React 笔记

Pionpill [1]

本文档为作者学习 React 及相关技术的笔记。

2022 年 12 月 23 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

React 是 Facebook 开源的目前最主流的前端框架，其核心理念是 all-in-js。React 本身并不复杂，只需一定的 js 基础就可以很快入门。

本文撰写环境:

- React: 18.2.0
- IDE: VSCode 1.72
- Chrome: 91.0
- Node.js: 18.12
- OS: Window11

2022 年 12 月 23 日

目录

第一部分

React

I React 基础篇

1 React 简介

前端 UI 的本质问题是如何将来源于服务器端的动态数据和用户的交互行为高效地反映到复杂的用户界面上。React 另辟蹊径，通过引入虚拟 DOM、状态、单向数据流等设计理念，形成以组件为核心，用组件搭建 U 的开发模式。

React 的特点可以归结为以下 4 点:

- **声明式视图层**: React 采用 JSX 语法来声明视图层，可以在视图中绑定各种状态数据以及相关操作。
- **简易更新流**: 声明式的视图定义方式有助于简化视图层的更新流程。你只需要定义 u 状态，React 便会负责把它渲染成最终的 UI。
- **灵活渲染实现**: React 并不是把视图直接渲染成最终的终端界面，而是先把它们渲染成虚拟 DOM。虚拟 DOM 再在各个平台渲染 (react-dom 对应浏览器；Node 对应服务端；React Native 对应移动端)。
- **高效的 DOM 操作**: 基于 React 优异的差异比较算法，React 可以尽量减少虚拟 DOM 到真实 DOM 的渲染次数，以及每次渲染需要改变的真实 DOM 节点数。

1.1 开发环境

React 应用开发有两个必要环境:

- **Node.js**: React 在本地开发调试需要使用到 Node.js 环境中的 NPM，Webpack 等依赖。
- **NPM**: 模块管理工具，用来管理模块之间的依赖关系。也可以用 **Yarn** 代理，安装 Node。

辅助工具:

- **Webpack**: 模块打包工具，不仅可以打包 JS 文件，配合相关插件的使用，它还可以打包图片资源和样式文件，已经具备一站式的 JavaScript 应用打包能力，是 React 开发的必要工具。
- **Babel**: Babel 是一个 JavaScript 编译器，为了浏览器兼容性考虑，需要把 ES6 或以后的语法编译成 ES5 及之前的语法达到向前兼容的目的。
- **ESLint**: JavaScript 代码检查工具，由于 JS 的语法非常乱，同一种实现有多种写法，为了团队同一管理，会用 ESLint 进行风格检测。

这些工具的使用方法比较繁琐，可以直接用 React 提供的脚手架工具构建工程。在官方文档中提供了以下几个命名用于快速构建工程:

```
# 在当前目录下创建 my-app 项目
```

```
npx create-react-app my-app
# 运行项目
cd my-app
npm start
```

通过这种方式创建的 React 项目结构如下 (仅重要文件/文件夹):

```
my-app
|- README.md          # react 相关的指令介绍(可删除)
|- .gitignore         # 版本控制
|- package.json       # 项目信息
|- package-lock.json  # 项目绑定信息
|- node_modules       # 工程依赖的模块，会被 .gitignore 忽视
|- public             # 外部访问文件
    |- index.html     # 应用入口界面
    |- manifest.json  # 应用注册信息
|- src            # 项目源代码，主要工作区
    |- index.js    # 源代码入口
    |- react-app-env  # 应用变量环境
```

放入 public 文件夹下的资源可以被直接引用。

还有很多其他文件，但主要的，启动一个项目会进入 `public/index.html` 界面，而这个界面一般加载了 `src/index.js` 脚本。开发者一般在 `src` 文件夹中写入功能。

此外，如果使用 typescript 开发，创建项目指令如下:

```
npx create-react-app my-app --template typescript
```

会新增几个 ts 管理文件。

2 组件基础，类组件，基础用法

2.1 JSX

JSX 是一种用于描述凹的 JavaScript 扩展语法，React 使用这种语法描述组件的 UI, 在 TypeScript 可以使用 TSX。

React 的一个核心思想是 all-in-js，即脚本控制所有。React 认为 HTML，CSS，JS 三者天生就存在耦合关系；与其分离三者各自书写长段的代码，不如由 JS 统一控制三者构成组件，让组件功能单一化。Vue 也采用了这种策略，只不过更多 Vue 开发者倾向于使用 template，而不是完全由 js 代替操作。

JSX 的语法和 XML 相同，都是使用成对的标签构成一个树状结构的数据，例如：

```
const element = (
    <div>
        <h1>Hello, World!</h1>
    </div>
)
```

JSX 有以下语法规范:

- 节点数必须仅有一个根标签，通常是 `<div>` 或用 React 定义的 `<Wrapper>`。
- Html 原生标签用小写表示，React 定义的标签用大写表示。除此以外，React 定义的标签与原生标签使用上没有区别。
- 在 JSX 中使用 JavaScript 表达式需要用大括号包起来。且至多包括一条语句 (可以是箭头函数)。如果是纯字符串则正常使用。
- 格式上，如果只有一个标签可以不写 (), 否则 JSX 需要被包含在 () 中。

由于 js 语法和 Html 语法关键字有重合，例如 `class`。因此部分属性的名称会有所改变，主要的变化有：HTML 中的 `class` 要写成 `className`；事件属性要用小驼峰命名法，例如 `onclick -> onClick`。此外 JSX 中的注释需要用大括号包起来: `{/**/}`。

JSX 的本质

JSX 本质上只是一种语法糖，一般的，用.js 文件写 JSX 语法也不会有问题，`Node` 编译器会自动识别。习惯上，我们将含有 JSX 语法的文件命名为 `.jsx` 以表示存在 UI 组件，其他脚本逻辑则保留在 `.jx` 文件中。

下面是 JSX 语法转换后的语句:

```
// JSX
const element = <div className= 'foo' >Hello, React</div>
// 转换后
const element = React.createElement('div',{className:'foo'},'Hello, React');
```

2.2 类组件

2.2.1 定义组件

组件是 React 的核心概念，是 React 应用程序的基石。组件将应用的 U 拆分成独立的、可复用的模块，React 应用程序正是由一个一个组件搭建而成的。

定义一个组件有两种方式，由 ES6 `class` 定义类组件或用函数组件。现在主流的方案是函数组件 +Hooks。

类组件

使用 `class` 定义类组件有两个条件:

- `class` 继承自 `React.Component`。
- `class` 内部定义 `render` 方法，用于返回该组件的 UI 元素 (一般用 JSX 语法)。

```
import React, { Component } from "react";

class PostList extends Component {
  render() {
    return (
      <div>
        <span>Learn</span>
        <ul>
          <li>JSX</li>
          <li>React-DOM</li>
          <li>Redux</li>
        </ul>
      </div>
    );
  }
}

export default PostList;
```

这样我们就定义了一个 PostList 组件，只要引入它，就可以使用 `<PostList>` 标签。

将其挂载到 DOM 节点上:

```
import React from "react";
import ReactDOM from "react-dom";
import PostList from "./PostList";

ReactDOM.render(<PostList />, document.getElementById("root"));
```

这样，我们可以将一份复杂的 HTML 节点数分解成部分可重用的 React 组件。

函数组件

由于 JS 定义函数的方法特别多，这里只写最主流的方案:

```
const PostList = (props) => {
  return (
    <div>
      <span>Learn</span>
      <ul>
        <li>JSX</li>
        <li>React-DOM</li>
        <li>Redux</li>
      </ul>
    </div>
  );
}
```

函数组件无法获取 state 和自身的生命周期，需要通过 React v16.8 给出的 Hooks，下文均以类组件为例，后面会单独介绍 Hooks。

组件与元素

React 组件可以看作一个 Html 标签，React 元素则是一个普通的 JavaScript 对象。在 JSX 中可以这样写:

```
// 组件
<div>
  <CustomButton/>
</div>
// 元素
<div>
  {CustomButton}
</div>
```

2.2.2 组件的 `props`

就前面的例子而言，如果我们要重用 `PostList` 组件，需要修改里面的内容怎么办，重新定义一个 `PostList2` 组件，继续用硬编码的方式写入内容？这显然违背了重用的理念。

React 定义的组件允许我们自定义标签属性 (HTML 中标签的属性，为了区分对象属性下文称为标签属性)。组件的 `props` 属性用于把父组件中的数据或方法传递给子组件。在类组件和函数组件中 `props` 调用方式不同:

- 在类组件中，继承自 `React.Component` 的组件会有一个 `this.props` 调用属性。
- 在函数组件中，函数的唯一参数 `props` 代表了属性。

我们重写上面的代码 (有部分改动):

```
class PostList extends Component {
  render() {
```

```
    const {title, author, date} = this.props; // 解构
    return (
      <div>
        <span>{title}</span>
        <ul>
          <li>{author}</li>
          <li>{date}</li>
        </ul>
      </div>
    );
  }
}
```

在调用时，我们只需要通过标签属性就可以指定 props 的值，React 会自动将自定义组件的标签属性装入 props 属性中:

```
<PostList title="React" author="Pionpill" date="2022-12-21">
```

不过，无论是函数组件还是方法组件都不能修改 props 的值。

React 提供了 PropTypes 这个对象，用于校验组件属性的类型。PropTypes 包含组件属性所有可能的类型，我们通过定义一个映射对象实现组件属性类型的校验。

```
import PropTypes from 'prop-types';

class PostItem extends React. Component{}

PostItem.propTypes = {
  post: PropTypes.object,
  onVote: PropTypes.func
};
```

如果属性值是对象或者数值,我们仍然无法知道其内部具体的数据,这时可以使用 PropTypes.shape 或 PropTypes.arrayOf 方法:

```
style: PropTypes.shape ({
  color:  PropTypes.string,
  fontSize: PropTypes.number
}),
sequence: PropTypes.arrayOf(PropTypes.number)
```

2.2.3 组件的 state

props 代表组件的外部状态，标签属性是外部传输进来的，只能调用不能修改。state 则表示内部状态，可以通过 setState() 方法对其进行修改。

在类组件中使用 state 的唯一方法是在构造方法 constructor 中通过 this.state 定义组件的初始状态并调用。

```
class PostList extends Component {
  constructor(props) {
```

```
    super(props);  // 这一句强制要求必须有
    this.state = {
      vote: 0
    };
  }

  handle() {
    let vote = this.state.vote; // 无法直接操作 state 值
    vote++;
    this.setState({
      vote: vote,
    });
  }

  render() {
    const {title, author, date} = this.props; // 解构
    return (
      <div>
        <button onClick={()=>this.handle()}><button> // 箭头函数
      </div>
    );
  }
}
```

操作 state 一般可分为以下三个步骤:

- 在构造函数中，通过 this.state 定义 state 的数据。
- 将对 state 的操作封装在函数中，且在函数中只能通过 setState 方法修改 state 值。
- 在 JSX 中调用函数，只能用箭头函数方式。

2.2.4 组件样式

React 可以将样式表当作一个模块，在组件中导入样式表并且使用:

```
import '.style.css' ;
function Welcome(props) {
  return <h1 className= 'foo' >Hello, {props.name}<hl>;
}
```

React 的核心理念之一是: all-in-js 因此，更推荐直接而在组件中定义样式:

```
function Welcome(props) {
  const style = {
    width: "100%",
    height: "50px",
    backgroundColor:"blue",
  };
  return <h1 style = {style}>Hello World!</h1>
}
```

React 原生定义内联样式属性必须采用小驼峰法命名。如果采用这种方案更推荐使用支

持 React 的 CSS 框架, 比如 `styled-component`。

2.2.5 组件的生命周期

通常，组件的生命周期可以被分为三个阶段：挂载阶段、更新阶段、卸载阶段。

挂载阶段组件被创建，执行初始化，并被挂载到 ODM 中，完成组件的第一次渲染。依次调用的生命周期方法有:

- **`constructor`**

  `class` 的构造方法,接收一个 `props` 参数,必须在这个方法中首先调用 `super(props)` 才能保证 `props` 被传入组件中。`constructor` 通常用于初始化组件的 `state` 以及绑定事件处理方法等工作。

- **`componentWillMount`**

  在组件被挂载到 DOM 前调用，且只会被调用一次。这个方法在实际项目中很少会用到，因为可以在该方法中执行的工作都可以提前到 `constructor` 中。在这个方法中调用 `this.setState` 不会引起组件的重新渲染。

- **`render`**

  唯一必要的方法 (其他方法可以省略)，根据组件的 `props` 和 `state` 返回一个 React 元素，用于描述组件的 UI。render 并不负责组件的实际渲染工作，它只是返回一个 UI 的描述，真正的渲染出页面 DOM 的工作由 React 自身负责。`render` 是一个纯函数，在这个方法中不能执行任何有副作用的操作，所以不能在 `render` 中调用 `this.setState`，这会改变组件的状态。

- **`componentDidMount`**

  在组件被挂载到 DOM 后调用，且只会被调用一次。这时候已经可以获取到 DOM 结构，因此依赖 DOM 节点的操作可以放到这个方法中。这个方法通常还会用于向服务器端请求数据。在这个方法中调用 `this.setState` 会引起组件的重新渲染。

组件被挂载到 DOM 后，组件的 `props` 或 `state` 可以引起组件更新。`props` 引起的组件更新，本质上是由渲染该组件的父组件引起的，也就是当父组件的 `render` 方法被调用时，组件会发生更新过程；不过无论 `props` 是否改变，只要调用 `render` 就会引起组件更新。`state` 引起的组件更新，是通过调用 `this.setState` 修改组件 `state` 触发的。组件更新阶段调用生命周期方法有:

- **`componentWillReceiveProps(nextProps)`**

  只在 `props` 引起的组件更新过程中，才会被调用。方法的参数 `nextProps` 是父组件传递给当前组件的新的 `props`。往往需要比较 `nextProps` 和 `this.props` 来决定是否执行 `props` 发生变化后的逻辑，比如根据新的 `props` 调用 `this.setState` 触发组件的重新渲染。

  组件更新过程中，只有在组件 `render` 及其之后的方法中，`this.state` 指向的才是更新后的 `state`。在 `render` 之前的方法，`this.state` 依然指向更新前的 `state`。

  `setState` 方法不会触发该方法，否则可能会进入死循环，毕竟该方法更新机制之

一就是调用 setState。

- **shouldComponentUpdate(nextProps, nextState)**

  该方法通过比较 nextProps, nextState 决定是否要执行更新过程，返回布尔值。当方法返回 false 时，组件的更新过程停止，后续的方法也不会再被调用。该方法可以减少不必要的渲染，从而优化组件性能。

- **componentWillUpdate(nextProps, nextState)**

  作为组件更新发生前执行某些工作的地方，一般很少用到。这里不能调用 setState，否则可能引起循环问题，下一个也是。

- **Render**

- **componentDidUpdate(prevProps, prevState)**

  组件更新后被调用，可以作为操作更新后的 DOM 的地方。两个参数代表更新前的 props 和 state。

卸载阶段，只有一个生命周期方法:

- **componentWillUnmount**

  在组件被卸载前调用，可以在这里执行一些清理工作，比如清除组件中使用的定时器，清除 componentDidMount 中手动创建的 DOM 元素等，以避免引起内存泄漏。

只有类组件才具有生命周期方法，函数组件是没有生命周期方法的。

2.2.6 React16 新特性

render 新的返回类型

React 16 之前 render 方法必须返回单个元素，现在 render 方法支持两种新的返回类型: 数组和字符串:

```
// 返回数组
class ListComponent extends Component{
  render() {
    return [
      <li key= ”A” >First item</li>,
      <li key=”B”>Second item</li>,
      <li key=”C”>Third item</li>
    ];
  }
}
// 返回字符串
class StringComponent extends Component {
  render () {
    return "Just a strings";
  }
}
```

错误处理

React 16 之前，组件在运行期间如果执行出错，就会阻塞整个应用的渲染，这时候只能刷新页面才能恢复应用。React16 引入了新的错误处理机制，默认情况下，当组件中抛出错误时，这个组件会从组件树中卸载，从而避免整个应用的崩溃。

这种方式比起之前的处理方式有所进步，但用户体验依然不够友好。React16 还提供了一种更加友好的错误处理方式——错误边界（ErrorBoundaries）。错误边界是能够捕获子组件的错误并对其做优雅处理的组件。

定义了 `componentDidCatch(error, info)` 这个方法的组件将成为一个错误边界:

```
class ErrorBoundary extends React.Component {
  // ......
  componentDidCatch(error, info) {
    // 输出错误日志
    console.log(error, info);
  }
}
```

Portals

React 16 的 Portals 特性让我们可以把组件渲染到当前组件树以外的 DOM 节点上，这个特性典型的应用场景是渲染应用的全局弹框，使用 Portals 后，任意组件都可以将弹框组件渲染到根节点上，以方便弹框的显示。Portals 的实现依赖 ReactDOM 的一个新的 API:

```
ReactDOM.createPortal(child, container)
```

第一个参数 `child` 是可以被渲染的 React 节点，例如 React 元素、由 React 元素组成的数组、字符串等，`container` 是一个 DOM 元素，`child` 将被挂载到这个 DOM 节点。

自定义 DOM 属性

React 16 之前会忽略不识别的 HTML 和 SVG 属性，现在 React 会把不识别的属性传递给 DOM 元素。

```
// React16 之前
<div />
// React16 之后
<div custom-attribute=” something" />
```

2.3 组件技巧

2.3.1 列表与 Keys

看一下 JavaScript 的 `map()` 方法，它接受三个参数: `value, index, arr` 分别代表数据值，数据索引，数据所属的列表:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
// [2, 4, 6, 8, 10]
```

在 React 中把数组转换为元素的过程也是类似的：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

为什么要给 `<li>` 标签添加 `key` 属性呢? 因为 React 使用 `key` 属性来标记列表中的每个元素，当列表数据发生变化时，React 就可以通过 `key` 知道哪些元素发生了变化，从而只重新渲染发生变化的元素来提高渲染效率。

不要将 `index` 作为 `key`，这在列表重排时会引起性能问题。此外，`key` 只有在数组上下文才有含义。

列表可以直接通过 `map()` 嵌入带 JSX 中，利用前面提到的 React 元素语法:

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

2.3.2 事件处理

使用 HTML 标签绑定事件这样写:

```
<button onclick="handle()"> XXX </button>
```

使用 React 则需要这样写:

```
<button onclick={handle}> XXX </button>
```

此外，在 HTML 中绑定的事件可以通过返回 `false` 阻止事件发生，例如:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

而 React 则定义了一个专用的方法:

```
function handleClick(e) {
  e.preventDefault();
  console.log('The link was clicked.');
}
```

有一个地方需要注意，在 JavaScript 中，class 的方法默认不会绑定 this(注意，只是方法，属性还是会绑定)，因此在 JSX 中不能直接调用函数，但是可以将函数视作属性调用。

针对 JavaScript 复杂的 `this` 指向问题，React 定义了三种处理函数书写方案:

- **箭头函数**

  箭头函数中的 `this` 指向的是函数定义时的对象，所以可以保证 `this` 总是指向当前组件的实例对象。但如果直接在箭头函数中写逻辑会让代码变得很乱，因此通常利用箭头函数 `this` 的特性传递真正的函数:

  ```
  render (
    <button onClick={ (event)=>{this.handleClick(event);} }> Button </button>
  )
  ```

  直接在 `render` 方法中为元素事件定义事件处理函数，最大的问题是，每次 `render` 调用时，都会重新创建一个新的事件处理函数，带来额外的性能开销，组件所处层级越低，这种开销就越大，因为任何一个上层组件的变化都可能会触发这个组件的 `render` 方法。一般情况下这种开销不必在意。

- **组件方法**

  直接将组件的方法赋值给元素的事件属性，同时在类的构造函数中，将这个方法的 `this` 绑定到当前对象。

  ```
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  ```

  这种方式的好处是每次 `render` 不会重新创建一个回调函数，没有额外的性能损失。但在构造函数中，为事件处理函数绑定 `this`，尤其是存在多个事件处理函数需要绑定时，这种模板式的代码还是会显得烦琐。

- **属性初始化语法**

  使用 ES7 的 `propertyinitializers` 会自动为 `class` 中定义的方法绑定 `this`。

```
handleClick = (event) => {
  console.log("111");
}

render {
  return (
    <button onClick={this.handleClick}> Button </button>
  )
}
```

2.3.3 表单

在 HTML 中，有些元素如表单元素自身维护着一些状态 (输入的内容)，这些状态默认情况下是不受 React 控制的。我们称这类状态不受 React 控制的表单元素为非受控组件。在 React 中，状态的修改必须通过组件的 `state`，非受控组件的行为显然有悖于这一原则。为了让表单元素状态的变更也能通过组件的 `state` 管理，React 采用受控组件的技术达到这一目的。

如果一个表单元素的值是由 React 来管理的，那么它就是一个受控组件。React 组件渲染表单元素，并在用户和表单元素发生交互时控制表单元素的行为，从而保证组件的 `state` 成为界面上所有元素状态的唯一来源:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('提交的名字: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          名字:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="提交" />
      </form>
    );
  }
```

```
}
```

由于在表单元素上设置了 `value` 属性，因此显示的值将始终为 `this.state.value`，这使得 React 的 `state` 成为唯一数据源。

常见的受控组件包括: `textarea`，`select`，`input`。

使用受控组件虽然保证了表单元素的状态也由 React 统一管理，但需要为每个表单元素定义 `onChange` 事件的处理函数，然后把表单状态的更改同步到 React 组件的 `state`，这一过程是比较烦琐的，一种可替代的解决方案是使用非受控组件。

使用非受控组件需要有一种方式可以获取到表单元素的值，React 中提供了一个特殊的属性 ref，用来引用 React 组件或 DOM 元素的实例。

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

状态提升

在 React 中，将多个组件中需要共享的 `state` 向上移动到它们的最近共同父组件中，便可实现共享 `state`。这就是所谓的“状态提升”。

实现状态提升的方案之一是在父组件中 `state` 存储数据。再将父组件的 `state` 传递给子组件的 `props`。很明显由于 `props` 被改变，子组件会从新 `render`。

3 Hook，函数组件，高级用法

最早支持 Hook 的 React 版本是 16.8。Hook 是完全可选的一项技术，Hook 技术使用与否不影响项目功能。但在本文撰写的 2022 年，函数组件配合 Hook 技术已经成为了主流写法; 同时这章会和借函数组件讲解一些高级用法，这些用法 (除了 Hook) 在类组件也同样适用。

在 React 官网对 Hook 的介绍中，有一句: *没有计划从 React 中移除 class*。由此可以看出 Hook 可以让函数组件实现所有的类组件功能，并且更为便捷高效。

3.1 基础 Hook

Hook 的本质就是 JavaScript 函数，它可以让你“钩入”React 的特性。`State Hook` 和 `Effect Hook` 是 Hook 的两个核心函数，分别让函数组件获取了 `state` 与生命周期。

3.1.1 State Hook

下面是使用 Hook 技术的一个函数组件:

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

function Example() {
  // 声明一个叫 “count” 的 state 变量。
  const [count, setCount] = useState(0);

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

在函数组件中，我们没有 `this`，所以我们不能分配或读取 `this.state`。我们直接在组件中调用 `useState` Hook。

`useState` 会返回一个有两个元素的数组：当前状态和一个让你更新它的函数。它类似类组件的 `this.setState`，但是它不会把新的 `state` 和旧的 `state` 进行合并 (因此它是 `const` 的，也可以直接操作 `state`)。`useState` 唯一的参数就是初始 `state`。

state 特性

前面我们说过，`state` 会 `props` 的改变通常会需要重新 `render` 组件。因此我们对 `state` 的操作必须十分小心，只有必要的，关联组件状态的数据才需要放在 `state` 中。

在上面的代码中，我们使用 `const` 修饰的数据接受了 `useState` 的返回值，这说明 `state`

数据本身是不可修改的，为此类组件中 React 专门提供了一个 setState 方法修改，Hook 也提供了对应的方法用于更新 state。

state 的更新是异步的，组件的 state 并不会立即改变，setState 只是把要修改的状态放入一个队列中，React 会优化真正的执行时机，并且出于性能原因，可能会将多次 setState 的状态修改合并成一次状态修改。因此不要依赖当前的 state 值计算下一个 state 值。

由于 state 本身应该是不可修改的对象，因此尽可能地使用 string，number 等数据类型，如果要用到数组或其他可变类型，则需要考虑是应该修改这些类型的数据，才是创建一个新的对应类型传给 state。最好的方法自然是使用 const 关键字。

3.1.2 Effect Hook

Effect Hook 对应的函数是 useEffect，它给函数组件增加了操作副作用的能力。它跟类组件中的 componentDidMount、componentDidUpdate 和 componentWillUnmount 具有相同的用途，只不过被合并成了一个 API。

```
// 相当于 componentDidMount 和 componentDidUpdate:
useEffect(() => {
  // 使用浏览器的 API 更新页面标题
  document.title = `You clicked ${count} times ${count}`;
});
```

当你调用 useEffect 时，就是在告诉 React 在完成对 DOM 的更改后 (对应生命周期:componentDidMount、componentDidUpdate) 运行你的“副作用”函数。由于副作用函数是在组件内声明的，所以它们可以访问到组件的 props 和 state。与类组件生命周期不同的是，useEffect 调度的 effect 不会阻塞浏览器更新屏幕。

useEffect 可以返回一个函数，React 将会在执行清除操作时调用它 (对应生命周期: componentWillUnmount)，因此常被命名为 cleanup。

在某些情况下，每次渲染后都执行清理或者执行 effect 可能会导致性能问题。在类组件中，我们可以通过在 componentDidUpdate 中添加对 prevProps 或 prevState 的比较逻辑解决，在 useEffect 中可以通过添加第二个可选参数实现相同效果:

```
useEffect(() => {
  document.title = `You clicked ${count} times ${count}`;
}, [count]); // 仅在 count 更改时更新
```

3.1.3 自定义 Hook

自定义 Hook 主要用于解决组件之间逻辑共享问题。当我们想在两个函数之间共享逻辑时，我们会把它提取到第三个函数中。而组件和 Hook 都是函数，所以也同样适用这种方式。

自定义 Hook 是一个函数，其名称以“use”开头，函数内部可以调用其他的 Hook:

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

```

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

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

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

  return isOnline;
}
```

与 React 组件不同的是，自定义 Hook 不需要具有特殊的标识。我们可以自由的决定它的参数是什么，以及它应该返回什么（如果需要的话）。换句话说，它就像一个正常的函数。但是它的名字应该始终以 `use` 开头，这样可以一眼看出其符合 Hook 的规则。

此处 `useFriendStatus` 的 Hook 目的是订阅某个好友的在线状态。这就是我们需要将 `friendID` 作为参数，并返回这位好友的在线状态的原因。

使用自定义 Hook 只需要在函数组件中调用即可。

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

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

3.2 进阶 Hook

3.2.1 Context Hook

Context 提供了一个无需为每层组件手动添加 `props`，就能在组件树间进行数据传递的方法。例如当前认证的用户、主题或首选语言。

```
// 调用 API 创建一个 Context
const ThemeContext = React.createContext('light');
function MyElement() {
  const theme = useContext(ThemeContext); // 使用 Context Hook
  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
```

```
  );
}
```

React 提供的关于 Context 的 API 有这些 (不包含 Context Hook):

- **`React.createContext`**

```
const MyContext = React.createContext(defaultValue);
```

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 `Provider` 中读取到当前的 `context` 值。只有当组件所处的树中没有匹配到 Provider 时，其 `defaultValue` 参数才会生效。这有助于在不使用 Provider 包装组件的情况下对组件进行测试。

- **`Context.Provider`**

```
<MyContext.Provider value={/* 某个值 */}>
```

每个 Context 对象都会返回一个 Provider React 组件，它允许消费组件订阅 context 的变化。Provider 接收一个 `value` 属性，传递给消费组件。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 `value` 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 `shouldComponentUpdate` 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

- **`Class.contextType`**

挂载在 class 上的 `contextType` 属性会被重赋值为一个由 React.createContext() 创建的 Context 对象。这能让你使用 `this.context` 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

- **`Context.Consumer`**

```
<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染*/}
</MyContext.Consumer>
```

这里，React 组件也可以订阅到 context 变更。这能让你在函数式组件中完成订阅 context。

- **`Context.displayName`**

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';
<MyContext.Provider> // "MyDisplayName.Provider" 在 DevTools 中
<MyContext.Consumer> // "MyDisplayName.Consumer" 在 DevTools 中
```

context 对象接受一个名为 displayName 的 property，类型为字符串。React DevTools 使用该字符串来确定 context 要显示的内容。

看看例子:

```
// theme-context.js
```

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};
export const ThemeContext = React.createContext(
  themes.dark // 默认值
);
// themed-button.js
import {ThemeContext} from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;
    return (
      <button
        {...props}
        style={{backgroundColor: theme.background}}
      />
    );
  }
}
ThemedButton.contextType = ThemeContext;
export default ThemedButton;
```

useHook 本质上就是替换了类组件中的 `static contextType = MyContext` 或者 `<MyContext.Consumer>`。仍然需要在上层组件树中使用 `<MyContext.Provider>` 来为下层组件提供 context。

3.2.2 Ref Hook

Ref 转发是一项将 ref 自动地通过组件传递到其一子组件的技巧。

看下面这个常规组件:

```
function FancyButton(props) {
  return (
    <button className="FancyButton">
      {props.children}
    </button>
  );
}
```

React 组件隐藏其实现细节，包括其渲染结果。其他使用 `FancyButton` 的组件通常不需要获取内部的 DOM 元素 `button` 的 `ref`。这很好，因为这防止组件过度依赖其他组件的 DOM

结构。

但是一旦组件变得复杂，当我们需要操作组件中的子元素时，由于 React 对组建的封装，如果直接操作原生 DOM 往往会无法轻易获取某个元素。Ref 转发是一个可选特性，其允许某些组件接收 ref，并将其向下传递（换句话说，“转发”它）给子组件。

在下面的示例中，`FancyButton` 使用 `React.forwardRef` 来获取传递给它的 `ref`，然后转发到它渲染的 DOM `button`：

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// 你可以直接获取 DOM button 的 ref:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

这样，使用 `FancyButton` 的组件可以获取底层 DOM 节点 `button` 的 ref，并在必要时访问，就像其直接使用 DOM `button` 一样。使用 Ref 转发后发生了如下事件:

- 我们通过调用 `React.createRef` 创建了一个 React ref 并将其赋值给 `ref` 变量。
- 我们通过指定 `ref` 为 JSX 属性，将其向下传递给 `<FancyButton ref=ref>`。
- React 传递 `ref` 给 forwardRef 内函数 `(props, ref) => ...`，作为其第二个参数。
- 我们向下转发该 `ref` 参数到 `<button ref=ref>`，将其指定为 JSX 属性。
- 当 `ref` 挂载完成，`ref.current` 将指向 <button> DOM 节点。

第二个参数 `ref` 只在使用 `React.forwardRef` 定义组件时存在。常规函数和 class 组件不接收 `ref` 参数，且 `props` 中也不存在 `ref`。

`useRef` 返回一个可变的 ref 对象，其 `.current` 属性被初始化为传入的参数（`initialValue`）。返回的 ref 对象在组件的整个生命周期内保持不变。

一个常见的用例便是命令式地访问子组件：

```
const TextInputWithFocusButton: React.FC = () => {
  const inputEl: MutableRefObject<any> = useRef(null)
  const handleFocus = () => {
    // `current` 指向已挂载到 DOM 上的文本输入元素
    inputEl.current.focus()
  }
  return (
    <p>
      <input ref={inputEl} type="text" />
      <button onClick={handleFocus}>Focus the input</button>
    </p>
  )
}
```

- 使用 `useRef` 创建一个 ref 并保存在 `inputEl` 中。

- 在 `return` 语句中，将 `inputEl` 通过 `ref` 标签属性绑定在 `<input>` 元素中。
- 在 `<button>` 点击事件中使用 `inputEl` 进而获得 `<input>` 元素进行控制。

本质上，`useRef` 就像是可以在其 `.current` 属性中保存一个可变值的“盒子”。

如果你将 ref 对象以 `<div ref=myRef />` 形式传入组件，则无论该节点如何改变，React 都会将 ref 对象的 `.current` 属性设置为相应的 DOM 节点。然而，`useRef()` 比 ref 属性更有用。它可以很方便地保存任何可变值。