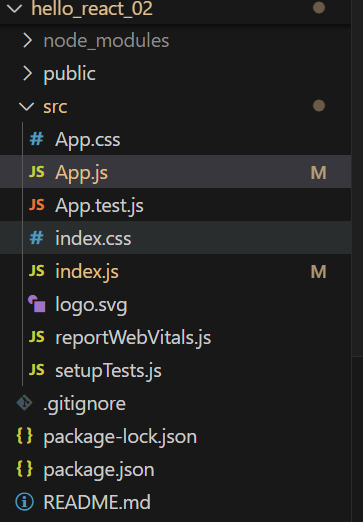
npx create-react-app XXX 创建一个新的react-app

npm start 启动服务

npm build 打包

package.json 依赖 react版本

index.js 是项目的入口

import React from 'react';

import ReactDOM from 'react-dom/client';

是React的两个核心包

App.js为项目的根组件

import App from './App';

渲染流程为：App.js渲染到index.js，然后index.js把内容渲染到public中的index.html

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <App />

);

将App渲染到id为root的根组件上。

JSX为Javascript和HTML的缩写，表示在JS代码中编写HTML模板结构

return (

    <div className="App">

      This is a react app.

    </div>

  );

Return中的就是JSX，优势1. HTML声明式模板 2. Javascript的可编程能力

JSX并不是标准的JS语法，是JS的语法扩展，需要特定的解析工具解析运行。通过{}识别Javascript的表达式，比如常见的变量，函数调用和方法调用。

1. 使用引号传递字符串

2. 使用Javascript变量

3. 函数调用和方法调用

4. 使用Javascript对象

{/\* 使用引号传递字符串 \*/}

{"This is a message."}

{/\* 使用Javascript变量 \*/}

{"Count is " + count}

{/\* 函数调用 \*/}

{getCount()}

{/\* 方法调用 \*/}

{new Date().getDate()}

{/\* 使用Javascript对象 \*/}

<div style={{color: "blue"}}>This is a div</div>

JSX实现列表渲染

需要使用原生Javascript中的map方法遍历渲染 注意需要key值

const list = [

  {id: 1001, name: "Vue"},

  {id: 1002, name: "React"},

  {id: 1003, name: "Angular"}

]

{/\* 渲染列表 \*/}

      <ul>

        {list.map(item => <li key={item.id}>{item.name}</li>)}

      </ul>

JSX实现条件渲染

通过逻辑运算符&&或者三元表达式实现

{/\* 条件渲染 \*/}

{/\* 逻辑与 && \*/}

{isLogin && <span>Please Login in</span>}

{/\* 三元运算符 \*/}

{isLogin ? <span>Username</span> : <span>Please Login in</span>}

JSX实现复杂条件渲染

通过自定义函数+if语句

const flag = 3;

// 定义函数，根据flag返回模板

function getFlagTemplate(){

  if(flag === 0){

    return <div>flag = 0</div>;

  } else if(flag === 1){

    return <div>flag = 1</div>;

  } else {

    return <div>flag = 3</div>;

  }

}

{/\* 复杂情况渲染 \*/}

{getFlagTemplate()}

React进行事件绑定

on + 事件名称 = {事件处理程序} 小驼峰

const clickButton = () => {

  console.log("click");

}

<button onClick={clickButton}>Click</button>

使用事件对象参数

通过设置形参e实现

// 事件参数

const clickButton1 = (e) => {

  console.log("click", e);

}

{/\* 获得对象参数 \*/}

<button onClick={clickButton1}>Click1</button>

传递自定义参数

通过箭头函数实现，事件需要绑定一个函数引用

// 传递自定义参数

const clickButton2 = (id) => {

  console.log("click", id++);

}

{/\* 传递自定义参数 \*/}

<button onClick={() => clickButton2(0)}>Click2</button>

同时传递事件对象和自定义参数

改造两处

// 传递事件参数和自定义参数

const clickButton3 = (id, e) => {

  console.log("click", ++id, e);

}

{/\* 传递事件参数和自定义参数 \*/}

<button onClick={(e) => clickButton3(0, e)}>Click3</button>

React中的组件

组件定义：一个组件就是用户界面的一部分，有自己的逻辑和外观，组件之间可以互相嵌套，也可以复用多次。

一个组件就是首字母大写的函数，渲染只需要按照标签来写

function Button() {

  return <button>Click Button1</button>

}

{/\* React组件 \*/}

<Button></Button>

const Button2 = () => {

  return <button>Click Button2</button>

}

<Button2></Button2>

useState

useState是一个React Hook(函数), 它允许我们向组件添加一个状态变量，从而影响组件的渲染结果。

当状态变量发生改变，UI试图也会跟着变（数据驱动视图）

const [count1, setCount] = useState(0);

useState是一个函数，返回值是一个数组

数组中的第一个参数是状态变量，第二个参数是set函数用来修改状态变量

useState的参数将作为count1的初始值

  //useState实现一个计数器，点击自增

  //1. 调用useState添加一个状态变量

  //count1 就是状态变量

  //setCount 就是修改状态变量的方法

  const [count1, setCount] = useState(0);

  //2. 点击事件回调

  const handleClick = () => {

    //作用： 1. 修改count1的值 2. 重新使用新的count1渲染UI

    setCount(count1 + 1);

  }

  {/\* useState函数 \*/}

  <button onClick={handleClick}>{count1}</button>

React修改状态的规则

状态不可变

在React中，状态被认为是只读的，只能替换而不能修改

  let [count2, setCount2] = useState(0);

  const handleClick2 = () => {

    //直接修改无法引发视图更新

    // count2++;

    // console.log(count2);

    setCount2(count2++);

  }

  {/\* useState修改状态规则 \*/}

  <button onClick={handleClick2}>{count2}</button>

修改对象状态

  const [form, setForm] = useState({name : "1"})

  const changeForm = () => {

    setForm({

      ...form,

      name : "2"

    })

  }

<button onClick={changeForm}>Change Form Name {form.name}</button>

React组件基础样式方案

通过className控制

.foo {

  color: blue;

  font-size: 10px;

}

import './index.css'

{/\* React基础样式控制 \*/}

{/\* 通过className控制 \*/}

<span className='foo'>This is className foo</span>

案例实现：评论

1. 渲染评论列表

使用useState维护评论列表

使用map方法对列表数据遍历渲染

2. 删除评论

通过比较判断user.uid === item.user.uid是否成立，显示删除按钮

通过点击事件，使用filter进行过滤

setCommentList(commentList.filter(item => item.rpid !== id));

3. 渲染Tab+点击高亮显示

点击时把type(独一无二的标识)记录下来，然后和遍历时的每一项type匹配，匹配到就负责高亮类名。

className={`nav-item ${type === item.type && 'active'}`}

4. 排序功能

import \_ from 'lodash'

setCommentList(\_.orderBy(commentList, 'like', 'desc'));

https://lodash.com/docs/4.17.15#sortBy

classnames优化类名控制

方便的通过条件动态控制class类名的显示

import classNames from 'classnames'

{tabs.map(item => <span className={classNames('nav-item', {active: type === item.type})} onClick={() => changeTabHandle(item.type)} key={item.type}>{item.text}</span>)}

受控表单绑定

使用React组件的状态（useState）控制表单的状态

1. 准备一个React状态值

React

(state)

<Input />

(value)

state绑定到input的value属性

把input最新的value值设置给state

2. 通过value属性绑定状态，通过onChange属性绑定状态同步的函数

  //受控表单绑定

  //1. 声明一个react状态 - useState

  //2. 核心绑定流程

  //2.1 通过value属性绑定react状态

  //2.2 绑定onChange事件 通过事件参数e拿到输入框最新的值 反向修改react状态

  const [value, setValue] = useState('');

{/\* 表单受控绑定 \*/}

 <input value={value} type='text' onChange={(e) => setValue(e.target.value)}></input>

React中获取DOM

使用useRef钩子函数

1. 使用useRef创建ref对象，并于JSX绑定

2. 在DOM可用时，通过inputRef.current拿到DOM对象

  const inputRef = useRef(null);

  const showDOM = () => {

    console.log(inputRef.current.value);

  }

  {/\* React中获取DOM \*/}

  <input type='text' ref={inputRef}></input>

  <button onClick={showDOM}>Show DOM</button>

案例实现：评论

发表评论

1. 获取评论内容

2. 点击发布按钮发布评论

value={commentValue}

onChange={(e) => setCommentValue(e.target.value)}

<div className="send-text" onClick={submitComment}>发布</div>

  const [commentValue, setCommentValue] = useState('');

  const submitComment = () => {

    setCommentList([

      ...commentList,

      {

        rpid: 4,

        user: {

          uid: user.uid,

          avatar: user.avatar,

          uname: user.uname,

        },

        content: commentValue,

        ctime: '10-19 09:00',

        like: 0,

      }

    ])

  }

rpid处理和时间处理

1. rpid通过uuid实现

2. ctime通过dayjs生成固定格式

import { v4 as uuidV4 } from 'uuid'

rpid: uuidV4(),

import dayjs from 'dayjs'

ctime: dayjs(new Date()).format('MM-DD hh:mm'),

清空内容并重新聚焦

1. 清空内容可以把控制input的value设置为空

setCommentValue('');

2. 只用DOM的focus方法

  const commentRef = useRef(null);

   <textarea

   className="reply-box-textarea"

   placeholder="发一条友善的评论"

   value={commentValue}

   ref={commentRef}

   onChange={(e) => setCommentValue(e.target.value)}

   />

    //使用DOM重新聚焦

    commentRef.current.focus();

组件通信

组件通信就是组件之间的数据传递

父子通信，兄弟通信，跨层通信

将父组件的数据传递给子组件使用

实现步骤：

1. 父组件传递数据：在子组件标签上绑定数据

2. 子组件接收数据：子组件通过props参数接收数据

//父传子

//1. 父组件传递数据 子组件身上绑定属性

//2. 子组件接收数据 props参数接收

function Son1(props) {

  //props: 是一个对象，包含了父组件中传来的所有数据

  return <div>{props.name}</div>;

}

const name = "This is app name"

{/\* 组件通信-父传子 \*/}

{/\* name为属性名，可以自由定义 \*/}

<Son1 name={name}></Son1>

props说明

1. props可以传递任意的数据

数字、字符串、布尔值、数组、对象、函数、JSX



2. props是只读对象

子组件只能读取props中的数据，不能修改，修改需要父组件完成。

3. 特殊的prop children

当我们把内容嵌套在子组件标签中时，父组件会自动在名为children的prop属性中接收该内容

在标签内写东西，会自动识别，放在props.children中

      <Son2>

        <span>this is span.</span>

      </Son2>

function Son2(props) {

  return <div>This is son2. {props.children}</div>;

}

子传父

在子组件中调用父组件中的函数并传递参数

父组件定义函数，将函数发送给子组件。子组件返回值给父组件，让父组件调用

  const getMeg = (msg) => {

    console.log(msg);

  }

      {/\* 子传父 \*/}

      <div>

        this is app

        <Son3 onGetSon3Msg = {getMeg} />

      </div>

function Son3({onGetSon3Msg}) {

  const sonMsg = 'This is son3 message';

  return (

  <div>

    this is son3

    <button onClick={() => onGetSon3Msg(sonMsg)}>sendMsg</button>

  </div>

  )

}

可以通过useState修改状态

使用状态提升实现兄弟组件通信

通过共同的父组件进行兄弟组件之间的数据传递 子传父+父传子

function Son4({sendMegToSon5}){

  const msg = "This is son4 message";

  return(

    <button onClick={() => sendMegToSon5(msg)}>Click To Son5</button>

  )

}

function Son5(props){

  return(

    <div>Son5 {props.msg}</div>

  )

}

  const [son4Msg, setSon4Msg] = useState('');

  const sendMsgToSon5 = (msg) => {

    setSon4Msg(msg);

  }

{/\* 使用状态提升完成兄弟间通信 \*/}

      <div>

        Son4

        <Son4 sendMegToSon5 = {sendMsgToSon5}></Son4>

      </div>

      <div>

        <Son5 msg = {son4Msg}></Son5>

      </div>

使用Content机制跨层级组件通信

1. 使用createContext方法创建一个上下文对象Ctx

2. 在顶层组件（APP）中通过Ctx,Provider组件提供数据

3. 在底层组件（B）中通过useContext钩子函数获取消费数据

{/\* 使用content机制跨层级组件通信 \*/}

      {/\* 1. 使用createContext方法创建一个上下文对象Ctx

          2. 在顶层组件（APP）中通过Ctx,Provider组件提供数据

          3. 在底层组件（B）中通过useContext钩子函数获取消费数据 \*/}

  const appMsg = 'This is app msg'

<div>

        <MsgContext.Provider value={appMsg}>

          This is APP

          <A />

        </MsgContext.Provider>

      </div>

// APP -> A -> B

const MsgContext = createContext();

function A() {

  return(

    <div>

      This is A component

      <B />

    </div>

  )

}

function B() {

  const msgB = useContext(MsgContext);

  return(

    <div>

      This is B component {msgB}

    </div>

  )

}

useEffect的概念理解

useEffect是一个React Hook函数，用于React组件中创建不是由事件引起而且是由渲染本身引起的操作，例如AJAX

需求：在组件渲染完毕后，立刻从服务端获取频道列表数据并显示到页面中

useEffect(() => { }, [ ])

参数1是一个函数，名为副作用函数，在函数内部可以放置要执行的操作

参数2是一个数组（可选），在数组中放置依赖项，不同依赖会影响第一个参数函数的执行，当一个空数组的时候，副作用函数只会在组件渲染完毕后执行一次

const URL1 = 'http://geek.itheima.net/v1\_0/channels';

  const [dataList, setDataList] = useState([])

  useEffect(() => {

    //额外的操作 获取频道列表

    async function getList() {

      const res = await fetch(URL1)

      const list = await res.json()

      setDataList(list.data.channels);

    }

    getList()

  }, [])

{/\* useEffect的概念理解 \*/}

          <div>

            <ul>

              {dataList.map(item => <li key={item.id}>{item.name}</li>)}

            </ul>

          </div>

根据传入依赖项的不同，会有不同的执行表现

|  |  |
| --- | --- |
| 依赖项 | 副作用函数的执行时机 |
| 没有依赖项 | 组件初始渲染+组件更新时执行 |
| 空数组依赖 | 只在初始渲染时执行一次 |
| 添加特定依赖项 | 组件初始渲染+特性依赖项变化时执行 |

  // 1. 没有依赖项 初始渲染时执行+组件更新

  const [count3, setCount3] = useState(0)

  useEffect(() => {

    console.log("副作用函数执行1");

  }) //任何组件的变化都会重新刷新

  // 2. 传入空依赖 初始渲染时执行

  useEffect(() => {

    console.log("副作用函数执行2");

  }, [])

  // 3. 传入特定的依赖项 初始渲染时执行+依赖项变化时执行

  useEffect(() => {

    console.log("副作用函数执行3");

  }, [count3]) //只有count3发生了改变才会改变，高度绑定count3

      {/\* useEffect的不同依赖项 \*/}

          <button onClick={() => setCount3(count3 + 1)}>+{count3}</button>

useEffect清除副作用

useEffect编写的由渲染本身引起的对接组件外部的操作，社区称为副作用操作，比如在useEffect开启一个定时器，在组建卸载时把定时器清理掉，该过程就是清理副作用。

useEffect(() => {

//副作用实现逻辑

return () => { //清除副作用逻辑

}

}, [ ])

清除副作用函数最常见的执行时机就是在组件卸载时自动执行

在Son6组件中开启一个定时器，卸载时清除这个定时器

function Son6() {

  //1. 渲染时开启一个定时器

  useEffect(() => {

    const timer = setInterval(() => {

      console.log("clock execute");

    }, 1000)

    return() => {

      //清除副作用

      clearInterval(timer);

    }

  }, [])

  return <div>This is Son6</div>

}

  const [show, setShow] = useState(true);

{/\* useEffect清除副作用 \*/}

          <div>

            {show && <Son6 />}

            <button onClick={() => setShow(false)}>Delete Son6</button>

          </div>

自定义Hook函数

指以use打头的函数，通过自定义hook函数可以用来实现逻辑的封装和复用

  //该布尔切换的逻辑 当前组件耦合在一起 不方便复用

  //自定义Hook解决

  function useToggle() {

    //可复用的逻辑代码

    const [divShow, setDivShow] = useState(true);

    const changeDivStatus = () => setDivShow(!divShow)

    //哪些状态和回调函数需要在其他组件中使用 return

    return {

      divShow,

      changeDivStatus

    }

  }

  // 封装自定义hook的通用思路

  // 1. 声明一个以use打头的函数

  // 2. 在函数体内封装可复用的逻辑（只要是可复用的逻辑）

  // 3. 把组件用到的状态或者回调return出去（以对象或者数组）

  // 4. 在哪个组件中要用到这个逻辑，就执行这个函数，解构出状态和回调

  const {divShow, changeDivStatus} = useToggle()

{/\* 自定义Hook函数 \*/}

      <div>

        {divShow && <div>This is div</div>}

        <button onClick={changeDivStatus}>toggle</button>

      </div>

React Hooks使用规则

1. 只能在组件中或者其他自定义Hook函数中调用

2. 只能在组件的顶层调用，不能嵌套在if, for, 其它函数中

优化B站评论demo

1. 使用请求接口的方式获取评论列表并渲染

2. 使用自定义Hook函数封装数据请求的逻辑

3. 把评论中的每一项抽象成一个独立的组件实现渲染

使用json-server工具模拟接口服务，通过axios发送接口请求

使用useEffect调用数据

  const [commentList, setCommentList] = useState([])

  useEffect(() => {

    //请求数据

    async function getData(){

      //axios请求数据

      const res = await axios.get('http://localhost:3004/list')

      setCommentList(res.data)

    }

    getData()

  }, [])

//封装

function useDataLoader() {

    //获取接口数据渲染

    const [commentList, setCommentList] = useState([])

    useEffect(() => {

      //请求数据

      async function getData(){

        //axios请求数据

        const res = await axios.get('http://localhost:3004/list')

        setCommentList(res.data)

      }

      getData()

    }, [])

    return {

      commentList,

      setCommentList

    }

}

  const {commentList, setCommentList} = useDataLoader()

抽象原则：只能组件和UI组件分离

//封装Item组件

function Item({item, onDel}) {

  return (

    <div className="reply-item">

            {/\* 头像 \*/}

            <div className="root-reply-avatar">

              <div className="bili-avatar">

                <img

                  className="bili-avatar-img"

                  alt=""

                  src={item.user.avatar}

                />

              </div>

            </div>

            <div className="content-wrap">

              {/\* 用户名 \*/}

              <div className="user-info">

                <div className="user-name">{item.user.uname}</div>

              </div>

              {/\* 评论内容 \*/}

              <div key={item.rpid} className="root-reply">

                <span className="reply-content">{item.content}</span>

                <div className="reply-info">

                  {/\* 评论时间 \*/}

                  <span className="reply-time">{item.ctime}</span>

                  {/\* 评论数量 \*/}

                  <span className="reply-time">点赞数:{item.like}</span>

                  {/\* 判断user.uid === item.user.uid \*/}

                  { user.uid === item.user.uid &&

                    <span className="delete-btn" onClick={() => onDel(item.rpid)}>

                      删除

                    </span>}

                </div>

              </div>

            </div>

          </div>

  )

}

{commentList.map(item => <Item key={item.id} item={item} onDel={deleteHandler}></Item>)}