搜索组件

搜索组件开发

自定义hook

axios基本用法

http跨域解法

by eric wen

搜索组件

分析UI

l 点击搜索按钮，跳转到结果页

l 搜索框下面一排，显示搜索历史

l 下拉框内：搜索历史 历史保存在本地，没历史时，为空logo，点击跳入搜索页面，关键词自动填充

l 下拉框内：商品查看历史，本质上，这不属于搜索，这属于浏览记录，一般情况下在用户中心，这里直接调用接口数据，没数据时为空Logo，点击跳入搜索页面，关键词自动填充

l 初始化时，url有关键词，则填入input ，执行搜索

l 维护关键词历史列表

组件设计

类型				
Props				
State	Focusing 搜索框聚焦控制，控制下拉框	keys搜索历史，初始值由自定义hook返回	Records 浏览商品记录，数据由useeffect接口返回	
functions	handleSearch 按钮跳转搜索页面，携带关键词，在搜索页面时，不跳转，此处需要判断组件本身是否在搜索页面	HandleclearKeys清楚搜索历史，调用自定义hook返回的方法，可主动渲染	clearRcords，清除商品浏览记录。	Search 搜索商品列表
Hooks	UseSearchKeys维护本地搜索记录，可主动驱动搜索组件更新	useGoodsRecords 维护商品浏览记录		

组件开发

第一步 ui，实现下拉框开关

第二步 实现handleSearch

不在搜索页面，存关键词到本地，携带关键词跳转

在搜索页面，存关键词到本地，直接搜索

初始化时，url带有key,则填入input ,直接搜索

第三步 开发hook useSearchKeys

把维护localstorage的逻辑封装在hooks里，为什么，因为在实际工作中，除了存localstorage，还会在实际app套壳中，通过原生方法再存本地，往往是异步的

自定义hook是什么

- 自定义 Hook 是一个以 `use` 开头的 JavaScript 函数（例如 `useFetch` 或 `useLocalStorage`）。
- 它可以调用 React 内置的 Hook（如 `useState`、`useEffect` 等），并将相关的逻辑封装起来。
- 自定义 Hook 的主要目的是**复用状态逻辑**，而不是复用 UI。

自定义 Hook 的特点

1. **命名约定**：自定义 Hook 必须以 `use` 开头，这是 React 的约定，便于识别哪些函数包含 Hook 逻辑。
2. **独立性**：每个组件调用自定义 Hook 时，都会拥有独立的状态和逻辑，不会相互干扰。
3. **复用性**：通过自定义 Hook，可以避免重复编写相同的逻辑。

思考：util 类型的函数也是封装逻辑复用，有何不同?

自定义 Hook 的常见场景

1. **数据获取**：封装 API 请求逻辑（如 `useFetch`）。
2. **表单处理**：管理表单状态和验证逻辑（如 `useForm`）。
3. **订阅外部数据源**：如 WebSocket、事件监听器等。
4. **本地存储管理**：同步 React 状态与浏览器的 `localStorage` 或 `sessionStorage`。
5. **动画或定时器**：管理复杂的动画逻辑或定时器。

useSearchKeys开发

1、加载本地数据

2、添加key

3、清除key

4、返回usestate和函数

引用自定义hook

实现自定义hook useGoodsRecords

1、加载数据

2、添加

3、清除

和searchKeys不同，商品浏览数据是存在服务端的，所以本部要有接口请求操作，这里我们用axios

Axios

Axios 是一个基于 Promise 的 HTTP 客户端，广泛用于浏览器和 Node.js 中发起网络请求。它简单易用且功能强大，支持拦截请求和响应、自动转换 JSON 数据、取消请求等功能。

以下是 Axios 的基础用法：

1. 安装 Axios

在使用 Axios 之前，需要先安装它：

```
npm install axios
```

或者直接通过 CDN 引入：

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

2. 基本用法

(1) 发起 GET 请求

```
axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data); // 获取返回的数据
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

(2) 发起 POST 请求

```
axios.post('https://api.example.com/data', {
    name: 'John',
    age: 30
  })
  .then(response => {
    console.log(response.data); // 获取返回的数据
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

(3) 并发请求

可以同时发起多个请求，并等待它们全部完成：

```
axios.all([
  axios.get('https://api.example.com/data1'),
  axios.get('https://api.example.com/data2')
])
  .then(axios.spread((response1, response2) => {
    console.log(response1.data);
    console.log(response2.data);
  }))
  .catch(error => {
    console.error('Error:', error);
  });
```

3. 配置选项

Axios 提供了丰富的配置选项，可以通过对象传递给请求方法或全局设置。

(1) 单次请求的配置

```
axios({
  method: 'get',
  url: 'https://api.example.com/data',
  params: { id: 123 }, // URL 参数
  headers: { 'Authorization': 'Bearer token' } // 自定义请求头
})
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

(2) 全局默认配置

你可以为所有请求设置默认值：

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = 'Bearer token';
axios.defaults.headers.post['Content-Type'] = 'application/json';

// 现在所有请求都会自动带上这些配置
axios.get('/data')
  .then(response => {
    console.log(response.data);
  });
```

4. 拦截器

Axios 支持请求和响应拦截器，可以在请求发送前或响应到达后对其进行处理。

(1) 请求拦截器

```
axios.interceptors.request.use(config => {
  // 在发送请求之前做些什么（如添加 token）
  config.headers.Authorization = 'Bearer token';
  return config;
}, error => {
  return Promise.reject(error);
});
```

(2) 响应拦截器

```
axios.interceptors.response.use(response => {
  // 对响应数据做点什么（如统一处理错误码）
  if (response.status === 200) {
    return response.data; // 直接返回数据部分
  }
  return response;
}, error => {
  // 对响应错误做点什么
  if (error.response.status === 401) {
    console.error('未授权，请重新登录');
  }
  return Promise.reject(error);
});
```

5. 取消请求

Axios 支持取消正在进行的请求，这对于优化用户体验非常有用。

```
const source = axios.CancelToken.source();

axios.get('https://api.example.com/data', {
  cancelToken: source.token
})
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    if (axios.isCancel(error)) {
      console.log('请求已取消', error.message);
    } else {
      console.error('Error:', error);
    }
  });

// 取消请求
source.cancel('用户手动取消了请求');
```

调用接口

1、用户获取

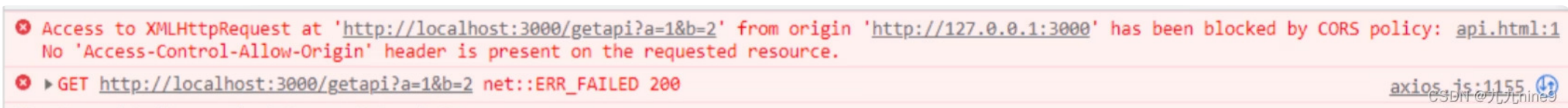
2、用户添加

3、用户清除

认识http跨域

HTTP前端跨域是指浏览器因为同源策略的限制，阻止了来自不同源（协议、域名、端口任何一个不同）的HTTP请求。这种安全机制防止了某些恶意行为，比如未经许可的资源访问。

典型报错



跨域解决办法

1. **CORS (跨域资源共享)**：服务器设置特定的HTTP头部，允许来自特定源的跨域请求。

```
'access-control-allow-origin': '*',
  'access-control-allow-methods': 'GET, POST, PUT, DELETE, OPTIONS',
   'access-control-allow-headers': 'Content-Type, Authorization, X-Requested-With'
```

2、**JSONP (仅限GET请求)**：通过动态创建<script>标签来加载数据，利用<script>标签没有跨域限制的特点。

3、代理服务器：在同源的后端服务器上设置代理，前端向代理发起请求，由代理转发到目标服务器并返回结果。

```
location /api {
  proxy_pass http://www.xx.com/api; # 后端服务器地址
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
}
```

前端开发环境用devserver的proxy解决

1、安装http-proxy-middleware

2、新建setupProxy.js

3、配置代理规则

```
const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function (app) {
    app.use(
        '/functions',
        createProxyMiddleware({
            target: 'https://rsvokvjzqdsfxyxobrks.supabase.co/functions',
            changeOrigin: true,
            logger:console,
            onProxyReq: (proxyReq, req, res) => {
                /* handle proxyReq */
                console.log('proxyReq',req.url);
            },
            onProxyRes: (proxyRes, req, res) => {
                /* handle proxyRes */
            },
            onError: (err, req, res) => {
                /* handle error */


            },
        })
    );
};
```

完成逻辑