连击按钮三次之后的效果图：



同样的效果，按照注释里的提示，我们也可以通过编写 class 组件来实现：

复制代码

```
import React from 'react';
// 定义类组件
class IncreasingTodoList extends React.Component {

  // 初始化 state
  state = { count: 0 }
  // 此处调用上个 demo 中 useEffect 中传入的函数
  componentDidMount() {
    this.addTodoItem()
  }
  // 此处调用上个 demo 中 useEffect 中传入的函数
  componentDidUpdate() {
    this.addTodoItem()
  }
  // 每次 count 增加时，都增加对应的待办项
```

```
  addTodoItem = () => {
    const { count } = this.state
    const todoList = document.getElementById("todoList")
    const newItem = document.createElement("li")
    newItem.innerHTML = `我是第${count}个待办项`
    todoList.append(newItem)
  }

  // 定义渲染内容
  render() {
    const { count } = this.state
    return (
      <div>
        <p>当前共计 {count} 个todo Item</p>
        <ul id="todoList"></ul>
        <button
          onClick={() =>
            this.setState({
              count: this.state.count + 1,
            })
          }
        >
          点我增加一个待办项
        </button>
      </div>
    )
  }
}
```

通过这样一个对比，类组件生命周期和函数组件 useEffect 之间的转换关系可以说是跃然纸上了。

在这里，我提个醒：初学 useEffect 时，我们难免习惯于借助对生命周期的理解来推导对 useEffect 的理解。但长期来看，若是执着于这个学习路径，无疑将阻碍你真正从心智模式的层面拥抱 React-Hooks。

有时候，我们必须学会忘记旧的知识，才能够更好地拥抱新的知识。对于每一个学习 useEffect 的人来说，生命周期到 useEffect 之间的转换关系都不是最重要的，最重要的是在脑海中构建一个“组件有副作用 → 引入 useEffect”这样的条件反射——**当你真正抛却类组件带给你的刻板印象、拥抱函数式编程之后，想必你会更加认同“useEffect 是用于为函数组件引入副作用的钩子”这个定义**。

useEffect 快速上手

useEffect 可以接收两个参数，分别是回调函数与依赖数组，如下面代码所示：

```
useEffect(callBack, [])
```

复制代码

useEffect 用什么姿势来调用，本质上取决于你想用它来达成什么样的效果。下面我们就以效果为线索，简单介绍 useEffect 的调用规则。

- 每一次渲染后都执行的副作用：传入回调函数，不传依赖数组。调用形式如下所示：

```
useEffect(callBack)
```

复制代码

- 仅在挂载阶段执行一次的副作用：传入回调函数，**且这个函数的返回值不是一个函**数，同时传入一个空数组。调用形式如下所示：

```
useEffect(()=>{
  // 这里是业务逻辑
}, [])
```

复制代码

- 仅在挂载阶段和卸载阶段执行的副作用：传入回调函数，**且这个函数的返回值是一个函数**，同时传入一个空数组。假如回调函数本身记为 A， 返回的函数记为 B，那么将在挂载阶段执行 A，卸载阶段执行 B。调用形式如下所示：

```
useEffect(()=>{
  // 这里是 A 的业务逻辑

  // 返回一个函数记为 B
  return ()=>{
  }
}, [])
```

复制代码

这里需要注意，这种调用方式之所以会在卸载阶段去触发 B 函数的逻辑，是由 useEffect 的执行规则决定的：**useEffect 回调中返回的函数被称为“清除函数”**，当 React 识别到清除函数时，会在卸载时执行清除函数内部的逻辑。**这个规律不会受第二个参数或者其他因素的影响，只要你在 useEffect 回调中返回了一个函数，它就会被作为清除函数来处理**。

- 每一次渲染都触发，且卸载阶段也会被触发的副作用：传入回调函数，且这个函数的返回值是一个函数，同时不传第二个参数。如下所示：

```
useEffect(()=>{
  // 这里是 A 的业务逻辑

  // 返回一个函数记为 B
  return ()=>{
  }
})
```

复制代码

上面这段代码就会使得 React 在每一次渲染都去触发 A 逻辑，并且在卸载阶段去触发 B 逻辑。

其实你只要记住，如果你有一段副作用逻辑需要在卸载阶段执行，那么把它写进 useEffect 回调的返回函数（上面示例中的 B 函数）里就行了。也可以认为，这个 B 函数的角色定位就类似于生命周期里 componentWillUnmount 方法里的逻辑（虽然并不推荐你再继续钻生命周期的牛角尖，哈哈）。

- 根据一定的依赖条件来触发的副作用：传入回调函数（若返回值是一个函数，仍然仅影响卸载阶段对副作用的处理，此处不再赘述），同时传入一个非空的数组，如下所示：

```
useEffect(()=>{
  // 这是回调函数的业务逻辑

  // 若 xxx 是一个函数，则 xxx 会在组件卸载时被触发
  return xxx
}, [num1, num2, num3])
```

复制代码

这里我给出的一个示意数组是 [num1, num2, num3]。首先需要说明，数组中的变量一般都是来源于组件本身的数据（props 或者 state）。若数组不为空，那么 React 就会在新的一次渲染后去对比前后两次的渲染，查看数组内是否有变量发生了更新（只要有一个数组元素变了，就会被认为更新发生了），并在有更新的前提下去触发 useEffect 中定义的副作用逻辑。

Why React-Hooks：Hooks 是如何帮助我们升级工作模式的

在第 06 课时我们已经了解到，函数组件相比类组件来说，有着不少能够利好 React 组件开发的特性，而 React-Hooks 的出现正是为了强化函数组件的能力。现在，基于对 React-Hooks 编码层面的具体认知，想必你对“动机”的理解也已经上了一个台阶。这里我们就趁热打铁，针对“Why React-Hooks”这个问题，做一个加强版的总结。

相信有不少嗅觉敏锐的同学已经感觉到了——没错，这个环节就是手把手教你做“为什么需要 React-Hooks”这道面试题。以“Why xxx”开头的这种面试题，往往都没有标准答案，但会有一些关键的“点”，只要能答出关键的点，就足以证明你思考的方向是正确的，也就意味着这道题能给你加分。这里，我梳理了以下 4 条答题思路：

- 告别难以理解的 Class；
- 解决业务逻辑难以拆分的问题；
- 使状态逻辑复用变得简单可行；
- 函数组件从设计思想上来看，更加契合 React 的理念。

关于思路 4，我在上个课时已经讲得透透的了，这里我主要是借着代码的东风，把 1、2、3 摊开来给你看一下。

1. 告别难以理解的 Class：把握 Class 的两大“痛点”

坊间总有传言说 Class 是“难以理解”的，这个说法的背后**是 this 和生命周期这两个痛点**。

先来说说 this，在上个课时，你已经初步感受了一把 this 有多么难以捉摸。但那毕竟是个相对特殊的场景，更为我们所熟悉的，可能还是 React 自定义组件方法中的 this。看看下面这段代码：

复制代码

```
class Example extends Component {
  state = {
    name: '修言',
    age: '99';
  };
  changeAge() {
    // 这里会报错
    this.setState({
      age: '100'
    });
  }
  render() {
    return <button onClick={this.changeAge}>{this.state.name}的年龄是{this.state.
  }
}
```

你先不用关心组件具体的逻辑，就看 changeAge 这个方法：它是 button 按钮的事件监听函数。当我点击 button 按钮时，希望它能够帮我修改状态，但事实是，点击发生后，程序会报错。原因很简单，changeAge 里并不能拿到组件实例的 this，至于为什么拿不到，我们将在第 15课时讲解其背后的原因，现在先不用关心。单就这个现象来说，略有一些 React 开发经验的同学应该都会非常熟悉。

为了解决 this 不符合预期的问题，各路前端也是各显神通，之前用 bind、现在推崇箭头函数。但不管什么招数，**本质上都是在用实践层面的约束来解决设计层面的问题**。好在现在有了 Hooks，一切都不一样了，我们可以在函数组件里放飞自我（毕竟函数组件是不用关心 this 的）哈哈，解放啦！

至于生命周期，它带来的麻烦主要有以下两个方面：

- 学习成本
- 不合理的逻辑规划方式

对于第一点，大家都学过生命周期，都懂。下面着重说说这“不合理的逻辑规划方式”是如何被 Hooks 解决掉的。

2. Hooks 如何实现更好的逻辑拆分

在过去，你是怎么组织自己的业务逻辑的呢？我想多数情况下应该都是先想清楚业务的需要是什么样的，然后将对应的业务逻辑拆到不同的生命周期函数里去——没错，**逻辑曾经一度与生命周期耦合在一起**。

在这样的前提下，生命周期函数常常做一些奇奇怪怪的事情：比如在 componentDidMount 里获取数据，在 componentDidUpdate 里根据数据的变化去更新 DOM 等。如果说你只用一个生命周期做一件事，那好像也还可以接受，但是往往在一个稍微成规模的 React 项目中，一个生命周期不止做一件事情。下面这段伪代码就很好地诠释了这一点：

复制代码

```
componentDidMount() {
  // 1. 这里发起异步调用
  // 2. 这里从 props 里获取某个数据，根据这个数据更新 DOM

  // 3. 这里设置一个订阅

  // 4. 这里随便干点别的什么

  // ...
}
componentWillUnMount() {
  // 在这里卸载订阅
}
componentDidUpdate() {
  // 1. 在这里根据 DidMount 获取到的异步数据更新 DOM

  // 2. 这里从 props 里获取某个数据，根据这个数据更新 DOM（和 DidMount 的第2步一样）
}
```

像这样的生命周期函数，它的体积过于庞大，做的事情过于复杂，会给阅读和维护者带来很多麻烦。最重要的是，**这些事情之间看上去毫无关联，逻辑就像是被“打散”进生命周期里了一样**。比如，设置订阅和卸载订阅的逻辑，虽然它们在逻辑上是有强关联的，但是却只能被分散到不同的生命周期函数里去处理，这无论如何也不能算作是一个非常合理的设计。

而在 Hooks 的帮助下，我们完全可以把这些繁杂的操作**按照逻辑上的关联拆分进不同的函数组件里：我们可以有专门管理订阅的函数组件、专门处理 DOM 的函数组件、专门获取数据的函数组件等。Hooks 能够帮助我们实现业务逻辑的聚合，避免复杂的组件和冗余的代码**。

3. 状态复用：Hooks 将复杂的问题变简单

过去我们复用状态逻辑，靠的是 HOC（高阶组件）和 Render Props 这些组件设计模式，这是因为 React 在原生层面并没有为我们提供相关的途径。但这些设计模式并非万能，它们在实现逻辑复用的同

时，也破坏着组件的结构，其中一个最常见的问题就是“嵌套地狱”现象。

Hooks 可以视作是 React 为解决状态逻辑复用这个问题所提供的一个原生途径。现在我们可以通过自定义 Hook，达到既不破坏组件结构、又能够实现逻辑复用的效果。

要理解上面这两段话，需要你对组件设计模式有基本的理解和应用。如果你读下来觉得一头雾水，也不必心慌。对于组件状态复用这个问题（包括 HOC、Render Props 和自定义 Hook），现在我对你的预期是“知道有这回事就可以了”。如果你实在着急，可以先通过文档中的相关内容简单了解一下。在专栏的第三模块，我会专门把这块知识提出来，放在一个更合适的上下文里给你掰开来讲。

保持清醒：Hooks 并非万能

尽管我们已经说了这么多 Hooks 的“好话”，尽管 React 团队已经用脚投票表明了对函数组件的积极态度，但我们还是要谨记这样一个基本的认知常识：事事无绝对，凡事皆有两面性。更何况 React 仅仅是推崇函数组件，并没有“拉踩”类组件，甚至还官宣了“类组件和函数组件将继续共存”这件事情。这些都在提醒我们，**在认识到 Hooks 带来的利好的同时，还需要认识到它的局限性**。

关于 Hooks 的局限性，目前社区鲜少有人讨论。这里我想结合团队开发过程当中遇到的一些瓶颈，和你分享实践中的几点感受：

- **Hooks 暂时还不能完全地为函数组件补齐类组件的能力**：比如 getSnapshotBeforeUpdate、componentDidCatch 这些生命周期，目前都还是强依赖类组件的。官方虽然立了“会尽早把它们加进来”的 Flag，但是说真的，这个 Flag 真的立了蛮久了……（扶额）
- **“轻量”几乎是函数组件的基因，这可能会使它不能够很好地消化“复杂”**：我们有时会在类组件中见到一些方法非常繁多的实例，如果用函数组件来解决相同的问题，业务逻辑的拆分和组织会是一个很大的挑战。我个人的感觉是，从头到尾都在“过于复杂”和“过度拆分”之间摇摆不定，哈哈。耦合和内聚的边界，有时候真的很难把握，**函数组件给了我们一定程度的自由，却也对开发者的水平提出了更高的要求**。
- **Hooks 在使用层面有着严格的规则约束**：这也是我们下个课时要重点讲的内容。对于如今的 React 开发者来说，如果不能牢记并践行 Hooks 的使用原则，如果对 Hooks 的关键原理没有扎实的把握，很容易把自己的 React 项目搞成大型车祸现场。

总结

在本课时，我们结合编码层面的认知，辩证地探讨了 Hooks 带来的利好与局限性。现在，你对于 React-Hooks 的基本形态和前世今生都已经有了透彻的了解，也真刀真枪地感受到了 Hooks 带来的利

好。学习至此，相信你已经建立了对 React-Hooks 的学习自信。

接下来，我们将续上本课时结尾处的“悬念”，向 React-Hooks 的执行规则发问，同时也将进入 React-Hooks 知识链路真正的深水区。