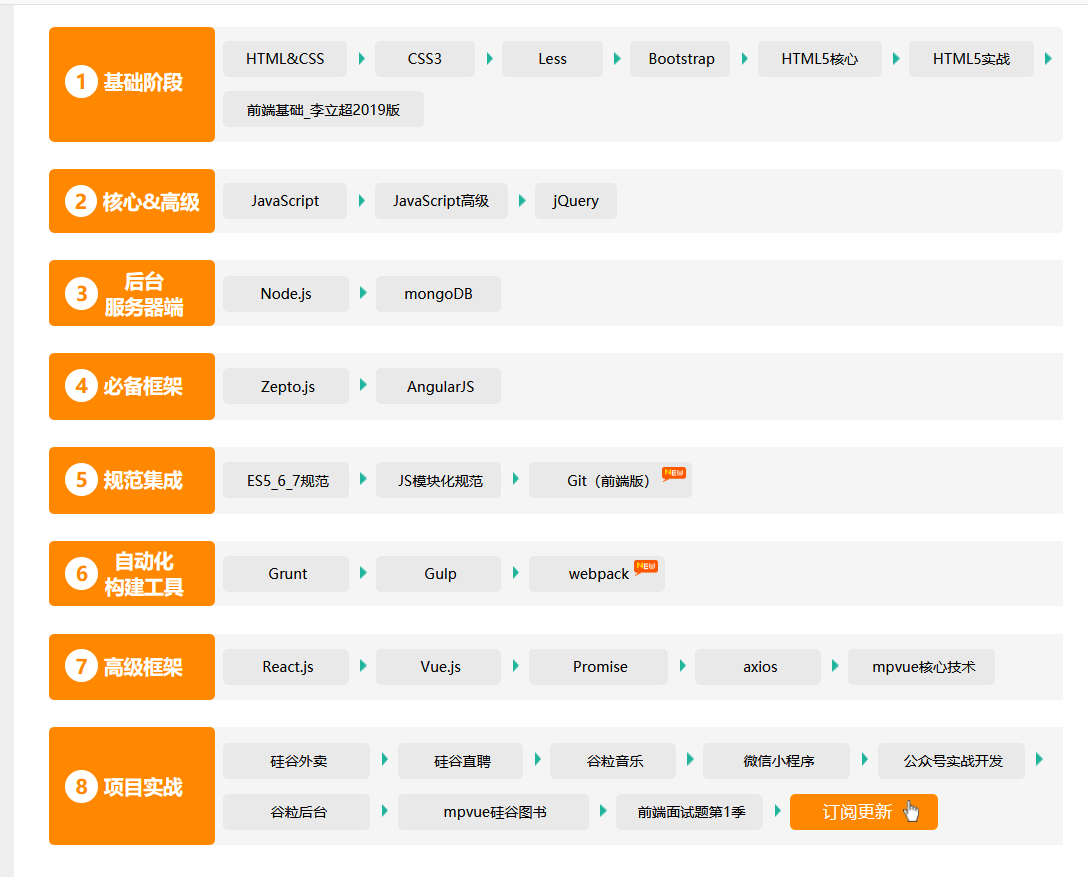
# Dewey前端的一些总结



1. Html&Css

## css布局问题

## reset.CSS

重置浏览器标签的样式表,因为浏览器的品种很多，每个浏览器的默认样式也是不同的，比如button标签，在IE浏览器、Firefox浏览器以及Safari浏览器中的样式都是不同的，所以，通过重置button标签的CSS属性，然后再将它统一定义，就可以产生相同的显示效果。

/\*\*

 \* Eric Meyer's Reset CSS v2.0 (http://meyerweb.com/eric/tools/css/reset/)

 \* http://cssreset.com

 \*/

<style>

    html, body, div, span, applet, object, iframe,

h1, h2, h3, h4, h5, h6, p, blockquote, pre,

a, abbr, acronym, address, big, cite, code,

del, dfn, em, img, ins, kbd, q, s, samp,

small, strike, strong, sub, sup, tt, var,

b, u, i, center,

dl, dt, dd, ol, ul, li,

fieldset, form, label, legend,

table, caption, tbody, tfoot, thead, tr, th, td,

article, aside, canvas, details, embed,

figure, figcaption, footer, header, hgroup,

menu, nav, output, ruby, section, summary,

time, mark, audio, video{

  margin: 0;

  padding: 0;

  border: 0;

  font-size: 100%;

  font: inherit;

  font-weight: normal;

  vertical-align: baseline;

}

/\* HTML5 display-role reset for older browsers \*/

article, aside, details, figcaption, figure,

footer, header, hgroup, menu, nav, section{

  display: block;

}

ol, ul, li{

  list-style: none;

}

blockquote, q{

  quotes: none;

}

blockquote:before, blockquote:after,

q:before, q:after{

  content: '';

  content: none;

}

table{

  border-collapse: collapse;

  border-spacing: 0;

}

/\* custom \*/

a{

  color: #7e8c8d;

  text-decoration: none;

  -webkit-backface-visibility: hidden;

}

::-webkit-scrollbar{

  width: 5px;

  height: 5px;

}

::-webkit-scrollbar-track-piece{

  background-color: rgba(0, 0, 0, 0.2);

  -webkit-border-radius: 6px;

}

::-webkit-scrollbar-thumb:vertical{

  height: 5px;

  background-color: rgba(125, 125, 125, 0.7);

  -webkit-border-radius: 6px;

}

::-webkit-scrollbar-thumb:horizontal{

  width: 5px;

  background-color: rgba(125, 125, 125, 0.7);

  -webkit-border-radius: 6px;

}

html, body{

  width: 100%;

  font-family: "Arial", "Microsoft YaHei", "黑体", "宋体", "微软雅黑", sans-serif;

}

body{

  line-height: 1;

  -webkit-text-size-adjust: none;

  -webkit-tap-highlight-color: rgba(0, 0, 0, 0);

}

html{

  overflow-y: scroll;

}

/\*清除浮动\*/

.clearfix:before,

.clearfix:after{

  content: " ";

  display: inline-block;

  height: 0;

  clear: both;

  visibility: hidden;

}

.clearfix{

  \*zoom: 1;

}

/\*隐藏\*/

.dn{

  display: none;

}

</style>

1. JS

## 1.方法

Array

**array.concat(item...)**：返回一个新数组，它包含array的浅复制（shallow copy）并将1个或多个参数item附加在其后。

## 正则

1. 后台
2. git有关

## 1.github

1. jQuery
2. webpack
3. Vue

## Vue

Vue概述

国产框架

作者：尤雨溪

**渐进式JavaScript框架**

（库：主要提供API;框架：提供基础性的服务为主）

声明式渲染🡺组件系统🡺客户端路由🡺集中式状态管理🡺项目构建

特点：易用：熟悉HTML，CSS，JavaScript知识后，可快速上手Vue

灵活：在一个库和一个完整框架之间自如伸缩

高效：20KB运行大小，超快虚拟DOM

Vue基本使用

**最基本的使用：**

<body>

    <div id="app">

        <div>{{meg}}</div>

*<!-- 插值表达式 -->*

    </div>

    <script src="../js/vue.js"></script>

    <script>

        var vm = new Vue({

            el: '#app',

            data: {

                meg: 'dewey'

            }

        })

    </script>

</body>

**el:**元素的挂载位置（值可以是CSS选择器或者DOM元素）

**data：**模型数据（值是一个对象）

**插值表达式用法：**将数据填充到HTML标签中；插值表达式支持基本的计算操作

**Vue代码运行原理分析：**

* **概述编译过程的概念（Vue语法🡺原生语法）**

Vue扩展插件

1. **vue-cli：vue脚手架**
2. **vue-resource(axios)：ajax请求**
3. **vue-router：路由**
4. **vuex：状态管理**
5. **vue-lazyload：图片懒加载**
6. **vue-scroller：页面滑动相关**
7. **minty-ui：基于vue的UI组件库（移动端）**
8. **element-ui：基于vue的UI组件库（PC端）**

前端渲染

1. **如何理解前端渲染？**

把数据填充到HTML标签中

**2．前端渲染方式：**

原生js拼接字符串：

使用前端模板引擎

使用vue特有的模板语法

Vue模板语法：

指令：

**本质就是自定义属性**

**指令的格式：以v-开始**

v-cloak指令：（cloak：遮盖，掩盖）

插值表达式存在问题：会出现页面闪动问题（主要是因为vue先将花括号的内容打印出来，再进行插值表达式中内容替换）

解决方法：使用v-cloak指令

解决该问题的原理：先通过样式隐藏内容，然后在内存中进行值的替换，替换好值之后再显示最终的值

**v-cloak用法：**

1. 提供样式[v-cloak]{display:none}
2. 在插值表达式所在的标签中添加v-cloak指令

**v-cloak指令：可以使用v-cloak指令设置样式，这些样式会在Vue实例编译结束时，从绑定的HTML元素上移除。**

**（注：在简单项目中，使用v-cloak指令是解决屏幕闪动的好方法。但在大型、工程化的项目中（webpack，vue-router）只有一个空的div元素，元素中的内容是通过路由挂载来实现的，这时我们就不需要用到v-cloak指令啦）**

**代码示例：**

<style>

    [v-cloak] {

        display: none;

    }

</style>

<body>

    <div id="app">

<div v-cloak>{{meg}}</div>

*<!-- 插值表达式 -->*

    </div>

    <script src="../js/vue.js"></script>

    <script>

        var vm = new Vue({

            el: '#app',

            data: {

                meg: 'dewey'

            }

        })

    </script>

</body>

v-text：（等同于{{}}表达式渲染数据，不能解析标签）

用于渲染普通文本，无论何时，绑定的数据对象上msg属性发生了改变，插值处的内容都会更新。

v-html：（不仅可以渲染数据，还可以解析标签，但不安全）

如果你想输出真正的HTML，你需要使用v-html指令，v-text仅渲染标签，不能解析HTML代码

v-pre：（填充原始信息）

显示原始信息，跳过编译过程（分析编译过程）

<div v-cloak>{{meg}}</div>

meg: '<strong>Hello</strong> Vue!'；

v-html输出为 ‘Hello Vue！’；

v-text输出为‘<strong>Hello</strong> Vue!’；

v-pre输出为‘{{meg}}’

数据响应式：

* **如何理解响应式**
  + html5中的响应式（屏幕尺寸的变化导致样式的变化）
  + 数据的响应式（数据的变化导致页面内容的变化）
* **什么是数据绑定**
  + 数据绑定：将数据填充到标签中
* **v-once 只编译一次**
  + 显示内容之后不再具有响应式功能
  + **应用场景：**如果显示的信息后续不需要再修改，你们可以使用v-once，这样可以提高性能

双向数据绑定：

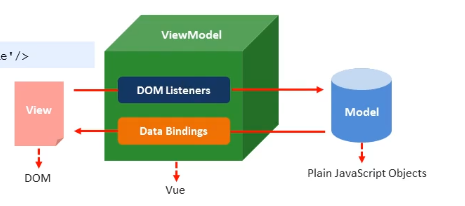
1. **什么是双向数据绑定？**
   1. **输入值的变化会导致上面的内容跟着变化**
2. **双向数据绑定分析**

v-model指令

<div><input type="text" v-model='meg'></div>

原理：<input v-bind:value="sth" v-on:input="sth = $event.target.value" />

1. **MVVM设计思想：**
   1. **M（model）：**数据
   2. **V（view）：**视图
   3. **VM（View-Model）**



事件绑定：

1. **Vue如何处理事件？**
   1. **v-on指令（v-on:click=‘’；可简写成@click=‘’）**
2. **事件函数的调用方式：（函数放在methods里面）**
   1. **直接绑定函数名称**

<div><button v-on:click = 'say'>hello</button></div>

* 1. **调用函数**

    <div><button v-on:click = 'say()'>hello</button></div>

1. **事件函数参数传递**
   1. **普通参数和事件对象(事件对象：$event)）（$event.target属性）**
      1. **如果事件直接绑定函数名称（就是 没有加（）），那么默认会传递事件对象作为事件函数的第一个参数**
      2. **如果事件绑定函数调用，那么事件对象必须作为最后一个参数显示传递，并且事件对象的名称必须是$event**
2. **事件修饰符**
   1. **.stop 阻止冒泡**
      1. **传统方式： event,stopPropagation()**
      2. **Vue方式：v-on:click.stop=‘’**
   2. **.prevent 阻止默认行为**
      1. **传统方式：event.preventDefault()**
      2. **Vue方式：v-on:click.prevent=‘’**
   3. **其他修饰符见官网**
   4. **（修饰符可连写，但需考虑顺序问题）**
   5. **按键修饰符**
      1. **.enter 回车键：v-on:keyup.enter=’’**
      2. **.delete 删除键：v-on:keyup.delete=’’**
   6. **自定义修饰符（规则：自定义按键符是自定义的，但是对应的值必须是按键对应event.keyCode值）**
      1. **全局配置config.keyCodes对象：Vue.config.keyCodes.a = 65 （因为65是对应a这个键的唯一标识，所以这里将65命名为a，也可以命其它名字）**
      2. **v-on：keyup.a=’’（这就相当于是 v-on:keyup.65=‘’，这样点击a键 就能执行这个点击事件）**

属性绑定

1. **Vue如何动态处理属性？**
   1. **v-bind指令用法：v-bind:href=‘url’(可缩写为 :href = ‘url’)**

**2.v-model的底层实现原理分析：**

<input v-bind:value = 'meg' v-on:input ='meg=$event.target.value'>

样式绑定

1. **class样式处理**
   1. **对象语法：v-bind:class=”{active:isActive}”**
   2. **数组语法：v-bind:class=’[activeClass,errorClass]’**
   3. **相关语法细节：**
      1. **对象绑定和数组绑定可以结合使用**
      2. **class绑定的值可以简化操作**
      3. **默认的class如何处理？答： 默认的class会保留**
2. **style样式处理**
   1. **对象语法：v-bind:style=’{color:activeColor,fontSize:fontSize}’**
   2. **数组语法：v-bind:style = ‘[baseStyle,overridingtyle]’**

分支循环结构

**1.分支结构**

* 1. **v-if**（v-if，v-else，v-else-if 只会将满足条件的渲染，不满足条件的将不会渲染）
  2. **v-else：**代表以上if情况都未满足，当作最后一个
  3. **v-else-if:**多个判断
  4. **v-show：**布尔值判断（无论true还是false都会渲染，为fasle则添加一个display：none样式）

1. **循环结构**
   1. **v-for遍历数组：**v-for=’item in list’ v-for=’(item,index) in list’
   2. **key的作用：帮助Vue区分不同的元素，从而提高性能** :key=’item.id’

(只要进行遍历，就添加key)

* 1. **v-for遍历对象：** v-for=’(value,key,index) in Object’
  2. **v-if和v-for结合使用：**v-if=’value==12’ v-for’(value,key,index) in Object’

Vue的常用特性

表单操作

用户的交互

* Input 单行文本
* textarea 多行文本
* select 下拉多选
* radio 单选框
* checkbox 多选框

表单域修饰符

* number：转化为数值
* trim：去掉开始和结尾的空格
* lazy：将input事件（每次输入内容的时候都会触发）切换为change事件（失去焦点的时候触发）（可用于验证）

<body>

    <div id="app">

*<!-- //单行文本： text -->*

        <div>姓名： <input type="text" v-model='uname'></div>

*<!-- //单选框 radio -->*

        <div>性别：

            <input type="radio" id='male' v-model='gender' value="male">

            <label for="male">男</label>

            <input type="radio" id='female' v-model='gender' value="female">

            <label for="female">女</label>

        </div>

*<!-- //多选框 checkbox -->*

        <div>爱好：

            <input type="checkbox" id="ball" v-model='hobby' value="ball"><label for="ball">篮球</label>

            <input type="checkbox" id="sing" v-model='hobby' value="sing"><label for="sing">唱歌</label>

            <input type="checkbox" id="code" v-model='hobby' value="code"><label for="code">写代码</label>

        </div>

*<!-- //下拉单选 select -->*

        <div>职业：

            <select v-model='occuption1'>

                <option value="0">请选择职业。。。</option>

                <option value="1">教师</option>

                <option value="2">软件工程师</option>v

                <option value="3">律师</option>

            </select>

        </div>

*<!-- //下拉多选 select -->*

        <div>副职业：

            <select v-model='occuption2' multiple='true'>

                <option value="0">请选择职业。。。</option>

                <option value="1">教师</option>

                <option value="2">软件工程师</option>v

                <option value="3">律师</option>

            </select>

        </div>

*<!-- //多行文本 textarea -->*

        <div>个人简历：

            <textarea v-model='desc'></textarea>

        </div>

        <div><button type="submit" @click.prevent='handle'>

                Save

            </button></div>

    </div>

    <script src="../js/vue.js"></script>

    <script>

        var vm = new Vue({

            el: '#app',

            data: {

                uname: '3',

                gender: 'female',

                hobby: ['ball'],

                occuption1: 1,

                occuption2: ['1'],

                desc: '...'

            },

            methods: {

                handle: function () {

                    console.log(this.uname)

                    console.log(this.gender)

                    console.log(this.hobby)

                    console.log(this.occuption1)

                    console.log(this.desc)

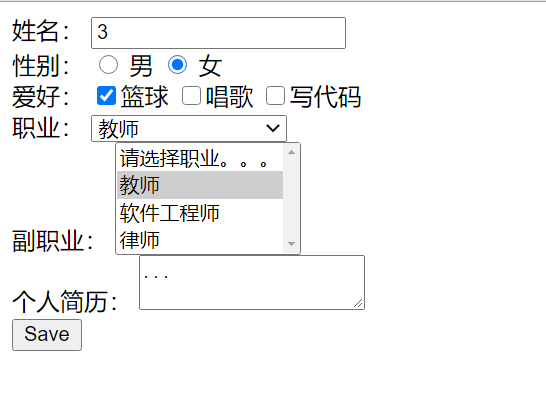
                }

            }

        })

    </script>

</body>



自定义指令

1. **为何需要自定义指令？**
   1. 内置指令不满足需求
2. **自定义指令的语法规则（获取元素焦点 v-focus）**

//自定义指令

        Vue.directory('focus' ,{

            inserted:function(el){

                //el表示指令所绑定的元素

                el.focus()

            }

        })

1. **自定义指令用法**
   1. **v-focus**
   2. **带参数的自定义指令（改变元素背景色）**

 //带参数的自定义指令

        Vue.directory('color', {

            inserted: function (el,binding) {

                //el表示指令所绑定的元素

                el.style.backgroundColor = binding.value.color

            }

        })

**用法：**

   <input type="text"  v-color='{color:'orange'}'>

* 1. **局部指令：**
     1. **组件中 有个directives属性**
     2. **将自定义指令代码 放入directives：{}里面**

计算属性

1. **为何需要计算属性？**
   1. **表达式的计算逻辑可能会比较复杂，使用计算属性可以使模板内容更加简洁**
2. **计算属性的用法：**

computed: {

                reverString:function(){

*return* this.msg.split('').reverse().join('')

                }

            },

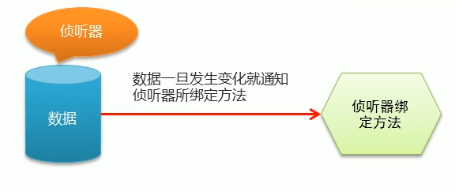
 <div>{{reverseString}}</div>

计算属性和方法的区别

* 计算属性是基于它们的依赖进行缓存的，可以节省性能
* 方法不存在缓存

侦听器

1. **侦听器的应用场景**
   1. **数据变化时执行异步或者开销大的操作**



1. **侦听器的用法&一个案例：验证用户名是否可用**
   1. **需求：输入框中输入姓名，失去焦点时 验证是否存在，如果已经存在，提示重新输入，如果不存在，提示可以使用**

<body>

    <div id="app">

        <div>firstName: <input type="text" v-model='firstName'></div>

        <div>lastName: <input type="text" v-model='lastName'></div>

        <div>fullName: {{fullName}}</div>

        <div>~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~</div>

        <div>用户名：<input type="text" v-model.lazy='username'> <span>{{tip}}</span></div>

    </div>

    <script src="../js/vue.js"></script>

    <script>

        var vm = new Vue({

            el: '#app',

            data: {

                firstName: '',

                lastName: '',

                fullName: '',

                username: '',

                tip: ''

            },

            methods: {

                chenkName: function (username) {

                    var that = this *//这里是因为下面用定时器时 中的this指的是window 而不是vue中的 所以 先将this在外面指定好*

                    setTimeout(function () {

*//模拟接口调用*

*if* (username == 'admin') {

                            that.tip = '用户名已经存在，请更换一个'

                        } *else* {

                            that.tip = '用户名可以使用'

                        }

                    }, 2000)

                }

            },

            watch: {

                firstName: function (val) {

                    this.fullName = val + ' ' + this.lastName

                },

                lastName: function (val) {

                    this.fullName = this.firstName + ' ' + val

                },

*// 侦听器*

*// 1.采用侦听器监听用户名的变化*

*// 2.调用后台接口进行验证*

*// 3，根据验证的结果调整提示信息*

                username: function (val) {

*//调用后台接口验证用户名的合法性*

                    this.chenkName(val)

*//修改提示信息*

                    this.tip = '正在验证中。。。'

                }

            },

        })

    </script>

</body>

过滤器

1. **过滤器的作用是什么？**
   1. **格式化数据，比如将字符串格式化首字母大写，将日期格式化为指定的格式等**
2. **自定义过滤器**

     Vue.filter('过滤器名称',function(value){

*//过滤器业务逻辑*

        })

或者：

在Vue中 进行局部过滤器

 filters: {

                upper: function (val) {

*return* val.charAt(0).toUpperCase() + val.slice(1)

                }

            }

1. **过滤器的使用**

<div>{{msg | upper}}</div>

        <div>{{msg|upper|lower}}</div>

//属性绑定

        <div v-bind:id='id|formatId'></div>

1. **带参数的过滤器及使用**

Vue.filter('format',function(value, arg ){

*//value就是过滤器传递过来的参数*

        })

使用：

   <div>{{date | format('yyyy-MM-dd')}}</div>

生命周期

1. 主要阶段

* **挂载（初始化相关属性）**
  + **beforeCreate：**在实例初始化之后，数据观测和事件配置之前被调用
  + **created：**在实例创建完成后被立即调用
  + **beforeMount：**在挂载开始之前被调用
  + **mounted：el被新创建的vm.$el替换，并挂载到实例上去之后调用该钩子**
    - * 一般放后台获取的数据
* **更新（元素或组件的变更操作）**
  + **beforeUpdate：**数据更新时调用，发生在虚拟DOM打补丁之前
  + **updated：**由于数据更改导致的虚拟DOM重新渲染和打补丁，在这之后会调用该钩子
* **销毁（销毁相关属性）**
  + **beforeDestroy：**实例销毁之前调用
  + **destroyed：**实例销毁后调用



数组相关API

1. **变异方法（修改原有数据）**
   1. **push()**
   2. **pop()**
   3. **shift()**
   4. **unshift()**
   5. **splice()**
   6. **sort()**
   7. **reverse()**
2. **替换数组（生成新的数组）**
   1. **filter()**
   2. **concat()**
   3. **slice()**

Vue修改响应式数据

* **Vue.set(vm.items,indexOfItem,newValue)**
* **vm.$set(vm.items,indexOfItem,newValue)**
  1. 参数一表示要处理的数组名称
  2. 参数二表示要处理的数组的索引
  3. 参数三表示要处理的数组的值

组件

**全局组件注册语法**

Vue.component(组件名称,{

data:组件数据,

templete:组件模板内容

})

<body>

    <div id="app">

        <button-counter></button-counter>

    </div>

    <script src="../js/vue.js"></script>

    <script>

        Vue.component('button-counter', {

            data: function () {

*return* {

                    count: 0

                }

            },

            template: '<button @click="handle">点击了{{count}}次</button>',

            methods: {

                handle: function () {

                    this.count += 2

                }

            }

})

        var vm = new Vue({

            el: '#app'

        })

    </script>

</body>

注：组件中 data要是个函数并且要有返回值

组件模板内容必须是单个根元素

组件模板内容可以是模板字符串（` `反引号）（ES6语法）

**组件命名方式**

短横线式

驼峰式（驼峰式组件 使用时需要换成短横线式）

**局部组件注册**

var ComponentA = {/\* …\*/}

var ComponentB = {/\* …\*/}

var ComponentC = {/\* …\*/}

new Vue({

el:’#app’,

components:{

‘component-a’ : ComponentA,

‘component-b’ : ComponentB,

‘component-c’ : ComponentC

}

})

组件间数据交互

为什么要先写组件再写实例？

vue的实例化会把全局注册的组件注册到当前实例中；实例化完成你再全局注册组件是对这个实例不起作用的

**父组件向子组件传值**

**1.组件内部通过props接收传递过来的值**

**Vue.component(‘menu-item’,{**

**props:[‘title],**

**template: ‘<div>{{title}}</div>’**

**})**

**2. 父组件通过属性将值传递给子组件**

**<menu-item title=”来自父组件的数据”></menu-item>**

**<menu-item :title=”title”></menu-item>**

<body>

    <div id="app">

        <menu-item title='你真棒'></menu-item>

    </div>

    <script src="../js/vue.js"></script>

    <script>

        Vue.component('menu-item', {

            data: function () {

*return* {

                    msg: 'wa'

                }

            },

            props: ['title'],

            template: '<div>{{msg +"~~~~"+ title}}</div>'

        })

        var vm = new Vue({

            el: '#app',

        })

    </script>

</body>

**注：**props属性名规则：在props中使用驼峰形式，模板中需要使用短横线的形式（字符串形式的模板不限制）

**子组件向父组件传值**

1. **子组件通过自定义事件向父组件传递信息**

**<button v-on:click=’$emit(“enlarge-text”)’>扩大字体</button>**

1. **父组件监听子组件的事件**

**<menu-item v-on:enlarge-text=’fonSize+=0.1’></menu-item>**

<body>

    <div id="app">

        <div :style='{fontSize: fontSize +"px"}'>sad</div>

        <menu-item v-on:enlarge-text='handle'></menu-item>

    </div>

    <script src="../js/vue.js"></script>

    <script>

        Vue.component('menu-item', {

            data: function () {

*return* {

                    msg: 'wa'

                }

            },

            template: `<div><button v-on:click='

            $emit("enlarge-text")

            '>扩大字体</button></div>`

        })

        var vm = new Vue({

            el: '#app',

            data: {

                parr: ['apple', 'orange', 'banana'],

                fontSize: 1

            },

            methods: {

                handle: function () {

                    this.fontSize += 5

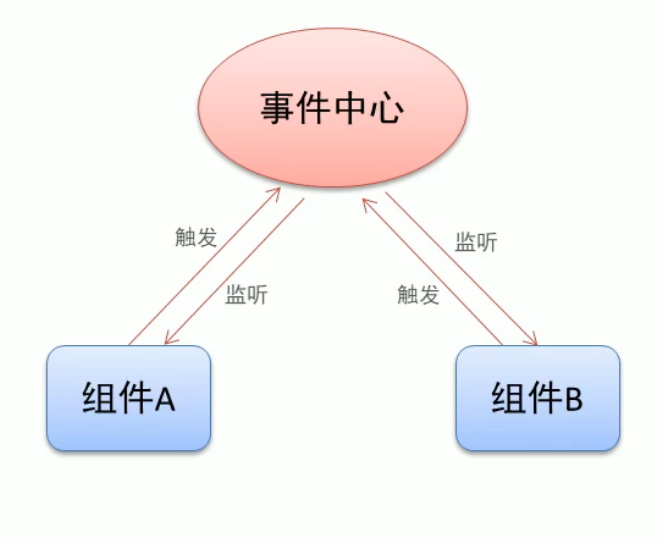
                }

            },

        })

    </script>

</body>



**非父子组件间传值**

1. **单独的事件中心管理组件间的通信**

**var eventHub = new Vue()**

1. **监听事件与销毁事件**

**eventHub.$on(‘add-todo’,addTodo)**

**eventHub.$off(‘add-todo’)**

1. **触发事件**

**eventHub.$emit(‘add-todo’,id)**

<body>

    <div id="app">

        <div><button @click='handle'>销毁</button></div>

        <test-tom></test-tom>

        <test-bob></test-bob>

    </div>

    <script src="../js/vue.js"></script>

    <script>

*//建立一个事件中心*

        var hub = new Vue()

*//组件*

        Vue.component('test-tom', {

                data: function () {

*return* {

                        num: 0

                    }

                },

                template: `<div><div>Tom:{{num}}</div>

                    <div>

                    <button @click='handle'>点击</button></div>

                    </div>`,

                methods: {

                    handle: function () {

*//触发事件*

                        hub.$emit('bob-event', 1)

                    }

                },

                mounted: function () {

*//监听事件*

                    hub.$on('tom-event', (val) => {

                        this.num += val

                    })

                }

            }),

            Vue.component('test-bob', {

                data: function () {

*return* {

                        num: 0

                    }

                },

                template: `<div><div>bob:{{num}}</div>

                    <div>

                    <button @click='handle'>点击</button></div>

                    </div>`,

                methods: {

                    handle: function () {

*//触发事件*

                        hub.$emit('tom-event', 2)

                    }

                },

                mounted: function () {

*//监听事件*

                    hub.$on('bob-event', (val) => {

                        this.num += val

                    })

                }

            })

        var vm = new Vue({

            el: '#app',

            data: {},

            methods: {

                handle: function () {

*//销毁事件*

                    hub.$off('tom-event')

                }

            }

        })

    </script>

</body>

组件插槽

**父组件向子组件传递内容**

**用法：**

1. **插槽位置**

**Vue.component(‘alert-box’,{**

**template:`**

**<div class=”demo=alert=box>**

**<strong>Error!</strong>**

**<slot></slot>**

**</div>**

**`**

**})**

1. **插槽内容**

**<alert-box>Something bad happened.</alert-box>**

**具名插槽用法**

1. **插槽定义**

 Vue.component(

            'alert-box', {

                template: `

        <div>

            <header>

            <slot name='header'></slot>

        </header>

        <main>

            <slot></slot>

        </main>

        <footer>

            <slot name ='footer'></slot>

        </footer>

        </div>

        `

            }

        )

**2．插槽内容**

      <alert-box>

            <h1 slot="header">这是头部</h1>

            <p>主要内容1</p>

            <p>主要内容2</p>

            <p slot='footer'>这是底部</p>

        </alert-box>

 <alert-box>

            <template slot="header">

                <div>

                    <p>这是头部</p>

                    <p>这是头部</p>

                </div>

            </template>

            <template slot="footer">

                <div>

                    <p>这是底部</p>

                    <p>这是底部</p>

                </div>

            </template>

            <p>主要内容1</p>

            <p>主要内容2</p>

        </alert-box>

**作用域插槽**

**应用场景：父组件对子组件的内容进行加工处理**

1. **插槽定义**

 Vue.component('fruit-list', {

            props: ['list'],

            template: `<div>

            <li :key='item.id' v-for='item in list'>

                <slot :info='item'>{{item.name}}</slot>

            </li>

        </div>`

        })

1. **插槽内容**

 <div id="app">

        <fruit-list :list='list'>

            <template slot-scope='slotProps'>

                <strong v-if='slotProps.info.id==3' class="current">

                    {{slotProps.info.name}}

                </strong>

                <span v-else> {{slotProps.info.name}}</span>

            </template>

        </fruit-list>

    </div>

前后端交互

1. 接口调用方式：
   1. 原生ajax
   2. 基于jQuery的ajax
   3. fetch
   4. axios
2. URL地址格式
   1. 传统形式的URL
      1. 格式：schema://host:[ort/path?query#fragment
         1. schema:协议。例如http、和https、ftp等
         2. host:域名或者ip地址
         3. port：端口，HTTP默认端口80，可以省略
         4. path：路径，例如/abc/a/b/c
         5. query:查询参数，例如uname=list&age=12
         6. fragment:锚点（哈希hash），用于定位页面的某个位置

b）Restful形式的URL

ii：HTTP请求方式

1. GET 查询
2. POST 添加
3. PUT 修改
4. DELETE 删除
5. Promise用法
   1. 异步调用
      1. 异步效果分析
         1. 定时任务
         2. Ajax
         3. 事件函数
      2. 多次异步调用的依赖分析
         1. 多次异步调用的结果顺序不确定
         2. 异步调用结果如果存在依赖需要嵌套
   2. Promise 是异步编程的一种解决方案，从语法上讲，Promise是一个对象，从它可以获得异步操作的消息
      1. 使用Promise的好处：
         1. 可以避免多层异步调用嵌套问题（回调地狱）
         2. Promise对象提供了简洁的API，使得控制异步操作更加容易
   3. Promise基本用法
      1. **实例化Promise对象，构造函数中传递函数，该函数中用于处理异步任务**
      2. **resolve和reject两个参数用于处理成功和失败两种情况，并通过p.then获取处理结果**

    <script>

        var p = new Promise(function (resolve, reject) {

*//这里用于实现异步任务*

            setTimeout(function () {

                var flag = false

*if* (flag) {

*//正常情况*

*//成功时调用 resolve()*

                    resolve('hello')

                } *else* {

*//异常情况*

*//失败时调用reject()*

                    reject('出错了')

                }

                p.then(function (data) {

                    console.log(data)

                }, function (info) {

                    console.log(info)

                })

            }, 100)

        })

        p.then(function (ret) {

*//从resolve得到正常结果*

        }, function (ret) {

*//从reject得到错误信息*

        })

    </script>

**通过promise发送多次ajax请求 ，避免回调地狱**

queryData()

        .then(function(data){

*return* queryData()

        })

        .then(function(data){

*return* queryData()

        })

        .then(function(data){

*return* queryData()

        })

then参数中的函数返回值

1. **返回Promise实例对象**
   1. **返回的该实例对象会调用下一个then**
2. **返回普通值**
   1. **返回的普通值会直接传递给下一个then，通过then参数中函数的参数接收该值**

Promise常用的API

1. **实例方法**
   1. **p.then()得到异步任务的正确结果**
   2. **p.catch() 获取异步信息**
   3. **p.finally()成功与否都会执行（尚且不是正式标准）**
2. **对象方法**
   1. **Promise.all() 并发处理多个异步任务，所有任务都执行完成才能得到结果**
   2. **Promise.race()并发处理多个异步任务，只要有一个任务完成就能得到结果**

Promise.all([p1,p2,p3]).then((result)=>{

                console.log(result)

            })

            Promise.race([p1,p2,p3]).then((result)=>{

                console.log(result)

            })

fetch概述

1. **基本特性**
   1. **更加简单的数据获取方式，功能更强大、更灵活，可以看做xhr的升级版**
   2. **基于Promise实现**
2. **语法结构**

  fetch(url).then(fn2)

                  .then(fn2)

                  ...

                  .catch(fn)

 fetch('http://127.0.0.1:8888/status').then(data => {

 //text()方法属于fetchAPI的一部分，它返回Promise的实例对象，用于获取后台返回的数据

*return* data.text()

        }).then(ret => {

            console.log(ret)

        })

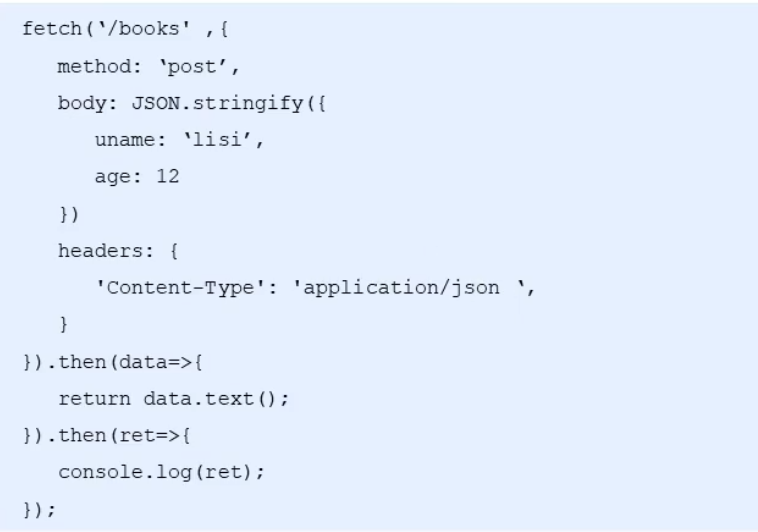
**fetch请求参数**

1. **常用配置选项**
   1. **method（String）：HTTP请求方法，默认为GET（GET、POST、PUT、DELETE）**
   2. **body（String）：HTTP的请求参数**
   3. **headers（Object）：HTTP的请求头，默认为{}**



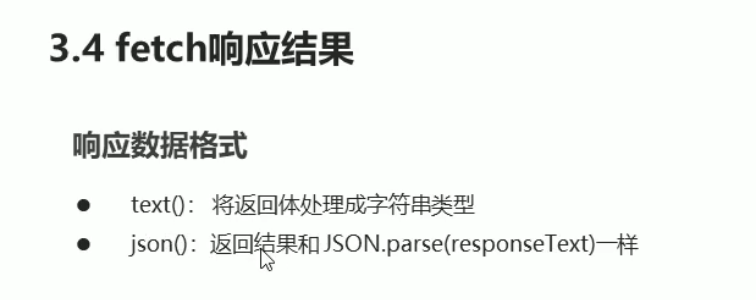








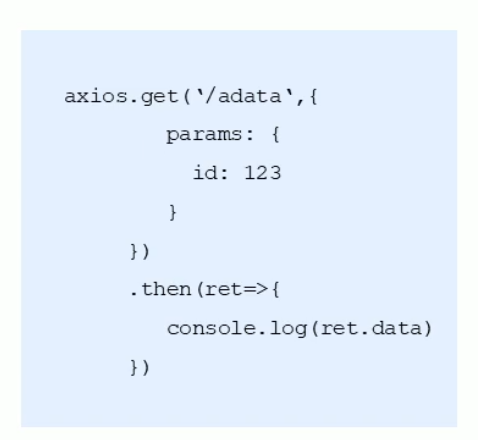




axios的基本特性

1. **支持浏览器和node.js**
2. **支持Promise**
3. **能拦截请求和响应**
4. **自动切换JSON数据**

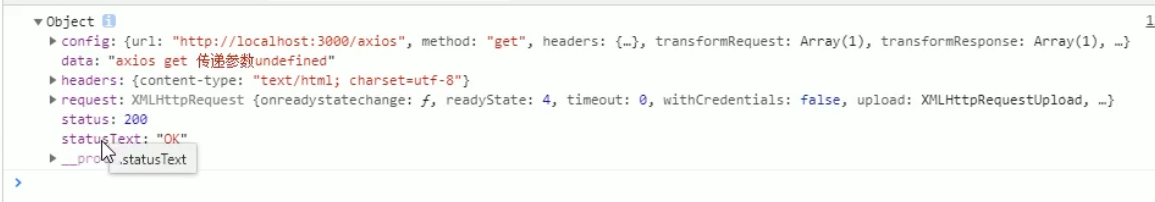












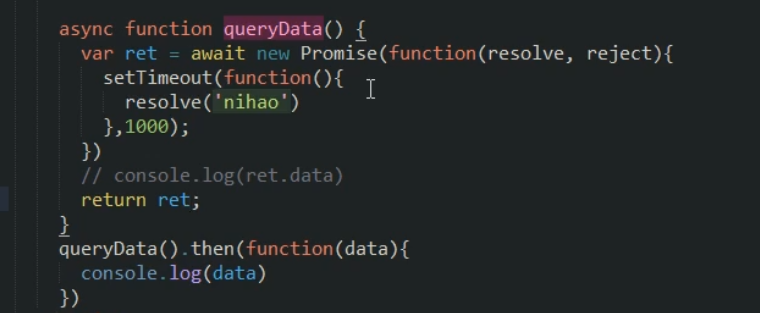


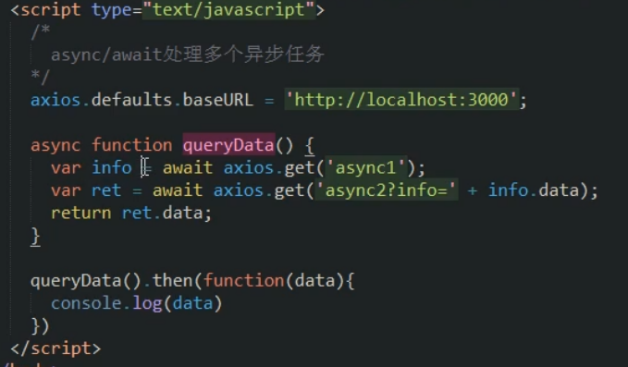




**个人理解：**

1. **加上async 后面接函数 ，则这个函数返回值就是Promise实例对象**
2. **await要在async函数里面**
3. **await axios.get(‘adata’)就相当于 axios.get(‘adata’).then(function(ret){})中的ret**
4. **await后面接着要是一个promise实例对象**





vuex

## Vue-router

## 其他内容

上传文件或者图片

1. promise和axios，以及ajax&http

## 1.http

****1.HTTP请求交互的基本过程****

1.前后应用从浏览器端向服务器发送HTTP请求（请求报文）

2.后台服务器接收到请求后，调度服务器应用处理请求，向浏览器端返回HTTP响应（响应报文）

3.浏览器接收到响应，解析显示响应体/调用监视回调

2.HTTP请求报文

1.请求行

method(请求类型) url

GET/product\_drtail?id=2

POST/login

2. 多个请求头

Host:www.baidu.com（域名/主机）

Cookie:BAIDUID=AD3BDFA706（服务器生成cookie数据储存在浏览器中）

Content-Type:application/x-www-form-urlencoded 或者 application/json （请求体内容的格式类型）

1. 请求体（get请求时没有）

username=tom&pwd=123（urlencodeed格式）

{“username”:”tom”,”pwd”:123}（Json格式）

3.HTTP响应报文

1.响应状态行 ：status（响应状态码） statusText（响应状态码对应的文本）

2.多个响应头：

Content-Type：text/html;charset=utf-8（响应体内容格式以及编码）

Set-Cookie:BD\_CK\_SAM=1;path=/（与请求头中的Cookie相对应）

3.响应体

html文本/json文本/js/CSS/图片...

4.post请求头体参数格式

1. Content-Type：application/x-www-form-urlencoded;charset=utf-8

用于键值对参数，参数的键值用=连接，参数之间用&连接

例如: name=%E5%B0%8F%E6%98%8E&age=12

2. Content-Type: application/json;charset=utf-8

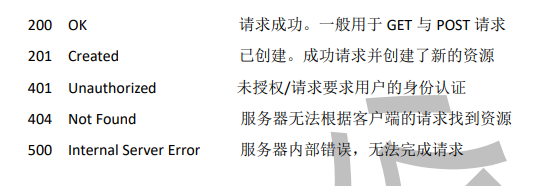
用于json字符串参数

例如: {"name": "%E5%B0%8F%E6%98%8E", "age": 12}

3. Content-Type: multipart/form-data

用于文件上传请求

5.常见的响应状态码



6.不同类型的请求及其作用

**1. GET：**从服务器端读取数据

**2. POST：**向服务器端添加新数据

**3. PUT：**更新服务器端已经有的数据

**4. DELETE：**删除服务器端数据

**7.API的分类**

**1.REST API : restful**

（1）发送请求进行CRUD（增删改查）哪个操作由请求方式来决定

（2）同一个请求路径可以进行多个操作

（3）请求方式会用到GET/POST/PUT/DELETE

（[自定义搭建一个REST API的后端接口](#_1.json-server)）

**2.非REST API : restless**

（1）请求方式不决定请求的CRUD操作

（2）一个请求路径只对应一个操作

（3）一般只有GET/POST

**8.HTTP协议**

协议详细规定了浏览器和万维网服务器之间互相通信的规则

## 2.XHR的理解和使用

**理解**：1.使用XMLHttpRequest（XHR）对象可以与服务器交互，也就是发送ajax请求

2.前端可以获取到数据，而无需让整个的页面刷新

3.这使得Web页面可以只更新页面的布局，而不影响用户的操作

**区别一般http请求与ajax请求：**

* + - 1. ajax请求是一个特别的http请求
      2. 对服务器端来说，没有任何区别，区别在浏览器端
      3. 浏览器端发请求：只有XHR或fetch发出的才是ajax请求，其它所有的都是非ajax请求
      4. 浏览器端接收到响应

一般请求：浏览器一般会直接显示响应体数据，也就是我们常说的刷新/跳转页面

ajax请求：浏览器不会对界面进行任何更新操作，只是调用监视的回调函数并传入响应相关数据

API：

1.XMLHttpRequest()：创建XHR对象的构造函数

2.status：响应状态码值，比如 202， 404

3.statusText：响应状态文本（与status相对应）

4.readyState：标识请求状态的只读属性

0：初始

1：open()之后

2：send()之后

3：请求中

4：请求完成

5.onreadystatechange：绑定readyState改变的监听

6.responseType：指定响应数据类型，如果是‘json’，得到响应后自动解析相应提数据

7.response：响应提数据，类型取决于responseType的指定

8.timeout：指定请求超时时间，默认为0代表没有限制

9.ontimeout：绑定超时的监听

10.onerror：绑定请求网络错误的监听

11.open()：初始化一个请求，参数为：(method,url[,async])

12.send(data)：发送请求

13.abort()：中断请求

14.getResponseHeader(name)：获取指定名称的响应头值

15.getAllResponseHeaders()：获取所有响应头组成的字符串

16.setRequestHeader(name,value)：设置请求头

XHR的ajax封装（简单版 axios）

**特点：**

1. 函数的返回值是promise，成功的结果为response，异常的结果为error
2. 能处理多种类型的请求：GET/POST/PUT/DELETE
3. 函数的参数为一个配置对象

{

url:””, //请求函数

method：“”，//请求方式GET/POST/PUT/DELETE

params: (), //GET/DELETE请求的query参数

data:{}. //POST或DELETE请求的请求体参数

}

1. 响应json数据自动解析为js

**编码实现：**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>

</head>

<body>

    <button onclick="testGet()">发送GET请求</button>

    <button onclick="testPost()">发送POST请求</button>

    <button onclick="testPut()">发送PUT请求</button>

    <button onclick="testDelete()">发送Delete请求</button>

    <script>

        function axios({

            url,

            method = 'GET',

            params = {},

            data = {}

        }) {

*//返回一个Promise对象*

*return* new Promise((resolve, reject) => {

*//处理method（转大写）*

                method = method.toUpperCase()

*//处理query参数（拼接到url中）*

                let queryString = ''

                Object.keys(params).forEach(key => {

                    queryString += `${key}=${params[key]}&`

                })

*if* (queryString) { *//id=1&xxx=abc&*

                    queryString = queryString.substring(0, queryString.length - 1)

                    url += '?' + queryString

                }

*//1.执行异步ajax对象*

*//创建xhr对象*

                const request = new XMLHttpRequest()

*//打开连接（初始化请求，并没有请求）*

                request.open(method, url, true)

*//发送请求*

*if* (method === 'GET' || method === 'DELETE') {

                    request.send()

                } *else* *if* (method === 'POST' || method === 'PUT') {

                    request.setRequestHeader('Content-Type',

                        'application/json;charset=utf-8') *//告诉服务器请求体的格式是json*

                    request.send(JSON.stringify(data)) *//发送json格式请求体参数*

                }

*//绑定状态改变的监听*

                request.onreadystatechange = function () {

*//如果请求没有完成，直接结束*

*if* (request.readyState !== 4) {

*return*

                    }

*//如果响应状态码在[200,299]之间代表成功，否则失败*

                    const {

                        status,

                        statusText

                    } = request

*//如果请求成功，调用resolve（）*

*if* (status >= 200 && status <= 299) {

*//准备结*果数据对象response

                        const response = {

                            data: JSON.parse(request.response),

                            status,

                            statusText

                        }

                        resolve(response)

                    } *else* {

                        reject(new Error('request error status is' + status))

                    }

                }

            })

        }

        function testGet() {

            console.log('get')

            axios({

                url: 'http://localhost:3000/posts',

                method: 'GET',

                params: {

                    id: 1,

                    xxx: 'abc'

                }

            }).then(

                response => {

                    console.log(response)

                },

                reject => {

                    alert(error.message)

                }

            )

        }

        function testPost() {

            axios({

                url: 'http://localhost:3000/posts',

                method: 'POST',

                data: {

                    "title": "json-server---",

                    "author": "typicode---"

                }

            }).then(

                response => {

                    console.log(response)

                },

                error => {

                    alert(error.message)

                }

            )

        }

        function testPut() {

            console.log('put')

            axios({

                url: 'http://localhost:3000/posts/1',

                method: 'put',

                data: {

                    "title": "tsddsd1",

                    "author": "dewey1"

                }

            }).then(

                response => {

                    console.log(response)

                },

                error => {

                    alert(error.message)

                }

            )

        }

        function testDelete() {

            console.log('delete')

            axios({

                url: 'http://localhost:3000/posts/2',

                method: 'delete',

            }).then(

                response => {

                    console.log(response)

                },

                error => {

                    alert(error.message)

                }

            )

        }

    </script>

</body>

</html>

## 3.AJAX梳理

AJAX：Asynchronous JavaScript And XML（异步的JS和XML）（XML：可扩展标记语言，被用来传输和存储数据，但已被Json给取代）

AJAX优势：无刷新获取数据

AJAX优点：1）无需刷新页面而与服务端进行通信

2）允许你根据用户事件来更新部分页面内容

缺点：1）没有浏览历史，不能回退

2）存在跨域问题

1. SEO不友好

## axios

axios是什么？

* + - 1. 前端最流行的ajax请求库
      2. react/vue官方都推荐使用axios发ajax请求
      3. 文档 : <https://github.com/axios/axios>

axios特点

1. 基本promise的异步ajax请求库
2. 浏览器端/node端都可以使用
3. 支持请求/响应拦截器
4. 支持请求取消
5. 请求/响应数据转换
6. 批量发送多个请求

axios常用语法

**axios(config)**：通用/最本质的发任意类型请求的方式

**axios(url[,config])**：可以只指定url发get请求

**axios.request(config)**：等同于axios（config）

**axios.get(url[,config])**：发get请求

**axios.delete(url[,config])**：发delete请求

**axios.post(url[,config])**：发post请求

**axios.put(url[,config])**：发put请求

**axios.default.xxx**：请求的默认全配置

**axios.interceptors.request.use()**：添加请求拦截器

**axios.interceptors.response.use()**：添加响应拦截器

**axios.create([config])**：创建一个新的axios（它没有下面的功能）

**axios.Cancel()**：用于创建取消请求的错误对象

**axios.CancelToken()**：用于创建取消请求的token对象

**axios.isCancel()**：是否是一个取消请求的错误

**axios.all(promises)**：用于批量执行多个异步请求

**axios.spread()**：用来指定接收所有成功数据的回调函数的方法

难点语法的理解和使用

**axios.create(config)**

1. 根据指定配置创建一个新的axios，也就是每个新axios都有自己的配置
2. 新的axios只是没有取消请求和批量发请求的方法其他所有语法都是一致的
3. 为什么设计这个语法？
   1. 需求：项目中有部分接口需要的配置与另一个部分接口需要的配置不太一样，如何处理
   2. 解决：创建两个新axios，每个都有自己特有的配置，分别应用到不同要求的接口请求中

**代码：**

  axios.defaults.baseURL = 'http://localhost:3000'

*// 使用axios发请求*

    axios({

      url: '/posts' *// 请求3000*

    })

*// axios({*

*//   url: '/xxx' // 请求4000*

*// })*

    const instance = axios.create({

      baseURL: 'http://localhost:4000'

    })

*// 使用instance发请求*

*// instance({*

*//   url: '/xxx'  // 请求4000*

*// })*

    instance.get('/xxx')

**拦截器函数/ajax请求/请求的回调函数的调用顺序**

1. 说明：**调用axios()并不是立即发送ajax请求，而是需要经历一个较长的流程**
2. **流程：**请求拦截器2 =>请求拦截器1 =>发ajax请求=>响应拦截器1=>响应拦截器2=>请求的回调
3. 注意：此流程是通过promise串连起来的，请求拦截器传递的是config，响应拦截器传递的是response

**代码：**

 /\*

    requestInterceptors: [{fulfilled1(){}, rejected1(){}}, {fulfilled2(){}, rejected2(){}}]

    responseInterceptors: [{fulfilled11(){}, rejected11(){}}, {fulfilled22(){}, rejected22(){}}]

    chain: [

      fulfilled2, rejected2, fulfilled1, rejected1,

      dispatchReqeust, undefined,

      fulfilled11, rejected11, fulfilled22, rejected22

    ]

    promise链回调: config

                  => (fulfilled2, rejected2) => (fulfilled1, rejected1)   // 请求拦截器处理

                  => (dispatchReqeust, undefined) // 发请求

                  => (fulfilled11, rejected11) => (fulfilled22, rejected22) // 响应拦截器处理

                  => (onResolved, onRejected) // axios发请求回调处理

    \*/

    // 添加请求拦截器(回调函数)

    axios.interceptors.request.use(

      config => {

        console.log('request interceptor1 onResolved()')

        return config

      },

      error => {

        console.log('request interceptor1 onRejected()')

        return Promise.reject(error);

      }

    )

    axios.interceptors.request.use(

      config => {

        console.log('request interceptor2 onResolved()')

        return config

      },

      error => {

        console.log('request interceptor2 onRejected()')

        return Promise.reject(error);

      }

    )

    // 添加响应拦截器

    axios.interceptors.response.use(

      response => {

        console.log('response interceptor1 onResolved()')

        return response

      },

      function (error) {

        console.log('response interceptor1 onRejected()')

        return Promise.reject(error);

      }

    )

    axios.interceptors.response.use(

      response => {

        console.log('response interceptor2 onResolved()')

        return response

      },

      function (error) {

        console.log('response interceptor2 onRejected()')

        return Promise.reject(error);

      }

    )

    axios.get('http://localhost:3000/posts')

      .then(response => {

        console.log('data', response.data)

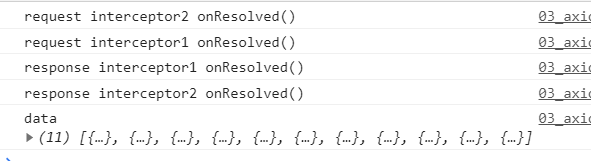
      })

      .catch(error => {

        console.log('error', error.message)

      })

**输出结果：**



取消请求：

**基本流程**：

* 1. 配置cancelToken对象
  2. 缓存用于取消请求的cancel函数
  3. 在后面特定时机调用cancel函数取消请求
  4. 在错误回调中判断如果error是cancel，做相应处理

**代码**

 let cancel  // 用于保存取消请求的函数

    function getProducts1() {

      axios({

        url: 'http://localhost:4000/products1',

        cancelToken: new axios.CancelToken((c) => { // c是用于取消当前请求的函数

          // 保存取消函数, 用于之后可能需要取消当前请求

          cancel = c

        })

      }).then(

        response => {

          cancel = null

          console.log('请求1成功了', response.data)

        },

        error => {

          cancel = null

          console.log('请求1失败了', error.message, error)

        }

      )

    }

    function getProducts2() {

      axios({

        url: 'http://localhost:4000/products2'

      }).then(

        response => {

          console.log('请求2成功了', response.data)

        },

        error => {

          cancel = null

          console.log('请求2失败了', error.message)

        }

      )

    }

    function cancelReq() {

      // alert('取消请求')

      // 执行取消请求的函数

      if (typeof cancel === 'function') {

        cancel('强制取消请求')

      } else {

        console.log('没有可取消的请求')

      }

    }

利用拦截器和取消请求来实现 只发送一个请求；当另一个请求发送则取消掉其他请求

*// 添加请求拦截器*

    axios.interceptors.request.use((config) => {

*// 在准备发请求前, 取消未完成的请求*

*if* (typeof cancel==='function') {

          cancel('取消请求')

      }

*// 添加一个cancelToken的配置*

      config.cancelToken = new axios.CancelToken((c) => { *// c是用于取消当前请求的函数*

*// 保存取消函数, 用于之后可能需要取消当前请求*

        cancel = c

      })

*return* config

    })

*// 添加响应拦截器*

    axios.interceptors.response.use(

      response => {

        cancel = null

*return* response

      },

      error => {

*if* (axios.isCancel(error)) {*// 取消请求的错误*

*// cancel = null*

          console.log('请求取消的错误', error.message) *// 做相应处理*

*// 中断promise链接*

*return* new Promise(() => {})

        } *else* { *// 请求出错了*

          cancel = null

*// 将错误向下传递*

*// throw error*

*return* Promise.reject(error)

        }

      }

    )

    let cancel  *// 用于保存取消请求的函数*

    function getProducts1() {

      axios({

        url: 'http://localhost:4000/products1',

      }).then(

        response => {

          console.log('请求1成功了', response.data)

        },

        error => {*// 只用处理请求失败的情况, 取消请求的错误的不用*

          console.log('请求1失败了', error.message)

        }

      )

    }

    function getProducts2() {

      axios({

        url: 'http://localhost:4000/products2',

      }).then(

        response => {

          console.log('请求2成功了', response.data)

        },

        error => {

          console.log('请求2失败了', error.message)

        }

      )

    }

    function cancelReq() {

*// alert('取消请求')*

*// 执行取消请求的函数*

*if* (typeof cancel === 'function') {

        cancel('强制取消请求')

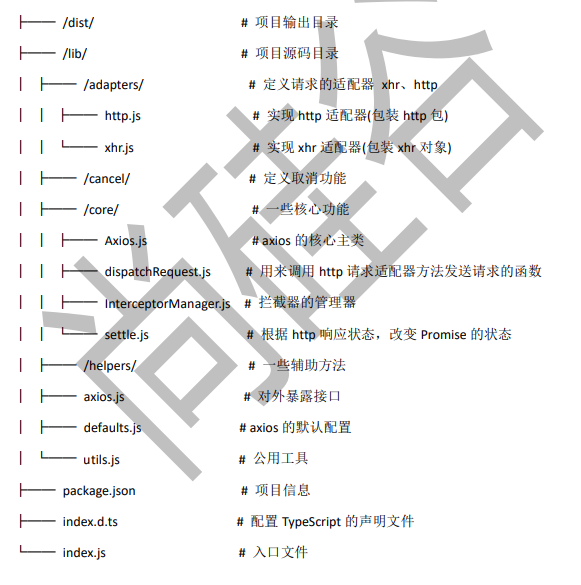
      } *else* {

        console.log('没有可取消的请求')

      }

    }

axios源码目录结构：



axios与Axios的关系？

1. 从语法上来说：axios不是Axios的实例（因为axios是个函数，而实例是个对象不可能是函数）
2. 从功能上来说：axios是Axios的实例
3. axios是Axios.prototype.request函数bind()返回的函数
4. axios作为对象有Axios原型对象上的所有方法，有Axios对象上所有属性（所以说，从功能上来说，axios是Axios的实例）

instance（实例）与axios的区别：

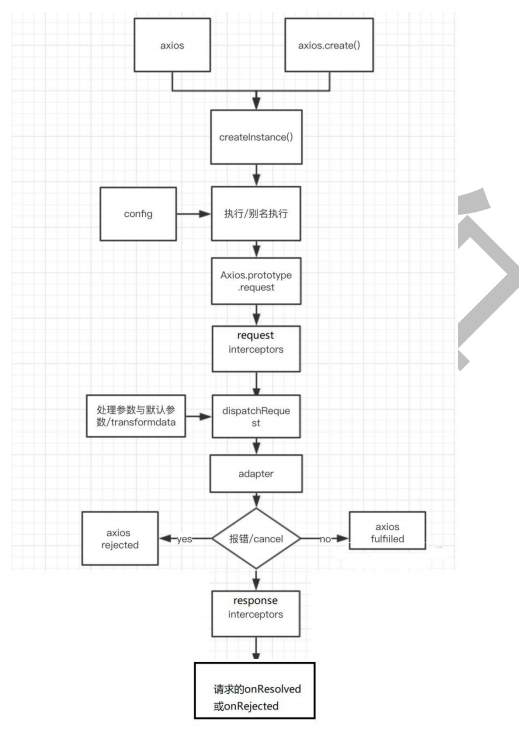
**相同：**

1. 都是一个能发任意请求的函数：request(config)
2. 都有发特定请求的各种方法：get()/post()/put()/delete()
3. 都有默认配置和拦截器的属性：defaults/interceptors

**不同：**

1. 默认匹配的值很可能不一样
2. instance没有axios后面添加的一些方法：create()/CancelToken()/all()（因为这些都是axios新添加的属性而不是Axios本身的所以instance没有）

axios运行的整体流程？



开始发请求

**整体流程：**

request(config) ==>dispatchRequest(config) ==>xhrAdapter(config)

**request(config)**

将请求拦截器/dispatchRequest()/响应拦截器 通过promise链串连起来，返回promise

/\*

  创建用于保存请求/响应拦截函数的数组

  数组的中间放发送请求的函数

  数组的左边放请求拦截器函数(成功/失败)

  数组的右边放响应拦截器函数

promise链回调: config

                  => (fulfilled2, rejected2) => (fulfilled1, rejected1)   // 请求拦截器处理

                  => (dispatchReqeust, undefined) // 发请求

                  => (fulfilled11, rejected11) => (fulfilled22, rejected22) // 响应拦截器处理

                  => (onResolved, onRejected) // axios发请求回调处理

  \*/

  var chain = [dispatchRequest, undefined];

  var promise = Promise.resolve(config);

  // 后添加的请求拦截器保存在数组的前面

  this.interceptors.request.forEach(function unshiftRequestInterceptors(interceptor) {

    chain.unshift(interceptor.fulfilled, interceptor.rejected);

  });

  // 后添加的响应拦截器保存在数组的后面

  this.interceptors.response.forEach(function pushResponseInterceptors(interceptor) {

    chain.push(interceptor.fulfilled, interceptor.rejected);

  });

  // 通过promise的then()串连起所有的请求拦截器/请求方法/响应拦截器

  while (chain.length) {

    promise = promise.then(chain.shift(), chain.shift());

  }

  // 返回用来指定我们的onResolved和onRejected的promise

  return promise;

};

**dispatchRequest(config)**

转换请求数据 ==>调用xhrAdapter()发请求 ==> 请求返回后转换响应数据，返回promise

**转换请求数据：**

 // 请求转换器

  transformRequest: [function transformRequest(data, headers) {

    // 指定headers中更规范的请求头属性名

    normalizeHeaderName(headers, 'Accept');

    normalizeHeaderName(headers, 'Content-Type');

    if (utils.isFormData(data) ||

      utils.isArrayBuffer(data) ||

      utils.isBuffer(data) ||

      utils.isStream(data) ||

      utils.isFile(data) ||

      utils.isBlob(data)

    ) {

      return data;

    }

    if (utils.isArrayBufferView(data)) {

      return data.buffer;

    }

    if (utils.isURLSearchParams(data)) {

      setContentTypeIfUnset(headers, 'application/x-www-form-urlencoded;charset=utf-8');

      return data.toString();

    }

    // 如果data是对象, 指定请求体参数格式为json, 并将参数数据对象转换为json

    if (utils.isObject(data)) {

      setContentTypeIfUnset(headers, 'application/json;charset=utf-8');

      return JSON.stringify(data);

    }

    return data;

  }],

**转换响应数据：**

// 响应数据转换器: 解析字符串类型的data数据

  transformResponse: [function transformResponse(data) {

    /\*eslint no-param-reassign:0\*/

    if (typeof data === 'string') {

      try {

        data = JSON.parse(data);

      } catch (e) { /\* Ignore \*/ }

    }

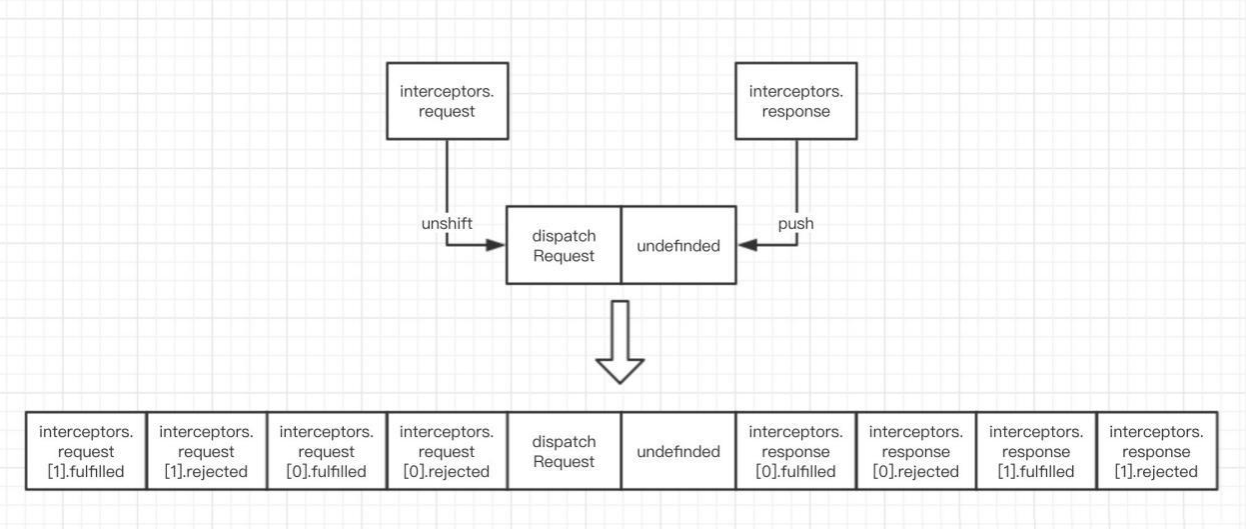
    return data;

  }],

**xhrAdapter(config)**

创建 XHR 对象，根据config进行相应设置，发送特定请求，并接收响应数据，返回promise

axios的请求/响应拦截器是什么？



**请求拦截器：**

在真正发送请求前执行的回调函数

可以对请求进行检查或配置进行特定处理

成功的回调函数，传递的默认是config（也必须是）

失败的回调函数，传递的默认是error

**响应拦截器：**

在请求得到响应后执行的回调函数

可以对响应数据进行特定处理

成功的回调函数，传递的默认是response

失败的回调函数，传递的默认是error

axios的请求/响应数据转换器是什么？

**请求转换器：**对请求头和请求体数据进行特定处理的函数

if(utils.isObject(data)){

seContentTypelfUnset(headers,’application/json;charset=utf-8’)

return JSON.stringify(data)

}

**响应转换器：**将响应体json字符串解析为js对象或数组的函数

response.data = JSON.parse(response.data)

response的整数结构

{

data,

status,

statusText.

headers,

config,

request

}

error的整数结构

{

message,

response,

request

}

如何让取消未完成的请求？

1. **当配置了cancelToken对象时，保存cancel函数**

创建一个用于将来中断请求的cancelPromise

并定义了一个用于取消请求的cancel函数

将cancel函数传递出来

1. **调用cancel()取消请求**
2. 执行cancel函数，传入错误信息message
3. 内部会让cancelPromise变为成功，且成功的值为一个Cancel对象
4. 在cancelPromise的成功回调中中断请求，并让发请求的promise失败，失败的reason为Cancel对象

## 5.promise

1. 其他

## 1.json-server

用来快速搭建REST API的工具包（将json文件放入后台，搭建一个后台接口， 并且可以对后台数据进行CRUD（增删改查）操作（通过get，post，put，delete四种请求类型））

**1.使用json-server**

1.在线文档：<https://github.com/typicode/json-server>

2.下载：**npm install -g json-server**

3.目标根目录下创建数据库json文件：db.json

{

"posts": [

{ "id": 1, "title": "json-server", "author": "typicode" }

],

"comments": [

{ "id": 1, "body": "some comment", "postId": 1 }

],

"profile": { "name": "typicode" }

}

4.启动服务器执行命令：**json-server –watch db.json**

## 2.建立一个简单的服务器

1.安装 node

2.安装express

3.编写简单服务器的代码

**express.js**

//1. 引入express

const express = require('express')

// 2.创建应用对象

const app = express()

//3.创建路由规则

//  request 是对请求报文的封装

//  response 是对响应报文的封装

app.get('/',(request,response)=>{

    // 设置响应

    response.send('Hello')

})

// 4.监听端口启动服务

app.listen(8000,()=>{

    console.log('服务已启动')

})

1. 启动 node express.js

**实例**

var express = require('express'); //express框架模块

var path = require('path'); //系统路径模块

var fs = require('fs'); //文件模块

var bodyParser = require('body-parser'); //对post请求的请求体进行解析模块

var app = express();

app.use(bodyParser.urlencoded({ extended: false })); //bodyParser.urlencoded 用来解析request中body的 urlencoded字符，只支持utf-8的编码的字符，也支持自动的解析gzip和 zlib。返回的对象是一个键值对，当extended为false的时候，键值对中的值就为'String'或'Array'形式，为true的时候，则可为任何数据类型。

//设置允许跨域请求

app.all('\*', function(req, res, next) {

    res.header('Access-Control-Allow-Origin', '\*'); //访问控制允许来源：所有

    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept'); //访问控制允许报头 X-Requested-With: xhr请求

    res.header('Access-Control-Allow-Metheds', 'PUT, POST, GET, DELETE, OPTIONS'); //访问控制允许方法

    res.header('X-Powered-By', 'nodejs'); //自定义头信息，表示服务端用nodejs

    res.header('Content-Type', 'application/json;charset=utf-8');

    next();

});

//创建get接口

function getApi(api,dirname) {

    app.get(api, function(req, res) {

        //console.log(req.body); //获取请求参数

        var file = path.join('f:\\VUE\_znywxt\\zd\\data\\', dirname);

        // var file = path.join('f:\\VUE\_znywxt\\zd\\data\\status.json'); //文件路径，\_\_dirname为当前运行js文件的目录

        //var file = 'f:\\nodejs\\data\\test.json'; //也可以用这种方式指定路径

        //读取json文件

        fs.readFile(file, 'utf-8', function(err, data) {

            if (err) {

                res.send('文件读取失败');

            } else {

                res.send(data);

            }

        })

    })

}

getApi('/machine','/machine.json')

getApi('/picture','/picture.json')

getApi('/reason','/reason.json')

getApi('/status','/status.json')

var hostName = '127.0.0.1'; //ip

var port = 8888; //端口

app.listen(port, hostName, function() {

    console.log(`服务器运行在http://${hostName}:${port}`);

});

## 关于时间复杂度的理解

时间复杂度：

评估执行程序所需的时间。可以估算出程序对处理器的使用程度。

空间复杂度：

评估执行程序所需的存储空间。可以估算出程序对计算机内存的使用程度。

**注：设计算法时，一般是要先考虑系统环境，然后权衡时间复杂度和空间复杂度，选取一个平衡点。不过，时间复杂度要比空间复杂度更容易产生问题，因此算法研究的主要也是时间复杂度，不特别说明的情况下，复杂度就是时间复杂度。**

3.1时间复杂度

**时间频度：**一个算法中的语句执行次数。记为**T(n)**

**时间复杂度：**T(n)中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数，记作**T(n)=O(f(n))**，它称为算法的渐进时间复杂度，简称时间复杂度。

* 1. 大O表示法

算法复杂度可以从最理想情况、平均情况和最坏情况三个角度来评估，由于平军情况大多和最坏情况持平，而且评估最坏情况也可以避免后顾之忧，因此一般情况下，我们设计算法时都要直接评估最坏情况的复杂度。

**O(1)：常数阶**

**O(n)：线性阶**

**O(logn)：对数阶**

**O()：平方阶**

**推导大O阶：**

1. 用常数1来取代运行时间函数中所有加法常数。
2. 修改后的运行次数函数中，只保留最高阶项。
3. 如果最高阶项存在且不是1，则去除与这个项相乘的常数。

