React 笔记

Pionpill [1]

本文档为作者学习 React 及相关技术的笔记。

2023 年 3 月 10 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，需要的前置知识如下:

- HTML5
- CSS3
- JavaScript(ES6+)

最好拥有 TypeScript 基础，文中靠后的一些章节会使用到。

本文的主要参考文献如下:

- React 中文官网: `https://react.docschina.org/`
- Redux 中文官网: `https://cn.redux.js.org/`
- React 学习手册: [美]Alex Banks 安道译中国电力出版社 2021 (第二版)
- React 进阶之路: 徐超清华大学出版社 2018

本文撰写环境:

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

2023 年 3 月 10 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 React 开发环境

1.1.1 默认开发环境 (Webpack)

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。

另外还有几个辅助工具:

- **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，(过去) 是 React 开发的必要工具，现在可以使用 Vite 替换 Webpack，下文详解。
- **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，同时也负责编译 JSX 语法。
- **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，风格不尽相同，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。

create-react-app

`create-react-app` 是一个单独的包，用于快速构建 React 项目，开源地址: `https://github.com/facebook/create-react-app`。

但实际上我们使用改包创建完 React 项目后，`package` 文件中并没有对应的包信息，这与 `npx` 命名的执行逻辑有关。使用 `npx` 命令会自动查找当前依赖包中的可执行文件，如果没有就在 `PATH` 中寻找，再找不到就会帮助我们安装，且安装的包在使用后会被卸载。

也就是说我们使用上述指令后，`npx` 帮我们安装了 `create-react-app` 包，然后构建了项目，最后将 `create-react-app` 包删除。渣男行为属于是。

非常遗憾的是 create-react-app 项目已经很久没有更新了，社会活力低下。而且由于采用 webpack 打包方式，创建的项目体量一旦上去，热加载时间很慢。

1.1.2 Vite 开发环境

Vite 是新一代前端工具链，现在大型项目逐渐由 Webpack 转向 Vite。这两者有如下几点不同:

表 1.1 Vite 与 Webpack 对比

	Vite	Webpack	备注
编写语言	Go	JavaScript	Go 是纳秒级响应，理论上速度快 10-100 倍
加载模式	懒加载	勤加载	Vite 利用 ESModule 按需加载必要模块
定位	打包 + 工具	纯打包	
使用	需下载	node 集成	

Webpack 项目开始时，ES6 尚未推出，因此 Webpack 并没有引入 ESModule。Webpack 的打包策略是加载所有的文件进行编译，最终转换为原生 js 再让浏览器显示，随着项目体积增加，打包时间也线性增加，一个复杂的项目，使用 Webpack 热加载往往需要几秒钟的时间才能看到界面。

Vite 利用 ESModule 的特性，只加载必要的文件，所编译的组件会自动导入其他模块并完成编译，也即按需加载。因此 Vite 加载时间始终是 O(1)，界面几乎是瞬间加载完成。

Vite 不能完全替代 Webpack，但现在绝大多数新项目都是用 Vite 打包。

继续上文，如果使用 vite 创建项目就不能使用 create-react-app:

```
npm create vite@latest app --template react-ts
```

Vite 官方地址: `https://cn.vitejs.dev/`

1.2 React 编程风格

个人总结的 React 三个直观的编程风格: 声明式，all-in-js，vDOM。

- **声明式**: 编程风格。

  声明式是函数式编程的一种风格，与之对应的是命令式。命令式很好理解，就是具体的命令，逻辑处理；命令式函数会显式地表明内部逻辑。声明式可以简单理解为只调用相关操作，不考虑内部具体逻辑。在实际项目中表现为将具体功能封装后，主要函数只负责调用。因此在 React 项目中很少直接看到函数该怎么做，而是关注于做什么。
- **all-in-js**: 组件化。

  前端三剑客中 HTML 负责结构，CSS 负责样式，JavaScript 负责逻辑。这样的分化存在一个问题，当项目过大后，一点小改动会使得整个前端牵一发而动全身。可能因为 HTML 的改变，DOM 结构发生变化；因为 CSS 的改变，预料之外的元素样式发生了变化...

  React 认为这三者天然存在耦合性，因此用 JSX 控制 HTML，将三者统一由 js 控制，而

将整个界面分为零散的小组件进行管理。如果使用 `styled-component` 或 `TailWindCss`，js 可以进一步控制 css，真正实现 all-in-js。

- **vDOM**: 性能优化。

  在 React 中可以创建 React 元素，即可以“自定义”新标签 (虽然这些标签实际上还是原生 DOM)，React 标签可以进一步封装。同时 vDOM 可以更加智能地让原生界面进行重绘与回流，React 内部做了大量优化让开发者既能快速开发，又不用担心性能。

此外，React 比 Vue 的一大优势是强大的生态，除了 react 项目本身，还有 react-native，next.js，redux.js 等众多优秀的项目。

2 React 运行机制与 JSX 语法

2.1 React 运行机制

在浏览器中[1]使用 React 需要引入两个核心库: React 和 ReactDOM。前者负责创建视图，后者负责在浏览器中渲染 UI。

React 与 ReactDOM 库

我们知道，HTML 本质上是一系列指令，让浏览器构建 DOM。浏览器加载 HTML 并渲染用户界面时，构成 HTML 文档的元素变成 DOM 元素。我们通过 JavaScript 调用 DOM API 可以修改 DOM。React 是代我们更新浏览器 DOM 的一个库。有了 React 库，我们不再直接与 DOM API 交互，则是让 React 收到指令后帮助我们渲染和协调元素。

React DOM 由 React 元素组成。React 元素是对真正 DOM 元素的描述，换句话说，React 元素是如何创建浏览器 DOM 的指令。

我们可以通过 `React.createElement` 创建一个表示 `h1` 的 React 元素:

```
React.createElement("h1", {id: "recipe-o"}, "Baked Salmon");
```

渲染时 (渲染由 ReactDOM 库负责)，React 把这个元素转换成真正的 DOM 元素:

```
<h1 id = "recipe-o">Baked Salmon</h1>
```

如果在控制台输出这个元素，会看到如下内容:

```
{
    $$typeof: Symbol(React.element), // $$ 对象类型，一般都是 React.element 或者 HTML 元素
    "type": "h1",  // HTML 或 SVG 元素
    "key": null,   // 类似索引，用于快速获取元素，提高效率
    "ref": null,
    "props": {id: "recipe-o", children: "Baked Salmon"}, // 元素属性，children 表示嵌套的文本
    "_owner": null,
    "_store": {},
}
```

创建 React 组件后，由 ReactDOM 负责渲染，渲染所需的 `render` 方法就在 ReactDOM 中。

```
const dish = React.createElement("h1", null, "Baked Salmon");
ReactDOM.render(dish, document.getElementById("root"));
```

在 DOM 中与渲染有关的一些共都在 ReactDOM 包中，React16 之后，`render` 方法可以接受一个数组以渲染多个元素。

比较特殊的，`props.children` 的值可以是 React 元素对象或元的对象组成的数组:

```
{
    "type": "ul"
```

[1]第一部分所有内容均在浏览器环境中，下文不再说明。

```
    "props": {
        "children": [
            {"type": "li", "props":{...} ...}
            {"type": "li", "props":{...} ...}
            {"type": "li", "props":{...} ...}
            {"type": "li", "props":{...} ...}
        ]
    }
}
```

2.2 JSX 语法

上一节我们使用了 React 包中的函数直接操作 React DOM，这样做可以清晰地看出 React 是如何运作的，但是实际开发中，这样写比较复杂，可读性不高，因此有了 JSX 语法。

JSX 是一种用于描述 UI 的 JavaScript 扩展语法，React 使用这种语法描述组件的 UI, JSX 本质上还是返回一个 React 元素。

JSX 的语法和 XML 类似，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:

- 节点树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: `{/**/}`。

Fragment

JSX 语法规定创建的 React 元素必须有且仅有一个根标签，通常情况下我们可以用 `<div>` 标签作为父标签，再做一些对应的样式调整。但有时候我们不需要多余的父标签，这会创建很多非必要的标签，导致 DOM 层级过深，此时可以使用 `<React.Fragment>` 标签表示一个 React 片段，且不会创建新标签。更简洁地，我们可以使用 `<>` 代替 `<React.Fragment>`:

```
return (
    <>
        <h1> Hello </h1>
        <h1> World </h1>
    </>
)
```

JSX 的本质

JSX 本质上只是一种语法糖，对于部分编译器，用.js 文件写 JSX 语法也不会有问题。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');
```

JavaScript 表达式

在 JSX 语法中可以使用 JavaScript 表达式，一堆花括号之间的 JavaScript 代码会做求值，比如:

```
<ul>
    {props.values.map((value, index) => (<li key="{i}">{value}</li>))}
<ul>
```

3 React 类组件

类组件已经完成了它的使命，自 React16 给出 Hooks 后，React 主流的方向是使用函数组件。不过一些老项目仍在使用类组件，类组件的用法了解即可。如果有一定基础，也可以跳过类组件这一小节。

3.1 类组件

3.1.1 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。

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>
```

3.1.2 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)
```

3.1.3 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。

3.1.4 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`。

3.1.5 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。

3.1.6 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" />
```

3.2 组件技巧

3.2.1 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>
  );
}
```

3.2.2 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>
  )
}
```

3.2.3 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。

4 React 状态管理

这一节开始，所介绍的内容都是函数组件与 Hooks。本节及以后只会用到函数组件。

4.1 useState 钩子

4.1.1 useState

在函数组件中，我们没有 this(更准确地说，不会用 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 让组件通过 state 保存状态数据:

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

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

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

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(下文也叫做 setState)。

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，还是创建一个新的对应类型传给 state。

组件树的状态

为了避免状态乱用与更好地管理状态，衍生出了两种组件树之间传递状态的方案，分别是沿组件树向下发送状态和向上发送交互。

沿组件树向下发送状态很好理解，子元素接受父元素的状态作为参数。向上发送交互也很好理解，在子组件中调用 `setState` 方法即可，这里主要看一下向上发送交互:

```
export default function StarRating({ style = {}, totalStars = 5, ...props }) {
  const [selectedStars, setSelectedStars] = useState(0);
  return (
    <div style={{ padding: 5, ...style }}>
      {createArray(totalStars).map((n, i) => (
        <Star
          key={i}
          selected={selectedStars > i}
          onSelect={() => setSelectedStars(i + 1)}
          {...props}
        />
      ))}
      <p>
        {selectedStars} of {totalStars} stars
      </p>
    </div>
  );
}
```

向上发送交互的过程:

- 子标签 `<Star>` 通过 `onSelect` 函数调用 `setSelectedStars` 方法。
- 父组件 `selectedStar` 被改变。
- ReactDOM 识别到组件 state 发生改变，尝试重绘。

异步更新

前面说过，`state` 的更新是异步的，多次 `setState` 的状态修改会合并成一次状态修改，如果我们需要避免异步更新导致数据不准确可以这样写:

```
const [count, setCount] = useState(0);
function handleClickFn() {
  setTimeout(() => {
    setCount((prevCount) => {
      return prevCount + 1
    })
  }, 3000);
}
```

当使用上述方法更新 (也叫函数式更新) state 的时候，就不会出现异步问题，因为它可以获取之前的 state 值，也就是代码中的 prevCount 每次都是最新的值。

4.1.2 useReducer

useReducer 是 useState 的替代方案:

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

- reducer: 处理函数。
- initialArg: 初始值。

如果 state 逻辑比较复杂或者一个 state 依赖之前的 state，使用 useReducer 更合适。

reducer 函数接受两个参数，一个当前 state 值，一个 action 表示 dispatch 传入的值:

```
/* 当state需要维护多个数据且它们互相依赖时，推荐使用useReducer
组件内部只是写dispatch({...})
处理逻辑的在useReducer函数中。获取action传过来的值，进行逻辑操作
*/

function reducer(state, action) {
  const { type, nextName } = action;
  switch (type) {
    case "ADD":
      return {
        ...state,
        age: state.age + 1
      };
    case "NAME":
      return {
        ...state,
        name: nextName
      };
  }
  throw Error("Unknown action: " + action.type);
}

export default function ReducerTest() {
  const [state, dispatch] = useReducer(reducer, { name: "qingying", age: 12 });
  function handleInputChange(e) {
    dispatch({
      type: "NAME",
      nextName: e.target.value
    });
  }
  function handleAdd() {
    dispatch({
      type: "ADD"
    });
  }
  const { name, age } = state;
  return (
    <>
      <input value={name} onChange={handleInputChange} />
      <br />
```

```
      <button onClick={handleAdd}>添加1</button>
      <p>
        Hello,{name}, your age is {age}
      </p>
    </>
  );
}
```

相当于将数据处理逻辑抽象到了 reducer 中处理，setState 改为 dispatch 只负责传递参数。

本质上 useReducer 和 useState 实现的效果是相同的，因此 useReducer 也会导致 state 改变，需要重新 render。可以将 useReducer 看作一种设计模式，将具体的逻辑抽象出去，更符合 React 声明式理念。

4.2 useRef 钩子

4.2.1 useRef 基础

useRef 和 useState 一样，可以在函数组件中直接创建:

```
const refContainer = useRef(initialValue);
```

- 返回一个可变的 ref 对象，该对象仅有一个 current 属性，初始值为传入的参数。
- 返回的 ref 对象在组建的整个生命周期保持不变。
- 更新 current 值不会导致 render。
- 更新 useRef 是副作用，应该放在 useEffect 里。

在 React 运行机制一章里，我们通过输出 React 元素对象，可以看到所有元素对象都有一个 ref 成员。可以直接使用标签属性绑定 ref 值。

```
const TextInputWithFocusButton = () => {
   const inputEl = useRef(null);
   const handleFocus = () => {
       inputEl.current.focus();
   }
   return (
       <p>
           <input ref={inputEl} type="text" />
           <button onClick={handleFocus}>Focus the input</button>
       </p>
   )
}
```

这样，我们就将 inputEl 绑定到了 input 标签上，以后可以通过 inputEl.current 直接操作 <input> 元素。

除了 useRef 还有个 createRef，crateRef 唯一的区别是每次组件更新都会重新返回新

的引用，除此之外没有区别。

4.2.2 useRef 共享数据

useState 有一个特性，不同渲染之间无法共享 state 状态值。如果我们在函数组件中写函数并使用 state，当我们更改状态的时候，React 会重新渲染组件，每次的渲染都会拿到独立的 state 值，并重新定义对应的函数，每个函数体里的 state 值也是它自己的，这会导致函数内的 state 值无法更新[2]。

如果我们要在函数中获取实时状态，有两种方案，一是在函数外定义一个全局变量，由于变量是定义在组件外，所以不同渲染间是可以共用该变量, 所以每次都会获取最新的变量值。这种方案的一个坏处是会定义全局变量，如果多个组件都有全局变量可能会让代码变得比较难维护，尤其是多个组件用同一个全局变量。

第二种方案是使用 useRef:

```
const LikeButton: React.FC = () => {
  let like = useRef(0);
  function handleAlertClick() {
    setTimeout(() => {
      alert(`you clicked on ${like.current} ${like.current}`);
    }, 3000);
  }
  return (
    <>
      <button onClick={() => { like.current = like.current + 1; }}>
        {like.current}赞
      </button>
      <button onClick={handleAlertClick}>Alert</button>
    </>
  );
};
export default LikeButton;
```

这样的好处是数据在组件内，和其他组件没有关系。同时也说明 useRef 可以返回任意类型，不仅仅是绑定 React 元素。

4.2.3 forwardRef

有的时候，我们需要获取 React 组件的某个 dom 元素，但是子组件已经被我们封装了，无法直接通过 React 组件获取内部 dom 元素节点。传统的方法是通过 Web API 一层层向下找 dom 节点。这有两个缺陷，一是如果封装的很深，获取 dom 节点的代码会很长; 二是回流可能会导致 dom 结构改变，用硬编码的方式会找不到 dom 节点。

于是 React 提供了 React.forwardRef 函数:

```
React.forwardRef((props, ref) => {})
```

[2]这个情况用的比较少，所以没有贴源码，有兴趣可以通过链接看原文

- 该函数返回一个 React 组件。
- 这个组件会接受 `ref` 参数，用于绑定内部标签的 `ref`。
- 父组件通过传入 `ref` 实参与子组件对应标签的 `ref` 绑定。
- 父组件绑定后，可以通过 `ref` 获取到对应的 dom 元素。

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

4.2.4 `useImperativeHandle`

直接暴露给父组件带来的问题是某些情况的不可控，父组件可以拿到 DOM 后进行任意的操作，但多数情况下我们希望限定只做某些操作。通过 `useImperativeHandle` 可以值暴露固定的操作。

```
useImperativeHandle(ref, createHandle, [deps])
```

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

这样，渲染 <FancyInput ref=fancyInputRef /> 的父组件就只可以调用 fancyInputRef.current.focus()。

`useImperativeHandle` 一般与 `forwardRef` 配合使用。

4.3 `useContext` 钩子

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。

`useContext` 不同于 `useState` 与 `useRef`，他需要先通过 `React.createContext` 创建数据。再通过 `useContext()` 或者 `Context.Provider` 使用数据。

最常用的使用方式如下:

```
// 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。

比较少用的，结合 `<Context.Provider>` 与 `<Context.Consumer>`:

```
const ThemeContext = createContext()

class App extends React.Component {
  render () {
    return (
      // 使用 Context.Provider 包裹后续组件，value指定值
      <ThemeContext.Provider value={'red'}>
        <Bottom></Bottom>
      </ThemeContext.Provider>
    )
  }
}

class Bottom extends React.Component {
  render () {
    return (
      // Context.Consumer Consumer消费者使用Context的值
      // 但子组件不能是其他组件，必须渲染一个函数，函数的参数就是Context的值
      <ThemeContext.Consumer>
        {
          theme => <h1>ThemeContext的值为{theme}</h1>
        }
      </ThemeContext.Consumer>
    )
  }
}
```

5 React 渲染管理

5.1 useEffect 钩子

5.1.1 useEffect

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 (render 之后) 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。

```
useEffect(() => {
  return () => "bye"; // 清理时调用
})
```

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 (类型是数组) 实现相同效果:

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

5.1.2 useLayoutEffect

useLayoutEffect 与 useEffect 作用相同: 处理 render 之后的副作用，但在组件生命周期调用顺序不同:

- 渲染 (render);
- 调用 useLayoutEffect;
- 浏览器绘制，将组件元素添加到 DOM;
- 调用 useEffect;

5.2 useMemo 钩子

本节参考文献: https://blog.csdn.net/sinat_17775997/article/details/94453167

useMemo 用来判读一个函数组件是否要重新渲染，主要目的是减少不必要的重绘来提升性能。作用和类组件的 shouldComponentUpdate 函数相同。

```
useMemo(fn, arr)
```

函数组件只要 props 或 state 发生改变就会重绘，不会判断改变后的值是否相同。这样会导致性能下降，useMemo 会比较传入的第二个参数是否发生了变化再判断是否要执行 fn。

只有第二个参数匹配，并且其值发生改变，才会多次执行执行，否则只执行一次，如果为空数组，fn 只执行一次。

```
export default function WithoutMemo() {
    const [count, setCount] = useState(1);
    const [val, setValue] = useState('');

    function expensive() {
        console.log('compute');
        let sum = 0;
        for (let i = 0; i < count * 100; i++) {
            sum += i;
        }
        return sum;
    }

    return <div>
        <h4>{count}-{val}-{expensive()}</h4>
        <div>
            <button onClick={() => setCount(count + 1)}>+c1</button>
            <input value={val} onChange={event => setValue(event.target.value)}/>
        </div>
    </div>;
}
```

上面这个反例中，expensive 函数值依赖于 count，但由于 val 也是 state，它的改变会让整个组件重绘，expensive 也会随之重新计算。这时候我们可以使用 useMemo 优化:

```
export default function WithMemo() {
  const [count, setCount] = useState(1);
  const [val, setValue] = useState('');
  const expensive = useMemo(() => {
      console.log('compute');
      let sum = 0;
      for (let i = 0; i < count * 100; i++) {
          sum += i;
      }
      return sum;
  }, [count]);

```

```
  return <div>
      <h4>{count}-{expensive}</h4>
      {val}
      <div>
          <button onClick={() => setCount(count + 1)}>+c1</button>
          <input value={val} onChange={event => setValue(event.target.value)}/>
      </div>
  </div>;
}
```

这样 expensive 就只依赖 count，只有依赖项改变才会触发 expensive 重新计算值，否则返回之前的值。

5.3 useCallback 钩子

本节参考文献: https://blog.csdn.net/sinat_17775997/article/details/94453167

useCallback 和 useMemo 类似，都用于优化渲染，不同的是，它但会的是缓存的函数。

```
const fnA = useCallback(fnB, [a])
```

上面的 useCallback 会将我们传递给它的函数 fnB 返回，并且将这个结果缓存；当依赖 a 变更时，会返回新的函数。

使用场景是：有一个父组件，其中包含子组件，子组件接收一个函数作为 props；通常而言，如果父组件更新了，子组件也会执行更新；但是大多数场景下，更新是没有必要的，我们可以借助 useCallback 来返回函数，然后把这个函数作为 props 传递给子组件；这样，子组件就能避免不必要的更新。

```
import React, { useState, useCallback, useEffect } from 'react';
function Parent() {
    const [count, setCount] = useState(1);
    const [val, setVal] = useState('');

    const callback = useCallback(() => {
        return count;
    }, [count]);
    return <div>
        <h4>{count}</h4>
        <Child callback={callback}/>
        <div>
            <button onClick={() => setCount(count + 1)}>+</button>
            <input value={val} onChange={event => setVal(event.target.value)}/>
        </div>
    </div>;
}

function Child({ callback }) {
    const [count, setCount] = useState(() => callback());
    useEffect(() => {
```

```
        setCount(callback());
    }, [callback]);
    return <div>
        {count}
    </div>
}
```

useEffect、useMemo、useCallback 都是自带闭包的。也就是说，每一次组件的渲染，其都会捕获当前组件函数上下文中的状态 (state, props)，所以每一次这三种 hooks 的执行，反映的也都是当前的状态，你无法使用它们来捕获上一次的状态。对于这种情况，我们应该使用 ref 来访问。

5.4 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';
}
```

5.4.1 useDebugValue()

```
useDebugValue(value)
```

用于在 React 开发者工具中显示自定义 hook 的标签。

`useDebugValue` 接受一个格式化函数作为可选的第二个参数。该函数只有在 Hook 被检查时才会被调用。它接受 debug 值作为参数，并且会返回一个格式化的显示值。

6 React 请求数据

Web 基础与 Http 协议不讲，只讲 Fetch API 以及 useFetch。

6.1 Fetch API

Fetch 是收发 Http 的一种技术，自 ES6 后加入 JavaScript，使用 Fetch 不需要添加外部包，它有如下特点:

- Fetch API 提供了一个获取资源的接口（包括跨域请求），用于取代传统的 XMLHttpRequest 的，在 JavaScript 脚本里面发出 HTTP 请求。
- Fetch API 是基于 promise 的设计，返回的是 Promise 对象，它是为了取代传统 xhr 的不合理的写法而生的。
- 相比 axios，Fetch API 更底层，没有被进一步封装。

6.1.1 Promise

本小节参考文章: `https://blog.csdn.net/m0_52040370/article/details/127197204`

Promise 是异步编程的一种解决方案，比传统的解决方案 (回调函数和事件) 更合理和更强大, 它是一个 ECMAScript 6 提供的类，目的是更加优雅地书写复杂的异步任务。

Promise 对象有以下两个特点:

- 对象的状态不受外界影响。Promise 对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
- 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise 对象的状态改变，只有两种可能：从 pending 变为 fulfilled 和从 pending 变为 rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。如果改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

Promise 语法格式:

```
new Promise(function (resolve, reject) {
  // resolve 表示成功的回调
  // reject 表示失败的回调
}).then(function (res) {
  // 成功的函数
}).catch(function (err) {
  // 失败的函数
})
```

```
const promise = new Promise((resolve,reject)=>{
```

```
    //异步代码
    setTimeout(()=>{
        // resolve(['111','222','333'])
        reject('error')
    },2000)
})
promise.then((res)=>{
    //兑现承诺，这个函数被执行
    console.log('success',res);
}).catch((err)=>{
    //拒绝承诺，这个函数就会被执行
    console.log('fail',err);
})
```

在 then，catch 中传入的函数会作为参数 resolve 和 reject 传入上方，随后在构造函数的回调函数中写入异步代码。

此外，promise 的 then 方法会返回实例本身，因此可以采用链式编程。

Promise 有一个 all 方法用于将多个 Promise 实例包装成一个新的 Promise 实例。

```
const q1 = pajax({
    url:"http://localhost:3000/looplist"
})

const q2 = pajax({
    url:"http://localhost:3000/datalist"
})
Promise.all([q1,q2]).then(res=>{
    console.log(res)
    console.log("隐藏加载中...")
}).catch(err=>{
    console.log(err)
})
```

II React 模块

7 Axios

7.1 Http 与 Restful API

Http

HTTP(HyperText Transfer Protocol)，即超文本传输协议，基于 TCP/IP 通信协议来传递数据。HTTPS(HyperText Transfer Protocol Secure)，即超文本传输安全协议。HTTPS 经由 HTTP 进行通信，但用 SSL/TLS 加密数据包。

默认情况下，http 使用 80 端口，https 使用 443 端口。

HTTP 有以下三个注意点:

- 无连接: 限制每次连接只处理一个请求，处理完即断开。
- 媒体独立: 任何类型的数据都可以通过 HTTP 发送。
- 无状态: 协议对于事务处理没有记忆能力。

一个 Http 请求报文有以下主要部分:

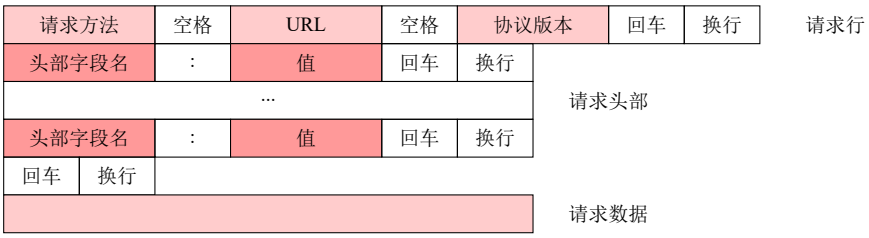

图 7.1 HTTP 请求报文

- 请求行: 请求方法字段，URL，字段，HTTP 协议版本。
  - 方法字段: 即 GET, HEAD, POST 等。
  - URL: 常见形式是网址 (当不限于此)。
  - 协议版本: 如 HTTP/1.1。
- 请求头部: 由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。
- 空行: 用于分开请求头与请求数据。
- 请求数据: GET 方法中没有，一般在 POST 方法中使用。

例如一个报文如下:

```
// 请求首行
POST /hello/index.jsp HTTP/1.1
//请求头信息
Host: localhost
```

```
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: GB2312,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://localhost/hello/index.jsp
Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98
Content-Type: application/x-www-form-urlencoded
Content-Length: 14
// 这里是空行
//POST有请求正文 （Get没有，为空）
username=hello
```

响应报文的结构类似，包含状态行，消息报头，空行，正文。其中状态行包括: HTTP 协议版本，状态码，状态信息。

常见的状态码如下:

表 2.1 Http 常见状态码

状态类型	状态码	说明
信息	100	正在处理
成功	200	成功处理
重定向	301	永久重定向
	302	临时重定向
	303	资源在另一个 URL，通过 GET 方法获取
	304	自从上次请求后，请求网页未修改过。不返回新的内容
重定向	400	语法错误，服务器无法理解
	401	需要携带认证信息
	403	权限等拒绝请求
	404	资源不存在
重定向	500	服务器错误
	503	服务器停机或维护或超载

Restful API

现代 URL 基本都用的 Restful API。Restful API 提供了四种请求类型:

- **GET**: 对应数据库查询操作，参数写在 URL 中。
- **POST**: 对应数据库更新操作，请求不会覆盖。参数放在 body 中，分段发送数据。
- **PUT**: 对应数据库更新操作，后来的请求会替换前一个请求。
- **DELETE**: 对应数据库删除操作。

7.2 AJAX 与 Promise

axios 是使用 promise 对 ajax 的封装。

AJAX

ajax(async javascript and XML) 指异步 JavaScript 和 XML。是一种使用现有标准的新方法。允许服务器部分更新网页内容。

ajax 工作步骤:

- 客户端发送请求，创建 ajax 对象，并将请求交给 ajax。
- ajax 将请求交给服务器。
- 服务器进行业务处理，返回数据给 ajax。
- ajax 对象接受数据。
- javascript 写入新的数据。

Promise

Promise 是 JS 中进行异步编程的新的解决方案。`Promise` 是一个构造函数，接收一个函数为参数，该参数包含两个函数: `resolve`, `reject`。分别代表异步执行成功和失败后的回调函数。`Promise` 对象只会改变一次，返回一个结果，即要么成功，要么失败。

Promise 有两大优点:

- 指定回调函数方法更灵活。
- 支持链式编程。

Promise 有三种状态: pending(等待), fulfilled(成功), rejected(失败)。

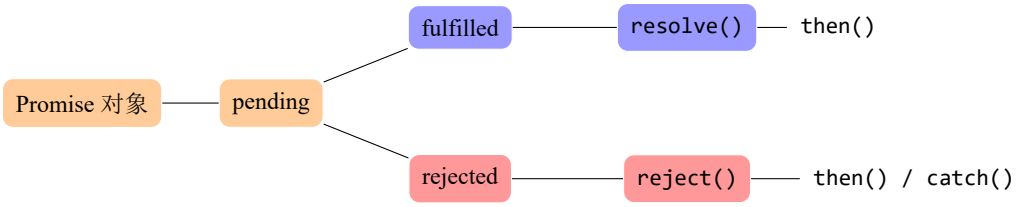

图 7.2 Promise 执行过程

Promise 的构造函数如下:

- `Promise` 构造函数: `Promise(excutor)` 。
- `excutor` 函数: `(resolve, reject) => {}`。
- `resolve` 函数: `value => {}`。
- `reject` 函数: `reason => {}`。

Promise 的常用方法 (回调函数签名和前面一致), 这些方法返回值都是 `Promise` 对象本身 (链式编程):

- 原型方法
  - **Promise.prototype.then(onResolved, onRejected)**

    用来预指定成功和失败的回调函数，成功回调函数是必选的，失败回调函数可选。
  - **Promise.prototype.catch(onRejected)**

    用来捕获与处理错误，相当于 then(undefined, onRejected)
  - **Promise.prototype.finally(onFinally)**

    无论成功失败都会执行的方法。
- 状态改变
  - **Promise.resolve(value)**

    value 为成功的数据或 promise 对象。返回一个成功/失败的 promise 对象。作用上会将本身状态由 pending 改为 resolved。
  - **Promise.reject(reason)**

    reason 为失败的原因。返回一个失败的 promise 对象。作用上会将本身状态由 pending 改为 rejected。
- 组合方法
  - **Promise.all(iterable[Promise])**

    将多个 Promise 包装为一个，如果其中一个 rejected 则转换为失败态，如果全部 fulfilled 则转化为成功态。
  - **Promise.race(iterable[Promise])**

    将多个 Promise 包装为一个，状态取决于最先改变状态的实例。
  - **Promise.any(iterable[Promise])**

    将多个 Promise 包装为一个，如果其中一个 fulfilled 则转换为成功态，如果全部 rejected 则转化为失败态。
  - **Promise.allSettled(iterable[Promise])**

    将多个 Promise 包装为一个，等全部执行完成后，转换为 fulfilled 态。

常见的单个 promise 对象调用形式:

```
promise
.then(result => {···})
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

7.3 Axios

Axios 是一个基于 promise 网络请求库，作用于 node.js 和浏览器中。在服务端它使用原生 node.js http 模块, 而在客户端 (浏览端) 则使用 XMLHttpRequests。

```
npm install axios
```

7.3.1 Axios API

Axios 方法返回值不做说明都是 `Promise` 对象。

`axios()`

直接通过构造函数创建请求有两种方式:

- **`axios(config)`**
- **`axios(url [,config])`**

例如:

```
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
}).then(response => {...});
```

请求方式别名

为了方便，给出了几个别名方法，主要用于省略常用配置项:

- **`axios.request(config)`**
- **`axios.get(url, [,config])`**
- **`axios.post(url, [, data [,config]])`**
- ……

`axios.create([config])`

创建一个 axios 实例可以用于省略通过用的配置项:

```
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});
```

同时 axios 实例也提供了 `get,post` 等方法。

7.3.2 参数说明

请求配置

请求配置包含以下常用参数，它只有一个必须项: `url`, 如果不指定方法，默认为 `get`。

表 2.2 请求配置选项

键	一般类型	说明
url	string	URL, 必填
method	string	http 请求方法，默认为'get'
baseUrl	string	加在 url 前面，起简化作用
header	Object	头，键都是小写
params	Object	参数，加在 URL 后
data	Object	请求数据
data	string	发送请求体数据的可选语法
timeout	number	超时毫秒数
auth	Object	放 token 等凭证信息
withCredentials	boolean	跨域请求是否需要凭证
responseType	string	返回信息类型
validateStatus	Function(status)	根据状态码决定接收还是拒绝
responseEncoding	string	解码方式
maxRedirects	number	最大重定向次数
transformRequest	Array[Function]	用于处理请求数据
transformResponse	Array[Function]	用于处理返回数据
proxy	Object	代理数据

响应结构

响应结构即请求的返回信息 `response`:

表 2.3 响应结构

键	一般类型	说明
status	number	响应码
statusText	string	响应状态信息
data	Object	服务器提供的数据
headers	Object	响应头
config	Object	配置信息
request	Object	此响应的请求

7.3.3 全局配置

默认配置

默认配置将作用于每个请求，使用 `axios.default` 进行配置:

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

axios 配置的优先级如下:

- 具体请求的 `config` 参数。
- 实例的 `config` 配置。
- 全局 `default` 配置。
- `lib/default.js` 库的默认配置。

拦截器

拦截器用于在请求或响应被 `then, catch` 处理前拦截:

```
// 添加请求拦截器
axios.interceptors.request.use(function (config) {
  // 在发送请求之前做些什么
  return config;
}, function (error) {
  // 对请求错误做些什么
  return Promise.reject(error);
});

// 添加响应拦截器
axios.interceptors.response.use(function (response) {
  // 2xx 范围内的状态码都会触发该函数。
  // 对响应数据做点什么
  return response;
}, function (error) {
  // 超出 2xx 范围的状态码都会触发该函数。
  // 对响应错误做点什么
  return Promise.reject(error);
});
```

可以移除拦截器:

```
const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

第二部分

Redux

III Redux 基础

8 Redux 介绍

8.1 简介与安装

Redux 是一款 JS 应用的状态容器，提供可预测的状态管理，也是 React 使用最多的状态管理容器 (基本上是标配)。Redux 官方团队维护了 React 框架版的 React-Redux(但我们一般用 Redux Toolkit)。React Redux 8.x 需要 React 16.8.3 或更高的版本 / React Native 0.59 或更高的版本，否则无法使用 hooks。

Redux Toolkit

Redux Toolkit 是 Redux 官方提供的构建包，可以简化大多数 Redux 任务，也是目前主流的 Redux 使用方案[1]。NPM 安装如下:

```
npm install @reduxjs/toolkit
```

构建 Redux 应用

从头开始构建 redux 应用指令如下:

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

对已有的项目添加 React Redux 指令如下，这三个包具体使用哪个视需求而定:

```
npm install @reduxjs/toolkit react-redux redux
```

Redux 理念

就像 React 的 all-in-js, 声明式式编程一样，Redux 主要作用以及理念如下:

- **集中式**状态管理: 所有的状态会被独立整合在一起，就像 DOM 结构一样，Redux 有自己唯一的状态管理容器。
- **可预测**函数式编程: Redux 中的所有状态处理函数都是可预测的，不存在随机数，当前时间等不可预测的内容。有什么样的输入就必定有相同的输出。

React 的状态管理存在一定的缺陷，React 组件的状态改变由 props/state 负责，父子组件之间的状态传递只能通过这两类数据的改变传递，如果组件深度过高，组件之间的状态传递

[1]在本文前几章会用到原生 Redux，实际开发中 Redux Toolkit 使用较多

会显得十分繁琐且难以管控。React 推出的 `useContext` 在一定程度上解决了这个痛点，但是对于大规模应用，单单引入 `useContext` 显得乏力，Redux 则推出了一套完备的理念来进行单独的状态管理。

Redux 有着丰富的社区资源，最主流的几个工具如下:

- **React-Redux**: Redux 的官方 React 版，可以让 React 组件访问 `state` 片段和 `dispatch actions` 更新 `store`，从而同 Redux 集成起来。
- **React-Toolkit**: 官方推荐的编写 Redux 逻辑的方法，提供了更简单的使用方式。
- **React-DevaTools**: 显示 Redux 存储中状态随时间变化的历史记录。

8.2 Redux 术语与概念

8.2.1 单向数据流

首先来看一段 React 代码:

```
function Counter() {
  // State: counter 值
  const [counter, setCounter] = useState(0)

  // Action: 当事件发生后，触发状态更新的代码
  const increment = () => {
    setCounter(prevCounter => prevCounter + 1)
  }

  // View: 视图定义
  return (
    <div>
      Value: {counter} <button onClick={increment}>Increment</button>
    </div>
  )
}
```

这一个小组件包含以下三部分:

- **state**: 状态，即组件中的变化数据，具体为 `useState` 内容。
- **view**: 视图，即显示的内容。
- **action**: 动作，即点击这一交互事件。

这三者的关系可以简单描述为: 用户通过 view 触发 action，action 改变 state 继而触发绘制新的 view。

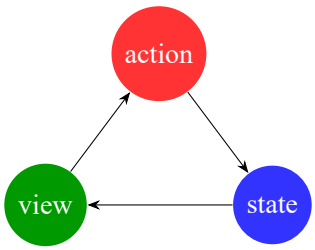

图 8.1 单向数据流

前面我们说过，如果 state 比较简单，传递深度不超过三层组件，用改变 props/state 的方法是十分简单的。但如果多个不同位置的组件需要共享，获取，改变相同的 state 时，就会变得十分复杂。Redux 提供的方案是提取共享的 state，单独抽象出来管理。这样一来 state 和部分 action(至于如何触发 action，一些前置处理仍由 React 负责) 将由 Redux 管理。

用过 `useState` 钩子都知道，只有 state 改变，才会触发组件重绘。这里的改变是指基本类型值变化或者新对象代替旧对象。例如下面数据改变不会触发重绘:

```
const arr = [1,2];
arr[1] = 3;
return arr;
```

如果要触发重绘，应该这样做:

```
const arr = [1,2];
const arr2 = arr.concat(3);
return arr2;
```

使用 Redux 时，务必使所有数据都是不可改变的 (即对象类型返回新的对象，而不是改变其属性)。

8.2.2 Redux 关键术语

Action

`action` 是一个具有 `type` 字段的对象。表示为程序中发生的事件，其中 `type` 是字符串类型，起标识作用。`action` 可以有其他字段，标识附加信息。

```
const addTodoAction = {
  type: "todos/todoAdded",
  payload: "buy milk"
}
```

对应的有一个 Action Creator，用于创建并返回一个 `action` 对象:

```
const addTodo = text => {
  return {
    type: "todos/todoAdded",
    payload: text
  }
}
```

Reducer

reducer 是一个函数，接受当前的 state 和一个 action 对象，必要时决定如何更新，并返回更新状态。函数签名: (state, action) => newState。作用类似于监听器，接受事件并进行处理，他必须遵守如下规则:

- 仅使用 state 和 action 计算新的状态值。
- 禁止修改 state，应该返回新的 state。
- 禁止异步逻辑，依赖随机值或其他副作用代码。

所有的 reducer 处理逻辑如下:

- 检查 reducer 是否关心这个 action，如果是，更新 state 并返回。
- 如果不是，返回原来的 state。

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  // 检查 reducer 是否关心这个 action
  if (action.type === 'counter/increment') {
    // 如果是，复制 `state`
    return {
      ...state,
      // 使用新值更新 state 副本
      value: state.value + 1
    }
  }
  // 返回原来的 state 不变
  return state
}
```

Store

store 用于存储应用的 state。store 是通过传入一个 reducer 来创建的，并且有一个名为 getState 的方法，它返回当前状态值：

```
import { configureStore } from '@reduxjs/toolkit'

const store = configureStore({ reducer: counterReducer })
console.log(store.getState())
// {value: 0}
```

store 有一个方法叫 dispatch。更新 state 的唯一方法是调用 store.dispatch() 并传入一个 action 对象。store 将执行所有 reducer 函数并计算出更新后的 state，调用 getState() 可以获取新 state。

```
store.dispatch({ type: 'counter/increment' })
console.log(store.getState())
```

`dispatch` 看名字就知道，采用了分发器设计模式。一个事件传入 Redux，最先由 `dispatch` 捕获，继而有它传递给各个 `reducer` 继续处理。

```
const increment = () => {
  return {
    type: 'counter/increment'
  }
}

store.dispatch(increment())
```

`Selector` 函数可以从 `store` 状态树中提取指定的片段。随着应用变得越来越大，会遇到应用程序的不同部分需要读取相同的数据，`selector` 可以避免重复这样的读取逻辑：

```
const selectCounterValue = state => state.value
const currentValue = selectCounterValue(store.getState())
console.log(currentValue)
```

8.2.3 Redux 数据流

Redux 的数据流结构如下:

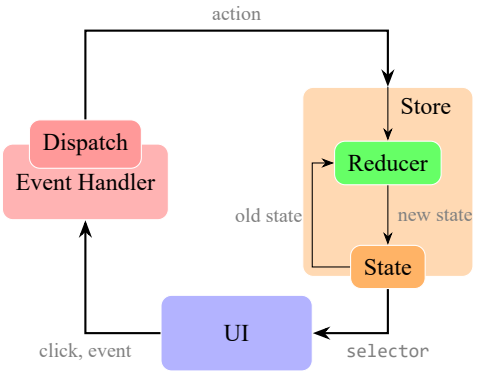

图 8.2 Redux 数据流

具体来说，对于 Redux，我们可以将这些步骤分解为更详细的内容：

- 初始启动
  - 使用最顶层的 root reducer 函数创建 Redux store
  - store 调用一次 root reducer，并将返回值保存为它的初始 state
  - 当视图首次渲染时，视图组件访问 Redux store 的当前 state，并使用该数据来决定要呈现的内容。同时监听 store 的更新，以便他们可以知道 state 是否已更改。
- 更新环节
  - 应用程序中发生了某些事情，例如用户单击按钮
  - dispatch 一个 action 到 Redux store，例如 `dispatch(type: 'counter/increment')`
  - store 用之前的 `state` 和当前的 `action` 再次运行 reducer 函数，并将返回值保存为新的 `state`

– store 通知所有订阅过的视图，通知它们 store 发生更新
– 每个订阅过 store 数据的视图组件都会检查它们需要的 `state` 部分是否被更新。
– 发现数据被更新的每个组件都强制使用新数据重新渲染，紧接着更新网页

9 Redux 构建应用

9.1 应用结构

9.1.1 Toolkit 方法

`configureStore()`

构建一个 Redux 容器，首先需要创建 `store`，而 `store` 又必须获取 `reducer` 和 `state`。下面是一个常见的 `store` 构建方式。

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export default configureStore({
  reducer: {
    counter: counterReducer
  }
})
```

`configureStore` 要求我们传入一个 `reducer` 对象。对象中的键名 key 将定义最终状态树中的键名 key。

- 键: 对应状态名，例如 `counter` 对应有一个 `state.counter`。
- 值: 负责更新对应的 `state`。

`configureStore` 默认会自动在 store setup 中添加几个中间件以提供良好的开发者体验。

Redux Slice

“slice”是应用中单个功能的 Redux reducer 逻辑和 action 的集合, 通常一起定义在一个文件中。该名称来自于将根 Redux 状态对象拆分为多个状态“slice”。

```
import { configureStore } from '@reduxjs/toolkit'
import usersReducer from '../features/users/usersSlice'
import postsReducer from '../features/posts/postsSlice'
import commentsReducer from '../features/comments/commentsSlice'

export default configureStore({
  reducer: {
    users: usersReducer,
    posts: postsReducer,
    comments: commentsReducer
  }
})
```

例如上面代码中: `state.users`, `state.posts...` 均为独立的“slice”. 由于 `usersReducer` 负责更新 `state.users` slice，我们将其称为“slice reducer”函数。

Redux store 需要在创建时传入一个“root reducer”函数。因此，如果我们有许多不同的 slice reducer 函数，理论上我们需要手动将其余 reducer 并入 root reducer。

```
function rootReducer(state = {}, action) {
  return {
    users: usersReducer(state.users, action),
    posts: postsReducer(state.posts, action),
    comments: commentsReducer(state.comments, action)
  }
}
```

Redux 有一个名为 `combineReducers` 的函数，它会自动为我们执行此操作。它接受一个全是 slice reducer 的对象作为其参数，并返回一个函数，该函数在调度操作时调用每个 slice reducer。

```
const rootReducer = combineReducers({
  users: usersReducer,
  posts: postsReducer,
  comments: commentsReducer
})
```

当我们将 slice reducer 的对象传递给 `configureStore` 时，它会将这些对象自动传递给 `combineReducers` 以便我们生成根 reducer。

`createSlice()`

Redux Toolkit 的 `createSlice()` 函数简化了 reducer 的书写过程，它负责生成 action 类型字符串、action creator 函数和 action 对象的工作。你所要做的就是为这个 slice 定义一个名称，编写一个包含 reducer 函数的对象，它会自动生成相应的 action 代码。

`name` 选项的字符串用作每个 action 类型的第一部分，每个 reducer 函数的键名用作第二部分。因此，action 类型即为“counter”名称 + “increment”，例如: `type: "counter/increment"`。

除此之外，我们还需要传入初始状态值，`createSlice` 会自动生成与我们编写的 reducer 函数同名的 action creator。

```
import { createSlice } from '@reduxjs/toolkit'

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    increment: state => {
      // Redux Toolkit 允许我们在 reducers 写 "可变" 逻辑。
      // 并不是真正的改变 state 因为它使用了 immer 库
      // 当 immer 检测到 "draft state" 改变时，会基于这些改变去创建一个新的
      // 不可变的 state
      state.value += 1
```

```
    },
    decrement: state => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    }
  }
})

export const { increment, decrement, incrementByAmount } = counterSlice.actions
export default counterSlice.reducer
```

前面我们说过，不要再 `reducer` 中更改 `state` 原始对象，但是上面代码中，我们改变了 (slice 肯定做了什么)。例如下面代码:

```
state.value = 123;
```

这是非法的，我们直接修改了 `state`，正确的做法是:

```
return {
  ...state,
  value: 123
}
```

产生副本，并覆盖原值，但是这样的逻辑非常繁琐，而且反直觉。所以 Toolkit 帮助我们简化了这一过程: `createSlice` 内部使用了一个名为 `Immer` 的库。`Immer` 使用一种称为“Proxy”的特殊 JS 工具来包装你提供的数据，当你尝试”mutate“这些数据的时候，奇迹发生了，`Immer` 会跟踪你尝试进行的所有更改，然后使用该更改列表返回一个安全的、不可变的更新值，就好像你手动编写了所有不可变的更新逻辑一样。

但是，这只有在 Redux Toolkit 的 `createSlice` 和 `createReducer` 中可以用，其他函数中没有提供类似的方案。

Thunk 异步逻辑

thunk 是一种特定类型的 Redux 函数，可以包含异步逻辑，Thunk 使用两个函数编写:

- 一个内部 thunk 函数，以 `dispatch` 和 `getState` 作为参数。
- 外部创建者函数，创建并返回 thunk 函数。

```
// 下面这个函数就是一个 thunk ，它使我们可以执行异步逻辑
// 你可以 dispatched 异步 action `dispatch(incrementAsync(10))` 就像一个常规的 action
// 调用 thunk 时接受 `dispatch` 函数作为第一个参数
// 当异步代码执行完毕时，可以 dispatched actions
export const incrementAsync = amount => dispatch => {
  setTimeout(() => {
    dispatch(incrementByAmount(amount))
  }, 1000)
}
```

我们可以像使用普通 Redux action creator 一样使用它们：

```
store.dispatch(incrementAsync(5))
```

但是，使用 thunk 需要在创建时将 `redux-thunk` middleware（一种 Redux 插件）添加到 Redux store 中。幸运的是，Redux Toolkit 的 `configureStore` 函数已经自动为我们配置好了，所以我们可以继续在这里使用 thunk。

当你需要进行 AJAX 调用以从服务器获取数据时，你可以将该调用放入 thunk 中。

```
// 外部的 thunk creator 函数
const fetchUserById = userId => {
  // 内部的 thunk 函数
  return async (dispatch, getState) => {
    try {
      // thunk 内发起异步数据请求
      const user = await userAPI.fetchById(userId)
      // 但数据响应完成后 dispatch 一个 action
      dispatch(userLoaded(user))
    } catch (err) {
      // 如果过程出错，在这里处理
    }
  }
}
```

9.1.2 React 整合 Redux

来看看怎么将 Redux 嵌入到 React 组件中 (省略不必要的代码):

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment, incrementByAmount, incrementAsync, selectCount } from
    './counterSlice'

export function Counter() {
  const count = useSelector(selectCount)
  const dispatch = useDispatch()
  const [incrementAmount, setIncrementAmount] = React.useState('2')

  return (
    <div>
      <button
        aria-label="Increment value"
        onClick={() => dispatch(increment())}
      >
        +
      </button>
      <span>{count}</span>
      <button
        aria-label="Decrement value"
        onClick={() => dispatch(decrement())}
      >
```

```
        -
      </button>
    </div>
  )
}
```

useSelector()

我们只使用 `useState` 存储了 `incrementAmount`，`count` 使用了 `useSelector` 存储。他是 Redux 的钩子函数。

`useSelector` 这个 hooks 让我们的组件从 Redux 的 `store` 状态树中提取它需要的任何数据。它接受一个函数，用于获取 store 的 state 值：

```
export const selectCount = state => state.counter.value
```

这样做显得很麻烦，为什么不直接导出 `value`。Redux 禁止 store 与 React 直接交互，只提供了有限且必要的钩子。

useDispatch()

类似的，`useDispatch` 用于访问具体的方法。

```
<button onClick={() => dispatch(increment())}>
  +
</button>
```

这两个钩子提供了向 Redux 容器交互的能力，但是我们并灭有导入 store，hooks 如何与 Redux store 对话呢:

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

9.1.3 什么时候使用 Redux

React 原生的 `useState` 钩子与 Redux 提供的 `store` 都可以管理状态，这两者的功能是否重合? 回到 Redux 要解决的问题上。Redux 用于管理全局状态，如果仅是组件局部的状态，不应该又 Redux 进行管理，因为这样没有必要，除了某个组件，并没有其他组件会用到这个状态。因此是否使用 Redux 的关键取决于是否有多个 “距离远” 的组件需要共享同一状态。

可以看看官网的建议:

- 应用程序的其他部分是否关心这些数据？

- 你是否需要能够基于这些原始数据创建进一步的派生数据？
- 是否使用相同的数据来驱动多个组件？
- 能够将这种状态恢复到给定的时间点（即时间旅行调试）对你是否有价值？
- 是否要缓存数据（即，如果数据已经存在，则使用它的状态而不是重新请求它）？
- 你是否希望在热重载视图组件（交换时可能会丢失其内部状态）时保持此数据一致？

此外，用 Redux 进行状态管理，是一件比较厚重的事，会带来性能上以及逻辑上的开销。例如我们要修改表单，临时修改的数据是否应该由 Redux 管理，更好的方式应该是由表单组件 `useState` 来管理那些没有提交的数据，只有在表单提交时用 `dispatch` 更新 store。