前端项目复盘 （常用技术点回顾）

一. 数据暴露与引入

1. 1数据暴露：

> 有3种形式：分别暴露、统一暴露、默认暴露

```
export let school="lccd";
export let teach=(params)=>{
    console.log(`我可以教你${params}`);
}
// 分别暴露




let school2="zjutuser";
let read=(params)=>{
    console.log(`我会读${params}`);
}

export {
    school2,
    read,
}
//统一暴露




export default{
    school3:"ZJUT",

    sleep:()=>{
        console.log(`我会睡觉`);
    }

}
//默认暴露
```

1.2数据引入：

> 有3种形式：通用引入、解构引入、简便形式引入

通用引入：

```
import * as obj from "./part1.js";
import * as obj2 from "./part2.js";
import * as obj3 from "./part3.js";

//通用的导入形式（注意：默认暴露的数据，需要加一层default来寻找）




console.log(obj.school);
obj.teach("haha")

console.log("-----------------------------------");

console.log(obj2.school2);
obj2.read("水浒传")



console.log("-----------------------------------");

console.log(obj3.default.school3);
obj3.default.sleep()
//默认暴露的数据要使用加一个default


//通用形式的使用
```

解构引入：

```
import {school,teach} from "./part1.js";
import {school2,read} from "./part2.js";
import {default as m1} from "./part3.js";


//解构赋值的形式




console.log(school);
teach("数学！ ");

```

```
16
17
18  console.log("----------------------------------");
19
20
21
22  console.log(school2);
23  read("西游记");
24
25
26
27  console.log("----------------------------------");
28
29
30
31  console.log(m1.school3);
32  m1.sleep();
33
34
35  //解构形式的使用
```

简便形式：

```
1  import m2 from "./part3.js"
2  //简便的形式（只能针对默认暴露）
3
4
5  console.log(m2.school3);
6  m2.sleep();
7
8  //简便形式的使用
```

1.3总结与注意：

```
1  //所以一般来说（最适合）：
2  //1.通用引入对付分别暴露
3  //2.解构赋值对付同意暴露
4  //3.简便写法对付默认暴露（因为默认暴露用其他解决，太别扭了）
5  //一般在REACT/VUE的项目中更频繁使用默认暴露的形式
```

```
1
2  //在node.js中默认不支持import 只支持commonJS
3
4  //想使用import 需要将package.json分件里面的type改成module，如下面一样：
```

```
{
 "name": "test_use",
 "version": "1.0.0",
 "description": "",
 "main": "app.js",
 "type":"module",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "",
 "license": "ISC"
}

```

二. React组件动态加载

2.1理论：

> **React项目动态加载的主要作用是提高应用程序的性能和效率。通过动态加载，我们可以将组件和模块分离成更小的块，并在需要时动态地加载它们**
> *当React项目的代码量较大时，就需要考虑动态加载优化，以提高应用程序的性能和效率。动态加载优化可以帮助我们将组件和模块分离成更小的块，并在需要时动态地加载它们。这样做可以减少初始加载时间，从而提高应用程序的响应速度*

需要使用动态加载的场景：

1. 应用程序使用了大量的第三方库或框架。如果我们不对这些库进行按需加载，那么它们的代码可能会被打包到应用程序中，从而增加了应用程序的体积和加载时间
2. 应用程序包含大量组件或页面。如果我们一次性将所有组件都加载到应用程序中，那么应用程序的初始加载时间可能会非常长。相反，如果我们只加载当前所需的组件或页面，那么应用程序的响应速度会更快
3. 应用程序需要支持低网络环境。在低网络环境下，大量的静态资源文件可能会导致应用程序加载缓慢或者加载失败。通过动态加载优化，我们可以减少初始加载的大小，从而提高应用程序的稳定性和可用性

2.2 组件的动态加载：

> **首先：因为lazy,和Suspense标签在Hooks与Class中均可以使用，所以用不用hooks基本没影响**

开始，我们将创建两个组件：父组件（fa.js）和子组件（son.js）。在父组件中，我们将使用React的useState和useEffect hooks来动态加载子组件。以下是详细的步骤：

1. 在son.js中创建一个简单的子组件
2. 在fa.js中创建一个父组件
3. 使用useState hook初始化一个名为 `isChildvisible` 的状态变量，用于控制子组件的显示
4. 使用useEffect hook监听 `isChildvisible` 状态变量的变化，当它变为true时，动态导入子组件并更新状态
5. 在父组件的渲染函数中，根据 `isChildvisible` 状态变量决定是否显示子组件

具体的代码实现：

```jsx
import React from 'react';

const Son = () => {
 return <div> 即将加载的子组件 </div>;
};

export default Son;

//son.js的代码（和正常的组件没任何区别
```

```jsx
import React, { useState, lazy, Suspense } from 'react';

// 使用 lazy 函数动态加载子组件
const SonComponent = lazy(() => import('./son'));

export default function FaComponent() {
 const [showSon, setShowSon] = useState(false);

 return (
  <div>
   <button onClick={() => setShowSon(!showSon)}>Toggle Son Component</button>

   {showSon && (
    // 使用 Suspense 组件包裹动态加载的组件，并设置 fallback 属性
    <Suspense fallback={<div>正在加载！请稍后....</div>}>
     <SonComponent />
    </Suspense>
   )}
  </div>
 );
}


// father.js的实现
```

2.3 组件库的按需引入：

> **每个组件库的按需加载形式不一样，需要看官网的页面介绍，或者可以直接使用babel-plugin-import工具来实现**

下面以semi-design组件库为例展示实现：

1. 先安装 `babel-plugin-import` 和 `babel-preset-semi-design` ：

```
yarn add babel-plugin-import babel-preset-semi-design

/*
babel-preset-semi-design和babel-plugin-import是两个不同的Babel插件，它们各自扮演不同的角色。以下是它们的区别
```

```
babel-preset-semi-design 是一个预设（preset），它是一组预定义的Babel插件集合，可以一次性加载多个插件，以便于开发者快速配置Babel环境。babel-preset-semi-design包含了许多用于转换ES6+语法和最新JavaScript特性的插件，这些插件可以在打包时将代码转换为浏览器能够识别的语法。

babel-plugin-import 是一个按需加载组件的插件。它会根据需要动态导入组件，从而减小应用程序的打包体积。当使用import { Button } from 'antd'这种方式引入Ant Design的Button组件时，babel-plugin-import会自动将其转换为import Button from 'antd/lib/button'这种按需加载方式。



因此，如果您想要使用Semidesign的UI组件库，并且希望实现按需加载组件的功能，可以同时安装babel-plugin-import和babel-preset-semi-design插件，分别用于实现按需加载和ES6+语法转换等功能。

需要注意的是，这两个插件虽然可以一起工作，但是它们的使用方法和作用是不同的。在使用时，需要根据自己的需求选择是否使用这两个插件，并合理配置它们。
*/
```

2. 在 `.babelrc` 文件中添加配置：

```
{
  "presets": ["semi-design"],
  "plugins": [
    ["import", { "libraryName": "@semicomplete/semidesign", "style": true }]
  ]
}

```

3.在代码中按需引入Semidesign组件，例如：

```
import React from 'react';
import { Button } from '@semicomplete/semidesign';

const MyComponent = () => {
  return <Button>Click me</Button>;
};

export default MyComponent;

```

2. 4路由组件：

路由组件的动态即在会在封装路由表那块详细讲：

三. React-RouterV6构建路由表

主要就分为两步就可以了 （**第一步：创建路由映射，第二部：注册路由表 第三部：使用组件映射**）

3.1 封装路由文件：

```
import { Navigate } from 'react-router-dom'
//跳转引入

import Login from '../pages/Login/Login'
import MainPage from '../pages/MainPage/MainPage'
import AddingPage from '../pages/AddingPage/AddingPage';
import ListPage from '../pages/ListPage/ListPage';
import ListPageMobile from '../pages/ListPage/ListPageMobile';
import SettingPage from '../pages/SettingPage/SettingPage';
import FilterPage from '../pages/Filter/FilterPage';
//页面的引入

let routerTab=[
    {
        path:'/login',
        element:<Login/>
    },
    {
        index:true,
        element:<Navigate to="/login"/>
    },//路由重定向到登录界面
    {
        path:"/index",
        element:<MainPage/>,
        children:[
            {
                path:"/index/adding",
                element:<AddingPage/>
            },
            {
                path:"/index/listDesktop",
                element:<ListPage/>
            },
            {
                path:"/index/listMobile",
                element:<ListPageMobile/>
            },
            {
                path:"/index/filter",
                element:<FilterPage/>
            },
            {
                path:"/index/setting",
                element:<SettingPage/>
            },
            {
                index:true,
                element:<Navigate to="/index/setting"/>
```

```
        },
      ]
    }
]
//routerTab是集中的路由对象，将他暴露出来


export default routerTab;
```

3.2 注册路由：

在高阶组件部分添加（*往往是app.js文件或入口文件的第一级组件*）

```
import routerTab from "./config/router";
import { useRoutes } from "react-router-dom";

function App() {
  const routerMap=useRoutes(routerTab);//注册路由映射表


  return (
    <div className="App">
      {routerMap}
      {/* 路由映射表 */}
    </div>
  );
}

export default App;

```

3.3 使用组件映射：

```
import { Outlet, useLocation, useNavigate } from 'react-router-dom';




//在需要使用该组件的地方加上outlet就行
<div>
  <div> 固定组件 </div>
  <div> <outlet/> </div>
</div>

//这个容器报了两个组件，一个是固定组件，一个是路由映射组件爱你
```

四. 新一代的状态管理库--zustand

4.1 理论介绍：

> **基于 `Flux` 模型实现的小型、快速和可扩展的状态管理解决方案，拥有基于 `hooks` 的舒适的API，非常地灵活且有趣.**

为什么是 zustand 而不是 redux?

- 轻巧灵活
- 将 `hooks` 作为消费状态的主要手段
- 不需要使用 `context provider` 包裹你的应用程序
- 可以做到瞬时更新(不引起组件渲染完成更新过程)

为什么是 zustand 而不是 react Context?

- 不依赖 `react` 上下文，引用更加灵活
- 当状态发生变化时 `重新渲染的组件更少`
- 集中的、基于操作的状态管理

为什么是 zustand-vue 而不是 pinia?

- 基于不可变状态进行更新，`store` 更新操作相对更加可控
- 将 `composition api` 作为消费状态的主要手段，而不是 Vue.use 全局注入

4.2流程与步骤：

4.3代码使用（小案例）

4.3.1 定义store仓库：

```
import { create } from 'zustand'//导入zustand的create方法

export const useStuStore = create((set) => ({
    schoolArr:["zjut","zju","pku","unn"],
    infObj:[{score:122,name:"lhc"},{score:111,name:"lhp"},{score:145,name:"lhd"}],//上面两个是仓库的初始值


    addSchool: () => {
        set((state) => ({schoolArr:[...state.schoolArr,obj]}))
    }//定义仓库的相应reducer


}))

```

4.3.2 组件调用：

```
// stu组件中的代码
import { useStuStore } from '../store/student'//导入创建好的仓库

const student = () => {
   const { schoolArr, infoObj } = useStuStore()//直接解构仓库的同时，就直接创建了状态对象不需要在useState了
   const { addSchool } = useStuStore()//由于是全局状态（所以reducer可以在全局任意一个组件中使用，这里不麻烦了，只在本组件中使用了）
   return (
      <div>
         {
            schoolArr.map((val,index)=>{
               return(
                  <div>
                     <span>学校{index}:</span>
                     <span>{val[index]}</span>
                  </div>
               )
            })
         }
         <button onclick={(next)=>{addSchool(next)}}></button>
      </div>
   )
}
export default Person

```

五. 使用Hooks封装小型状态管理库：

5.1使用场景和原因

1. **简单易用：**相比redux等大型状态管理库，自己封装的状态管理库更加简单易用。它不需要太多的配置和学习成本，也不需要引入额外的依赖。
2. **轻量化：**自己封装的状态管理库通常只包含应用程序所需的最基本功能，因此它更轻量化，可以提高应用程序的性能。
3. **更好的可维护性：**在自己封装的状态管理库中，我们可以更容易地理解和调试代码，因为它是自己编写的。这也使得我们更容易维护代码，并对其进行修改和扩展。
4. **更灵活的设计：**自己封装的状态管理库可以根据应用程序的需求进行设计，而不需要考虑其他库的约束。这意味着我们可以实现一些特定的业务逻辑，同时还可以保持代码的简洁性和可读性。
5. **更好的类型安全：**使用TypeScript可以实现更好的类型安全，减少了出错的可能性。
6. **更好的开发体验：**使用hooks(useReducer和useContext)可以让我们更方便、快速地构建应用程序，提高开发效率和体验。

5.2基本的代码实现

5.2.1 构建仓库：

> 封装的仓库应该包含：以下几个要素：状态数据初始值、状态的reducer、承接状态的上下文、整体的状态管理器

```
import React, { createContext, useReducer } from 'react';

const initialState = {
  count: 0,
}; //真正的状态数据（初始化的状态）

const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    case 'DECREMENT':
      return { ...state, count: state.count - 1 };
    default:
      throw new Error(`Unhandled action type: ${action.type}`);
  }
};//构建reducer（处理操作的函数）



export const StoreContext = createContext();

export const StoreProvider = ({ children }) => {
  const [state, dispatch] = useReducer(reducer, initialState);
  const value = { state, dispatch };
  return <StoreContext.Provider value={value}>{children}</StoreContext.Provider>;
};
/*
在这里，我们导出了一个StoreContext对象和一个StoreProvider组件。StoreProvider组件使用useReducer()钩子函数来创建状态管理器，并将其作为值传递给StoreContext.Provider。由于我们希望在整个应用程序中都可以访问这个状态管理器，因此我们需要将它放在最外层的组件中。
*/
//构建承载该全局状态的上下文
//并将这个上下文暴露出去



export const useStore = () => {
  const { state, dispatch } = useContext(StoreContext);
  const increment = () => dispatch({ type: 'INCREMENT' });
  const decrement = () => dispatch({ type: 'DECREMENT' });
  return { state, increment, decrement };
};
/*
最后，我们将创建一个自定义hook函数useStore，用于在组件中使用状态管理器。

```

```
在这里，我们使用useContext()函数来获取当前的状态管理器，并将其解构为state和dispatch。然后，我们返回一个包含state、increment和decrement的对象，以便在组件中使用。
*/

//封装一个状态整体（数据与函数放在一起）的状态管理器
```

5.2.2 仓库的调用：

```
import React from 'react';
import { StoreProvider, useStore } from './store';

function Counter() {
  const { state, increment, decrement } = useStore();
  return (
    <div>
      <button onClick={increment}>+</button>
      <span>{state.count}</span>
      <button onClick={decrement}>-</button>
    </div>
  );
}

function App() {
  return (
    <StoreProvider>
      <Counter />
    </StoreProvider>

  );
}
//StoreProvider应该套在所有需要此个状态的最上层组件，像用户信息及其token信息这样的全局数据就要直接放在入口文件的下一层(app.js)就可以了

/*
我们使用useStore()钩子函数来访问状态管理器，并从state中获取count属性。然后，我们将increment和decrement操作绑定到相应的按钮上，以便在用户单击时调用它们。最后，在App组件中，我们将StoreProvider组件作为父级组件来包裹所有需要访问状态管理器的子组件。

*/
```

六. 使用Axios封装集中式网络请求库：

6.1 作用（为什么）：

1. **代码简洁易懂：**通过集中式管理网络请求，可以减少重复的代码和逻辑，使代码更加简洁易懂。
2. **维护方便：**所有的网络请求都放在同一个地方进行管理，方便维护和修改，也可以统一处理异常情况。
3. **功能强大：**Axios支持多种HTTP请求方式，如GET、POST、PUT和DELETE等，还能设置请求头、响应拦截器等功能，使得开发人员可以很方便地定制请求参数和处理响应数据。
4. **可扩展性强：**通过封装Axios，可以方便地实现对于不同API接口的调用，同时也方便地扩展其他网络请求库，如fetch等。
5. **提高效率：**通过集中式管理网络请求，可以提高开发效率，避免因为网络请求的散乱而导致开发效率降低。

6.2 需要完成的步骤：

1. **配置请求的统一拦截器：**需要对请求用户进行第一层鉴权（token是否有/过期）
2. **配置相应的统一拦截器：**对相应的信息进行第一层处理与错误请求的拦截
3. **封装基本方法的函数：**（get/post方法等等）
4. **创建统一的请求对象：**（创建HTTP对象并将所封装的方法包含在内，并将HTTP暴露出去）
5. **构建URL的统一管理和基地址的配置**
6. **设计每个请求的对应方法：**实际的业务需求的函数，请求直接使用HTTP对象，URL直接找5所创建的地方
7. **业务函数暴露：**将所有业务函数构建成整体的业务请求对象，并将业务请求对象暴露出去，供组件调用

6.3 使用axios构建：

> **其中第一部到第四部均可以在一个文件中构建,创建requestPackageing.jsx文件**

```
//其中第一部到第四部均可以在一个文件中构建
//创建requestPackageing.jsx文件
import axios from "axios";
import msag from "./Response";
import { Notification } from "@douyinfe/semi-ui";
import ConstantTab from "./Constant";




axios.defaults.timeout = 100000;//设置超时信息
axios.defaults.baseURL = "127.0.0.1:7001/admin";//设置请求基地址


//基本上都是axios的封装与配置
//但本项目是对axios包装了两层（一层是axios的api，一层是统一处理一级封装函数）；留出更多处理数据的空间，也相对更规范些
//最终是将集中处理的函数默认暴露出去，直接在requestUse的集中请求文件中调用他即可
//没了解过可以看axios的中文文档：https://www.axios-http.cn/

/**
 * 第一步：  http 请求的统一拦截器
 */
axios.interceptors.request.use(
```

```
  (config) => {
    const tokenJson = localStorage.getItem('token');
    if(tokenJson){
      //本项目的用户信息存储到localStorage里面了 也可以用store，因为一般登录后还是会将登录信息存到redux里面一份，
      //但不存到localsStorage的话，当前页面刷新时，redux中的内容会消失。
      const tokenStr=JSON.stringify(tokenJson);
      const {token,openID}=tokenStr;
      if(token&&Date.now()-openID<=3600000*2){
          config.headers.jwt_token = token //请求头加上token信息
      }
      //检验token是否存在，和openID是否过期（这里设的是2天，自己根据自己的项目改时间戳就行）
      else{
        Notification.error(ConstantTab.TokenExpireNotify)
      }
    }
    else{
      Notification.error(ConstantTab.TokenDisNotify)
    }

    return config
  },
  (error) => {

    //token没有或者已经过期了;这里做出处理(返回登录页并清除token信息)
    return Promise.reject(error);
  }
);









/**
 * 第二步：  http 响应的拦截器
 */
axios.interceptors.response.use(
  (response) => {
    //相应状态码在2XX以内会触发
    if (response.data.errCode === 2) {
      console.log("过期");
    }
    return response;
  },
  (error) => {
    //响应状态码超出2XX会触发这里，如3XX 4XX 5XX;
    console.log("请求出错：", error);
  }
);


//第三部：  封装请求方法

```

```
/**
 * 封装get方法
 * @param url  请求url
 * @param params  请求参数
 * @returns {Promise}
 *
 * 注意 axios--get的参数要么接在url后面，要么包一个对象放后面
 */
export function get(url, params = {}) {
  return new Promise((resolve, reject) => {
    axios.get(url, {params: params}).then(
      (response) => {
        resolve(response.data);
      })
      .catch((error) => {
        reject(error);
      });
  });
}

/**
 * 封装post请求
 * @param url
 * @param data
 * @returns {Promise}
 */

export function post(url, data) {
  return new Promise((resolve, reject) => {
    axios.post(url, data).then(
      (response) => {
        resolve(response.data);
      },
      (err) => {
        reject(err);
      }
    );
  });
}
```

```
//第四步：构建统一的HTTP请求对象

//统一接口处理，返回数据
/**
 * @param fetch 请求类型(get/post/put/patch)
 * @param url   请求路径（相对路径，基准路径在baseUrl已经设过
 * @param data  数据体
 * @returns {Promise}
 */
export default function (fecth, url, param) {
  let _data = "";
```

```
  return new Promise((resolve, reject) => {
    switch (fecth) {


      case "get":
        console.log("begin a get request,and url:", url);
        get(url, param)
          .then(function (response) {
            resolve(response);
          })
          .catch(function (error) {
            console.log("get request GET failed.", error);
            reject(error);
          });
        break;


      case "post":
        post(url, param)
          .then(function (response) {
            resolve(response);
          })
          .catch(function (error) {
            console.log("get request POST failed.", error);
            reject(error);
          });
        break;


      default:
        break;
    }
  });
}


```

然后创建一个apiurl.jsx,用于整请求URL的管理

```


// let baseURL={
//     local:"http://localhost:7001/admin/",
//     online:"/api/admin/",
// }

// let ipUrl=baseURL.local;


//baseUrl已经在packaging.jsx中配置过了，这里不需要了
```

```
let servicePath = {

    getArticleList:'/article/home/index',

    registerByUserName:'/register/username',

    registerByEmail:'/register/email',

    loginByUserName:'/login/username',

    loginByEmail:'/login/email',

    getTableForCPU_R23:'/login/email',

    getTableForMobile_AVG_CPU:'/table/mobile/avgcpu',

    getTableFilter:'/table/mobile/filter',

    getLineChart:'/chart/linechart',

    getPaiChart:'/chart/paichart',

    getColumnChart:'/table/chart/column',

    getCardData:'/card/data',

}

export default servicePath;
```

再创建一个requestUsing.jsx,用于写实际业务请求函数的位置

```

import http from "./requestPackaging";//引入axios的封装文件
import { Notification} from '@douyinfe/semi-ui';
import servicePath from "./apiUrls";  //引入请求路径库




/**
 * 获取首页列表
 */
const getArticleList=()=>{
  return new Promise((resolve, reject) => {
    http("get",servicePath.getArticleList).then(res => {
      resolve (res);
    },error => {
      console.log("网络异常~",error);
```

```
    reject(error)
   })
  })
}

const getTableForCPU_R23=()=>{
  return new Promise((resolve, reject) => {
    http("get",servicePath.getTableForCPU_R23).then(res => {
      resolve (res);
    },error => {
      console.log("网络异常~",error);
      reject(error)
    })
  })
}//获取表格（关于cpu_r23）的数据









export {
  getArticleList,
  getTableForCPU_R23,
}

```

最后，在实际组件中调用它几可以了

```
import { getTableForCPU_R23} from '../../config/requestUse';




    const handleTabsChange=(val)=>{
        switch (val) {
            case "6":
              changeMainData(data.slice(11,17))
              break;
            case "7":
              changeMainData(data.slice(5,17))
              break;
            case "8":
              changeMainData(data.slice(11,16))
              break;

```

```
            default:
              break;
          }//mock虚拟数据


        getTableForMobile_AVG_CPU().then(
            (val)=>{
                mainData=val.mainData;
            },
            ()=>{}
        )
        // 后端有数据之后就把这里开启就可
    }

```