如何用 Vue 实现前端权限控制（路由权限 + 视图权限 + 请求权限）

为什么做前端权限控制

前端权限控制并不是新生事物，早在后端MVC时代，web系统中就已经普遍存在对按钮和菜单的显示/隐藏控制，只不过当时它们是由后端程序员在jsp或者php模板中实现的，随着前后端分离架构的流行，前后端以接口为界实现开发解耦，权限控制也一分为二，前端权限控制的所有权才真正回到了前端。

可能有的同学会想，前后端分别做一套控制，是不是将事情复杂化了，而且从根本上讲前端没有秘密，后端才是权限的关键，那是不是只在后端做控制就可以了。对于这个问题我们首先应该明确，前后端权限控制他们的控制对象、控制目的和控制手段都不一样，如果仅从技术实现的角度讲，确实只在后端做控制就足够了，但在实际项目中，前端权限控制也有其不可或缺的作用，主要体现为三点：

1. 提升突破权限的门槛
2. 过滤越权请求，减轻服务端压力
3. 提升用户体验

第一点可以理解为前端权限控制是系统安全的排头兵，虽然不是主力，但起码手动输url、控制台发请求、开发者工具改数据这种级别的入侵可以防范掉；第二点是为了省钱，不该发的请求干脆就让他发不出去，带宽都是钱买的；第三点是从用户体验角度出发，一个设计优良的系统理应根据权限为每个用户展现特定的内容，避免在界面上给用户带来困扰，这是前端的本职工作，也是我个人做前端权限最大的动力之一。

前端权限控制具体指什么

前端权限归根结底是请求的发起权，请求的发起可能由页面加载触发，也可能由页面上的按钮点击触发，总的来说，所有的请求发起都触发自前端路由或视图，所以我们可以从这两方面入手，对触发权限的源头进行控制，最终要实现的目标是：

1. 路由方面，用户登录后只能看到自己有权访问的导航菜单，也只能访问自己有权访问的路由地址，否则将跳转4xx提示页；
2. 视图方面，用户只能看到自己有权浏览的内容和有权操作的控件；
3. 最后再加上请求控制作为最后一道防线，路由可能配置失误，按钮可能忘了加权限，这种时候请求控制可以用来兜底，越权请求将在前端被拦截。

怎么做前端权限控制

控制的第一步是知道用户拥有哪些权限，所以用户登录后第一件事是获取权限数据。

权限数据至少应该包括路由权限和资源权限。路由权限顾名思义，就是用户可访问的路由集合，以此作为设置前端路由和生成导航菜单的依据；资源权限是用户可访问的资源集合，“资源”概念来自RESTful架构，如果对“资源”感到陌生也可以简单理解成用户能够发起的所有请求集合，以此作为视图控制和请求拦截的依据。

这里插入讲一下“角色”这个概念，可能有的系统会通过角色来做权限控制，我理解的角色就是特定几个资源打包后的快捷方式，比如拥有总经理这个角色意味着拥有a,b,c这三个资源，副总经理就只有b,c两个资源，为用户赋予角色的本质是为用户赋予角色背后的资源。引入角色这个概念的好处是，后台可以通过赋角色的方式，很方便的为某一类用户赋予特定的资源集合，而角色的作用应该仅限于此，尤其不应该将角色用做前端权限控制的依据，因为角色背后的资源权限是后端动态可配的，我们也可以创建一个名字叫做“总经理”的角色，但其实一个资源都没有，所以前端应该始终关注资源权限本身，而只将角色视为用户的一个普通属性就好了。

有了权限数据下一步就是分别-实现对路由、视图、请求的控制。

路由控制首先要实现动态菜单，这样就可以对常规访问方式进行限制；对于非常规访问方式比如手动修改url，可以从前端路由处着手做控制，路由控制的思路有两种，一种是初始化即挂载全部路由，每次路由跳转前做校验；另一种是只挂载用户拥有的路由，相当于从源头上做了控制。前者的缺点很明显，每次路由跳转都要做一遍校验是对计算资源的浪费，另外对于用户无权访问的路由，理论上就不应该挂载。后者解决了上述问题，但仔细想这里存在一个悖论，要按需挂载路由就需要知道用户的路由权限，要知道用户的路由权限就需要用户先登录进来，但路由没有加载应用也没有初始化，用户从哪儿登录？这里又可以有两种解决思路，一种是单独做一个登录页，登录后带着用户凭据跳转到前端应用；另一种是先初始化一个只有登录路由的应用，用户登录后动态添加路由，当然这需要框架提供支持。

视图控制需要实现一个可以在视图层调用的权限验证方法，输入用户期望的权限，输出是否拥有该权限，将调用这个方法的结果，作为界面上需要验证权限的控件或元素显示与否的依据。

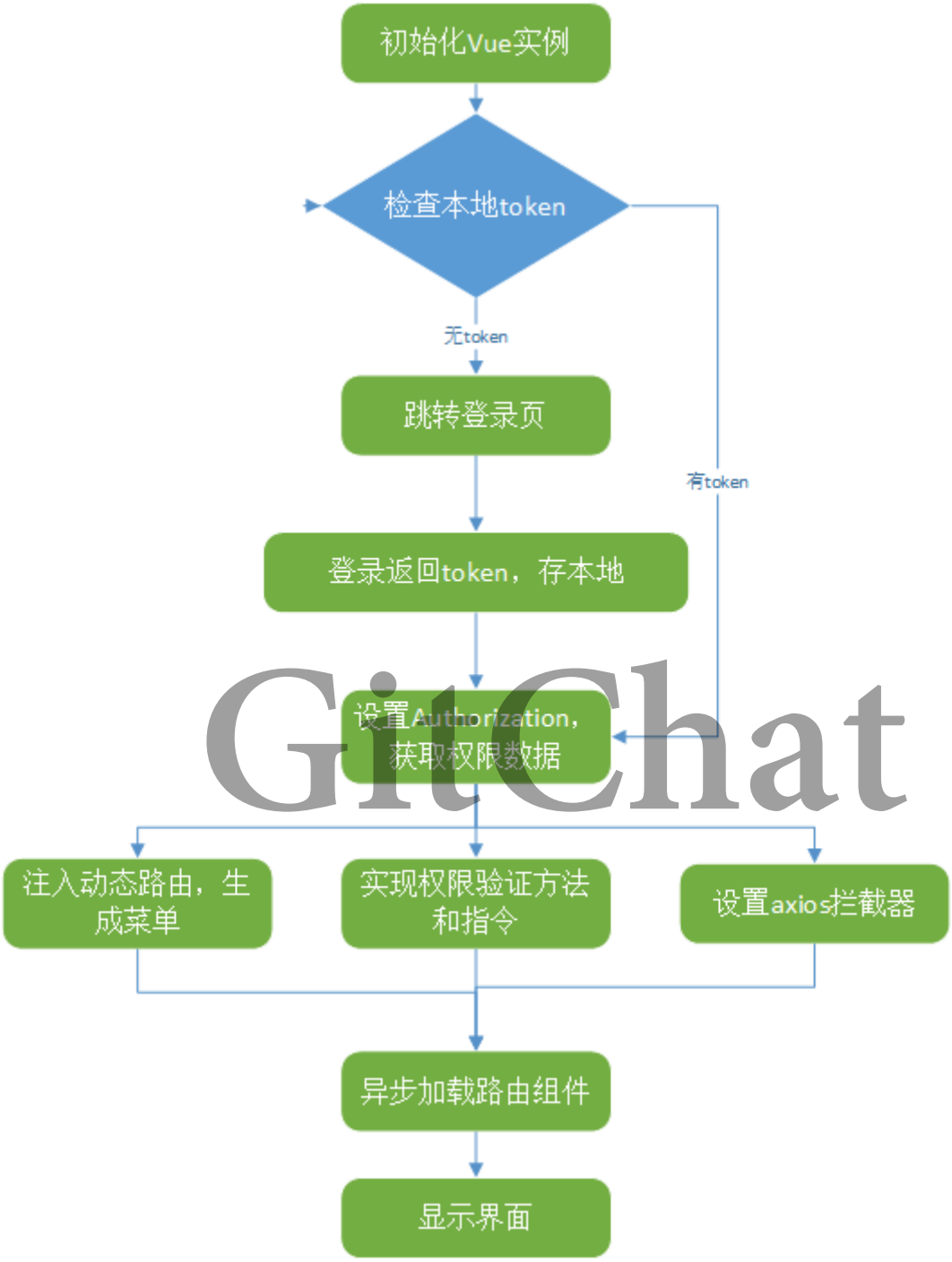
请求控制实际上就是为你使用的HTTP库实现一个请求拦截器，对将要发起的请求与用户资源权限进行匹配，拦截越权请求。这里值得一提的是对于携带参数的url，需要先进行模式约定，比如 `/people/1` 这个url可以在权限中描述为 `/people/**` ，那么拦截器中就要先将这种url处理成约定后的格式，然后再进行权限验证。

基于Vue的实现方案

概述

到目前为止我们谈的都是脱离具体技术栈的实现思路，理论上可以用任何技术栈实现这个思路，但我在项目中用的是Vue，所以下面介绍的实现细节全部基于Vue。

先来看整个流程：



从第一步“初始化Vue实例”到“获取权限数据”之间做的其实是用户鉴权，这一步跟权限控制关系不大，怎么做都可以，这里的做法是用户登录后获得一个token，然后在请求Headers中设置“Authorization”。token会存进sessionStorage里，用户刷新将直接使用本地token授权，并重新获取权限数据，如果本地token失效，那么后端应该返回401状态码，前端跳回登陆界面。

从“获取权限数据”到“异步加载路由组件”之间做的是用户权限初始化，分别用 addRoutes() 方法实现动态路由及菜单，实现全局权限验证方法及指令，以及实现axios请求拦截。

因为用的是动态路由方案，当动态路由注入时异步路由组件会开始加载，首次访问通常是加载首页组件，如果是用户刷新，地址栏还保留着之前浏览的的url，那么动态路由注入后也会正确的加载对应的路由组件，显示对应的界面。

下面我们着重来看权限初始化部分的实现细节，因为所有的初始化操作都基于后端给的权限数据，所以我们先来约定权限数据的数据格式：

路由权限数据是如下格式的对象数组

```
[
    {
      "id": "1",              //路由id
      "name": "菜单1",         //路由名称
      "parent_id": null,      //父级路由id
      "route": "route1"       //路由地址
    },
    {
      "id": "2",
      "name": "菜单1-1",
      "parent_id": "1",
      "route": "route2"
    }
 ]
```


资源权限数据是如下格式的对象数组

```
 [
    {
      "id": "2c9180895e172348015e1740805d000d", //资源id
      "name": "账号-获取",                       //资源名称
      "url": "/accounts",                       //资源请求url
      "method": "GET"                           //资源请求方法
    },
    {
      "id": "2c9180895e172348015e1740c30f000e",
      "name": "账号-删除",
      "url": "/account/**",
      "method": "DELETE"
    }
]
```

路由控制

动态路由

最初实例化的路由里仅包含登录和404之类的基本路径，而我们期待完整的路由是这样的：

```
[{
  path: '/login',
  name: 'login',
  component: (resolve) => require(['../views/login.vue'], resolve)
}, {
  path: '/404',
  name: '404',
  component: (resolve) => require(['../views/common/404.vue'],
resolve)
}, {
  path: '/',
  name: '首页',
  component: (resolve) => require(['../views/index.vue'], resolve),
  children: [{
    path: '/route1',
    name: '栏目1',
    meta: {
      icon: 'icon-channel1'
    },
    component: (resolve) => require(['../views/view1.vue'], resolve)
  }, {
    path: '/route2',
    name: '栏目2',
    meta: {
      icon: 'ico-channel2'
    },
    component: (resolve) => require(['../views/view2.vue'], resolve),
    children: [{
      path: 'child2-1',
      name: '子栏目2-1',
      meta: {

      },
      component: (resolve) => require(['../views/route2-1.vue'],
resolve)
    }]
  }]
}, {
  path: '*',
  redirect: '/404'
}]
```


一级路由只增加了一个首页，以及最后兜底的404，其他功能模块都作为首页的子路由，这么做主要是为了可以在首页实现全局导航菜单，实际项目中也可以调整这个路由结构。下一步我们关注的重点应该是获取首页的子路由们，思路是事先在本地存一份整个项目的完整路由数据，根据用户的路由权限对完整路由进行筛选。具体说一下筛选的实现，先将路由权限数据处理成如下结构：

```
let hashMenus = {
   "/route1":true,
   "/route1/route1-1":true,
   "/route1/route1-2":true,
   "/route2":true,
   ...
}
```

然后遍历本地完整路由，在循环中将路径拼接成上述结构中的key格式，通过 hashMenus[route] 判断路由是否匹配。

如果你有更好的筛选方法，或者后端返回的路由权限数据与约定不同，也可以酌情修改这部分的逻辑，只要最终能得到可用的路由数据就可以。注意在调用 addRoutes() 方法时，404页面的模糊匹配一定要放在数组的最后，否则其后的路由都不会生效。

动态菜单

用户的实际路由数据可以直接用来生成导航菜单，但首先有一个小问题，路由数据是在根组件中得到的，而导航菜单存在于首页组件中，我们需要用某种方式将菜单数据传递到首页，方法有很多，考虑到菜单数据在整个用户会话过程中不会发生改变，而且除了生成菜单之外就没有其他共享价值了，所以这里就用了最简单直接的办法，把菜单数据挂在根组件上，在首页里用 this.$parent.menuData 获取。

另外，导航菜单很可能会有一些个性化需求，比如添加栏目图标，这可以通过在路由中添加 meta 数据实现，例如将图标class或unicode存到路由meta里，模板中就可以访问到meta数据，用来生成图标标签，类似的需求也都可以这样来做。

另一个问题可能在多角色系统中比较常遇到，就是当不同角色都有一个名字相同但功能不同的路由，会发生路由名称冲突，举例来说，系统管理员和企业管理员都有一个叫做”账号管理”的路由，但他们的操作对象不同，实际上这就是两个完全不同的路由，所以路由的name肯定要有所区分，为了能在前端导航菜单上都能显示”账号管理”这个名字，我们可以为路由再起一个别名，放进 meta.name ，生成导航菜单时优先展示别名就可以了。

视图控制

全局验证方法

验证方法的的实现本身很简单，全局混入一个 $_has() 方法，内部实现无非是将所需权限与拥有权限做比对，返回一个布尔值。重点在于工程实践上的优化，怎么能让这件事做起来更方便，通常的做法可能是下面这样的：

```
<div v-if="$_has('delete,/people')">删除</div>
```

像这样的按钮一个页面上可能有多个，每个页面都需要手动的去维护权限信息，而且过程中还要频繁的在模板和脚本之间、当前组件文件和api文件之间来回切换，去查阅每一个权限对应资源的url和方法具体是什么，这样的流程显然非常容易出错，开发体验也很不好，经过摸索和总结，最终使用的方案是将权限信息和请求api维护在一起，组成一个资源对象，验证方法接收资源对象为参数，方法内部自动获取对象中的权限信息用做验证。这样做的好处是在写资源的请求方法时可以顺手维护上资源的权限信息，这样一来在前端模板中就不需要出现具体的权限信息，只要给到这个资源对象的名称就行了，另外权限验证方法应该允许多个权限联合验证，所以将参数格式改成数组，最终用法是这样的：

```
模板：
<div v-if="$_has([people.delete])" @click="people.delete.r()">删除
</div>

脚本：
import * as people from "../api/people";
```

资源对象示例：

```
// "../api/people"
const delete = {
  p: ['delete,/people'],                    //资源权限
  r: params => {
    return instance.delete(`/people`)       //资源请求方法
  }
}

export {
    delete
}
```


验证方法的实现比较简单就不展开了，将权限验证方法全局混入就可以在项目中很容易的配合 v-if 实现元素显示控制， v-if 这种方式的优点在于除了可以校验权限外，还可以在表达式中结合业务数据做更多样性的判断，从而实现随业务变化的动态视图控制。

自定义指令

v-if 的响应特性是把双刃剑，因为表达式在应用运行过程中会频繁触发，但实际上在一个用户的会话周期内其权限极少会发生变化， v-if 产生的大量运算都是不必要的，多数时候我们希望只在视图载入时做一次校验决定元素的去留，这个需求可以通过自定义指令实现：

```
Vue.directive('has', {
  bind: function(el, binding) {
    if (!Vue.prototype.$_has(binding.value)) {
      el.parentNode.removeChild(el);
```

```
        }
      }
    });
```

自定义指令内部仍然是调用全局验证方法，但优点在于只会在元素初始化时执行一次，多数情况都应该使用自定义指令实现界面元素的权限控制。

请求控制

请求控制是利用axios拦截器实现的，原理是在请求拦截器中获取本次请求的url和method信息，再与资源权限数据做比对，判断请求是否合法从而决定是否拦截。

普通请求很容易处理，遍历资源权限数据，直接判断 `request.method` 和 `request.url` 是否吻合就可以了。对于带参数的url就不能用全文匹配了，而应该用模式匹配，这里需要前后端先协商一致，后端返回的资源权限数据中，需要将url的参数用通配符代替，前端的请求拦截器中也要将带参数url处理成跟后端一致的格式，这样才能正确校验这类url，例如以下这两种常见的参数格式及其代替写法：

1. url：/resources/1
   权限：/resources/**
   解释：一个名词后跟一个参数，参数通常表示名词的id

2. url：/store/1/member
   权限：/store/*/member
   解释：两个名词之间夹带一个参数，参数通常表示第一个名词的id

格式的匹配和参数替换可以用正则表达式实现，可能遇到的一个问题是，如果你要发起一个url为”/aaa/bbb”的请求，默认会匹配为上述第一种格式，然后被处理成”/aaa/**”进行权限校验，而如果这里的”bbb”并不是参数而是url的一部分，那么你可以将url改成”/aaa/bbb/”，在最后加一个”/”以绕过格式匹配。

如果你的项目还需要其他的通配符格式，只需要在拦截器中实现对应的匹配和转化方法就可以了。

Vue-Access-Control

上述就是我对前端权限管理的一点经验，完整方案已经整理成开源项目Vue-Access-Control，若对实现细节有疑问可以参考对应部分的代码，如果这个项目对你有帮助，也请多多star，不要客气，项目地址见下方。

项目本身也是一个可运行的DEMO，演示地址和测试账号同样见下方。

仓库地址：https://github.com/tower1229/Vue-Access-Control

项目主页：http://refined-x.com/Vue-Access-Control/

演示地址：vue-access-control.refined-x.com

测试账号：

1. username: root
   password: 任意
2. username: client
   password: 任意