登录–整体认识和路由配置

整体认识

登录页面的主要功能就是表单校验和登录登出业务

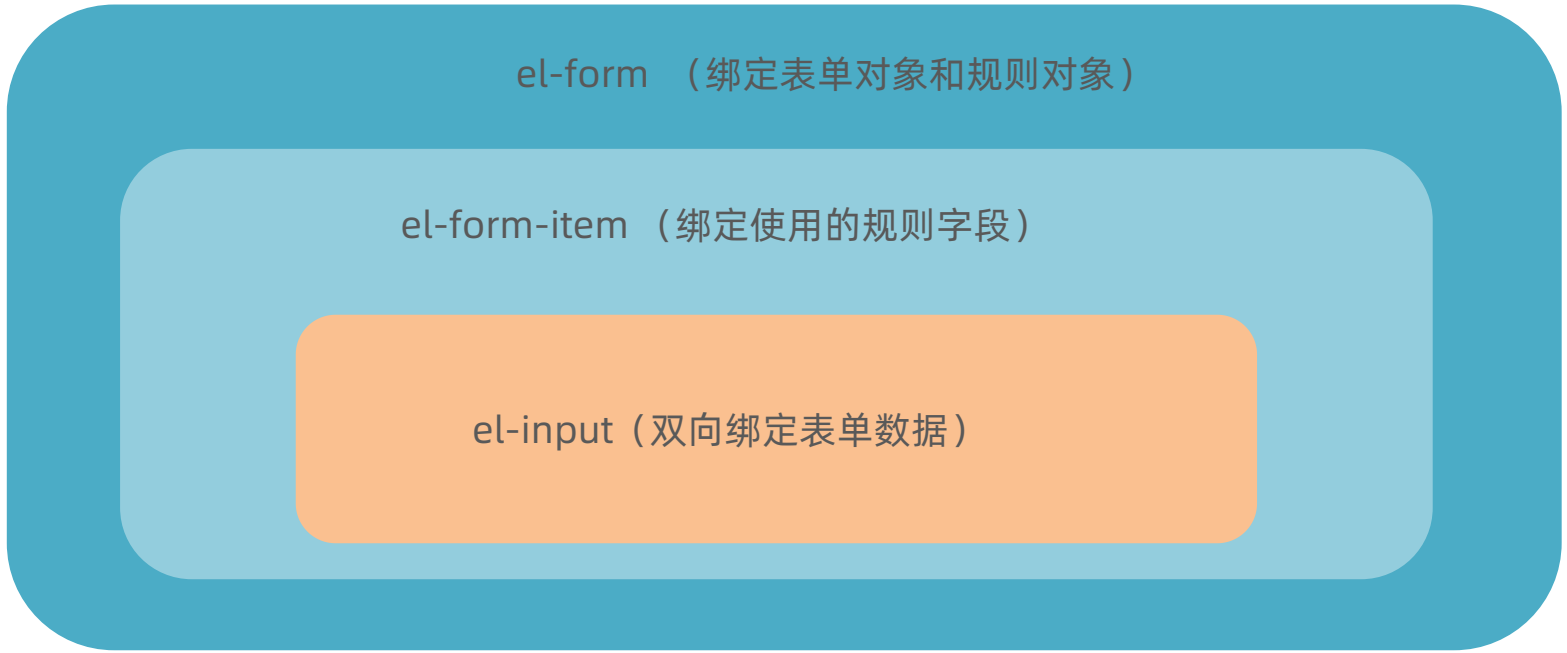
登录–表单校验实现

为什么需要校验

作用：前端提前校验可以省去一些错误的请求提交，为后端节省接口压力

表单如何进行校验

ElementPlus表单组件内置了表单校验功能，只需要按照组件要求配置必要参数即可（直接看文档）



思想：当功能很复杂时，通过多个组件各自负责某个小功能，再组合成一个大功能是组件设计中的常用方法

表单校验步骤

1. 按照接口字段准备表单对象并绑定
2. 按照产品要求准备规则对象并绑定
3. 指定表单域的校验字段名
4. 把表单对象进行双向绑定

> 用户名：不能为空，字段名为 account
>
> 密码：不能为空且为6-14个字符，字段名为 password
>
> 同意协议：必选，字段名为 agree

自定义校验规则

ElementPlus表单组件内置了初始的校验配置，应付简单的校验只需要通过配置即可，如果想要定制一些特殊的校验需求，可以使用自定义校验规则，格式如下：

```
{
  validator: (rule, val, callback) => {
    // 自定义校验逻辑
    // value: 当前输入的数据
    // callback: 校验处理函数 校验通过调用
  }
}
```

校验逻辑：如果勾选了协议框，通过校验，如果没有勾选，不通过校验

整个表单的内容验证

思考：每个表单域都有自己的校验触发事件，如果用户一上来就点击登录怎么办呢？

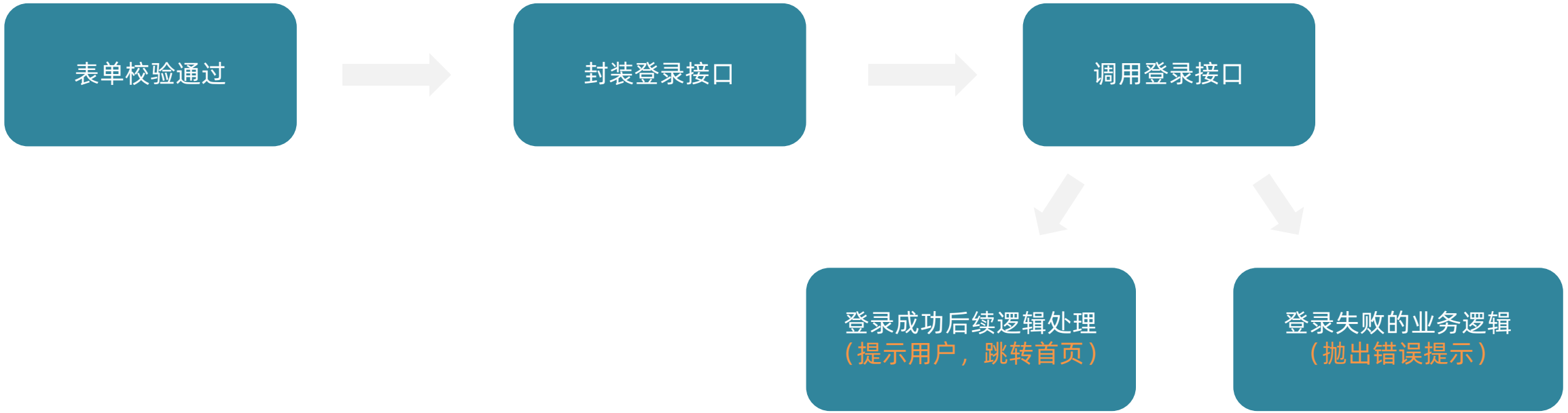
答：在点击登录时需要对所有需要校验的表单进行统一校验

```
formEl.validate((valid) => {
  if (valid) {
    console.log('submit!')
  } else {
    console.log('error submit!')
    return false
  }
})
```

1. 获取form组件实例 → 2. 调用实例方法

登录–基础登录业务实现

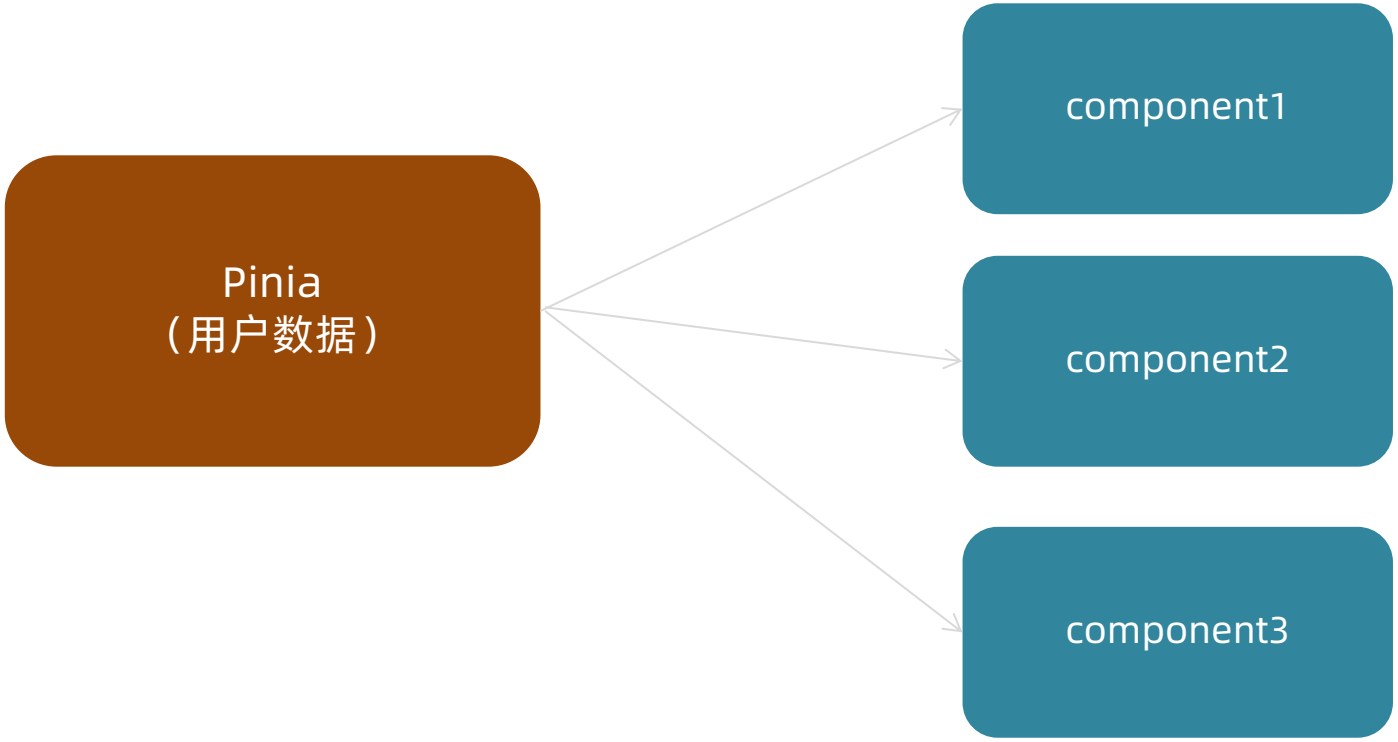
登录业务流程


表单校验通过
封装登录接口
调用登录接口
登录成功后续逻辑处理
（提示用户，跳转首页）
登录失败的业务逻辑
（抛出错误提示）

登录–Pinia管理用户数据

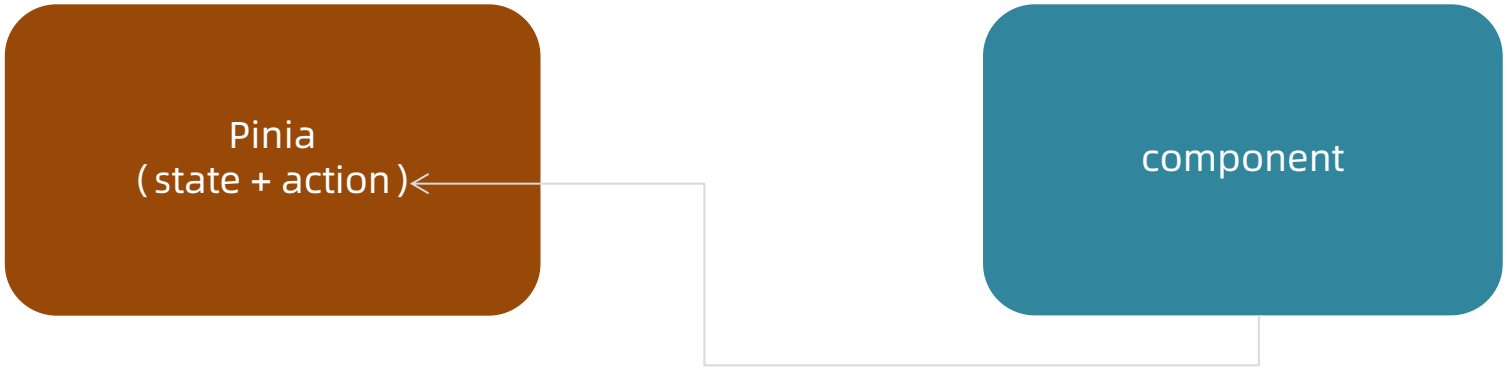
为什么要用Pinia管理数据

由于用户数据的特殊性，在很多组件中都有可能进行共享，共享的数据使用Pinia管理会更加方便



如何使用Pinia管理数据

遵循理念：和数据相关的所有操作（state + action）都放到Pinia中，组件只负责触发action函数



关键代码总结

```
export const useUserStore = defineStore('user', () => {
  // 1. 定义管理用户数据的state
  const userInfo = ref({})
  // 2. 定义获取接口数据的action函数
  const getUserInfo = async ({ account, password }) => {
    const res = await loginAPI({ account, password })
    userInfo.value = res.result
  }
  // 3. 以对象的格式把state和action return
  return {
    userInfo,
    getUserInfo
  }
})
```

```
await userStore.getUserInfo({ account, password })
```

组件触发action → action函数内部接口获取数据 → 用户信息存入

登录–Pinia用户数据持久化

持久化用户数据说明

1. 用户数据中有一个关键的数据叫做Token （用来标识当前用户是否登录），而Token持续一段时间才会过期
2. Pinia的存储是基于内存的，刷新就丢失，为了保持登录状态就要做到刷新不丢失，需要配合持久化进行存储

目的：保持token不丢失，保持登录状态

最终效果：操作state时会自动把用户数据在本地的localStorage也存一份，刷新的时候会从localStorage中先取

关键步骤总结和插件运行机制

安装插件包 → pinia注册插件 → 需要持久化的store进行配置

```
const pinia = createPinia()
pinia.use(piniaPluginPersistedState)
```

```
{
  // 持久化配置 存入ls
  persist: {
    enabled: true
  }
}
```

运行机制：

在设置state的时候会自动把数据同步给localStorage，在获取state数据的时候会优先从localStorage中取

登录–登录和非登录状态的模版适配

需求理解


请先登录 | 帮助中心 | 关于我们


运动
杂项
搜一搜
0


chaichai | 退出登录 | 我的订单 | 会员中心


数码
运动
杂项
搜一搜
0

多模版适配的通用思路

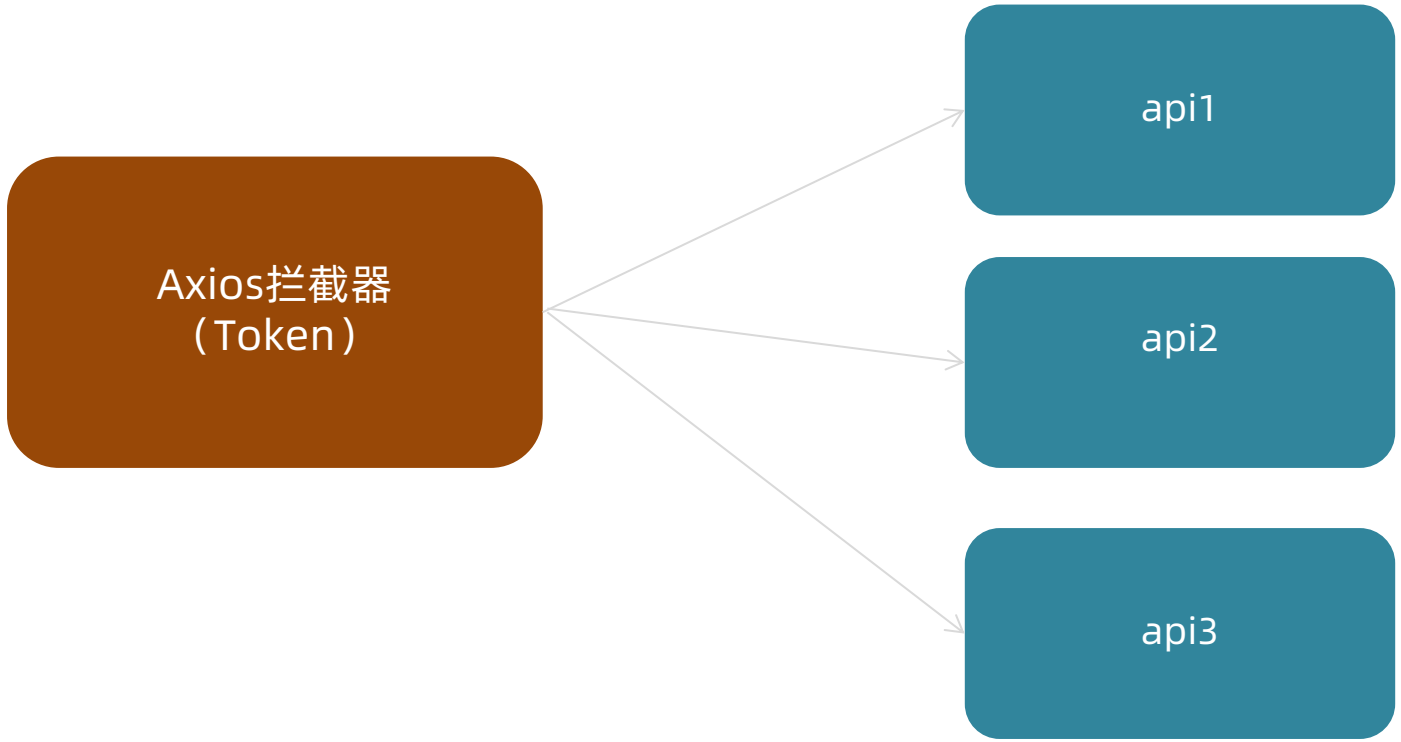
思路：有几个需要适配的模版就准备几个template片段，通过条件渲染控制显示即可

```
<template v-if="条件1">
  <!-- 模版1 -->
</template>
<template v-if="条件2">
  <!-- 模版2 -->
</template>
<template v-if="条件3">
  <!-- 模版3 -->
</template>
```

登录–请求拦截器携带Token

为什么要在请求拦截器携带Token

Token作为用户标识，在很多个接口中都需要携带Token才可以正确获取数据，所以需要在接口调用时携带Token。另外，为了统一控制采取请求拦截器携带的方案



如何配置

Axios请求拦截器可以在接口正式发起之前对请求参数做一些事情，通常Token数据会被注入到请求header中，格式按照后端要求的格式进行拼接处理

```
instance.interceptors.request.use(config => {
  const userStore = useUserStore()
  const token = userStore.userInfo.token
  if (token) {
    config.headers.Authorization = `Bearer ${token}`
  }
  return config
}, e => Promise.reject(e))
```

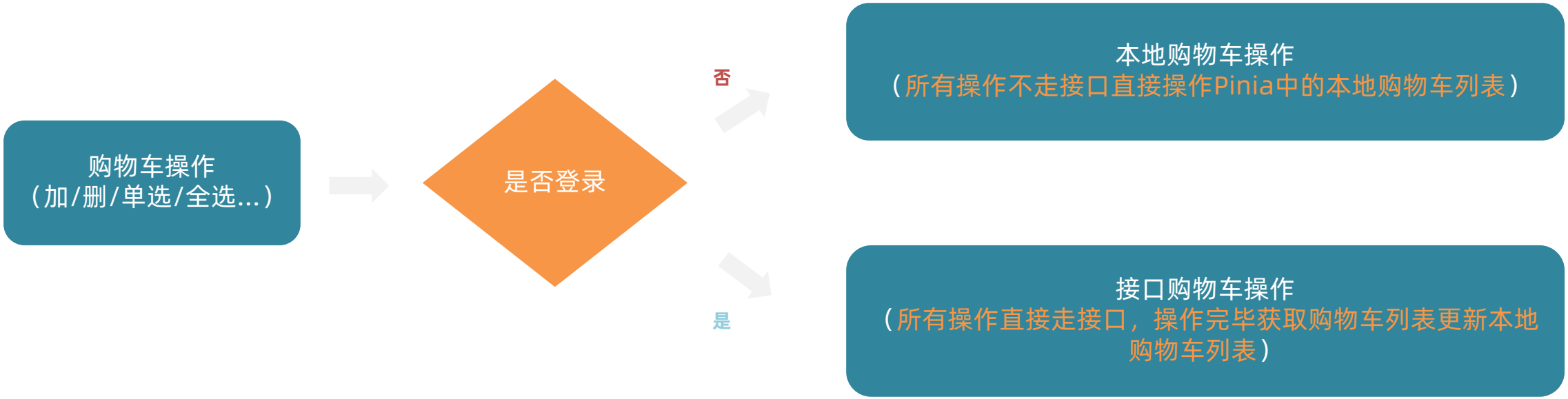
登录–退出登录功能实现

退出登录业务实现

点击退出登录弹确认框 → 点击确定按钮实现退出登录逻辑 → 登录逻辑包括：
1. 清除当前用户信息
2. 跳转到登录页面

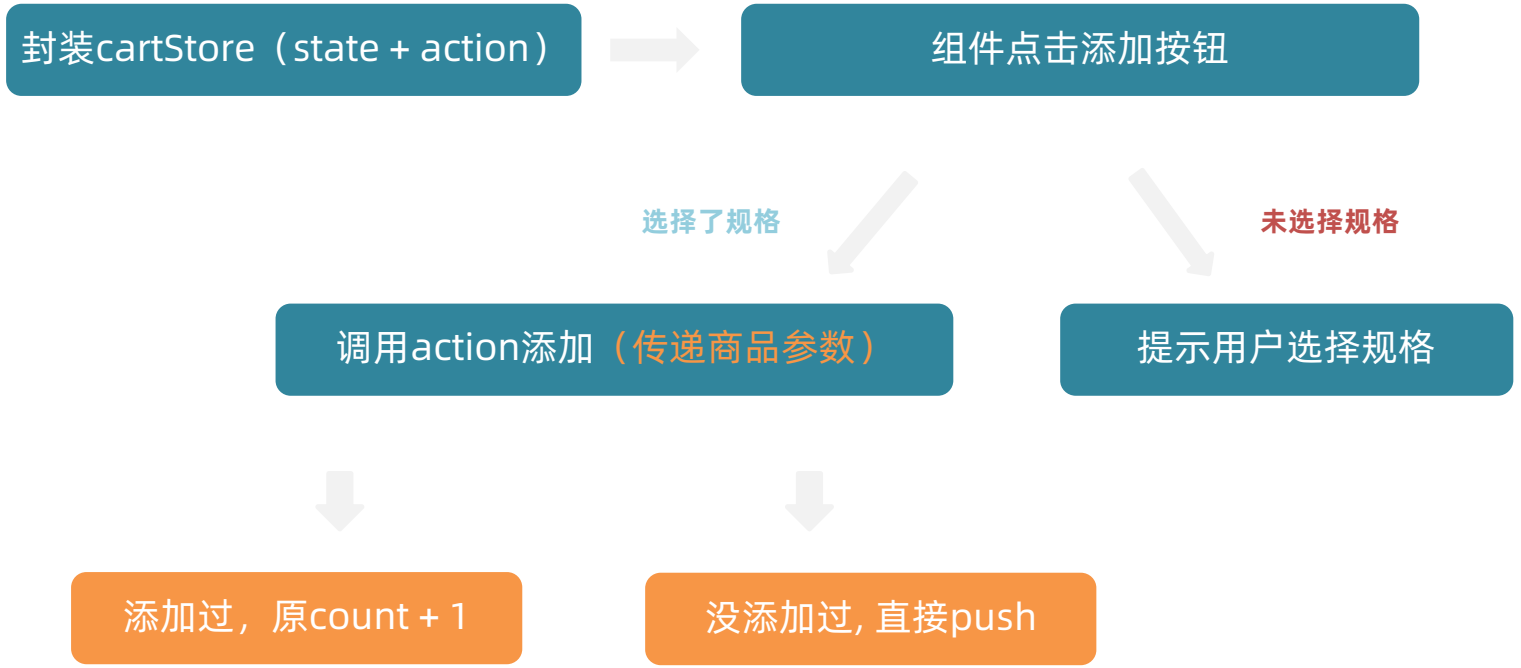
购物车功能实现

购物车业务逻辑梳理拆解



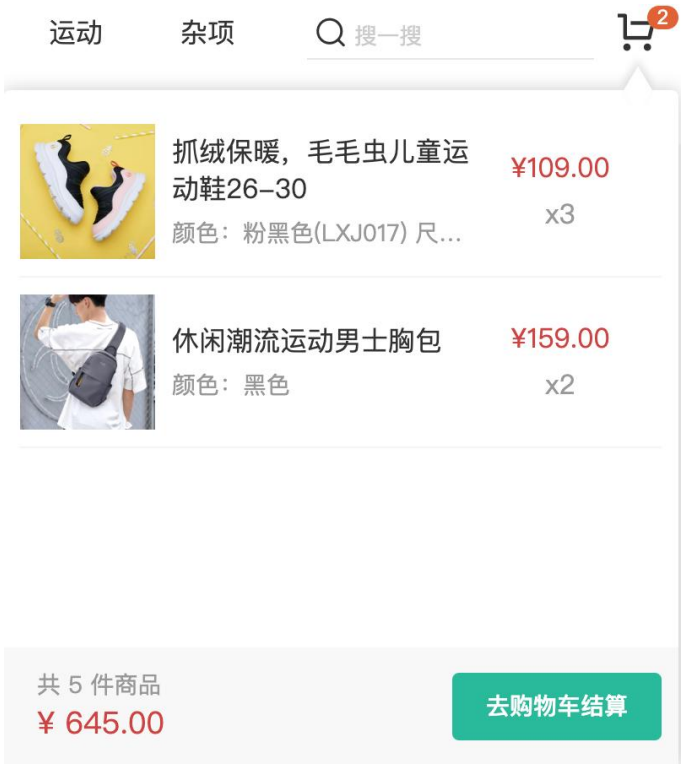
1. 整个购物车的实现分为俩个大分支，本地购物车操作和接口购物车操作
2. 由于购物车数据的特殊性，采取Pinia管理购物车列表数据并添加持久化缓存

本地购物车 - 加入购物车实现




```
{
    id: goods.value.id, // 商品id
    name: goods.value.name, // 商品名称
    picture: goods.value.mainPictures[0], // 图片
    price: goods.value.price, // 最新价格
    count: count.value, // 商品数量
    skuId: skuObj.skuId, // skuId
    attrsText: skuObj.specsText, // 商品规格文本
    selected: true, // 商品是否选中
}
```

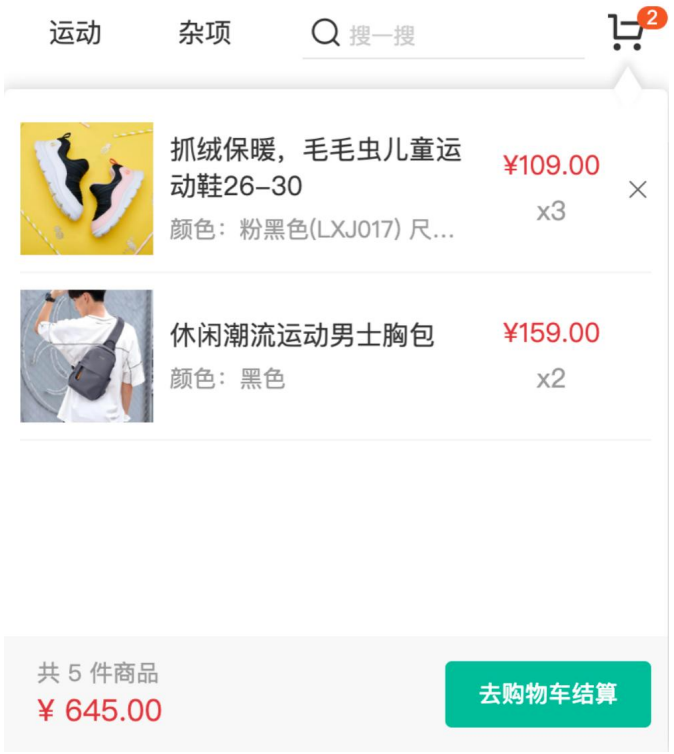
本地购物车 - 头部购物车列表渲染



准备头部购物车组件

从Pinia中获取数据渲染列表

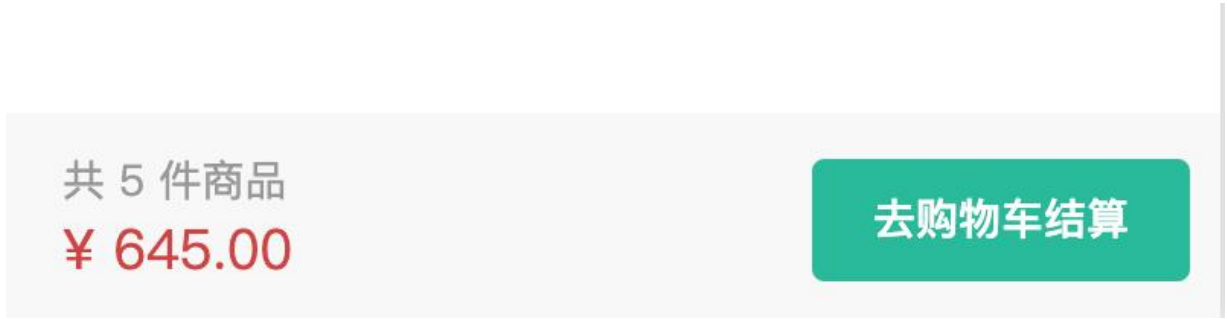
本地购物车 - 头部购物车删除实现



编写删除的action函数,实现删除逻辑

组件中调用action并传递 skuId

本地购物车 - 头部购物车统计计算



用什么来实现：计算属性

计算逻辑是什么：

1. 商品总数计算逻辑：商品列表中的所有商品 count 累加之和
2. 商品总价钱计算逻辑：商品列表中的所有商品的 count * price 累加之和

传智教育旗下高端IT教育品牌