用户组件

跨框架的组件

js 模块认知

手写createElement函数，增进对jsx的理解

cookie的基本认知

webpack打包npm包

by eric wen

分析UI

l 分未登录和登录两种状态，不同的状态有不同的UI

l 未登录状态下，提供登录注册功能

l 登录状态下，显示用户信息，提供基础的功能连接跳转

l 提供一个无UI模式，可以在应用需要的时候，唤起登录面板

l 特殊需求：需要跨框架使用，在react\vue\原生js环境均可直接使用

特殊需求的实际场景讲解：

l 老项目，新需求。一家有年头的盈利的公司都是有老项目的，而且老项目承担着盈利核心业务，不能轻易舍去，每次二次开发时，都要小心翼翼。用三方包的形式开发，可以在老项目直接引用。

l 老项目，新项目，同时开发新功能。需要在老项目用传统js开发一遍，在新项目用react或者vue再开发一遍，相同的逻辑和功能。

l 通用性功能，不管公司有多少新老项目，在用户端，都需要保持风格一致性。如行为验证码，登录注册，订单付款等等。这种就需要用三方包的形式开发，所有项目直接引用

这是一个非常实际的，所有人都会面临的情况。

如果不会，在面临时，只能用最笨的方法。

这也是开发独立js包的方法。是必学的。

组件设计

创建实例，并传入挂载dom，如未传入，默认无UI模式。

例：Const user=new user({root:Element})

入参设计

参数	默认值	描述
root	无	挂载dom容器，包体dom将挂载在这个容器下
isUI	false	是否为UI模式

暴露函数设计

函数	入参	功能
getUser	无	获取用户基本信息
openLogin	IsModal:false	唤起登录面板，默认弹框，true，则跳转到统一登录页面，登录页面可在配置中心配置，更新时可全应用自动更新
openRegister	isModal:false	唤起注册面板,默认弹框，true，则跳转到统一注册页面，注册页面可在配置中心配置，更新时可全应用自动更新
Logout		退出登录

事件设计

onInit	用户组件初始化成功时触发 传递用户用户登录状态
onChange	用户状态变化时触发，如会员升级等影响业务变化的数据变化
onLogout	用户退出
onLogin	用户登录

内部逻辑

时序	函数	
实例创建	Init	查看cookie信息，无token即未登录，有token，则调用接口，后端返回用户信息则为登录，否则为失败，手动删除cookie 触发oninit
渲染UI	renderUI	init完成后，渲染UI
用户交互登录面板	renderLogin	渲染登录面板
登录	Login	调用登录接口，触发onlogin
用户注册面板	renderRegister	渲染注册面板
退出	Logout	调用退出接口，触发onlogout

组件开发

包工程

理论上，我们用原生js把逻辑封装在function里，然后把function挂在window下面，这样就可以在任何框架下使用。

跨框架的本质就是使用原生js。

但是这样的话，在react工程下，不能使用import

所以，为了让我们的包使用方便，我们打包的时候打成umd模式，即兼容commonjs、amd 全局变量

什么是模块化?

模块化，即一个一个js文件。

在其它语言里，是天然支持的，但js不是。

js在原生运行环境，想要加载相互依赖的js包，就要按顺序显示引用，极易出错

所以就需要一个在文件内部，用代码的方式引用。

如:

import XX from ‘xx’

Const xx=require(‘xx’)

但原生环境不支持。

于是开发者就发明了模块化方案，以解决这种js先天缺陷。

简单了解一下js相关的模块化方案。

1、CommonJS 最初由Node.js采用，CommonJS是一种同步加载的模块系统，主要用于服务器端。它使用require()函数来导入模块，并通过module.exports或exports对象导出模块的功能。

```
// 导入模块 const moduleA = require('./moduleA');

// 导出功能 module.exports = { sayHello: function() { console.log('Hello!'); } };
```

2、AMD (Asynchronous Module Definition) AMD是为浏览器环境设计的一种异步加载模块的方式，主要解决浏览器端模块加载效率问题。RequireJS是一个实现了AMD规范的库。 // 定义一个模块 define(['dependency'], function(dependency) { return { // 模块功能 }; });

```
// 使用模块 require(['moduleA'], function(moduleA) { // 使用moduleA中的功能 });
```

3、ES6 Modules,现在我们最常见的 // 导出 export const myFunction = () => console.log('Hello from ES6 module!');

```
// 导入 import { myFunction } from './myModule.js'; myFunction();
```

4、. Umd

UMD是一种旨在兼容多种环境（包括浏览器全局变量、AMD、CommonJS等）的模块定义模式。它使得同一个模块可以在不同的JavaScript环境中工作

```
(function (root, factory) {

if (typeof define === 'function' && define.amd) {

// AMD

define(['dependency'], factory);

} else if (typeof exports === 'object') {

// CommonJS

module.exports = factory(require('dependency'));

} else {

// 浏览器全局变量

root.returnExports = factory(root.dependency);

}

}(this, function (dependency) {

// 模块实现

}));
```

创建工程

新建文件夹 eshop-user

npm install --save-dev webpack webpack-cli

在项目的根目录下创建一个名为webpack.config.js的文件，并添加如下配置

```
const path = require('path');

module.exports = {
    mode: 'production', // 设置为 production 以优化输出
    entry: './src/index.js', // 入口文件
    output: {
        filename: 'eshopUser.js', // 输出文件名
        path: path.resolve(__dirname, 'dist'), // 输出目录
        library: {
            name: 'EshopUser',
            type: 'umd',
            export: 'default'
        }
    },
};
```

umd包验证

新建一个html引用包

开发内部逻辑

1、获取用户登录态

2、渲染UI

3、执行钩子函数

4、引入css

5、处理css ，安装loader和plugins、配置webpackconfig

6、开发登录

7、开发注册

cookie

Cookie是前端的本地存储方案之一。

但前端现在很少主动使用。

主要是用于存储token钥匙，每次http接口发出，都会自动携带同域cookie，后端需要这些cookie来判断安全性和用户信息。

存储cookie原生方法

```
function setCookie(name, value, days) {

let expires = "";

if (days) {

const date = new Date();

date.setTime(date.getTime() + (days  24  60  60  1000));

expires = "; expires=" + date.toUTCString();

}

document.cookie = name + "=" + (value || "") + expires + "; path=/";

}
```

解析cookie原生方法

```
const cookieStr = document.cookie;

// 拆开cookie字符串，然后遍历

const cookies = {}

// cookie的使用方法

cookieStr.split(';').forEach(function (cookie) {

var parts = cookie.split('=');

var name = parts[0].trim();

var value = parts[1].trim();

cookies[name] = value;

});
```

渲染UI的方法

1、字符串模板，缺点，绑定事件麻烦

2、创建如react和vue那样的createElement工具函数，可动态绑定

createElement

直接看代码，一看就会

```
/**
 * 创建DOM元素的辅助函数
 * @param {Object} options - 元素配置选项
 * @param {string} options.tag - 元素标签名，默认为'div'
 * @param {string} options.className - 元素的类名
 * @param {string} options.text - 元素的文本内容
 * @param {Array} options.children - 子元素列表
 * @param {Object} options.attrs - 元素属性键值对
 * @param {Object} options.events - 元素事件处理函数键值对
 * @returns {HTMLElement} 创建的DOM元素
 */
export function createElement(options) {
    const {
        tag = 'div',
        className = '',
        text = '',
        children = [],
        attrs = {},
        events = {},
    } = options;

    const element = document.createElement(tag);

    // 添加类名
    if (className) {
        element.className = className;
    }

    // 添加文本
    if (text) {
        element.textContent = text;
    }

    // 添加属性
    Object.entries(attrs).forEach(([key, value]) => {
        element.setAttribute(key, value);
    });

    // 添加事件
    Object.entries(events).forEach(([event, handler]) => {
        element.addEventListener(event, handler);
    });

    // 添加子元素
    children.forEach(child => {
        element.appendChild(child);
    });

    return element;
}
```

继续开发

3、完善渲染逻辑

4、引入css

5、处理css ，安装loader和plugins、配置webpackconfig

6、开发登录

7、开发注册