React 前端开发

第三讲 React 初探

PDLi

2023 年 12 月 10 日

目录

今天课程的主要内容

1. 补充：Promise 的使用
    1. 异步编程的概念
    2. Promise 的基本使用
    3. async & await
2. React 初探
    1. React 的基本概念
    2. JSX 语法规则
    3. 创建 React 项目
    4. React 组件
    5. state - 状态
    6. props - 属性
    7. ref - 引用

补充：Promise 的使用

异步编程的概念

在 JavaScript 中，异步编程是一种处理异步操作的编程范式，它允许程序在等待某些操作完成的同时继续执行其他任务。异步编程对于处理事件、处理 I/O 操作、发起网络请求等场景非常有用，因为这些操作可能需要一些时间来完成，而在等待的过程中，程序可以执行其他任务，提高了程序的性能和响应能力。

协程：协程是一种比线程更加轻量级的存在，它可以在不同的线程之间切换，但是协程的切换不是由系统控制，而是由程序自身控制，因此协程的切换不会引起线程的切换，也不会引起线程上下文的切换，所以协程的切换非常快。在某种程度上，协程可以用作实现异步编程的一种方式。异步编程涉及到管理非阻塞的操作，而协程是实现这一目标的一种机制。在一些编程语言和框架中，协程和异步编程的概念经常结合在一起，以提供更高效和清晰的异步代码编写方式。

Promise 的基本使用 I

Promise 是异步编程的一种解决方案，它是一个对象，用来表示一个异步操作的最终完成或者失败。

它提供了一种统一的 API，使得异步操作更加容易管理。在 JavaScript 中，Promise 是一个构造函数，可以通过 new 关键字来创建一个 Promise 对象。

创建 Promise 对象时，需要传入一个函数作为参数，这个函数被称为 executor，它接收两个参数 resolve 和 reject，分别表示异步操作执行成功和失败时的回调函数。在 executor 函数中，我们可以执行一些异步操作，当异步操作执行成功时，调用 resolve 函数，当异步操作执行失败时，调用 reject 函数。

Promise 的基本使用 II

```
const promise = new Promise((resolve, reject) => {
  // 异步操作
  // ...
  if (/* 异步操作成功 */) {
    resolve(/* 异步操作成功的结果 */);
  } else {
    reject(/* 异步操作失败的原因 */);
  }
});
```

Promise 的基本使用 III

Promise 对象创建成功后，可以通过 then 方法来指定异步操作执行成功时的回调函数，通过 catch 方法来指定异步操作执行失败时的回调函数。

```
1 promise.then((result) => {
2   // 异步操作执行成功时的回调函数
3   // result是异步操作执行成功的结果
4 }).catch((reason) => {
5   // 异步操作执行失败时的回调函数
6   // reason是异步操作执行失败的原因
7 });
```

Promise 的基本使用 IV

then 方法和 catch 方法都会返回一个 Promise 对象，因此可以通过链式调用的方式来指定多个异步操作执行成功或者失败时的回调函数。

```
promise.then((result) => {
  // 异步操作执行成功时的回调函数
  // result是异步操作执行成功的结果
}).then((result) => {
  // 异步操作执行成功时的回调函数
  // result是异步操作执行成功的结果
}).catch((reason) => {
  // 异步操作执行失败时的回调函数
  // reason是异步操作执行失败的原因
});
```

Promise 的基本使用 V

多个 Promise 对象可以通过 Promise.all 方法组合成一个新的 Promise 对象，这个新的 Promise 对象在所有的 Promise 对象都执行成功时执行成功，只要有一个 Promise 对象执行失败，这个新的 Promise 对象就执行失败。

Promise.all 方法接收一个 Promise 对象数组作为参数，返回一个新的 Promise 对象。then 的回调函数接收一个数组作为参数，这个数组包含了所有 Promise 对象执行成功的结果，catch 的回调函数接收一个参数，这个参数是第一个执行失败的 Promise 对象的执行失败的原因。

Promise 的基本使用 VI

```
 1 const promise1 = new Promise((resolve, reject) => {
 2 // 异步操作
 3 // ...
 4 if (/* 异步操作成功 */) {
 5     resolve(/* 异步操作成功的结果 */);
 6   } else {
 7     reject(/* 异步操作失败的原因 */);
 8   }
 9 });
10
11 const promise2 = new Promise((resolve, reject) => {
12 // 异步操作
13 // ...
14 if (/* 异步操作成功 */) {
15     resolve(/* 异步操作成功的结果 */);
16   } else {
17     reject(/* 异步操作失败的原因 */);
18   }
```

Promise 的基本使用 VII

```
19 });
20
21 Promise.all([promise1, promise2]).then((results) => {
22   // 所有的异步操作都执行成功时的回调函数
23   // results是所有异步操作执行成功的结果
24 }).catch((reason) => {
25   // 有一个异步操作执行失败时的回调函数
26   // reason是第一个异步操作执行失败的原因
27 });
```

Promise 的基本使用 VIII

我们可以使用 setTimeout 函数来模拟一个异步操作，它接收一个回调函数和一个延迟时间作为参数，当延迟时间过去后，回调函数会被调用。

```
 1 const promise1 = new Promise((resolve, reject) => {
 2   setTimeout(() => {
 3     resolve('promise1');
 4   }, 1000);
 5 });
 6
 7 const promise2 = new Promise((resolve, reject) => {
 8   setTimeout(() => {
 9     resolve('promise2');
10   }, 2000);
11 });
12
13 Promise.all([promise1, promise2]).then((results) => {
14   console.log(results);
15 }).catch((reason) => {
```

Promise 的基本使用 IX

```
16    console.log(reason);
17  });
```

出现错误的情况：

```
 1  const promise1 = new Promise((resolve, reject) => {
 2    setTimeout(() => {
 3      resolve('promise1');
 4    }, 1000);
 5  });
 6
 7  const promise2 = new Promise((resolve, reject) => {
 8    setTimeout(() => {
 9      reject('promise2');
10    }, 2000);
11  });
12
13  Promise.all([promise1, promise2]).then((results) => {
14    console.log(results);
```

Promise 的基本使用 X

```
15 }).catch((reason) => {
16   console.log(reason);
17 });
```

async & await I

async 函数是 ES2017 引入的新的语法，它可以让我们更加方便地编写异步代码。

它使我们拥有了像编写同步代码一样编写异步代码的能力。

使用 async 函数，我们可以在函数内部使用 await 关键字来等待一个 Promise 对象执行完成，await 关键字后面可以跟 Promise 对象，也可以跟 then 方法返回的结果，await 关键字后面的表达式会被暂停执行，直到 Promise 对象执行成功，await 关键字后面的表达式才会被执行。

在这种情况下，我们使用 try...catch 语句来捕获 await 关键字后面的 Promise 对象执行失败时抛出的异常。

async & await II

```
 1 const promise = new Promise((resolve, reject) => {
 2   setTimeout(() => {
 3     resolve('promise');
 4   }, 1000);
 5 });
 6 try {
 7   console.log('before await');
 8   const result = await promise;
 9   console.log('after await');
10   console.log(result);
11 } catch (reason) {
12   console.log(reason);
13 }
```

async & await III

async 函数返回一个 Promise 对象，因此我们可以通过 then 方法来指定异步操作执行成功时的回调函数，通过 catch 方法来指定异步操作执行失败时的回调函数。

```
 1 const promise = new Promise((resolve, reject) => {
 2   setTimeout(() => {
 3     resolve('promise');
 4   }, 1000);
 5 });
 6 async function asyncFunction() {
 7   try {
 8     console.log('before await');
 9     const result = await promise;
10     console.log('after await');
11     console.log(result);
12   } catch (reason) {
13     console.log(reason);
14   }
15 }
```

async & await IV

```
16 asyncFunction().then(() => {
17   console.log('asyncFunction resolved');
18 }).catch(() => {
19   console.log('asyncFunction rejected');
20 });
```

为什么要使用 async 函数？

回调地狱：在异步编程中，我们经常会遇到回调地狱的问题，当我们需要执行多个异步操作时，我们需要在每个异步操作执行成功时执行下一个异步操作，这样就会出现多层嵌套的回调函数，这种代码不仅难以阅读，而且难以维护。
例如，我们需要执行三个异步操作，每个异步操作执行成功后执行下一个异步操作，如果使用回调函数来实现，代码如下所示：

```
1 const promise1 = new Promise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('promise1');
4   }, 1000);
5 });
6 const promise2 = new Promise((resolve, reject) => {
7   setTimeout(() => {
8     resolve('promise2');
```

async & await VI

```
 9     }, 2000);
10   });
11   const promise3 = new Promise((resolve, reject) => {
12     setTimeout(() => {
13       resolve('promise3');
14     }, 3000);
15   });
16   promise1.then((result1) => {
17     console.log(result1);
18     promise2.then((result2) => {
19       console.log(result2);
20       promise3.then((result3) => {
21         console.log(result3);
22       });
23     });
24   });
```

async & await VII

async 函数可以解决回调地狱的问题，它可以让我们像编写同步代码一样编写异步代码，使得异步代码更加清晰易读。

```
const data1 = await promise1;
console.log(data1);
const data2 = await promise2;
console.log(data2);
const data3 = await promise3;
console.log(data3);
```

async & await VIII

注意: async & await 的传染性

await 关键字只能在 async 函数中使用，如果在 async 函数之外使用 await 关键字，会抛出语法错误。

React 初探

React 的基本概念 I

React 是一个用于构建用户界面的 JavaScript 库，它是 Facebook 开发的一个开源项目，它可以用于构建单页面应用程序，也可以用于构建移动端应用程序。

React 的核心思想是组件化，它将用户界面抽象成一个个组件，通过组合这些组件来构建复杂的用户界面。

React 使用 JSX 语法来描述用户界面，它是一种 JavaScript 的语法扩展，可以在 JavaScript 中编写 HTML 代码，它可以让我们在 JavaScript 中编写 HTML 代码，这样可以更加方便地描述用户界面。

React 的基本概念 II

函数式编程的思想：React 的组件是一个函数，它接收一个 props 对象作为参数，返回一个 React 元素，这个 React 元素描述了组件的用户界面。

$$UI = f(state, props)$$

其中，state 表示组件的状态，props 表示组件的属性，UI 表示组件的用户界面。

React 元素是 React 中最基本的构建块，它是一个普通的 JavaScript 对象，它描述了组件的用户界面，它包含了组件的类型、属性和子元素等信息。

React 元素是不可变的，一旦创建，就无法修改它的属性和子元素，如果需要更新 React 元素的属性或者子元素，需要创建一个新的 React 元素。

JSX 语法规则 I

JSX 是一种 JavaScript 的语法扩展，它可以在 JavaScript 中编写 HTML 代码，它可以让我们在 JavaScript 中编写 HTML 代码，这样可以更加方便地描述用户界面。

JSX 语法规则

1. 标签中混入 JavaScript 表达式时，需要用 {} 将 JavaScript 表达式包裹起来。
2. 标签中指定类名时，需要使用 className 属性，而不是 class 属性。（class 是 JavaScript 的关键字）
3. 对于内联的样式，需要使用 style 属性，需要传入一个 JavaScript 对象，而不能是 CSS 字符串。（style={{key: value}}）

JSX 语法规则 II

JSX 语法规则

4. JSX 标签必须闭合（
）
5. JSX 标签必须有唯一的根元素。（可以使用 或 <></> 包裹）
6. 标签首字母
   1. 如果首字母是小写，那么将会按照 HTML 标签的规则进行解析。
   2. 如果首字母是大写，那么将会按照 React 组件的规则进行解析。

JSX 语法规则 III

示例代码：

```
1 const element = (
2   <div className='container'>
3   <h1>Hello, world!</h1>
4   <h2>It is {new Date().toLocaleTimeString()}.</h2>
5   <div style={{ color: 'red' }}>Hello, world!</div>
6   <image src='https://www.baidu.com/img/flexible/logo/pc
        /result.png' />
7   </div>
8 );
```

创建 React 项目 I

使用 create-react-app 创建 React 项目

1. 安装 create-react-app 工具

```
1 npm install -g create-react-app
```

2. 创建 React 项目（使用 TypeScript）

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

React 组件 I

类组件与函数组件：

类组件

类组件是 React 中最基本的组件，它是一个类，它继承自 React.Component 类，它必须包含一个 render 方法，这个 render 方法必须返回一个 React 元素，它描述了组件的用户界面。

```
1 class Welcome extends React.Component {
2   render() {
3     return <h1>Hello, world!</h1>;
4   }
5 }
```

React 组件 II

函数组件

函数组件是 React 中最基本的组件，它是一个函数，它接收一个 props 对象作为参数，返回一个 React 元素，这个 React 元素描述了组件的用户界面。

```
1 type WelcomeProps = {
2   name: string;
3 };
4 function Welcome(props: WelcomeProps) {
5   return <h1>Hello, {props.name}</h1>;
6 }
```

在本课程中，我们主要使用函数组件来编写 React 组件。

state - 状态 I

state - 状态

state 是组件的状态，它是一个普通的 JavaScript 对象，它用于存储组件内部的状态数据，当组件的状态发生变化时，组件会重新渲染。

使用 useState 创建状态

useState 是 React 中的一个 Hook，它可以用于在函数组件中创建状态，它接收一个初始状态作为参数，返回一个数组，数组的第一个元素是当前的状态，数组的第二个元素是一个函数，这个函数可以用于更新状态。

state - 状态 II

示例：计数器组件

```
 1 function Counter() {
 2   const [count, setCount] = useState(0);
 3   return (
 4     <div>
 5       <p>You clicked {count} times</p>
 6       <button onClick={() => setCount(count + 1)}>
 7         Click me
 8       </button>
 9     </div>
10   );
11 }
```

state - 状态 III

注意：useState 的使用条件

useState 是 React 中的一个 Hook，它只能在函数组件中使用，如果在函数组件之外使用 useState，会抛出语法错误。

```
1 const [count, setCount] = useState(0);
```

状态的不可变性

state 是不可变的，一旦创建，就无法修改它的值，如果需要更新状态，需要调用 setCount 函数。

```
1 setCount(count + 1);
```

state - 状态 IV

状态改变时组件的重新渲染

当组件的状态发生变化时，组件会重新渲染。

```
 1 function Counter() {
 2   const [count, setCount] = useState(0);
 3   console.log('render');
 4   return (
 5     <div>
 6       <p>You clicked {count} times</p>
 7       <button onClick={() => setCount(count + 1)}>
 8         Click me
 9       </button>
10     </div>
11   );
12 }
```

props - 属性 I

props - 属性

props 是组件的属性，它是一个普通的 JavaScript 对象，它用于接收父组件传递过来的数据，它是只读的，不能修改它的值。

使用 props 传递数据

父组件可以通过 props 属性向子组件传递数据，子组件可以通过 props 属性接收父组件传递过来的数据。

props - 属性 II

示例：欢迎组件

```tsx
type WelcomeProps = {
  name: string;
};
function Welcome(props: WelcomeProps) {
  return <h1>Hello, {props.name}</h1>;
}
```

```tsx
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

props - 属性 III

子组件改变父组件的状态

子组件可以通过 props 属性调用父组件传递过来的函数，从而改变父组件的状态。

```
1 type WelcomeProps = {
2   name: string;
3   onClick: () => void;
4 };
5 function Welcome(props: WelcomeProps) {
6   return <h1 onClick={props.onClick}>Hello, {props.name
        }</h1>;
7 }
```

props - 属性 IV

```
1 function App() {
2   const [name, setName] = useState('Sara');
3   return (
4     <div>
5       <Welcome name={name} onClick={() => setName('Cahal
            ')} />
6       <h2>{name}</h2>
7     </div>
8   );
9 }
```

props - 属性 V

props 的只读性

props 是只读的，不能修改它的值，如果需要修改 props 的值，需要使用 state。

```
1 function Welcome(props: WelcomeProps) {
2   props.name = 'Cahal';
3   return <h1>Hello, {props.name}</h1>;
4 }
```

ref - 引用 I

ref - 引用

ref 是 React 中的一个 Hook，它可以用于获取 DOM 元素或者组件实例。

使用 ref 获取 DOM 元素

ref 可以用于获取 DOM 元素，它接收一个 DOM 元素作为参数，返回一个 ref 对象，这个 ref 对象的 current 属性指向 DOM 元素。

ref - 引用 II

示例：获取 DOM 元素

```
 1 function TextInputWithFocusButton() {
 2   const inputEl = useRef<HTMLInputElement | null>(null);
 3   const onButtonClick = () => {
 4     if (inputEl.current) {
 5       inputEl.current.focus();
 6     }
 7   };
 8   return (
 9     <>
10       <input ref={inputEl} type="text" />
11       <button onClick={onButtonClick}>Focus the input</
             button>
12     </>
13   );
14 }
```

ref - 引用 III

使用 ref 存储改变不会引起组件重新渲染的值

ref 也可以用于存储改变不会引起组件重新渲染的值，例如定时器的返回值。

```
 1 function Timer() {
 2   const intervalRef = useRef<number | null>(null);
 3   const [count, setCount] = useState(0);
 4   const start = () => {
 5     if (intervalRef.current) {
 6       return;
 7     }
 8     intervalRef.current = window.setInterval(() => {
 9       setCount((count) => count + 1);
10     }, 1000);
11   };
12   const stop = () => {
13     if (intervalRef.current) {
```

ref - 引用 IV

```
14          clearInterval(intervalRef.current);
15          intervalRef.current = null;
16        }
17      };
18      return (
19        <>
20          <p>{count}</p>
21          <button onClick={start}>Start</button>
22          <button onClick={stop}>Stop</button>
23        </>
24      );
25    }
```

ref - 引用 V

注意：ref 的使用条件

ref 是 React 中的一个 Hook，它只能在函数组件中使用，如果在函数组件之外使用 ref，会抛出语法错误。

```
1 const inputEl = useRef<HTMLInputElement | null>(null);
```