

Vue

Vue.js是目前最流行的前端MVVM框架

作者：尤雨溪（华人）前Google员工

是一套构建用户界面的渐进式的自底向上增量开发**MVVM**框架，**Vue**的核心库只关注视图层，它不仅易于上手，还便于与第三方库或既有项目整合。通过尽可能简单的**API**实现响应的数据绑定和组合的视图组件



Vue是一套渐进式框架的理解

每个框架都会都有自己的一些特点，会对开发者有一定的要求，这些要求就是主张，主张有强有弱，它的强势程度会影

可以在原有大系统的上面，把一两个组件改用**vue**实现，也可以整个用**vue**全家桶开发不会做职责之外的事

对于**Vue**自底向上增量开发的设计的理解

先写一个基础的页面，把基础的东西写好，再逐一去添加功能和效果，由简单到繁琐的这么一个过程。

Vue.js 目的

Vue.js的产生核心是为了解决如下三个问题

1.解决数据绑定问题。

2.**Vue.js**主要的目的是为了开发大型单页面应用。

3.支持组件化，也就是可以把页面封装成为若干个组件，把组件进行拼装，这样是让页面的复用性达到最高。

vue.js的核心思想

vue.js的核心思想包括：数据驱动和组件化。

Vue.js优势

简洁：`HTML 模板 + Vue 实例 + JSON 数据`

轻量：`17kb`, 性能好

设计思想：视图与数据分离，无需操作DOM

社区：大量的中文资料和开源案例

MVC 框架

什么是框架

封装与业务无关的重复代码，形成框架

框架的优势

使用框架提升开发效率（虽然使用框架要遵循框架的语法但是使用框架可以大大提高对于业务逻辑的操作）

MVC - 表示软件可分成三部分

模型（`Model`）数据的储存和处理，再传递给视图层相应或者展示

视图（`View`）前端的数据展示

控制器（`Controller`）对数据的接收和触发事件的接收和传递

为什么要使用 **MVC**

MVC 是一种专注业务逻辑，而非显示的设计思想

MVC 中没有DOM操作

将数据独立出来，方便管理

业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户

MVVM思想

Vue.js是一套构建用户界面的MVVM框架

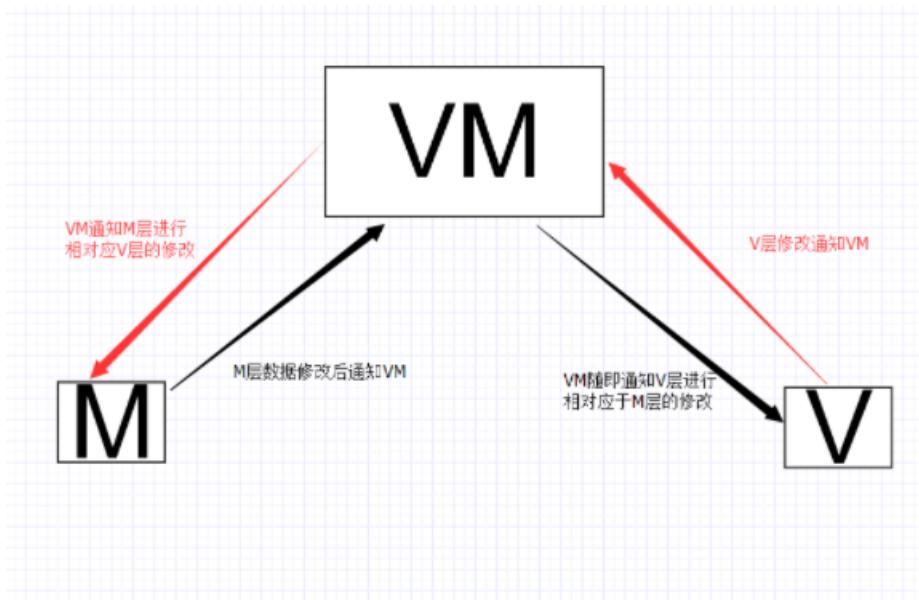
MVVM分为三个部分：分别是M（Model，模型层），V（View，视图层），VM（ViewModel，V与M连接的桥梁，也可以叫视图模型）

1、 M：模型层，主要负责业务数据相关；

2、 V：视图层，顾名思义，负责视图相关，细分下来就是html+css层；

3、 VM：V与M沟通的桥梁，负责监听M或者V的修改，是实现MVVM双向绑定的要点；因此开发者只需关注业务逻辑，不关心视图层的实现。

1. MVVM思想关系图



Vue声明式渲染

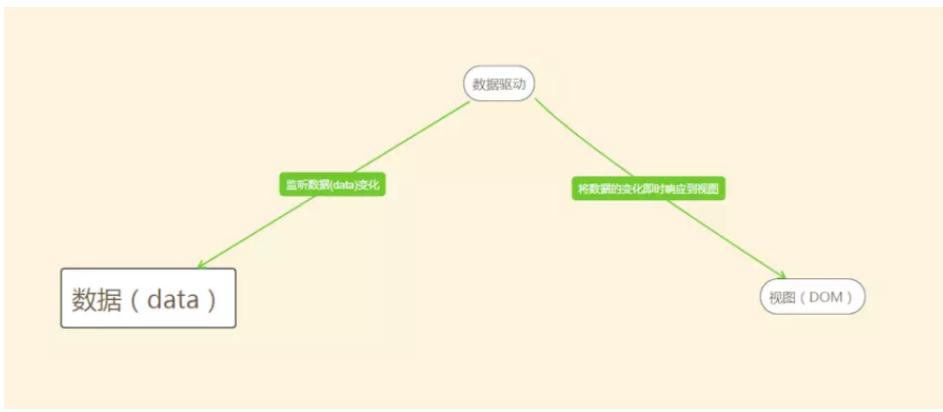
Vue.js 的核心是一个允许采用简洁的模板语法来声明式的将数据渲染进 DOM，也就是将模板中的文本数据写进DOM中

命令式渲染： 命令我们的程序去做什么，程序就会跟着你的命令去一步一步执行

声明式渲染： 我们只需要告诉程序我们想要什么效果，其他的交给程序来做。

Vue数据驱动

通过控制数据的变化来显示vue的数据驱动是视图的内容随着数据的改变而改变



Vue渲染方式

{()}--表达式 ----- 双大括号语法-----也叫模板语法

将双大括号中的数据替换成对应属性值进行响应式的展示

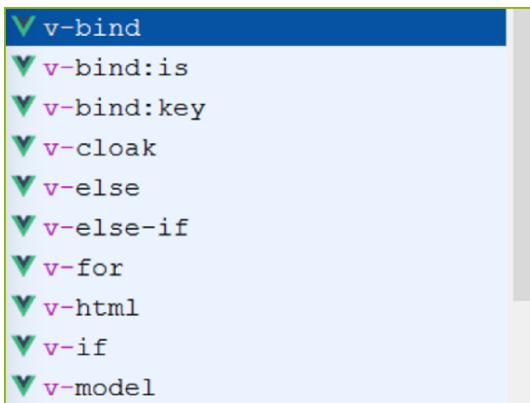
```
<div id="app">
  <span>Massage: {{ msg }}</span>
</div>

<script>
new Vue({
  el: "#app",
  data: {
    mag: 'hello!' // 改变msg的值，花括号里的msg也会改变
  }
})
</script>
```

Vue.js 指令

什么是 Vue.js 指令

指令是带有 v- 前缀的特殊属性



Vue.js 指令的用途

它们作用于HTML元素，指令提供了一些特殊的特性，将指令绑定在元素上时，指令会为绑定的目标元素添加一些特殊

Vue.js 指令的书写规范

书写位置：任意 HTML 元素的开始标签内

注意：一个开始标签内可写入多个指令，多个指令间使用空格分隔

常见指令

v-model 指令：表单上数据的双向绑定

```
<body>
  <!-- 作用：主要是用于表单上数据的双向绑定
        语法：v-model = 变量 -->
  <div id="box">
    <!-- 双向数据绑定就是：视图的数据发生改变 模型也发生改变，模型改变了，视图也随之发生改变 -->
    <input type="text" v-model="hello">
    <h3>{{hello}}</h3>

    <!-- 绑定到复选框上 会把数据修改成布尔值 -->
    <input type="checkbox" v-model="cb">
    <h3>{{cb}}</h3>
  </div>

  <script>
    new Vue({
      el: '#box',
      data: {
        hello: 'hahaha',
        cb: 'checkbox'
      }
    })
  </script>
</body>
```

作用：主要是用于表单上数据的双向绑定

语法：v-model = 变量

注：v-model 指令必须绑定在表单元素上

双向绑定

Vue框架核心的功能就是双向的数据绑定。 双向是指：HTML标签数据 绑定到 Vue对象，另外反方向数据也是绑
使用 `v-model` 指令来实现双向数据绑定 把视图数据与模型数据相互绑定

双向绑定--原理数据劫持

vue数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的

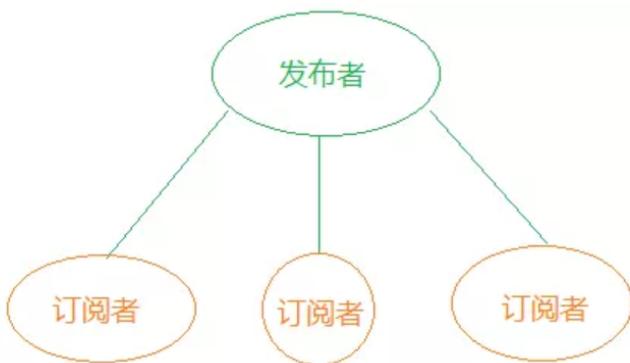
数据劫持：当我们访问或设置对象的属性的时候，都会触发`Object.defineProperty()`函数来拦截（劫持）

```
var user={  
    name:"小明"  
}  
  
//参数1监听对象 2监听属性  
Object.defineProperty(user,"name", {  
    get:function(){  
        console.log("正在读取");  
        return name;  
    },  
    //形参接受设置值  
    set:function(data){  
        console.log("正在设置");  
        name=data;  
    }  
})  
user.name="奚大官人"  
console.log(user.name);
```

双向绑定--原理发布者-订阅者模式

vue数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的

发布者-订阅者模式：其定义对象间一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知。



`v-show` 指令：元素的显示和隐藏

作用：控制切换一个元素的显示和隐藏

语法：`v-show = 表达式`

根据表达式结果的真假，确定是否显示当前元素

`true`表示显示该元素；`false`(默认)表示隐藏该元素

元素一直存在只是被动态设置了`display: none`

```
<body>
  <div id="box">
    <!-- v-show 控制元素的显示或者隐藏 true显示 false隐藏 -->
    <!-- 注意 v-show 隐藏元素是通过display来设置的隐藏 -->

    <input type="checkbox" v-model="bool"><span>{{bool?'显示':'隐藏'}}</span>
    <div v-show="bool">{{h}}</div>
  </div>

  <script>
    new Vue({
      el: '#box',
      data: {
        h: '显示和隐藏',
        bool: true,
      }
    })
  </script>
</body>
```

v-on 指令：元素绑定事件

作用：为 `HTML` 元素绑定事件监听

语法：`v-on: 事件名称='函数名称()'`

简写语法：`@事件名称='函数名称()'`

注：函数定义在 `methods` 配置项中

```
<body>
  <div id="box">
    <!-- v-on指令 绑定事件 -->
    <button v-on:click="fn()">点击</button>
    <!-- @是简写的写法 -->
    <button @click="fn()">点击</button>
  </div>

  <script>
    new Vue({
      el: '#box',
      data: {
        },
      methods: {
        fn(){
          alert(11)
        }
      }
    })
  </script>
</body>
```

v-for 指令: 遍历**data**中的数据

作用: 遍历 **data** 中的数据, 并在页面进行数据展示

语法: `v-for = '(item, index) in arr'`

item 表示每次遍历得到的元素

index 表示**item**的索引, 可选参数

```

<body>
  <div id="box">
    <div>
      <p v-for="(v,i) in arr"> {{v}},{{i}}</p>
    </div>
    <table border="1">
      <tr v-for="(v,i) in obj">
        <td>{{v.name}}</td>
        <td>{{v.age}}</td>
      </tr>
    </table>
  </div>
  <script>
    new Vue({
      el:'#box',
      data:{
        arr:['safa','dds','dsd'],
        obj:[
          {name:"小明1",age:223},
          {name:"小明2",age:223},
          {name:"小明3",age:223},
          {name:"小明4",age:223}
        ]
      }
    })
  </script>
</body>

```

v-bind 指令: 绑定HTML元素的属性

作用: 绑定 HTML 元素的属性

语法: `v-bind:属性名 = ‘表达式’ / 简写 : 属性名=‘表达式’`

绑定一个属性: ``

绑定多个属性(不能使用简写):

``

```

<body>
  <div id="box">
    <!-- v-bind 给html的属性插入变量 -->
    <a v-bind:href="ahref">{{f}}</a>
    <!-- v-bind简写 -->
    <a :href="ahref">{{f}}</a>
    <h3 v-on:click="fn()" v-bind:class="bool?'color':''">点击变色</h3>
  </div>

  <script>
    new Vue({
      el: '#box',
      data: {
        f: '百度',
        ahref: 'http://www.baidu.com',
        bool: true,
      },
      methods: {
        fn(){
          this.bool = !this.bool;
        }
      }
    })
  </script>
</body>

```

v-if指令

作用：判断是否加载固定的内容

语法：v-if = 表达式

根据表达式结果的真假，确定是否显示当前元素

true表示加载该元素；false表示不加载该元素

元素的显示和隐藏 是对Dom元素进行添加和删除

v-show与v-if区别：

v-if有更高的切换消耗（安全性高）。

v-show有更高的初始化的渲染消耗（对安全性无要求选择）

v-else 指令

作用：必须配合v-if使用否则无效。当v-if条件不成立的时候执行

v-else-if 指令

作用：当有一项成立时执行。

```
<div id="box">
  <input type="checkbox" v-model="bool">
  <div v-if="bool">登录了</div>
  <div v-else>没有登录</div>

  <select v-model="selectData">
    <option value="吃吃">吃吃</option>
    <option value="睡睡">睡睡</option>
    <option value="玩玩">玩玩</option>
  </select>

  <p v-if="selectData=='吃吃'">吃吃</p>
  <p v-else-if="selectData=='睡睡'">睡睡</p>
  <p v-else-if="selectData=='玩玩'">玩玩</p>
  <p v-else>什么都没选</p>
</div>
<script>
  new Vue({
    el: '#box',
    data: {
      bool: true,
      selectData: '',
    }
  })
</script>
```

v-text 指令

作用：操作网页元素中的纯文本内容。{{}}是他的另外一种写法

```
<div id="box" v-cloak>
  <h3>{{hh}}</h3>
  <h3 v-text="hh"></h3> //v-text 指令
</div>
<script>
  new Vue({
    el: '#box',
    data: {
      hh: 'vdshdfbhfdkb'
    }
  })
</script>
```

v-text与{{}}区别

v-text与`{}`等价，`{}`叫模板插值，**v-text**叫指令。

有一点区别就是，在渲染的数据比较多的时候，可能会把大括号显示出来，俗称屏幕闪动：

1. 解决闪烁

为了解决这种问题，可以采用以下两种方式：

①使用**v-text**渲染数据

②使用`{}`语法渲染数据，但是同时使用**v-cloak**指令（用来保持在元素上直到关联实例结束时候进行编译），**v-cloak**并不需要添加到每个标签，只要在el挂载的标签上添加就可以

```
<div id="box" v-cloak> //v-cloak并不需要添加到每个标签，只要在el挂载的标签上添加就可以
  <h3>{{hh}}</h3>
</div>
```

```
<style>
[v-cloak]{
  display: none;
}
</style>
```

v-html 指令

作用：双大括号会将数据解释为纯文本，而非 HTML。为了输出真正的 HTML，你需要使用 **v-html** 指令

语法：`<p v-html="text"></p>`

```
//v-html 把字符串的html 编译成DOM
<div id="box">
  <div v-html='a'></div>
</div>

<script>
new Vue({
  el: '#box',
  data: {
    a:<a href='http://www.baidu.com'>点我去百度</a>"
  }
})
</script>
```

v-once 指令

作用：当数据改变时，插值处的内容不会更新(会影响到该节点上的所有属性)

v-once: 只渲染一次

语法: <p v-once>{{text}}</p>

指令小结

1. **v-show**: 控制切换一个元素的显示和隐藏
2. **v-on**: 为 **HTML** 元素绑定事件监听
3. **v-model**: 将用户的输入同步到视图上
4. **v-for** : 遍历 **data** 中的数据，并在页面进行数据展示
5. **v-bind**: 绑定 **HTML** 元素的属性
6. **v-if**: 判断是否加载固定的内容
7. **v-else**: 当**v-if**条件不成立的时候执行
8. **v-else-if**: 当有一项成立时执行。
9. **v-text**: 操作网页元素中的纯文本内容
10. **v-html** : 输出真正的 **HTML**
11. **v-once**:

watch 监听

watch: 书写位置 与 **el data methods** 同级位置

可以监听模型数据，当模型数据改变的时候就会触发

```
watch:{  
    监听的data数据(newval,oldval){  
        console.log(newval,oldval)  
        //newval :新值      oldval: 原值  
    }  
}
```

watch初始化的时候不会运行，只有数据被改变之后才会运行

什么时候使用**watch**

当需要在数据变化时执行异步或开销较大的操作时，**watch**这个方式是最有用的。

计算属性与侦听器的区别：

当**watch**监听的值发生改变就会被调用，**watch**可以在数据变化时做一些异步处理或者开销大的操作
计算属性是计算依赖的值，当依赖的值发生改变才会触发。

交互

交互的应用场景（什么时候用到前后端交互）

从后端获取一些数据，将其进行展示或计算

将用户在页面中提交的数据发送给后端

Vue请求数据交互

vue请求数据有**Vue-resource**、**Axios**、**fetch**三种方式。

Vue-resource是Vue官方提供的插件，

axios是第三方插件，

fetch es6原生

Vue.js resource交互 2.0停止更新了

Vue自身不带处理HTTP请求 如果想使用HTTP请求必须要引入 **vue-resource.js** 库

它可以通过**XMLHttpRequest**发起请求并处理响应。

也就是说，**\$.ajax**能做的事情，**vue-resource**插件一样也能做到，而且**vue-resource**的API更为简洁。

Vue.js 交互借助于 **\$http** 完成

下载：npm install --save vue-resource

get 类型：

```
语法: this.$http.get("url",      //请求地址(字符串)
  {params: {key1:val1,key2:val2...}}).  //参数列表
  then(function(res){处理请求成功的情况},    //请求成功
  function(res){处理请求失败的情况})      //请求失败
```

POST类型：

```
语法: this.$http.post("url",
  {key1:val1,key2:val2...},
  {emulateJSON:true}).           //模拟json格式, 传递参数
  then(function(res){处理请求成功的情况},
  function(res){处理请求失败的情况})
```

Axios

Axios是第三方插件，不仅能在Vue里使用，还能再其他第三方库中使用例如react

```
npm install --save axios
```

get类型：

```
语法: axios.get('/路径?k=v&k=v')
  .then((ok)=>{})
  .catch((err)=>{})
```

post类型：

```
语法: axios.post('/user', {k:v,k:v })
  .then(function (ok) { })
  .catch(function (error) { });
```

Axios--post交互

数据请求不到？

试一试用jquery请求看看 发现是可以发送**post**数据的

问题就在传参方式上面。

需要使用URLSearchParams对象修改操作 URL传递参数的方法。

实例化对象：

```
let param = new URLSearchParams();
```

添加发送数据参数

```
param.append("key", "value");
```

```
let param = new URLSearchParams();
param.append("uname", "xixi");
param.append("age", 19);
axios.post('http://localhost:3000/post', param)
  .then(function (ok) {
    console.log(ok);
  }) any
  .catch(function (error) {
    console.log(error);
  });
});
```

Axios--综合交互

axios(url:'请求地址',method:'请求方式',data/params:{k:v}).then((ok)=>{})
使用get发送数据的时候 使用params: {key:val}发送数据
使用post发送数据需要使用 var param=new URLSearchParams();修改传参方法
使用param.append("uname","xixi")添加数据并且使用data发送数据

```
axios({
  url:"http://localhost:3000/get",
  method:"get",
  params:{uname:"xixi"}
}).then((ok)=>{
  console.log(ok)
})
```

```

var param=new URLSearchParams();
param.append("uname","xixi")
axios({
    url:"http://localhost:3000/post",
    method:"post",
    data:param
}).then((ok)=>{
    console.log(ok)
})

```

```

methods:{
    sendBtn(){      //点击发送请求
        this.bool=true;
        this.axiosurl("").then((res)=>{
            console.log(res)
            this.arr = res.data.data.commentList;
            this.bool=false;
        }).catch((rev)=>{
            console.log(err)
        })
    },
    axiosurl(url){      //封装axios
        return new Promise((resovle,reject)=>{
            axios({
                url,
                method:"get"
            }).then((res)=>{
                resovle(res)
            }).catch((err)=>{
                reject(err)
            })
        })
    }
},

```

過濾器

過濾器

过滤器作用：

在不改变数据的情况下，输出前端需要的格式数据

2.0中已经废弃了内置过滤器，需要我们自定义过滤器来使用filter

全局过滤器的定义方法

位置：创建实例之前

```
Vue.filter("过滤器名字", function(val){  
    return val.substr(1,2)  
});
```

Vue.js 局部过滤器

只能在当前vue注册内容中使用
在vue实例中与el属性data属性同级定义

```
filters: {  
    过滤器名字(val){  
        return 输出内容  
    }  
}
```

过滤器的调用方法：

```
{{ msg | 过滤器名字 }}
```

注意事项：

定义全局过滤器，必须放在Vue实例化前面
在没有冲突的前提下，过滤器可以串联

示例

```

<ul>
    <li v-for="(v,i) in arr">{{v|過濾器名字}}</li>
</ul>

// 
new Vue({
    el:"#box",
    data:{
        arr:[1,3,4,5,7,7,5,4,4,9]
    },
    filters:{
        過濾器名字(val){
            return val>5?'大于5':'小于5';
        }
    }
})

```

事件对象

【扩展】事件对象

语法：<div @click='fn(\$event)'></div>中，\$event为事件对象

作用：记录事件相关的信息

```

<input type="text" @keydown.ctrl="fun($event)">

methods:{
    fun(e){
        console.log(e.keyCode)
    }
}

```

事件修饰符

【扩展】事件修饰符

概念：v-on指令提供了事件修饰符来处理DOM事件细节

按键修饰符：.up, .down, .ctrl, .enter, .space等等

语法：@click.修饰符='fn()'

【扩展】事件修饰符

`prevent`修饰符：阻止事件的默认行为(`submit`提交表单)

`stop`修饰符：阻止事件冒泡

`capture`修饰符：与事件冒泡的方向相反，事件捕获由外到内

`self`: 只会触发自己范围内的事件，不包含子元素

`once`: 只会触发一次

注意：修饰符可以串联使用

计算属性

计算属性：就是vue实例中一个有计算`data`数据功能的属性

概念：

顾名思义，首先它是一种属性，其次它有“计算”这个特殊性质。

每次取得它的值得时候，它并不像普通属性那样直接返回结果，而是经过一系列的计算之后再返回结果。

同时只要在它的当中里引用了 `data` 中的某个属性，当这个属性发生变化时，

计算属性仿佛可以嗅探到这个变化，并自动重新执行。

为什么要用计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。

在模板中放入太多的逻辑会让模板过重且难以维护

例：

```
<p>{{text.toUpperCase().substr(2,1)}}</p>
```

计算属性--语法

```
computed: {  
    需要返回的数据(){  
        return 处理操作  
    }  
}
```

计算属性 **VS** 方法

用(计算属性)和(方法)改造：把数据转大写并且截取的例子

```
//计算属性
computed:{
    xt(){
        return this.text.toUpperCase().substr(2,1);
    }
}

//方法
methods:{
    textfun(){
        return this.text.toUpperCase().substr(2,2)
    }
}
```

计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变时才会重新求值。

方法绑定数据只要被调用，方法将总会再次执行函数。

计算属性相对于方法在处理特定场合下节省资源性能

计算属性与方法有什么区别用一句话来解释： 就是计算属性有缓存，方法没有 所以计算属性对性能的消耗更低

实例生命周期

1.什么是实例的生命周期

实例在创建到销毁经过的一系列过程叫生命周期

2.什么是生命周期钩子

在生命周期中被自动调用的函数叫做生命周期钩子

3.生命周期钩子函数的用途

每个 Vue 实例在被创建时都要经过一系列的初始化过程—例如，
需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。
同时在这个过程中也会运行一些叫做生命周期钩子的函数，
这给了用户在不同阶段添加自己的代码的机会。

3.1钩子函数有哪些

`beforeCreate`（创建实例）、`created`（创建完成）、
`beforeMount`（开始创建模板）、`mounted`（创建完成）、
`beforeUpdate`（开始更新）、`updated`（更新完成）、
`beforeDestroy`（开始销毁）、`destroyed`（销毁完成）

钩子函数的书写位置在`data`与`methods`同级位置书写。

3.2 设置数据请求的钩子

`created`里面，如果涉及到需要页面加载完成之后的话就用 `mounted`。

在`created`的时候，视图中的`html`并没有渲染出来，所以此时如果直接去操作`html`的`dom`节点，一定找不到相关的元素

而在`mounted`中，由于此时`html`已经渲染出来了，所以可以直接操作`dom`节点

实例的生命周期—扩展常见问题

1.什么是vue生命周期?

Vue 实例从创建到销毁的过程，就是生命周期。也就是从开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程，我们称这是 Vue 的生命周期。

2.vue生命周期的作用是什么？

生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易完成指定逻辑

3.vue生命周期总共有几个阶段？

它可以总共分为8个阶段：创建前/后，载入前/后，更新前/后，销毁前/销毁后

4.第一次页面加载会触发哪几个钩子？

第一次页面加载时会触发 `beforeCreate`, `created`, `beforeMount`, `mounted` 这几个钩子

5.DOM 渲染在 哪个周期中就已经完成？

DOM 渲染在 `mounted` 中就已经完成了。

6.简单描述每个周期？

`beforeCreate`（创建前） 在数据观测和初始化事件还未开始

`created`（创建后） 完成数据观测，属性和方法的运算，初始化事件，实例中的`el`属性还没有显示出来

`beforeMount`（载入前） 在挂载开始之前被调用，相关的`render`函数首次被调用。

实例已完成以下的配置：编译模板，把`data`里面的数据和模板生成`html`。注意此时还没有挂载`html`到页面上。

`mounted`（载入后） 在`el` 被新创建的 `vue.el` 替换，并挂载到实例上去之后调用。

实例已完成以下的配置：用上面编译好的`html`内容替换`el`属性指向的DOM对象。

完成模板中的`html`渲染到`html`页面中。此过程中进行`ajax`交互。

`beforeUpdate`（更新前） 在数据更新之前调用，发生在虚拟DOM重新渲染和打补丁之前。

可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。

`updated`（更新后） 在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。

调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。

然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。

该钩子在服务器端渲染期间不被调用。

`beforeDestroy`（销毁前） 在实例销毁之前调用。实例仍然完全可用。

`destroyed`（销毁后） 在实例销毁之后调用。

调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。

该钩子在服务器端渲染期间不被调用。

组件

1.什么是组件

组件即自定义控件，是Vue.js最强大的功能之一

2.组件的用途：

组件能够封装可重用代码，扩展HTML标签功能

3.组件的本质

自定义标签

4.组件的分类

全局组件

作用域：不同作用域内均可使用

局部组件

作用域：只在定义该组件的作用域内可以使用

5.组件的类型

1、页面级别的组件：页面级别的组件，通常是views目录下的.vue组件，是组成整个项目的一个大的页面。一般不会

2、业务上可复用的基础组件：在业务中被各个页面复用的组件，这一类组件通常都写到components目录下，然后通过import在各个页面中使用。

3、与业务无关的独立组件：与业务功能无关的独立组件。这类组件通常是作为基础组件，在各个业务组件或者页面组件中被使用。目前市面上比较流行的ElementUI和iview等中包含的组件都是独立组件。如果是自己定义的独立组件，比如富文本编辑器等，通常写在utils目录中。

6.全局组件

建议：组件名（字母全小写且必须包含一个连字符）。这会帮助你避免和当前以及未来的 [HTML](#) 元素相冲突。

定义位置：创建实例前定义全局组件

语法：

```
Vue.component('name', {  
  template: '<div></div>' //template: 'html代码'  
});
```

组件的调用方法：`<组件名></组件名>`

组件的作用域：全局范围内均可调用

示例：

```
<div id="demodiv">  
  <f></f> //调用  
  <hh></hh>  
</div>  
  
Vue.component("f", {  
  template: '<div>小可爱</div>'  
})  
Vue.component('hh', {  
  template: '<h3>小磊</h3>'  
})
```

1. 组件的名字使用驼峰怎么办？

那么在调用的时候 大写转小写 前面加个-

2. 如果组件有多个html怎么办（初学者最容易出错的）

必须必须必须必须必须有一个父元素来进行包裹

组件中如果想使用变量或者是方法 必须在组件自己内部创建

组件的作用域是独立的

7. 局部组件

语法：定义在vue实例中只能在当前实例范围内生效

局部组件的定义

定义位置：实例配置项中定义

`template`的设置：

```
template: 'html代码'  
template: '#template1'
```

数据的定义：

```
data: function(){  
    return {a:1,b:2}  
}
```

局部组件的调用：

组件的调用方法：`<组件名></组件名>`

组件的作用域：定义该组件的作用域内可调用

组件在命名时如果是驼峰命名法 那么在调用的时候用-替代大写 “`myText`”调用 `<my-text></my-text>`

props 选项

作用：`props`选项用来声明它期待获得的数据

`props` 本质：`props` 为元素属性

props 的声明

语法：如果是驼峰命名法需要把大写转小写前面加-

JS 中：

```
props:[ 'message1', 'messAge2' ...]
```

HTML 中：

```
<组件 message='val' mess-age2='val'></组件>
```

props 的使用

与 `data` 一样，`props` 可以用在模板中

可以在 `vm` 实例中像 `this.message` 这样使用

props 验证

仅仅只会在控制台抛出警告，但不会对程序造成影响，原因是`prop`验证是给程序员在开发的时候看的

我们可以为组件的 `prop` 指定验证要求，例如知道的这些数据的类型。

为了定制 `prop` 的验证方式，你可以为 `props` 中的值提供一个带有验证需求的对象，而不是一个字符串数组。

```
props:{  
    // 可以验证 String,Number,Boolean,Array,Object,Date,Function类型  
    ziaprop:Number,  
    // 传入数组验证多个类型  
    zibprop:[String, Number],  
    // 必填的内容  
    zicprop:{  
        type: String,  
        required: true  
    },  
    // 带有默认值的数字  
    zidprop:{  
        type: Number,  
        default: 100  
    }  
}
```

props 验证常见问题

为什么写的没有错但是没有错误提示？

生产版本也就是压缩版的文件删除了警告，
所以使用非压缩版的js文件就可以看到错误

```
<!-- props 完成正向传值 父亲给儿子数据 -->
<!-- props的作用就是用来接收组件外部传递进来的数据 -->
<!-- props语法:
      写在需要接收外部传递数据组件的 data methods等同级位置
      props:["变量","变量"]
-->
```

```
<div id="box">
    <father></father>
</div>
<!-- 组件之间的数据都是完全独立的 互相不能直接使用 -->

<template id="fatherTmp">
    <div class="father">
        <son v-for="(v,i) in arr" :title="v.title"></son>
    </div>
</template>

<template id="sonTmp">
    <div class="son">
        <div>{{title}}</div>
    </div>
</template>
<script>
    new Vue({
        el: "#box",
        // 子组件声明的位置 是在父组件之内
        // 注意：子组件在哪里调用呢？
        // 子组件是在父组件模板中进行调用的
        components: {
            "father": {
                template: "#fatherTmp",
                data(){
                    return {
                        f:"父变量",
                        arr:[
                            {title:"首页"},
                            {title:"我的"},
                            {title:"联系"},
                        ]
                    }
                },
                components: {
                    "son": {
                        template: "#sonTmp",
                        data(){
                            return {
                                s:"子变量"
                            }
                        },
                    },
                }
            }
        }
    })
</script>
```

```
methods:{  
},  
props:["title"]  
}  
}  
})  
})
```

slot (槽口)

slot的作用

用来混合父组件的内容与子组件自己的模板

数量不同，内容也不同的时候建议使用

slot 的使用

语法：

声明组件模板： 定义组件的时候留下slot等待调用的时候插入内容

```
<div>  
  <h2>我是子组件的标题</h2>  
  <slot>  
    只有在没有要分发的内容时才会显示。  
  </slot>  
</div>
```

调用组件模板： 调用的时候直接插入

```
<div>  
  <h2>我是子组件的标题</h2>  
  <slot>  
    只有在没有要分发的内容时才会显示。  
  </slot>  
</div>
```

示例：

子组件:

```
<template>
  <div>
    vfd
    <slot></slot>      //使用slot
  </div>
</template>
```

父组件:

```
<template>
  <div>
    <aa>
      <h3>jdsvdvlmlkm</h3>    //标签不显示,
      <h3>jdsvdvlmlkm</h3>    //可当使用了slot, 页面就会加载显示自定义标签中的标签
    </aa>
  </div>
</template>
import aa from 'a.vue', (引入, 调用, 使用)
```

具名 slot

<slot> 元素可以用一个特殊属性 `name` 来配置如何分发内容

多个slot可以有不同的名字

具名slot将匹配内容片段中有对应slot特性的元素

示例:

子组件:

```
<template>
  <div>
    vfd
    <slot name="s"></slot>    //有name具名的 slot
    <slot name="ww"></slot>    //有name具名的 slot
  </div>
</template>
```

父组件:

```
<template>
  <div>
    <aa>
      hdfdjfbfsbb
      <h3 slot="s">egrtrhtr</h3>      //有相对slot具名的 标签会添加
      <h3>jdsv3453dvlmlkm</h3>      //bo
      <h3 slot="ww">354636</h3>      // y
      <h3>jdsvdvlmlkm</h3>          //n
    </aa>
  </div>
</template>
import aa from 'a.vue', (引入, 调用, 使用)
```

案例:

子组件:

```
<template>
  <div class="slo">
    <div class="title">
      <slot name="title"></slot>
    </div>
    <div class="info">
      <slot name="info"></slot>
    </div>
  </div>
</template>
```

父组件:

```
<aa>
  <div slot="title">我是title</div>
</aa>
<aa>
  <div slot="title">我是title2</div>
  <p slot="info">我是info2</p>
</aa>
```



8.父子组件

子组件声明的位置 是在父组件之内

注意：子组件在哪里调用呢？

子组件是在父组件模板中进行调用的

```
components:{          //父组件
  "father":{
    template:"#fatherTmp",
    data(){
      return {
        f:"父变量"
      }
    },
    methods:{
    },
    components:{      //子组件
      "son":{
        template:"#sonTmp",
        data(){
          return {
            s:"子变量"
          }
        },
        methods:{
        }
      }
    }
  }
}
```

自定义事件 (**逆向传值)

父子组件中传值

父子组件间作用域相互独立所以没有办法直接调用，必须借助于自定义事件来进行传值

子组件传值给父组件叫 <逆向传值> (是不允许的 必须要有事件触发才能传值)

父组件传值给子组件叫 <正向传值> (不需要事件触发)

1. 抛出自定义事件监听

要传值必须要先抛出，在接收

语法：

```
this.$emit('event',val) //event: 自定义事件名称 val: 通过自定义事件传递的值(可选)
```

\$emit：实例方法，用来触发事件监听

2. 接收自定义事件监听

语法：

```
<component @抛出的事件名='函数不加()不加()'></component>
```

```
fn:function(val){ //val:自定义事件传递出的值  
}
```

示例：

子组件:

```
<template>
  <div>
    <!-- 逆向传值必须必须必须必须使用事件来触发 -->
    <button @click="fun()">点我进行逆向传值</button>
  </div>
</template>
<script>
export default {
  methods:{
    fun(){
      // 1.事件触发一个函数
      // 2.使用$emit来创建一个自定义事件 并且在其中传递要逆向传值的数据
      this.$emit("xiaoming",this.zitext)
      // 3.在子组件被调用的时候
    },
    data(){
      return{
        zitext:"我是子组件的数据"
      }
    },
  }
</script>
```

父组件:

```
<!-- 3.在子组件被调用的时候 使用@自定义事件名 =“父组件的函数 但是不加()不加()不加()” -->
<Home v-bind:hometext="text" @xiaoming="fun"/> //调用子组件
```

```
methods:{
  // 4.定义一个函数来接收子组件传递过来的数据 形参就是子组件传递的数据会自动穿入
  fun(val){
    console.log("我是父组件"+val)
    this.num=val
  }
}
```

项目环境配置

2.0vue-cli项目环境配置 (**了解)

项目环境配置（2.0）

安装 vue-cli: `npm install vue-cli -g` (可以自动的构建项目结构和项目目录)

安装webpack: `npm install webpack -g`

`cd` 到指定的项目路径中 并且初始化文件夹 `npm init`

创建项目: `vue init webpack 项目名`

切换到所创建的项目目录下: `cd 你创建的项目文件夹`

启动项目: `npm run dev`

项目目录结构分析（2.0）

文件解释:

`build` 中配置了webpack的基本配置，开发环境配置、生产环境配置（不建议修改）

`config` 中配置了路径端口值等

`node_modules` 为依赖的模块

`src` 放置组件和入口文件

`static` 放置静态资源文件

`index.html` 文件入口

卸载脚手架 :

如果已经全局安装了旧版本的vue-cli(1.x 或 2.x)，需要卸载: `npm uninstall vue-cli -g`
`cnpm uninstall vue-cli -g`

最新CLI环境搭建 (vue-cli @4)

10月16日，官方发布消息称Vue-cli 4.0正式版发布

安装和vue-cli3.0的是一模一样的，与3.0的脚手架，除了目录发生变化一些，其他的都一样

由于近期才推出 企业中还在使用3.0 但是4.0使用方式与3.0相同

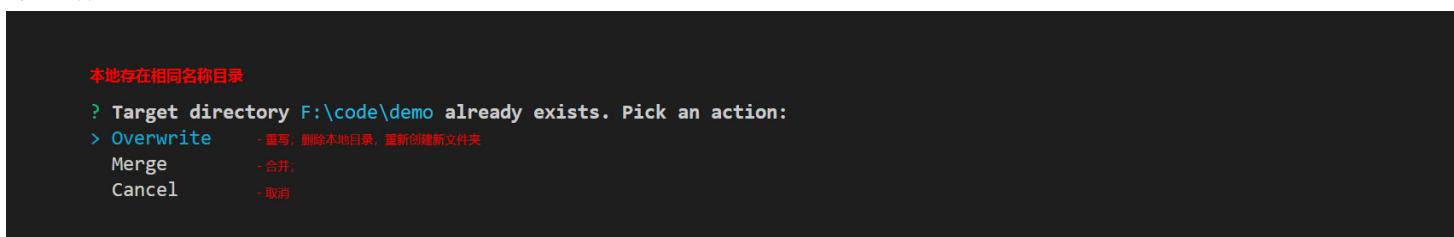
vue-cli @4 安装

`npm install -g @vue/cli`

`vue --version` 查看版本

`vue create 项目名`

本地存储在相同目录下:



本地存在相同名称目录
? Target directory F:\code\demo already exists. Pick an action:
> Overwrite - 覆写: 删除本地目录, 重新创建新文件夹
Merge - 合并
Cancel - 取消

配置方式:

配置方式

```
Vue CLI v3.11.0
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)      - 默认配置;
Manually select features       - 自定义配置;
```

项目需要什么东西：

```
项目需要些什么东西
? Check the features needed for your project: (Press <space> to select, <a> to toggle all, <i> to invert selection)
>(*) Babel
  ( ) TypeScript
  ( ) Progressive Web App (PWA) Support
  ( ) Router
  ( ) Vuex
  ( ) CSS Pre-processors
(*) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
    - 编译: es6 to es5;
    - 代码检测;
    - 浏览器兼容性
    - VUE路由
    - VUE状态管理 ( Vuex )
    - CSS预处理器
    - 代码检测和格式化
    - 单元测试
    - E2E 测试
      - 指定端口号。属于集成测试，通过编写测试用例时，自动化操作用户操作，确保组件的逻辑正常，程序员的数据传递如预期。
```

路由是采用history模式：

```
路由是否采用history模式
? Use history mode for router? (Requires proper server setup for index fallback in production) (Y/n)
```

CSS预编译器：

```
CSS预编译器
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by default): (Use arrow keys)
> Sass/SCSS (with dart-sass)      - SASS dart-sass 编译模式
  Sass/SCSS (with node-sass)        - SASS node-sass 编译模式
  Less                               - Less 编译模式
  Stylus                            - Stylus 编译模式
```

选择ESLint的代码规范：

```
选择ESLint的代码规范
? Pick a linter / formatter config: (Use arrow keys)
> ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

选择何时进行代码检测：

```
选择何时进行代码检测
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection
)
>(*) Lint on save                 - 保存时进行检测
  ( ) Lint and fix on commit      - 提交时进行检测
```

选择Babel PostCSS ESLint等配置文件的存放位置：

```
选择 Babel、PostCSS、ESLint等配置文件存放位置
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow keys)
> In dedicated config files - 配置保存在各自的配置文件中
  In package.json - 保存在package.json文件中
```

是否保存选择的配置：

```
是否保存选择的配置
? Save this as a preset for future projects? (y/N)

是: 下次构建项目时, 可选择此次配置
否: 不保存
```

```
是否保存选择的配置
? Save this as a preset for future projects? (y/N)

是: 下次构建项目时, 可选择此次配置
否: 不保存
```

cd 项目名

npm run serve

安装成功界面：



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project,
check out the [vue-cli documentation](#).

Installed CLI Plugins

[babel](#) [eslint](#)

Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

2.x、4.x的差异

2X:

```
✓ vue-admin
  > node_modules
  > public
  > src
  └ .browserslistrc
    ○ .eslintrc.js
    ♦ .gitignore
    JS babel.config.js
    { package-lock.json
    { package.json
    JS postcss.config.js
    ⓘ README.md
```

4X:

```
✓ vue2
  > build
  > config
  > node_modules
  > src
  > static
  ⚡ .babelrc
  ⚙ .editorconfig
  ○ .eslintignore
  ○ .eslintrc.js
  ♦ .gitignore
  JS .postcssrc.js
  ▷ index.html
  { package-lock.json
  { package.json
  ⓘ README.md
```

vue-cli 配置端口 自动开启 热更新

在项目的根路径下根路径下根路径下根路径下创建一个文件名为vue.config.js

```
module.exports = {
  ...
  devServer: {
    open: true, // 自动开启
    port: 3333, // 修改端口
    host: "127.0.0.1",
    hotOnly: true, // 热更新
  },
}
```

单文件组件

通过一个.vue为后缀的文件来完成一个组件的封装

模板在单文件组件中的书写方式:

```
<template>      <!-- 模板在单文件组件中的书写 -->
  <div>
    <h2 class="title">租房找室友</h2>
    <div class="father" v-for="(v,i) in arr" :key="i">  <!-- v-for key的使用 -->
      <Doubanson :pic="v.pic" :info="v.info" :title="v.title"/>
    </div>
  </div>
</template>

<script>
import Doubanson from './doubanson.vue'          //引入子文件
// export default (在一个文件中只能出现一次 但是我要暴露很多个 那么就用下面的暴露方式)
// export 也是暴露可以出现多次 (在使用的时候必须解构)
import {axiosL} from '@/api/axiosurl.js'        //用结构进入 axios 地址的函数

export default {
  components:{
    Doubanson
  },
  data(){
    return {
      arr:[]
    }
  },
  mounted() {
    axiosL().then(res=>{
      this.arr = res.data.data
      console.log(this.arr)
    }).catch(rej=>{
      console.log(rej)
    })
  },
}
</script>

<style scoped> /* scoped样式的表示: 当前样式仅对当前组件*/
  .title{
    line-height: 30px;
  }
</style>
```

app.vue

```
<template>
<div id="app">
  <!-- 3使用 -->
  <Demo/>
</div>
</template>

<script>
// 如果想引用组件
// 1.必须先引用
// import Demo from "./components/demo.vue"
// import Axioslink from "@/components/axioslink.vue"
export default {
  name: 'App',
  components: {
    // 2.调用
    // Demo,
    Axioslink
  }
}
</script>

<style lang="scss">
</style>
```

v-for key的使用

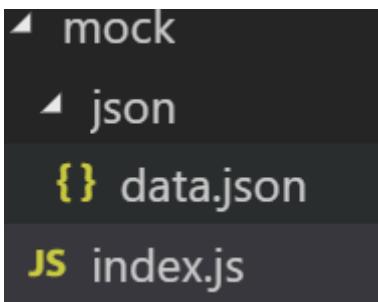
为遍历数组或元素中的唯一标识，增加或删减元素时，
通过这个唯一标识key判断是否是之前的元素，
vue会直接对已有的标签进行复用，不会整个的将所有的
标签全部重新删除和创建，只会重新渲染数据，
然后再创建新的元素直到数据渲染完为止

mock模拟数据

在前端开发过程中，有后台配合是很必要的。但是如果自己测试开发，或者后台很忙，没时间，那么我们需要自己提

```
npm install --save mockjs //在项目根目录下下载
```

在src文件夹下创建相关文件与文件夹



设置请求文件index.js

```
const Mock = require('mockjs');
//格式: Mock.mock( 设置的请求地址, post/get , 返回的数据);
Mock.mock('/user/userInfo', 'get', require('./json/data'));
Mock.mock('/home/banner', 'get', require('./json/data'));
```

引用mock在main.js

路由配置

路由的作用

路由：所有的路径都经由这个模块进行重新分配（改变URL，在不重新请求页面的情况下，更新页面视图。）
根据 url 锚点路径，在容器中加载不同的模块
完成 SPA（单页面应用）的开发

路由原理

利用锚点完成切换
页面不会刷新

路由

用 `Vue.js + Vue Router` 创建单页面应用，是非常简单的。将组件 (`components`) 映射到路由 (`routes`)，然后

1. 定义 (路由) 组件。

注意：可以使用`template`模板进行`html`封装，调用`id`属性更加方便

2. 定义路由

使用`component`来进行路由映射组件。`name`属性是命名路由通过一个名称来标识一个路由

3. 创建 `router` 实例，然后传 `routes` 配置

4. 创建和挂载根实例。

通过 router 配置参数注入路由，从而让整个应用都有路由功能

使用<router-link to="/url"></router-link>标签设置路由跳转

to属性用来设置跳转链接

路由出口:<router-view></router-view>表明路由模版显示的位置

一个路由项目的搭建示例！！！

vue create 名字

删除components和views下自带的文件， app.vue文件中删除（自带配置）

创建api和mock文件夹：

##mock文件夹下的index.js中配置：

```
let Mock = require("mockjs");
Mock.mock("wode/data", "get", require("./data/wode.json"))
```

*##main.js中配置：

```
require("./mock")
```

*##api/api.js利用axios发送数据

```
import axios from "axios";
export function jump(){
    return new Promise((resolve,reject)=>{
        axios({
            url:"wode/data",
            methods:"get"
        }).then(res=>{
            resolve(res)
        }).catch(rej=>{
            reject(rej)
        })
    })
}
```

在项目的根目录下创建vue.config.js

配置自动打开：

```
module.exports={
  devServer:{
    open:true,
    port:8888
  }
}
```

配置解析别名：

```
configureWebpack:{
  resolve:{
    alias:{
      //“别名”: “对应的文件夹”
      "com": "@/components"
    }
  }
}
```

在views文件夹下 创建一级路由页面：创建模板

```
<template>
  <div>
    购物车
  </div>
</template>
```

在router下的index.js中引进并配置路由规则

```
import Vue from 'vue'
import VueRouter from 'vue-router'

import Cart from '@/views/cart.vue';

Vue.use(VueRouter)

const routes = [
  {
    path: '/cart',
    name: 'Cart',
    component: Cart
  },
  { //重定向
    path: '/',
    redirect: "/index"
  },
  { //404页面
    path: '*',
    name: 'No',
    component: No
  }
]
```

main.js中配置路由：

```
new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

app.vue:写导航

```
<router-link to="/index"> index </router-link>
<router-link to="/cart"> cart </router-link>
<router-link to="/wode"> wode </router-link>

<router-view/> //路由出口
```

js跳转

使用 `this.$router` 全局路由的 `push()` 方法进行路由跳转

当 `<router-link>` 对应的路由匹配成功，将自动设置 `class` 属性值 `.router-link-active`。

通过自动设置的类名方便进行路由导航样式设置

常规参数只会匹配被 / 分隔的 URL 片段中的字符。如果想匹配任意路径，我们可以使用通配符 (*)

```
{name:"tema",path:"*",component:tema}
```

匹配任意开头的路径使用通配符 (*)

```
{name:"tema",path:"/demo-*",component:tema}
```

当使用通配符路由时，请确保路由的顺序是正确的，也就是说含有通配符的路由应该放在最后。路由 { path: '*' }

路由匹配优先级

同一个路径可以匹配多个路由，此时，匹配的优先级就按照路由的定义顺序：

谁先定义的，谁的优先级就最高。

编程式导航----> 使用js方式进行跳转路由

```
router.replace()
```

声明式 ----> 使用router-link进行跳转路由

```
<router-link :to=" " replace>
```

扩展路由跳转方式： `router.replace()` 替换

```
router.replace() 替换
```

与 `push()` 唯一的不同就是，它不会向 `history` 记录中添加新记录，而是跟它的方法名一样 — 替换掉当前的 `history` 记录。

`this.$router.go(n)` 这个方法的参数是一个整数，意思是在 `history` 记录中向前或者后退多少步，类似 `window.history.go(n)`。

```
func(){
    // 在浏览器记录中前进一步，等同于 history.forward()
    this.$router.go(1);
},
funb(){
    // 后退一步记录，等同于 history.back()
    this.$router.go(-1);
}
```

动态路由匹配 (路由传参)

动态路由也可以叫做路由传参

组件的显示内容经常会根据用户选择的内容不同来在同一个组件中渲染不同内容。

那么在这个时候就需要动态路由

动态路径参数

使用动态路由匹配中的 动态路径参数来进行路由配置。

注意：动态路径参数 以冒号：开头

```
{name:"tema",path:"/index/:id/:name",component:tema},
```

绑定参数

路由导航绑定参数的两种方式 但是注意 params只能通过路由配置中的name属性来引用路由

```
<router-link to="/index/参数1/参数2">index</router-link>
<!--第二种方式：不能忘 同时params只能通过name来引入路由--&gt;
&lt;router-link :to="{name:'tema',params:{id:'参数11',name:'参数22'}}&gt;index&lt;/router-link&gt;</pre>
```

js方式进行参数绑定

```
fun(){
    this.$router.push("/index/js参数1/js参数2");
    // 或者 但是注意params只能通过name来引入路由
    this.$router.push({name:'tema',params:{id:'js参数11',name:'js参数22'}})
}
```

获取路由传入参数

如果想得到路径参数那么使用\$route.params.id

```
const tema={template:<p>index{{this.$route.params.id}}--{{this.$route.params.name}}</p>};
```

或者是使用this实例中的this.\$route.params.id进行调用

```
created(){
    // 路由信息都被挂载到this实例中去
    console.log("路由参数是: "+this.$route.params.id+"---"+this.$route.params.name);
},
```

动态路由--query传参

1. 路由参数不需要添加内容

```
{name:"temp",path:"/home",component:temp},
```

2. 路由导航绑定参数的方式

```
<router-link to="/home?id=参数1&name=参数2">home</router-link>
<!-- 使用name引入路由 -->
<router-link :to="{name:'temp',query:{id:'参数11',name:'参数22'}}>home</router-link>
<!-- 使用path引入路由 -->
<router-link :to="{path:'/home',query:{id:'参数111',name:'参数222'}}>home</router-link>
```

3. js方式进行参数绑定

```
fun(){
  this.$router.push("/home?id=js参数1&name=js参数2");
  // 通过path引用路由
  this.$router.push({path: '/home',query:{id:'js参数11',name:'js参数22'}})
  // 通过name引用路由
  this.$router.push({name:'temp',query:{id:'js参数111',name:'js参数222'}})
}
```

params 与 query 区别

用法上的：

query要用path来引入，params要用name来引入，接收参数都是类似的，
分别是this.\$route.query.name和this.\$route.params.name。

url展示上的：

params类似于post，query更加类似于我们ajax中get传参，说的再简单一点，
前者在浏览器地址栏中不显示参数，后者显示，所以params传值相对安全一些。

\$router 和 \$route 的区别

\$router是VueRouter的一个对象，router的实例对象，
这个对象中是一个全局的对象，他包含了所有的路由包含了许多关键的对象和属性。

举例：history对象

\$route是一个跳转的路由对象，每一个路由都会有一个route对象，是一个局部的对象，
可以获取对应的name,path,params,query等

绑定参数 和 获取路由传入参数（示例）

```

***绑定参数
<router-link to="/wode/ding"> ding </router-link>
<router-link to="/wode/gou"> gou </router-link>
<ul>
  <li v-for="(v,i) in arr" :key="i">
    <p>
      <router-link :to="{name:'all',params:{title:v.content}}">{{v.title}}</router-link>
    </p>
  </li>
</ul>

<router-view></router-view>

```

***把数据带到详情页： 获取路由传入参数

```

<template>
  <div>
    <button @click="fun()">回</button>
    <p>{{this.$route.params.title}}</p>
  </div>
</template>

<script>
export default {
  methods: {
    fun(){
      this.$router.go(-1)
    }
  }
}
</script>

```

hash模式-history模式

1. hash模式

hash模式url里面永远带着#号，我们在开发当中默认使用这个模式。

2. history模式

history模式没有#号，是个正常的url适合推广宣传。

考虑url的规范那么就需要使用history模式，因为当然其功能也有区别，在开发app的时候有分享页面，这个分享出去的页面就是用vue做的，把这个页面分享到第三方的app里，有的app里面url是不允许带有#号的，所以要将#号去除那么就要使用history模式，history模式还有一个问题就是，做刷新操作，会出现404错误，那么就需要和后端人配合让他配置一下apache或是nginx的url重定向，重定向到你的首页路上。

history模式使用

```
const router = new VueRouter({
  mode: 'history',//设置history模式
  routes
})
```

history模式与hash模式区别

hash与history的区别

	hash	history
url显示	有#，很Low	无#，好看
回车刷新	可以加载到hash值对应页面	一般就是404掉了
支持版本	支持低版本浏览器和IE浏览器	HTML5新推出的API

路由懒加载

懒加载简单来说就是延迟加载或按需加载，即在需要的时候的时候进行加载。

为给客户更好的客户体验，首屏组件加载速度更快，解决白屏问题。做的一些项目越来越大。

vue打包后的js文件也越来越大，这会是影响加载时间的重要因素。

当构建的项目比较大的时候，懒加载可以分割代码块，提高页面的初始加载效率

常用的懒加载方式有两种：即使用vue异步组件懒加载 和 ES中的import

1. ES 提出的import(推荐使用)

```
const HelloWorld = () => import('需要加载的模块地址')
```

2. vue异步组件懒加载-- resolve

主要是使用了Promise.resolve()的异步机制，用require代替了import，实现按需加载

```
component: resolve=>(require(["引用的组件路径"], resolve))
```

`promise`是什么？

`Promise`是一种异步操作的解决方案，将写法复杂的传统的回调函数和监听事件的异步操作，用同步代码的形式表达出来。避免了多级异步操作的回调函数嵌套。

- 1、主要用于异步计算
- 2、可以将异步操作队列化，按照期望的顺序执行，返回符合预期的结果
- 3、可以在对象之间传递和操作`promise`，帮助我们处理队列

配置解析别名---修改文件夹引用别名

第一个参数：是你设置的别名 第二个参数：所指向的路径

```
configureWebpack:{  
    resolve:{  
        alias:{  
            // "别名": "对应的文件夹"  
            "com": "@/components"  
        }  
    }  
}
```

```
module.exports={  
...  
    configureWebpack:{  
        resolve:{  
            alias:{  
                // "别名": "对应的文件夹"  
                "com": "@/components"  
            }  
        }  
    },  
    devServer:{  
        port:8888, //端口号修改  
        open:true, //自动开启  
    }  
}
```

嵌套路由

嵌套路由的配置

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各

配置二级路由路径参数中使用 `children` 配置

```
{  
  path: '/cart',           //一级  
  name: 'cart',  
  component: cart,
```

配置二级路由：

(一)二级路由中路径不加 /

1. 在父组件中设置路由出口： router-view
2. 由于我们没有加 /，所以在路由导航的时候应该是： /一级路由/二级路由

```
children:[           //二级  
  {path:"ding",name:"ding",component:ding},  
  {path:"gou",name:"gou",component:gou}  
]
```

(二)二级路由路径中加 /

1. 在父组件中设置路由出口 router-view
2. 由于我们加了 /，所以在路由导航的时候应该是： /二级路由

```
children:[  
  {path:"/ding",name:"ding",component:ding},  
  {path:"/gou",name:"gou",component:gou}  
]  
,
```

路由重定向

重定向也是通过 routes 中的 redirect 属性配置来完成

```
{  
  path: '/',  
  redirect: '/index'    // 重定向 重新定位方向  
},
```

路由守卫/导航守卫

导航守卫

vue项目中经常在路由跳转前做一些验证，比如登录验证，是网站中的普遍需求。

导航守卫-全局前置守卫

当一个导航触发时，全局前置守卫(在进入组件之前)按照创建顺序调用。

vue-router 提供的 `router.beforeEach((to, from, next)=>{})`
可以方便地实现全局前置导航守卫

`to`: 即将要进入的目标 路由对象

`from`: 当前导航正要离开的路由

`next`: 下一步执行

The screenshot shows a file structure for a Vue project named 'myapp'. The 'src/router/index.js' file is open, containing the following code:

```
// 全局前置守卫/全局前置路由钩子/全局前置导航守卫/全局路由限制
router.beforeEach((to, from, next) => {
  // console.log(to)
  // console.log(from)
  // next()

  // 通过请求后台判断到用户是否登录了 如果登录让他能正常浏览项目 反之只能去login和phone的两个路由
  if(to.path === '/login' || to.path === '/phone'){
    next()
  }else{
    alert("您必须登录后才能访问!!!")
    next('/login')
  }
})

export default router
```

A red box highlights the entire code block, and a blue circle points to the first line of the code.

导航守卫-全局后置钩子

当一个导航触发时，全局后置钩子(在进入组件之后)调用。

vue-router 提供的 `router.afterEach((to, from) => {})` 实现全局后置守卫

`to`: 即将要进入的目标 路由对象

`from`: 当前导航正要离开的路由

The screenshot shows a file structure for a Vue project named 'myapp'. The 'src/router/index.js' file is open, containing the following code:

```
// 全局后置守卫
router.afterEach((to, from) => {
  console.log(to)
  console.log(from)
  alert("欢迎尊贵的会员！！！！")
```

A red box highlights the entire code block, and a blue circle points to the first line of the code.

导航守卫-路由独享的守卫

与全局前置守卫相比路由独享守卫只是对当前路由进行单一控制参数和全局前置守卫相同

在路由配置上直接定义 `beforeEnter` 进行路由独享守卫定义

```
{name: "temp", path: "/home", component: temp, beforeEnter: (to, from, next) =>
  alert("没有登陆不能访问！！！");
  next("/login");
  // 如果next()中什么都不传递那么就会直接进入到当前路由
  // next()
},
```

导航守卫-组件内的守卫

组件内守卫只会对当前组件生效。

`beforeRouteEnter` 在 **进入** 组件前调用

`beforeRouteLeave` **离开** 路由之前

导航守卫-组件内的守卫**beforeRouteEnter**

在组件中使用`beforeRouteEnter(to, from, next) {}`来进行进入组建前的钩子

```
var tema={
  template: "#tema",
  data:function(){
    return{
      text:"组件数据"
    }
  },
  beforeRouteEnter (to, from, next) {
  // alert("来到了组件"+this.text)
  // 因为通常在组件内守卫需要调用data数据完成指定逻辑所以必须在next()中使用回调函数进行调用
  // 如果不在的话 是无法调用的因为在调用beforeRouteEnter的时候因为当守卫执行前，组件实例还没被创建
    next((d)=>{
      alert("来到了组件--"+d.text)
    })
  }
};
```

导航守卫-组件内的守卫**beforeRouteLeave**

在组件中使用`beforeRouteLeave(to, from, next) {}`来进行离开组件的钩子

The screenshot shows a file tree on the left and a code editor on the right. The file tree includes: myapp, node_modules, public, src (assets, components, router), index.js, views (About.vue, Home.vue, login.vue, phone.vue), and App.vue. The code editor shows a script block with a red box highlighting the following code:

```
beforeRouteLeave (to, from, next) {
    console.log(to);
    console.log(from);

    if(confirm("您确定离开吗?")){
        next()
    }else{
        // 用户不想离开
        next(false)
    }
}
```

A large red box surrounds the entire code block, and the word "组件内" is written in red at the bottom right of the box.

```
beforeRouteLeave (to, from, next) {
  if(confirm("确定离开吗")==true){
    next();
  }else{
    // 不进行下一步(也就是不从当前路由离开)
    next(false)
  }
};
```

路由知识点扩展---数据获取

进入某个路由后，需要从服务器获取数据。例如，在渲染用户信息时，你需要从服务器获取用户的数据。我们可以通过两种方式来实现

导航完成之后获取：先完成导航，然后在接下来的组件生命周期钩子中获取数据

导航完成之前获取：导航完成前，在路由进入的守卫中获取数据，在数据获取成功后执行导航。

从技术角度讲，两种方式都不错 — 就看你想要的用户体验是哪种。

封装工具库

在**src**文件夹下创建一个**util**（工具文件夹）。把一些工具类的**js**文件进行统一管理

```

aa > myapp > src > util > JS index.js > stripscript
1   export function stripscript(s) {
2     var pattern = new RegExp(`[^~!@#$^&*()=|{}';',\\\[\\].<>/?~! @#￥.....&* () &;-|{`);
3     var rs = "";
4     for (var i = 0; i < s.length; i++) {
5       rs = rs + s.substr(i, 1).replace(pattern, '');
6     }
7     return rs;
8   }

```

在需要的位置引用使用

```
import {stripscript} from "util/index"
```

axios拦截器

axios模块



axios请求封装原理



axios拦截器类型

1. 请求拦截器

请求拦截器的作用是在请求发送前进行一些操作，
例如在每个请求体里加上token，统一做了处理如果以后要改也非常容易。

2. 响应拦截器

响应拦截器的作用是在接收到响应后进行一些操作
，例如在服务器返回登录状态失效，需要重新登录的时候，跳转到登录页。

*** axios拦截器初体验

1. 在util工具文件夹中创建request.js文件用来编写拦截器。

```
import axios from "axios"
// 创建axios 赋值给常量service
const service = axios.create();

// 添加请求拦截器 (Interceptors)
service.interceptors.request.use(function (config) {
    // 发送请求之前做写什么
    return config;
}, function (error) {
    // 请求错误的时候做些什么
    return Promise.reject(error);
});

// 添加响应拦截器
service.interceptors.response.use(function (response) {
    // 对响应数据做点什么
    return response;
}, function (error) {
    // 对响应错误做点什么
    return Promise.reject(error);
});
export default service
```

Interceptors

You can intercept requests or responses before they are handled by `then` or `catch`.

```
// Add a request interceptor
axios.interceptors.request.use(function (config) {
  // Do something before request is sent
  return config;
}, function (error) {
  // Do something with request error
  return Promise.reject(error);
});

// Add a response interceptor
axios.interceptors.response.use(function (response) {
  // Do something with response data
  return response;
}, function (error) {
  // Do something with response error
  return Promise.reject(error);
});
```

If you may need to remove an interceptor later you can.

```
const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

You can add interceptors to a custom instance of axios.

```
const instance = axios.create();
instance.interceptors.request.use(function () {/*...*/});
```

2. 在拦截器文件进行测试

```
// 添加响应拦截器
service.interceptors.response.use(function (response) {
  // 对响应数据做点什么
  return response;
}, function (error) {
  // 对响应错误做点什么
  return Promise.reject(error);
});

service.request({
  url:"http://localhost:3000/",
  method:"get"
}).then((ok)=>{
  console.log(ok)
})
```

export default service

3. 在页面中引用之后查看请求情况

```
import service from "@/util/request.js"
```

1. 在api文件夹的对应文件中引用拦截器文件

```
import service from "@/util/request.js"
```

2. 在封装请求文件中编写并暴露

```
import service from "@/util/request.js"
/**
 * 我是第一个请求
 */
export function ajaxLink(){
  service.request({
    url:"http://localhost:3000/",
    method:"get"
  }).then((ok)=>{
    console.log(ok)
  })
}
```

3. 在需要的地方引用调用请求

```
<script>

  import {ajaxLink} from "@/api/About.js"
  export default {
    beforeCreate() {
      // 调用封装的请求
      ajaxLink()
    },
  }
</script>
```

代理跨域

proxyTable解决跨域

在vue.config.js中

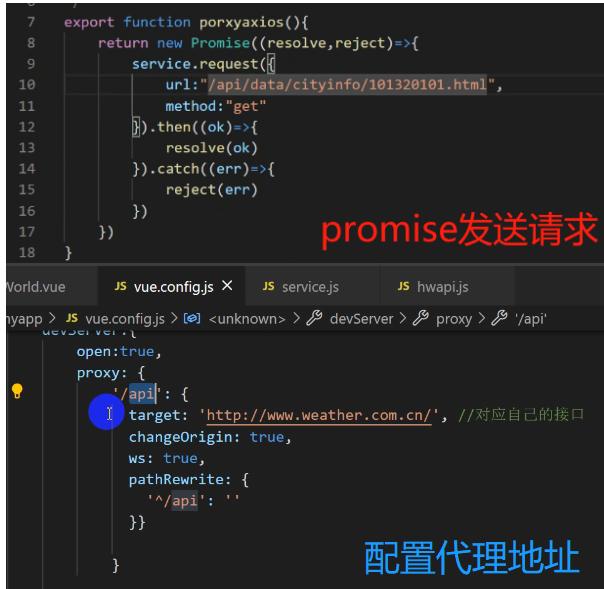
```
proxy: {
  '/api': {
    target: 'http://localhost:3000/', // 对应自己的接口
    changeOrigin: true,
    ws: true,
    pathRewrite: {
      '^/api': ''
    }
  }
}
```

```
module.exports={
  devServer:{
    port:8888,//端口号修改
    open:true,//自动开启
    host:"127.0.0.1",//修改host
    proxy: {
      '/api': {
        target: 'http://localhost:3000/', //对应自己的接口
        changeOrigin: true,
        ws: true,
        pathRewrite: {
          '^/api': ''
        }
      }
    }
}
```

"<http://www.weath.com.cn/data/cityinfo/101320101>"

```
7 export function porxyaxios(){
8   return new Promise((resolve,reject)=>{
9     service.request([
10       url:"/api/data/cityinfo/101320101.html",
11       method:"get"
12     ]).then((ok)=>{
13       resolve(ok)
14     }).catch((err)=>{
15       reject(err)
16     })
17   })
18 }
```

promise发送请求



封装url地址： 方便后期修改。



```
1023 > myapp > src > util > url > js hwurl.js > default
1   let url="http://localhost:8888"
2
3   export default[
4     login:url+"/login",
5     login:url+"/login",
6     login:url+"/login",
7     login:url+"/login",
8     login:url+"/login",
9   ]
```

封装url地址

动态组件

让多个组件使用同一个挂载点，并动态切换，这就是动态组件。

使用 `is` 特性来切换不同的组件

```
<component :is="com"></component>
```

```

<template>
  <div class="hello">
    index
    <br>
    <button @click="demo='demoa'">aa</button>
    <button @click="demo='demob'">bb</button>
    <component :is="demo"></component>
  </div>
</template>

<script>
import demoa from '@/components/a.vue'
import demob from '@/components/b.vue'

<template>
  <div>
    <button @click="com='demoa'">a</button>
    <button @click="com='demob'">b</button>
    <button @click="com='democ'">c</button>
    <component :is="com"></component>
  </div>
</template>

<script>
import demoa from "@/components/demoa.vue"
import demob from "@/components/demob.vue"
import democ from "@/components/democ.vue"
export default {
  components:{
    demoa,
    demob,
    democ,
  },
  data() {
    return {
      com:"demoa"
    }
  },
}

```

keep-alive

在上一个demo中我们不停的切换两个标签页的内容时候，会发现在练习我们中选择好的内容，切换路由之后会恢复初也就是说之前的状态丢失。原因是每次切换路由的时候，Vue 都创建了一个新的 组件实例

解决这个问题，我们可以用一个 `<keep-alive>` 元素将其路由出口包裹起来。

在切换过程中将状态保留在内存中，

防止重复渲染DOM，减少加载时间及性能消耗，提高用户体验性

```
<router-link to="/home">Home</router-link>
<router-link to="/index">index</router-link>
<keep-alive>
  <router-view></router-view>
</keep-alive>
```

keep-alive 属性

在vue 2.1.0 版本之后，`keep-alive`新加入了两个属性：

`include`(包含的组件缓存) 与 `exclude`(排除的组件不缓存，优先级大于`include`) 。

```
<!-- Topbara,Topbarb两个组件会被缓存 -->
  <keep-alive include="Topbara,Topbarb">
    <component :is="text"></component>
  </keep-alive>

</div>
<!-- 动态组件 -->
<button @click="com='Demoa'">点我去demoa</button>
<button @click="com='Demob'">点我去demob</button>
<button @click="com='Democ'">点我去democ</button>
<button @click="com='Demoinput'">点我去demoinput</button>
<keep-alive include="Demoa">
  <!-- 我现在用keepalive包裹了动态组件的4个内容 但是我只想让其中demoa缓存起来
      开发中遇见的问题 keepalive包裹了多个动态组件 只想让其中一个缓存怎么办?
  -->
  <component :is="com"></component>
</keep-alive>
```

keep-alive 的钩子函数

这两个生命周期函数一定是要在使用了`keep-alive`组件之上。

`activated` 类型: `func` 触发时机: `keep-alive`组件激活时使用;

`deactivated` 类型: `func` 触发时机: `keep-alive`组件停用时调用;

refs

ref和\$refs的使用

在Vue中一般很少会用到直接操作DOM，但不可避免有时候需要用到，这时我们可以通过`ref`和`$refs`这两个来实现

`ref` 被用来给元素或子组件注册引用信息，引用信息将会注册在父组件的`$refs`对象上，如果是在普通的DOM元素上使用，引用指向的就是DOM元素，如果是在子组件上，引用就指向组件的实例。

`$refs` 是一个对象，持有已注册过`ref`的所有的子组件。

1.用在dom元素上

可以获取DOM对象

绑定：`<p ref="demo">我是ref</p>`

获取：`this.$refs.demo`

示例：

```
<p ref="pp">我是ref 哈哈</p>
<button @click="fun()">点我让ref变色</button>
methods: {
  fun(){
    this.$refs.pp.style.color="hotpink"
  }
},
```

2.用在组件上

`ref`绑定到组件上可以快速访问到组件实例，及其相关属性方法

```
<Home ref="comHome"></Home>
<button @click="fun()">点我获取到home组件相关内容</button>
```

获取子组件的`data`数据/方法：

```
fun(){
  window.console.log(this.$refs.comHome.hometext)
  this.$refs.comHome.funb()
}
```

子组件:

```
export default {
  data() {
    return {
      zi:"sondata"
    }
  },
}
```

父组件:

```
<demoa ref="re"/>          //调用子组件
<button @click="fun11()">点我获取子组件的数据</button>
```

```
fun11(){
  console.log(this.$refs.re.zi)      //sondata
}
```

自定义指令

自定义指令

除了内置指令外， 用户自定义的指令

局部指令定义:

```
directives:{
  自定义指令的名字: {
    自定义指令钩子函数(el){
      操作逻辑
    }
  }
},
```

bind: 绑定指令到元素上，只执行一次

inserted: 绑定了指令的元素插入到页面中展示时调用，基本上都是操作这个钩子函数

update: 所有组件节点更新时调用

componentUpdated: 指令所在组件的节点及其子节点全部更新完成后调用

unbind: 解除指令和元素的绑定，只执行一次

示例:

```

<input type="text" v-xiaoming/>          //使用自定义指令
<h1 v-xiaohong>自定义指令</h1>

directives:{
  xiaoming:{
    inserted(el){
      // 这个指令是让输入框在页面加载完毕之后自动获得焦点
      el.focus()
    }
  },
  xiaohong:{
    inserted(el){
      el.style.color="red"
    }
  }
}

```

VUEX

什么是vuex

Vuex 是一个专为 Vue.js 应用程序开发中管理的一个模式。

通过创建一个集中的数据存储，方便程序中的所有组件进行访问



思考：

传统vue是单向数据流。如果是兄弟组件之间传值兄弟组件间的状态传递无能为力

我们经常会采用父子组件通过正向/逆向传值来对数据进行传递。以上的这些模式非常脆弱，通常会导致无法维护
vuex只能用于单个页面中不同组件（例如兄弟组件）的数据流通。

安装vuex

```
npm install vuex --save
```

配置vuex文件创建在src中创建store文件夹-->与store.js

单一状态树：

Vuex 使用单一状态—用一个对象就包含了全部的应用层级状态。

至此它便作为一个“唯一数据源”而存在。这也意味着，每个应用将仅仅包含一个 store 实例。

创建store实例：

```
import Vue from 'vue'  
import Vuex from 'vuex'  
  
Vue.use(Vuex)  
  
export default new Vuex.Store({  
  state: {  
  },  
  mutations: {  
  },  
  actions: {  
  },  
  modules: {  
  }  
})
```

1. vuex--State数据源

vuex中的数据源state，我们需要保存的数据就保存在这里

```
import Vue from "vue"  
import Vuex from "vuex"  
Vue.use(Vuex)  
export const store=new Vuex.Store({  
  state:{  
    arr:[  
      {id:1,title:"家里比海盗好1",content:"近平总书记这样称赞他们！ 三去一降一补 亮出实招实策"},  
      {id:2,title:"家里比海盗好2",content:"长三角部分外商追加投资调研 上海：当好改革开放排头兵"},  
      {id:3,title:"家里比海盗好3",content:"70年的新中国发生什么跨越 红军的故事永难忘 理上网课"},  
      {id:4,title:"家里比海盗好4",content:"86岁老兵捐款千万 自己最好的鞋15元 不忘初心 牢记使命"},  
      {id:5,title:"家里比海盗好5",content:"以绿色金融建设绿色一带一路 今年来中国商务运行稳中向好"},  
      {id:6,title:"家里比海盗好6",content:"中科院发布难题清单，提出问题比解决问题更重要"},  
      {id:7,title:"家里比海盗好7",content:"近平总书记这样称赞他们！ 三去一降一补 亮出实招实策"}  
    ],  
  },  
})
```

vuex--使用数据源

要使用首先在main.js引入vuex。

一对大括号的原因是，指定要从其他模块导入的变量名。

```
import {store} from './store/store'
```

```
new Vue({
  // 引入对外暴露的vuex对象
  store:store,
  el: '#app',
  router,
  components: { App },
  template: '<App/>'
})
```

1. 可以使用`$store.state.xx`调用

```
{{ $store.state.text }}

mounted(){
  console.log(this.$store.state.text)
}
```

2. 使用数据的组件中使用计算属性调用

```
this.$store.state.xxx
```

```
computed(){
  obj(){
    return this.$store.state.obj
  }
}
```

vuex--state 扩展知识点

一、Vuex就是一个仓库，仓库里面放了很多对象。其中`state`就是数据源存放地，对应于与一般Vue对象里面的`data`

二、`state`里面存放的数据是响应式的，Vue组件从`store`中读取数据，若是`store`中的数据发生改变，依赖这个数据的组件也会发生更新

三、它通过`mapState`把全局的`state` 和 `getters` 映射到当前组件的`computed` 计算属性中

2. Getters

vuex--Getters属性

`getters`相当于之前组件中学习的计算属性，`getters`属性主要是对于`state`中数据的一种过滤

使用场景：在项目开发中，有时候希望对`state`中的某个属性在多个组件中展示出不同状态

```
getters:{  
    // 接收state作为参数  
    newarr(state){  
        var newarr=state.arr.filter((v,i)=>{  
            // 过滤出id大于3的内容  
            if(v.id>3){  
                return v  
            }  
        })  
        // 不要忘了  
        return newarr;  
    }  
}
```

vuex--使用Getters数据

与使用`state`相同，在组件中的计算属性当中使用`this.$store.getters.xxx`来进行调用

```
newtextobj(){  
    return this.$store.getters.newdataobj  
}
```

vuex--Getters扩展知识点

vuex的Getter特性是？

- 一、`getters` 可以对`State`进行计算操作，它就是`Store`的计算属性
- 二、 虽然在组件内也可以做计算属性，但是`getters` 可以在多组件之间复用
- 三、 如果一个状态只在一个组件内使用一次或者使用了多次但是展示的形态相同，是可以不用`getters`

3.Mutations

vuex--Mutations :修改数据

`mutations`，里面装着一些改变数据方法的集合，就是把处理数据逻辑方法全部放在`mutations`里面（当触发事件的时候想改变`state`数据的时候使用`mutations`）

```
mutations:{  
    up(state){  
        state.arr.forEach((v,i)=>{  
            | v.id++  
        })  
    }  
}
```

注意：不能直接调用一个 mutations 中的处理函数 要使用
`this.$store.commit()` 来进行调用。

```
methods:{  
    fun(){  
        // this.$store.commit("调用mutations的函数名");  
        this.$store.commit("dataup");  
    }  
}
```

vuex--Mutations 提交载荷（Payload）

之前的只是一个简单的修改state中的属性

在实际项目中往往会有值传递给Mutations 给`store.commit`传一个附加参数，他就叫做mutation的载荷

```
<button @click="fun(i)>修改</button>  
  
funa(num){  
    this.$store.commit("uptwo",num)  
}  
  
mutations:{  
    uptwo(state,payload){  
        state.arr.foreach((v,i)=>{  
            if(i==payload){  
                state.arr[i].title = "obj"  
            }  
        })  
    }  
}
```

多个参数也可以传递一个对象

```
this.$store.commit('add', { 'num': 20 } )
```

****vuex--Mutations 问题

vuex中页面刷新数据丢失问题

使用vue进行开发的过程中，状态管理工具一般使用vuex进行管理。但是修改后的vuex数据存储在内存中，所以当前页面刷新数据会产生丢失现象

使用H5特性本地存储

解决：

```
created () {
    //在页面加载时读取localStorage里的状态信息
    if (localStorage.getItem("data") ) {
        //replaceState替换数据 Object.assign合并对象
        this.$store.replaceState(Object.assign({}, this.$store.state, JSON.parse(localStorage.get
    })
    //在页面刷新时将vuex里的信息保存到localStorage里
    window.addEventListener("beforeunload",()=>{
        localStorage.setItem("data",JSON.stringify(this.$store.state))
    })
},
},
```

4. Actions

vuex--Actions

Actions 进行操作，使用Actions 进行异步操作（异步请求）

```
actions:{
    actionfun(context){
        //调用mutations中的方法
        context.commit(dataup)
    }
}
```

分发 Action: Action 通过 this.\$store.dispatch("xxxx");方法触发

```
this.$store.dispatch("actionfun")
```

vuex--Actions 载荷（Payload）

Action同样支持载荷

```
<button @click="funb(i)>修改2</button>

funb(num){
    this.$store.dispatch("act",num)
}

actions:{
    act(context,payload){
        context.commit("uptwo",payload)
    }
}
```

vuex--Actions总结

Actions可以理解为通过将mutations里面处理数据的方法变成可异步的处理数据的方法，简单的说就是异步操作数据（但是还是通过mutation来操作，因为只有它能操作）

5. modules

vuex--modules

在Vue中State使用是单一状态树结构，应该所有的状态都放在state里面，如果项目比较复杂，那state是一个很大的对象，store对象也将变得非常大，难于管理。

module：可以让每一个模块拥有自己的state、mutation、action、getters，使得结构非常清晰，方便管理。

创建文件容纳模块

```
export let a={

    namespaced: true, //命名空间为true
    state: {
        text:"vuex数据"
    },
    mutations: {
    },
    actions: {
    },
}
```

在vuex中引用模块

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
import {a} from "./model/a.js"
export default new Vuex.Store({
  state: [
    ],
  mutations: {
    },
  actions: {
    },
  modules: {
    a
  }
})
```

使用数据: `$store.state.模块名.xxx`

```
<template>
  <div class="about">
    <h1>This is an about page--{{ $store.state.a.text }}</h1>
  </div>
</template>
<script>
export default {
  mounted(){
    console.log(this.$store.state.a.text)
  }
}</script>
```

修改数据: 和没有modules的时候一样。

```
add(){
  this.store.commit("add")
}
```

```
mutations:{
  add(state){
    state.text += "sdgsg";
  }
}
```

vuex--表单处理

当在严格模式中使用 Vuex 时，在属于 Vuex 的 state 上使用 `v-model` 会比较棘手。在用户输入时，`v-model` 会试图直接修改 state 数据。在严格模式中，由于这个修改不是在 mutation 函数中执行的，这里会抛出一个错误。

vuex--表单处理解决方式

Vuex 的思维解决，给 `<input>` 中绑定 `value`，然后侦听 `input` 或者 `change` 事件，在事件回调中调用 `action`。

```
<input type="text" name="uname" :value="newtextobj[0].age" @input="inputfun"/>
```

调用函数。

```
// 绑定事件对象通过e.target.value得到输入框的值并且传入并调用vuex actions
inputfun(e){
  this.$store.dispatch("actionfunb",e.target.value);
}
```

创建 `actions` 调用 `mutations` 并接收数据

```
actionfunb(context,v){
  context.commit("datauodate",v);
}
```

使用 `mutations` 来进行修改

```
datauodate(state,v){
  state.dataobj[0].age=v;
}
```

兄弟组件传值----中央事件总线

1. 在 `src/` 下创建一个文件夹用来存放 `xxx.js` 文件在其中只创建一个新的 `Vue` 实例，以后它就承担起了组件之间通信的桥梁了，也就是中央事件总线。

```
import Vue from "vue"
export default new Vue()
```

2. 创建一个 `A` 组件，引入事件总线的 `js` 文件，接着添加一个按钮并绑定一个点击事件，进行自定义事件抛出

```
<template>
  <div>
    |   aaa
    |   <button @click="fun()">点我兄弟组件传值</button>
  </div>
</template>

<script>
import eventBus from '@/assets/eventBus.js'
export default {
  methods:{
    fun(){
      eventBus.$emit("pao","我是数据")
    }
  }
</script>
```

3. 再创建一个B组件，引入eventBus事件总线，在mounted钩子，监听了自定义事件，并把传递过来的字符串参数传递给了on监听器的回调函数

\$on:监听当前实例上的自定义事件

```
<template>
  <div>
    | bbbbb{{text}}
  </div>
</template>

<script>
import eventBus from '@/assets/eventBus.js'
export default [
  data(){
    return{
      text:"默认值"
    }
  },
  mounted(){
    eventBus.$on("pao",(msg)=>{
      this.text=msg;
    })
  }
]
```

总结

- 1、创建一个事件总线，例如demo中的eventBus，用它作为通信桥梁
- 2、在需要传值的组件中用bus.emit触发一个自定义事件，并传递参数(emit前加美元符)
- 3、在需要接收数据的组件中用bus.\$on监听自定义事件，并在回调函数中处理传递过来的参数

打包上线

打包上线

```
npm run build
```

在项目下面会生成一个dist文件夹就是打包好的内容 运行其中的html文件即可

1. 运行之后会发现白屏 在控制台会发现有报错
2. 解决:修改资源路径为./

在vue.config.js中配置:

```
module.exports = {

  // 基本路径
  publicPath: './', //部署应用包时的基本 URL
  outputDir: 'dist', // 输出文件目录
  assetsDir: '', //放置生成的静态资源 (js、css、img、fonts) 的 (相对于 outputDir 的) 目录
  runtimeCompiler: false, //是否使用包含运行时编译器的 Vue 构建版本。设置为true可以使用template
  productionSourceMap: false, //生产环境是否生成 sourceMap 文件
  lintOnSave: true,
  chainWebpack(config) {
    config.resolve.alias
      .set('style', resolve('public/style'))
    config.output.filename('js/[name].[hash:16].js');//hash值设置
    config.output.chunkFilename('js/[id].[hash:16].js');
    // config.output.filename('css/[name].[hash:16].css');//hash值设置
  },
  configureWebpack: () => {
  },
  // css相关配置
  css: {
    // 是否使用css分离插件 ExtractTextPlugin
    extract: true,
    // 开启 CSS source maps?
    sourceMap: false,
    // css预处理器配置项
    loaderOptions: {},
    // 启用 CSS modules for all css / pre-processor files.
    modules: false
  },
  parallel: require('os').cpus().length > 1, //是否为 Babel 或 TypeScript 使用 thread-loader
  // PWA 插件相关配置
  // see https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-pwa
  pwa: {},
  // webpack-dev-server 相关配置
  devServer: {

    open: process.platform === 'darwin',
    host: '0.0.0.0',
    port: 8888,
    https: false,
    hotOnly: false,
    // 设置代理
    proxy: {
      '/api': {
        target: 'http://localhost:3000/', //对应自己的接口
        changeOrigin: true,
        ws: true,
        pathRewrite: {

```

```
        '^/api': ''
    }
}
},
// 第三方插件配置
pluginOptions: {
    // ...
}
}
```

设置路由模式

router-view中的内容显示不出来

需要关闭路由的history模式，因为当前模式需要后台配合。

nginx服务器配置try_files重定向

```
location / {
    try_files $uri $uri/ /index.html;
}
```

打包优化

productionSourceMap是用来报错时定位到代码位置(就是.map文件
作用：项目打包后，代码都是经过压缩加密的，如果运行时报错，输出
的错误信息无法准确得知是哪里的代码报错。)配置为false取消配
置。可以减少构建时间，优化构建速度

```
productionSourceMap: process.env.NODE_ENV === 'development'
```

process.env.NODE_ENV是判断生产环境或开发环境。

打包优化

图片压缩

```
npm install --save babel-polyfill
```

配置vue.config.js

```
chainWebpack: config => {
    // ======压缩图片 start=====
    config.module
        .rule('images')
        .use('image-webpack-loader')
        .loader('image-webpack-loader')
        .options({ bypassOnDebug: true })
        .end()
    // ======压缩图片 end=====
},
```

自动忽略**console.log**语句

```
npm install terser-webpack-plugin --save
```

配置vue.config.js

```
const TerserPlugin = require('terser-webpack-plugin')

configureWebpack: (config) => {
    // =====去除console=====
    if(process.env.NODE_ENV === 'production'){
        config.optimization.minimizer[0].options.terserOptions.compress.drop_console = true
    }
    // =====去除console end=====
```