

# Vue课程学习

定义：构建用户界面的渐进式框架 基于数据动态渲染页面 循序渐进的学习

优点：大大提升开发效率

缺点：需要理解记忆规则->官网

vue2:<https://v2.cn.vuejs.org/>

Vue的两种使用方式

- 1: 核心包开发 局部模块改造
- 2: 核心包和插件工程化开发 整站开发

## Vue快速上手

创建Vue实例

- 1: 准备容器
- 2: 引包
- 3: 创建Vue实例 `new Vue()`
- 4: 指定配置-->渲染数据
  - `el`: 指定挂载点
  - `data`: 提供数据

html引入

- 1: 开发版本 `<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>`
- 2: 生产版本 `<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>`

## 基础代码实现

基础代码实现

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    {{msg}}
    <a href="#">{{count}}</a>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
```

```

<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      msg: 'Hello world! ',
      count: '666'
    }
  })
</script>
</html>

```

## 插值表达式

插值表达式{{}}

1. 作用: 利用表达式进行赋值，渲染到页面中

表达式: 是可以被求值的代码，JS会将其计算出一个结果

2: 语法{{}}

注意点:

- 1: 使用的数据必需存在
- 2: 支持的是表达式而非语句 比如if for
- 3: 不能在标签中使用{{}}插值

## 插值表达式基础代码

插值表达式基础代码

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <!--
    插值表达式{{}}
  1. 作用: 利用表达式进行赋值，渲染到页面中
  表达式: 是可以被求值的代码，JS会将其计算出一个结果
  2: 语法{{}}
  注意点:
  1: 使用的数据必需存在
  2: 支持的是表达式而非语句 比如if for
  3: 不能在标签中使用{{}}插值
  -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    {{nickname}}
    <!-- 转大写 -->
    {{nickname.toUpperCase()}}
    {{nickname + '你好'}}
    {{age >=18 ? '成年' : '未成年'}}
  </div>
</body>
</html>

```

```

        <p>我叫{{friend.name}}我的描述{{friend.desc}}</p>
    </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
    //一旦引入核心包，在全局环境就有了Vue构造函数
    const app=new Vue({
        //通过el选择器，指定管理的是哪个盒子
        el: '#app',
        //通过data提供数据
        data:{
            nickname: 'tony',
            age: 19,
            friend: {
                name: '小明',
                desc: '热爱学习Vue'
            }
        }
    })
</script>
</html>

```

## Vue核心特性：响应式

如何访问or修改？

- 1: 访问数据："实例.属性名字"
- 2: 修改数据："实例.属性名字"="值"

定义：数据改变，视图会自动更新

聚焦于数据-数据驱动视图

使用Vue开发，关注业务的核心逻辑

## Vue指令

### v-html

指令：带有v-前缀的特殊标签属性 eg: v-html

eg:

v-html

作用：设置元素的innerHTML

语法：v-html='语法式'

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Document</title>
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
    <!-- 准备容器 -->
    <div id="app">
        <!-- 编写用于渲染的代码逻辑 -->
    </div>
</body>
</html>

```

```

        <div v-html="msg"></div>
        <div v-html="heima"></div>
    </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
    //一旦引入核心包，在全局环境就有了Vue构造函数
    const app=new Vue({
        //通过el选择器，指定管理的是哪个盒子
        el: '#app',
        //通过data提供数据
        data:{
            msg: '<a href="www.baidu.com">百度</a>',
            heima: '<h1>222</h1>'
        }
    })
</script>
</html>

```

## V-show 和V-if

### v-show

- 1.作用：用来控制元素显示隐藏
- 2.语法：v-show="表达式" 表达式值为true显示，false隐藏
- 3.场景：频繁切换显示隐藏的

### v-if

- 1.作用:控制元素显示隐藏(条件渲染)
- 2.语法：v-if="表达式" 表达式值为true显示，false隐藏 本质来说:条件渲染\
- 3.场景：不频繁切换显示隐藏的

区别：

v-show底层原理：切换css的display: none 来控制隐藏

v-if 底层原理 ：根据判断条件 控制元素的创建和移除(条件渲染)

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Document</title>
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <style>
        .box{
            border: 1px solid black;
            width: 100px;
            height: 100px;
            margin-top: 10px;
            border-radius: 5px;
            text-align: center;
        }
    </style>
</head>
<body>
<!-- 准备容器 -->
<div id="app">
    <!-- 编写用于渲染的代码逻辑 -->

```

```

    <div v-show="flag" class="box">我是v-show控制的盒子</div>
    <div v-if="flag" class="box">我是v-if控制的盒子</div>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      flag: true
    }
  })
</script>
</html>

```

## v-else 和v-else-if

- 1.作用：辅助V-IF进行判断渲染
- 2.注意：紧挨着v-if

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    <p v-if="gender === 1">性别：男</p>
    <p v-else>性别：女</p>
    <hr>
    <p v-if="score >= 90">成绩评定A:奖励电脑一台</p>
    <p v-else-if="score >= 70">成绩评定B:奖励周末旅游</p>
    <p v-else-if="score >= 60">成绩评定C:奖励零食礼包</p>
    <p v-else>成绩评定D:奖励一周不能玩手机</p>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      gender: 1,

```

```

        score: 100
      }
    })
  </script>
</html>

```

## v-on

1. 作用：注册事件=添加监听+提供处理逻辑
2. 语法：
  - <1>v-on:事件名字="内联语句" @事件名字="内联语句"
  - <2>v-on:事件名字="methods中的函数名" @事件名字="methods中的函数名"
3. 简写：@事件名
4. v-on调用传参 eg: <button @click="fn(参数1, 参数2)">切换显示隐藏</button>

```

<1>v-on:事件名字="内联语句" @事件名字="内联语句"
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    <!-- 点击 -->
    <button v-on:click="count--">-</button>
    <span>{{count}}</span>
    <button v-on:click="count++">+</button>
    <!-- 滑入 -->
    <button @mouseenter="count--">-</button>
    <span>{{count}}</span>
    <button @mouseenter="count++">+</button>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      count: 10
    }
  })
</script>
</html>

```

```

<2>v-on:事件名字="methods中的函数名" @事件名字="methods中的函数名"
<!DOCTYPE html>

```

```

<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
<!-- 准备容器 -->
<div id="app">
  <!-- 编写用于渲染的代码逻辑 -->
  <button @click="fn">切换显示隐藏</button>
  <h1 v-show="isShow">Hello world</h1>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      isShow: true
    },
    methods:{
      fn(){
        //让提供的所有methods中的函数,this都指向当前实例
        this.isShow=!this.isShow
      }
    }
  })
</script>
</html>

```

4.v-on调用传参 eg: <button @click="fn(参数1, 参数2)">切换显示隐藏</button>

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
  <style>
    .box{
      border: 2px solid black;
      width: 200px;
      height: 200px;
      text-align: center;
      padding-top:10px;
    }
  </style>
</head>
<body>
<!-- 准备容器 -->
<div id="app">

```

```

<!-- 编写用于渲染的代码逻辑 -->
<div class="box">
  <h3>小黑售卖机</h3>
  <button @click="buy(5)">可乐5元</button>
  <button @click="buy(10)">咖啡10元</button>
</div>
<p>银行卡余额: {{money}}</p>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      money:100
    },
    methods:{
      buy(price){
        this.money-=price
      }
    }
  })
</script>
</html>

```

## v-bind

- 1.作用：动态的设置html的标签属性->src url title.....
- 2.语法：v-bind:属性名="表达式" 或者 :属性名字="表达式"

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new vue({
    //通过el选择器，指定管理的是哪个盒子

```



```

    el: '#app',
    //通过data提供数据
    data:{
      image: './img/2.jpg',
      msg: 'Hello 波仔'
    }
  })
</script>
</html>

```

## 图片切换案例-波仔的学习之旅

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
  <style>
    .img{
      width: 100px;
      height: 100px;
    }
  </style>
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <button v-show="index >0" @click="index--">上一页</button>
    <!-- 编写用于渲染的代码逻辑 -->
    <div>
      
    </div>
    <button v-show="index < list.length -1" @click="index++">下一页</button>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      //定义索引
      index: 0,
      //定义数组
      list:[
        './img/2.jpg',
        './img/3.jpg',
        './img/4.jpg',
      ]
    }
  })
</script>

```

```
</html>
```

## v-for

1. 作用：基于数据循环,多次渲染整个元素 ->数组,对象,数字....
2. 遍历数组语法: `v-for="(item,index) in 数组"` item每一项 index下标

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <!--
    插值表达式
  -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    <h3>小黑水果店</h3>
    <ul>
      <li v-for="(item,index) in list">{{item}}-{{index}}</li>
    </ul>
    <!-- 编写用于渲染的代码逻辑 -->
    <h3>小黑水果店</h3>
    <ul>
      <li v-for="item in list">{{item}}</li>
    </ul>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包,在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器,指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      list:['西瓜','苹果','鸭梨']
    }
  })
</script>
</html>
```

## 小黑的书架

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
```

```

    <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
<!-- 准备容器 -->
<!--
    插值表达式
-->
<div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    <h3>小黑的书架</h3>
    <ul>
        <li v-for="(item,index) in bookList" :key="item.id">
            <span> {{item.name}}</span>
            <span>{{item.author}}</span>
            <!--注册点击事件-》通过ID进行删除-->
            <button @click="del(item.id)">删除</button>
        </li>
    </ul>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
    //一旦引入核心包，在全局环境就有了Vue构造函数
    const app=new Vue({
        //通过el选择器，指定管理的是哪个盒子
        el: '#app',
        //通过data提供数据
        data:{
            bookList:[
                {id:1,name: '《红楼梦》',author: '曹雪芹'},
                {id:2,name: '《西游记》',author: '吴承恩'},
                {id:3,name: '《水浒传》',author: '施耐庵'},
                {id:4,name: '《三国演义》',author: '罗贯中'}
            ]
        },
        methods:{
            del(id){
                //filter 根据条件保留满足条件的对应项，在得到一个数组。(不会改变原数组)
                this.bookList=this.bookList.filter(item => item.id !==id)
            }
        }
    })
</script>
</html>

```

## v-for中的key

语法：key属性="唯一标识"

作用：给列表添加的唯一标识，便于Vue进行列表项的正确排序复用 ->一定要加key

# v-model

1. 作用：给表单元素使用，双向数据绑定->可以快速获取和设置表单元素的内容  
<1>数据变化->试图自动更新  
<2>视图变化->数据自动更新
2. 语法：v-model='变量'
3. v-model 可以让数据和视图，形成双向数据绑定  
(1)数据变化,视图自动更新  
(2)视图变化,数据自动更新  
可以快速获取或设置表单元素的内容

## 实例代码

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    账户: <input type="text" v-model="username" > <br><br>
    密码: <input type="password" v-model="password" > <br><br>
    <button @click="login">登录</button>
    <button @click="reset">注册</button>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      username: '',
      password: ''
    },
    methods:{
      login(){
        console.log(this.username,this.password)
      },
      reset(){
        this.username=''
        this.password=''
      }
    }
  })
</script>
</html>
```

## 项目案例-小黑记事本

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link rel="stylesheet" href="./css/index.css" />
  <title>记事本</title>
</head>
<body>
  <!-- 主体区域 -->
  <section id="app">
    <!-- 输入框 -->
    <header class="header">
      <h1>小黑记事本</h1>
      <input placeholder="请输入任务" v-model="toolName" class="new-todo" />
      <button class="add" @click="add">添加任务</button>
    </header>
    <!-- 列表区域 -->
    <section class="main">
      <ul class="todo-list">
        <!-- 遍历渲染的标签 -->
        <li class="todo" v-for="(item,index) in list" :key="item.id">
          <div class="view">
            <span class="index">{{index+1}}</span> <label>{{item.name}}
          </label>
            <button @click="del(item.id)" class="destroy" ></button>
          </div>
        </li>
      </ul>
    </section>
    <!-- 统计和清空 -->
    <footer class="footer" v-show="list.length >0">
      <!-- 统计 -->
      <span class="todo-count">合 计:<strong> {{list.length}} </strong></span>
      <!-- 清空 -->
      <button class="clear-completed" @click="clear" >
        清空任务
      </button>
    </footer>
  </section>
  <!-- 底部 -->
  <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        toolName: '',
        list:[
          {id:1,name: '跑步一公里'},
          {id:2,name: '跳绳200次'},
          {id:3,name: '游泳100m'},
        ]
      },
      methods:{
```

```

        del(id){
            this.list=this.list.filter(item => item.id !==id)
        },
        add(){
            if (this.toolName.trim() === ''){
                alert("请输入任务名称")
                return
            }
            this.list.unshift({
                id: +new Date(),
                name: this.toolName
            })
            this.toolName= ''
        },
        clear(){
            this.list=[]
        }
    }
})
</script>
</body>
</html>

```

## 指令修饰符

通过“.”致命一些指令后缀,不同后缀封装了不同的处理事件->简化代码

1: 按键修饰符

@keyup.enter->键盘回车监听

2: v-model 修饰符

v-model.trim ->去掉首尾空格

v-model.number ->转数字

3: 事件修饰符

@事件名.stop ->阻止冒泡

@事件名.prevent ->阻止默认行为

## v-bind对于样式控制的增强-操作class

语法: class="对象/数组"

1. 对象->键就是类名,值是布尔值,如果值为true, 有这个类,否则没有这个类

eg: :class="{类名1:布尔值,类名2:布尔值}"

适用场景: 一个类名,来回切换

2. 数组->数组中所有的类,都会添加到盒子上,本质就是一个class列表

eg: :class="{类名1,类名2}"

适用场景: 批量添加或删除类

## 京东秒杀tab高亮

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Document</title>
    <meta name="viewport" content="width=device-width,initial-scale=1.0">

```

```

<style>
  .active{
    background-color: red;
  }
</style>
</head>
<body>
<!-- 准备容器 -->
<div id="app">
  <!-- 编写用于渲染的代码逻辑 -->
  <ul>
    <li v-for="(item,index) in list" :key="item.id"
@click="activeIndex=index">
      <a :class="{active: activeIndex===index}">
        {{item.name}}
      </a>
    </li>
  </ul>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      activeIndex:0,
      list:[
        {id: 1,name:'京东秒杀'},
        {id: 2,name:'每日特价'},
        {id: 3,name:'品类秒杀'}
      ]
    },
    methods:{

    }
  })
</script>
</html>

```

## v-bind对于样式控制的增强-操作style

语法: :style="样式对象"

eg: :style="{css属性名1:css属性值1,css属性名2:css属性值2}"

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">

```

```

<style>
  .box{
    width: 200px;
    height: 200px;
    background-color: pink;
  }
</style>
</head>
<body>
<!-- 准备容器 -->
<div id="app">
  <!-- 编写用于渲染的代码逻辑 -->
  <div class="box" :style="{width: '400px',height:'400px',backgroundColor:
'green'}">黑马程序员</div>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{

    },
    methods:{

    }

  })
</script>
</html>

```

## v-model应用于其他的表单元素

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>v-model应用于其他的表单元素</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
<!-- 准备容器 -->
<div id="app">
  <!-- 编写用于渲染的代码逻辑 -->
  <h3>小黑学习网</h3>
  姓名:
  <input type="text" v-model="username"><br><br>
  是否单身:
  <input type="checkbox" v-model="isSingle"><br><br><br>
  性别:
  <input v-model="gender" type="radio" name="gender" value="1">男
  <input v-model="gender" type="radio" name="gender" value="2">女
  <br><br><br>

```



```

    所在城市：
    <select v-model="cityId">
      <option value="bj">北京</option>
      <option value="sh">上海</option>
      <option value="cd">成都</option>
      <option value="nj">南京</option>
    </select>
    <br><br><br>
    自我描述：
    <textarea v-model="desc"></textarea>
    <br><br><br>
    <button>立即注册</button>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      username: '',
      issingle: true,
      gender: "1",
      cityId: '102',
      desc: ""
    }
  })
</script>
</html>

```

## 计算属性

概念：基于现有的数据，计算出来的新属性。依赖的数据变化，自动重新计算

语法：

1. 声明在`computer`配置项中，一个计算属性对应一个函数
2. 使用起来和普遍属性一样使用`{{ 计算属性名 }}`

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>计算属性</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
  <!-- 准备容器 -->
  <div id="app">
    <!-- 编写用于渲染的代码逻辑 -->
    <h1>小黑的礼物清单</h1>
    <table border=" true">
      <tr>
        <th>名字</th>

```

```

        <th>数量</th>
      </tr>
      <tr v-for="(item,index) in list":key="item.id">
        <td>{{item.name}}</td>
        <td>{{item.num}}</td>
      </tr>
    </table>
    <p>礼物总数: {{totalCount}}个</p>
  </div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包, 在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器, 指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      list:[
        {id:1,name:'篮球',num:1},
        {id:2,name:'玩具',num:2},
        {id:3,name:'铅笔',num:3}
      ]
    },
    computed:{
      totalCount(){
        //基于现有的数据去编写求值逻辑
        //计算属性函数内部,可以直接通过this 访问到app实例
        //需求:对this.list 数组里面的num进行求和 ->reduce
        let total=this.list.reduce((sum,item) => sum +item.num ,0)
        return total
      }
    }
  })
</script>
</html>

```

## computed计算属性VSmethods方法

### computed计算属性

作用:封装了一段对于数据的处理,求得一个结果

语法:

- 1: 写在computed配置项中
- 2: 作为属性,直接使用->this.计算属性{{计算属性}}

### methods方法

作用: 给实例提供了一个方法, 调用以处理业务逻辑

语法:

- 1: 写在methods配置项中
- 2作为方法,需要调用->this.方法名() {{方法名}} @事件名="方法名"、

缓存特性(提升性能)

计算属性会对计算出来的结果缓存, 再次使用直接读取缓存, 依赖项变化了, 会自动重新计算->并再次缓存

# 计算属性完整写法

计算属性默认的简写，只能读取访问，不能修改。  
如果要修改->需要些计算属性的完整写法

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>改名</title>
  <meta name="viewport" content="width=device-width,initial-scale=1.0">
</head>
<body>
<!-- 准备容器 -->
<div id="app">
  <!-- 编写用于渲染的代码逻辑 -->
  姓: <input type="text" v-model="firstName">+
  名: <input type="text" v-model="lastName">=
  <span>{{fullName}}</span><br><br>
  <button @click="changName">改名卡</button>
</div>

</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script>
  //一旦引入核心包，在全局环境就有了Vue构造函数
  const app=new Vue({
    //通过el选择器，指定管理的是哪个盒子
    el: '#app',
    //通过data提供数据
    data:{
      firstName: '刘',
      lastName: '备',
    },
    methods:{
      changName (){
        this.fullName='牛瑞祥';
      }
    },
    computed:{
      /*fullName(){
        return this.firstName+this.lastName
      }*/
      //完整写法->获取+设置
      fullName:{
        get(){
          return this.firstName+this.lastName
        },
        //当fullName计算属性被修改赋值时，执行set
        //修改的值，传递给set形参
        set(value){
          this.firstName=value.slice(0,1)
          this.lastName=value.slice(1)
        }
      }
    }
  })
}
```

```
    }  
  })  
</script>  
</html>
```

## watch侦听器（监视器）

作用：监视数据变化，执行一些业务逻辑或异步操作。

语法:

### 1简单写法->简单类型数据,直接监视

## 2完整写法->添加额外配置项

(1) `deep: true` 对复杂类型深度监视

(2) `immediate: true` 初始化立刻执行一次 `handler` 方法

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
      font-size: 18px;
    }
    #app {
      padding: 10px 20px;
    }
    .query {
      margin: 10px 0;
    }
    .box {
      display: flex;
    }
    textarea {
      width: 300px;
      height: 160px;
      font-size: 18px;
      border: 1px solid #dedede;
      outline: none;
      resize: none;
      padding: 10px;
    }
    textarea:hover {
      border: 1px solid #1589f5;
    }
    .transbox {
      width: 300px;
      height: 160px;
      background-color: #f0f0f0;
      padding: 10px;
      border: none;
    }
  </style>
</html>
```

```

    .tip-box {
      width: 300px;
      height: 25px;
      line-height: 25px;
      display: flex;
    }
    .tip-box span {
      flex: 1;
      text-align: center;
    }
    .query span {
      font-size: 18px;
    }

    .input-wrap {
      position: relative;
    }
    .input-wrap span {
      position: absolute;
      right: 15px;
      bottom: 15px;
      font-size: 12px;
    }
    .input-wrap i {
      font-size: 20px;
      font-style: normal;
    }
  </style>
</head>
<body>
<div id="app">
  <!-- 条件选择框 -->
  <div class="query">
    <span>翻译成的语言: </span>
    <select v-model="obj.lang">
      <option value="italy">意大利</option>
      <option value="english">英语</option>
      <option value="german">德语</option>
    </select>
  </div>

  <!-- 翻译框 -->
  <div class="box">
    <div class="input-wrap">
      <textarea v-model="obj.words"></textarea>
      <span><i>📄</i>文档翻译</span>
    </div>
    <div class="output-wrap">
      <div class="transbox">{{result}}</div>
    </div>
  </div>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script>
  // 接口地址: https://applet-base-api-t.itheima.net/api/translate
  // 请求方式: get
  // 请求参数:

```

```

// (1) words: 需要被翻译的文本（必传）
// (2) lang: 需要被翻译成的语言（可选）默认值-意大利
// -----

const app = new Vue({
  el: '#app',
  data: {
    words: '',
    obj: {
      words: '',
      lang: 'italy'
    },
    result: '',
    timer: null
  },
  // 具体讲解: (1) watch语法 (2) 具体业务实现
  watch: {
    obj: {
      deep: true, // 深度监视
      immediate: true, // 立刻执行，一进入页面handler就立刻执行一次
      handler (newValue) {
        clearTimeout(this.timer)
        this.timer = setTimeout(async () => {
          const res = await axios({
            url: 'https://applet-base-api-t.itheima.net/api/translate',
            params: newValue
          })
          this.result = res.data.data
          console.log(res.data.data)
        }, 300)
      }
    }
    // 'obj.words'(newValue){
    //   clearTimeout(this.timer)
    //   this.timer=setTimeout(async ()=>{
    //     const res= await axios({
    //       url: 'https://applet-base-api-t.itheima.net/api/translate',
    //       params:{
    //         words: newValue
    //       }
    //     })
    //     this.result=res.data.data
    //   },300)
    // }
  }
})
</script>
</body>
</html>

```

# 购物车

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="./css/inputnumber.css" />
    <link rel="stylesheet" href="./css/index.css" />
    <title>购物车</title>
  </head>
  <body>
    <div class="app-container" id="app">
      <!-- 顶部banner -->
      <div class="banner-box"></div>
      <!-- 面包屑 -->
      <div class="breadcrumb">
        <span><img alt="home icon" /></span>
        /
        <span>购物车</span>
      </div>
      <!-- 购物车主体 -->
      <div class="main" v-if="fruitList.length > 0">
        <div class="table">
          <!-- 头部 -->
          <div class="thead">
            <div class="tr">
              <div class="th">选中</div>
              <div class="th th-pic">图片</div>
              <div class="th">单价</div>
              <div class="th num-th">个数</div>
              <div class="th">小计</div>
              <div class="th">操作</div>
            </div>
          </div>
          <!-- 身体 -->
          <div class="tbody">
            <div class="tr" :class="{active: item.isChecked}" v-for="
(item,index) in fruitList" :key="item.id">
              <div class="td"><input type="checkbox" v-model="item.isChecked" />
            </div>
            <div class="td"></div>
            <div class="td">{{item.price}}</div>
            <div class="td">
              <div class="my-input-number">
                <button class="decrease" @click="sub(item.id)"
:disabled="item.num<=1"> - </button>
                <span class="my-input__inner">{{item.num}}</span>
                <button class="increase" @click="add(item.id)"> + </button>
              </div>
            </div>
            <div class="td">{{item.num * item.price}}</div>
            <div class="td"><button @click="del(item.id)">删除</button></div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

[illegible]



```

    },
  ],
  const app = new Vue({
    el: '#app',
    data: {
      // 水果列表
      fruitList: JSON.parse(localStorage.getItem('fruitList')),
    },
    methods: {
      del(id) {
        this.fruitList = this.fruitList.filter(item => item.id !== id)
      },
      add(id) {
        const frult = this.fruitList.find(item => item.id === id)
        frult.num++
      },
      sub(id) {
        const frult = this.fruitList.find(item => item.id === id)
        frult.num--
      }
    },
    computed: {
      isAll: {
        get() {
          return this.fruitList.every(item => item.isChecked)
        },
        set(value) {
          this.fruitList.forEach(item => item.isChecked = value)
        }
      },
      totalCount() {
        return this.fruitList.reduce((sum, item) => {
          if (item.isChecked) {
            return sum + item.num
          } else {
            return sum
          }
        }, 0);
      },
      totalPrice() {
        return this.fruitList.reduce((sum, item) => {
          if (item.isChecked) {
            return sum + item.price * item.num
          } else {
            return sum
          }
        }, 0)
      }
    },
    watch: {
      fruitList: {
        deep: true,
        handler(newValue) {
          localStorage.setItem('fruitList', JSON.stringify(newValue))
        }
      }
    }
  })
})

```

```
</script>
</body>
</html>
```

## Vue工程化

### 生命周期和四个阶段

**Vue周期:** 一个Vue实例从创建到销毁的整个过程

生命周期四个阶段: 创建, 挂载, 更新, 销毁

- 1: 创建阶段 响应式数据
- 2: 挂载阶段 渲染模板
- 3: 更新阶段 数据修改, 更新视图
- 4: 销毁阶段 销毁实例

### Vue生命周期函数-钩子函数

Vue生命周期过程中, 会自动运行一些函数, 被称为【生命周期钩子】->让开发者可以在【特定阶段】运行自己的代码

```
beforeCreate
Create
beforeMount
mounted
beforeUpdate
updated
beforeDestroy
destroyed
```

### 简单代码演示

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    * {
      margin: 0;
      padding: 0;
      list-style: none;
    }
    .news {
      display: flex;
      height: 120px;
      width: 600px;
      margin: 0 auto;
      padding: 20px 0;
      cursor: pointer;
    }
    .news .left {
      flex: 1;
      display: flex;
      flex-direction: column;
```

```

        justify-content: space-between;
        padding-right: 10px;
    }
    .news .left .title {
        font-size: 20px;
    }
    .news .left .info {
        color: #999999;
    }
    .news .left .info span {
        margin-right: 20px;
    }
    .news .right {
        width: 160px;
        height: 120px;
    }
    .news .right img {
        width: 100%;
        height: 100%;
        object-fit: cover;
    }
</style>
</head>
<body>
<!-- 准备容器 -->

<div id="app">
    <ul>
        <li class="news" v-for="(item,index) in list" :key="item.id">
            <div class="left">
                <div class="title">{{item.title}}</div>
                <div class="info">
                    <span>{{item.source}}</span>
                    <span>{{item.time}}</span>
                </div>
            </div>
            <div class="right">
                
            </div>
        </li>
    </ul>
</div>
</body>
<!-- 引包 -->
<script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script>
    //一旦引入核心包，在全局环境就有了Vue构造函数
    // 接口地址: http://hmajax.itheima.net/api/news
    // 请求方式: get
    const app=new Vue({
        //通过el选择器，指定管理的是哪个盒子
        el: '#app',
        //通过data提供数据
        data:{
            list:[]
        },
        async created(){

```





```

    <!-- 右侧图表 -->
    <div class="echarts-box" id="main"></div>
  </div>
</div>
<script src="./api/echarts.min.js"></script>
<script src="./api/vue.js"></script>
<script src="./api/axios.js"></script>
<script>
  /**
   * 接口文档地址:
   * https://www.apifox.cn/apidoc/shared-24459455-ebb1-4fdc-8df8-
0aff8dc317a8/api-53371058
   *
   * 功能需求:
   * 1. 基本渲染
   * 2. 添加功能
   * 3. 删除功能
   * 4. 饼图渲染
   */
  const app = new Vue({
    el: '#app',
    data: {
      list: [],
      name: '',
      price: ''
    },
    methods: {
      async getList() {
        const res = await axios.get('https://applet-base-api-
t.itheima.net/bill', {
          params: {
            creator: 'Coder-Su'
          }
        })
        this.list = res.data.data
        //更新图标
        this.myChart.setOption({
          series: [
            {
              data: this.list.map(item=>({value: item.price
, name: item.name}))
            }
          ]
        })
      },
      async add() {
        if (!this.name) {
          alert("没有输入")
          return
        }
        if (typeof this.price !== 'number') {
          alert("money格式不对")
          return
        }
        const res = await axios.post('https://applet-base-api-
t.itheima.net/bill', {
          creator: 'Coder-Su',

```

```

        name: this.name,
        price: this.price
      })
      this.getList()
      this.name=''
      this.price=''
    },
    async del(id) {
      const res = await axios.delete(`https://applet-base-api-
t.itheima.net/bill/${id}`)
      this.getList()
      if (res.status===200){
        alert("删除成功")
      }
    }
  },
  computed:{
    totalprice(){
      return this.list.reduce((sum,item)=>sum+item.price,0)
    }
  },
  created() {
    this.getList()
  },
  mounted(){
    this.myChart=echarts.init(document.getElementById('main'))
    this.myChart.setOption({
      title:{
        text: '消费账单列表',
        left: 'center'
      },
      tooltip:{
        trigger: 'item'
      },
      legend:{
        orient: 'vertical',
        left: 'left'
      },
      series:[
        {
          name:'消费账单',
          type: 'pie',
          radius: '50%',
          data:[

          ],
          emphasis:{
            itemStyle:{
              shadowBlur:10,
              shadowOffsetX: 0,
              shadowColor:'rgba(0, 0, 0, 0.5)',
            }
          }
        }
      ]
    })
  }
})

```

```
</script>
</body>
</html>
```

## 工程化开发&脚手架Vue CLI

开发Vue的两种方式

- 1: 核心包传统开发模式: 基于html/css/js文件, 直接引入核心包
- 2: 工程化开发模式: 基于构建工具(webpack)的环境中开发vue

### Vue CLI

基本介绍:

Vue CLI是vue官方提供的一个全局命令工具, 可以帮助我们创建一个vue项目的标准化基础架子

好处:

- 1: 开箱即用
- 2: 内置babel
- 3: 标准化

选中:

Babel  
Router  
Vuex  
CSS pre-processors  
Linter/Formatter  
Less  
ESLint+Standard config

## 文件的作用

node\_modules 第三方文件夹  
public 放html文件的地方  
src 源代码目录->以后写代码的文件夹  
assets 静态资源目录->存放图片, 字体  
components 组件目录->存放通用组件  
App.vue App跟组件  
main.js 入口文件  
babel.config.js babel配置文件  
jsconfig.json js配置文件  
package.json 项目配置文件->包含项目名, 版本号之类的  
README.md 项目说明文档  
vue.config.js vue-cli配置文件  
yarn.lock yarn锁文件

## 组件化开发&根组件

- 1: 组件化: 一个页面可以拆分成一个个组件, 每个组件有着自己独特的结构, 样式, 行为  
好处: 便于维护, 利于复用->提升开发效率
  - 2: 根组件: 整个应用最上层的组件, 包裹所有普通的小组件
- App.vue 文件的三个组成部分 结构, 行为, 样式
- template 结构 (有且只能一个根元素)
- script js逻辑
- style 样式 (可支持less, 需要装包)



## 普通组件的注册使用

两种方式

1: 局部注册: 只能在注册的组件内使用

创建 .vue 文件 (三个组成成分)

在使用的组件内导入并注册

```
<template>
  <div class="App">
    <HmHeader></HmHeader>
    <HmMain></HmMain>
    <HmFooter></HmFooter>
  </div>
</template>
<script>
import HmHeader from "../components/HmHeader.vue";
import HmMain from "@components/HmMain.vue";
import HmFooter from "@components/HmFooter.vue";
export default {
  components: {
    'HmHeader': HmHeader,
    'HmMain': HmMain,
    'HmFooter': HmFooter
  },
}
</script>

<style>
.App{
  width: 600px;
  height: 600px;
  background-color: #87ceab;
  margin: 0 auto;
  padding: 20px;
}
</style>
```

## 全局组件的注册使用

所有组件内都能使用

1: 创建 .vue 文件

2: main.js 中进行全局注册

2: 全局注册: 所有组件内都能使用

使用:

当成 html 标签使用 <组件></组件>

尽量采用大驼峰

main.js

// 编写导入的代码, 在代码顶部去编写

```
import HmButton from "@components/HmButton.vue";
```

// 3. 注册全局组件

```
Vue.component('HmButton', HmButton)
```

## 组件的三大组成部分(结构/样式/逻辑)

组件的样式冲突

默认情况：写在组件中的样式会全局生效->因此很容易造成多个组件之间的样式冲突问题

1. 全局样式：默认组件中的样式会作用到全局
2. 局部样式：可以给组件加上**scoped**属性，可以让样式只作用与当前组件

**scoped**原理：

1. 给当前组件模板的所有元素，都会被添加上一个自定义属性**data-v-hash**值  
利用哈希值可以区分开不同的组件
  2. **css**选择器后面，被自动处理，添加上了属性选择器
- 最终效果：必须是当前组件的元素，才会有这个自定义属性，才会被这个样式作用到

## data是一个函数

一个组件的**data**选项必须是一个函数->保证每个组件实例，维护独立的一份数据对象  
每次创建的组件实例，都会新执行一次**data**函数，得到一个新对象

```
<template>
  <div>
    BaseOne
    {{msg}}
  </div>
</template>
<script>
export default {
  data() {
    return {
      msg: 'Hello vue'
    }
  }
}
</script>
<style scoped>
/*
1. 默认的style样式，会作用与全局->全局样式
2. scoped样式，只作用与当前组件->局部样式
推荐加上scoped
*/
div{
  border: 3px solid blue;
  margin: 30px;
}
</style>
```

## 什么是组件通信

组件通信，就是指 组件与组件 之间的数据传递

组件的数据是独立的，无法直接访问其他组件的数据

想用其他组件的数据->组件通信

组件A->自己的数据  
|| 组件通信方案  
组件B->自己的数据

不同的组件关系 和 组件通信方案分类

组件关系分类：

1. 父子关系
2. 非父子关系

组件通信解决方案:

父子关系

||

props和 \$emit

非父子关系

||

1.provide \$inject

2.eventbus

父传子

父组件通过props将数据传递给子组件

子传父

子组件利用\$emit通知父组件,进行修改更新

父亲:

```
<template>
  <div>
    我是APP组件
    <BaseOne :title="myTitle" //子传父@changeTitle="changeTitle"></BaseOne>
  </div>
</template>
<script>
import BaseOne from "@/components/BaseOne.vue";
export default {
  components: {
    BaseOne
  },
  data() {
    return {
      myTitle: '学前端来Su'
    }
  },
  methods: {
    changeTitle(newTitle) {
      this.myTitle = newTitle;
    }
  }
}
```

</script>

<style>

</style>

儿子:

```
<template>
  <div>
    {{ title }}
    <button @click="changeTitle">修改title</button>
  </div>
</template>
<script>
export default {
  //父传子
  props: ['title'],
  //子传父
  methods : {
    changeTitle(){
      this.$emit('changeTitle','Su教育')
    }
  }
}
```

```

}
</script>
<style scoped>
/*
  1. 默认的style样式，会作用与全局->全局样式
  2. scoped样式，只作用与当前组件->局部样式
  推荐加上scoped
*/
div{
  border: 3px solid pink;
  width: 300px;
  height: 300px;
}
</style>

```

## props详解

什么是prop

**Prop**定义: 组件上注册的一些自定义属性

**Prop**作用: 向子组件传递数据

数组:

:hobby='hobby'

特点:

可以传递任意数量的prop

可以传递任意类型的prop

props校验

思考: 组件的prop可以乱传吗? 当然不可以

作用: 为组件的prop指定验证要求, 不符合要求, 控制台就会有错误提示->帮助开发者, 快速发现错误

语法:

1: 类型校验

props: { 校验的属性名: 类型(Number, string, boolean, Array, function) } eg: props:

{w: Number}

2: 非空校验

3: 默认值

4: 自定义校验

```

props: {
  title: {
    type: String,
    required: true, // 非空
    default: '默认值',
    validator(value) {
      return 是否通过验证
    }
  }
},

```

prop & data 单向数据流

共同点: 都可以给组件提供数据.

区别:

data的数据是自己的->顺便改

prop的数据是外部的->不能直接改, 要遵循 单向数据流

// 自己的数据顺便改, 外部的数据不能顺便改

便改

口诀: 谁的数据谁负责

## 非父子通信(扩展)-event bus事件总线

作用：非父子组件之间，进行简易消息传递。(复杂场景->Vuex)

1. 创建一个都能访问到的事件总线->utils/EventBus.js
2. A组件，监听Bus实例的事件
3. B组件，触发Bus实例的事件

```
<template>
  <div>
    我是A组件(接收方)
    <p>{{msg}}</p>
  </div>
</template>
<script>
import Bus from "@/utils/EventBus";
export default {
  data () {
    return {
      msg : ''
    }
  },
  created() {
    //2. 在接收方进行监听
    Bus.$on("sendMsg", (msg) => {
      console.log(msg)
      this.msg=msg
    })
  }
}
</script>
<style scoped>
/*
1. 默认的style样式，会作用与全局->全局样式
2. scoped样式，只作用与当前组件->局部样式
推荐加上scoped
*/
div{
  border: 3px solid pink;
  width: 300px;
  height: 300px;
}
</style>
```

```
<template>
  <div>
    我是B组件(发布方)
    <button @click="clickSend">发布通知</button>
  </div>
</template>
<script>
import Bus from "@/utils/EventBus";
export default {
  methods: {
```

```

        clickSend() {
            //1. 在发布方进行发送
            Bus.$emit("sendMsg", "我是B组件发布的消息");
        }
    }
}
</script>

<style scoped>
div{
    border: #00BE9A solid 3px;
    height: 300px;
    width: 300px;
}
</style>

```

## 非父子通信(扩展)-provide & inject

provide & inject作用:跨层级共享数据

1. 父组件provide提供数据
2. 子/孙组件inject取值使用

```

<template>
  <div>
    <BaseOne></BaseOne>
    <BaseTwo></BaseTwo>
    <BaseThree></BaseThree>
  </div>
</template>

<script>
import BaseOne from "@/components/BaseOne.vue";
import BaseTwo from "@/components/BaseTwo.vue";
import BaseThree from "@/components/BaseThree.vue";
export default {
  components: {
    BaseThree,
    BaseOne,
    BaseTwo
  },
  provide(){
    return {
      color: this.color,
      userInfo: this.userInfo
    }
  },
  data(){
    return {
      color: 'pink', //简单类型 非响应
      userInfo: { //复杂类型 响应式的
        name: '张三',
        age: 18
      }
    }
  }
}

```

```
</script>
```

```
<style>
```

```
</style>
```

```
<template>
```

```
  <div>
```

```
    我是A组件(接收方)
```

```
    <p>{{msg}}</p>
```

```
    <p>{{color}}</p>
```

```
    <p>{{userInfo.name}}</p>
```

```
    <p>{{userInfo.age}}</p>
```

```
  </div>
```

```
</template>
```

```
<script>
```

```
import Bus from "@utils/EventBus";
```

```
export default {
```

```
  inject: ['color', 'userInfo'],
```

```
  data () {
```

```
    return {
```

```
      msg : ''
```

```
    }
```

```
  },
```

```
  created() {
```

```
    //2.在接收方进行监听
```

```
    Bus.$on("sendMsg", (msg) => {
```

```
      console.log(msg)
```

```
      this.msg=msg
```

```
    })
```

```
  }
```

```
}
```

```
</script>
```

```
<style scoped>
```

```
/*
```

```
  1. 默认的style样式，会作用与全局->全局样式
```

```
  2. scoped样式，只作用与当前组件->局部样式
```

```
  推荐加上scoped
```

```
*/
```

```
div{
```

```
  border: 3px solid pink;
```

```
  width: 300px;
```

```
  height: 300px;
```

```
}
```

```
</style>
```

## V-model详解

原理:本质上就是一个语法糖。列如应用在输入框上,就是value属性和input事件的合写

作用:提供数据的双向绑定

1: 数据变,视图跟着变:value

2: 视图变,数据跟着变@input

注意:\$event用于在模板中,获取事件的相残

## 表单类组件的封装 & v-model简化代码

### 1. 表单类组件 封装

- a. 父传子: 数据 应该是父组件props传递过来, v-model拆解绑定数据
- b. 子传父: 监听输入, 子传父值给父组件修改

### 2. 父组件v-model简化代码, 实现子组件 和父组件数据的双向绑定

- a. 子组件中: props通过value接受, 事件触发input
- b. 父组件中: v-model给组件直接绑数据(:value+@input)

```
<template>
  <div>
    <BaseSelect v-model="selectId"></BaseSelect>
  </div>
</template>
<script>
import BaseSelect from "@components/BaseSelect.vue";
export default {
  components: {
    BaseSelect
  },
  data(){
    return{
      selectId : '101'
    }
  }
}
</script>
<style>
</style>
```

```
<template>
  <div>
    <select :value="value" @change="handleChange">
      <option value="101">北京</option>
      <option value="102">上海</option>
      <option value="103">武汉</option>
      <option value="104">深圳</option>
      <option value="105">广州</option>
    </select>
  </div>
</template>
<script>
export default {
  props:{
    value: String
  },
  methods:{
    handleChange(e){
      this.$emit('input', e.target.value)
    }
  }
}
</script>
<style>
</style>
```



## .sync修饰符

作用:可以实现子组件与父组件数据的双向绑定,简化代码

特点:**prop**属性名,可以自定义,非固定为**value**

场景:封装弹框类的基础组件,**visible**属性 **true**显示**false**隐藏

本质: 就是:**属性名**和**@update:属性名** 合写

```
<template>
  <div>
    <button @click="isShow = true">退出按钮</button>
    <BaseSelect :visible.sync="isShow"></BaseSelect>
  </div>
</template>

<script>
import BaseSelect from "@/components/BaseSelect.vue";
export default {
  components: {
    BaseSelect
  },
  data(){
    return{
      isShow:false
    }
  }
}
</script>
<style>
</style>
```

```
<template>
  <div v-show="visible">
    <select >
      <option value="101">北京</option>
      <option value="102">上海</option>
      <option value="103">武汉</option>
      <option value="104">深圳</option>
      <option value="105">广州</option>
    </select>
    <button @click="close">x</button>
  </div>
</template>
<script>
export default {
  props:{
    visible: Boolean
  },
  methods:{
    handleChange(e){
      this.$emit('input',e.target.value)
    },
    close(){
      this.$emit('update:visible',false)
    }
  }
}
```

```

    }
  </script>
  <style>
  </style>

```

## ref和\$refs

作用: 利用ref和\$refs可以用于获取Dom元素或组件实例

特点: 寻找范围->当前组件内(更精准稳定)

### a. 获取dom:

1. 目标标签->添加ref属性
2. 恰当时机, 通过this.\$refs.xxx获取目标标签

### b. 获取组件实例

1. 目标组件-添加Ref属性
2. 恰当时机, 通过this.\$refs.xxx获取目前组件就可以调用组件对象里面的方法

```

<template>
  <div>
    <BaseSelect ref="BaseForm"></BaseSelect>
    <button @click="handleGet">获取数据</button>
    <button @click="handleSet">充值数据</button>
  </div>
</template>

<script>
import BaseSelect from "@/components/BaseSelect.vue";
export default {
  components: {
    BaseSelect
  },
  data(){
    return{
      isShow:false
    }
  },
  methods:{
    handleGet(){
      alert('账号为'+this.$refs.BaseForm.getValue().account+
        '密码为'+this.$refs.BaseForm.getValue().password
      )
    },
    handleSet(){
      this.$refs.BaseForm.resetValues()
    }
  }
}
</script>
<style>
</style>

```

```

<template>
  <div>
    <form action="" >
      账号:<input type="text" v-model="account">
      密码:<input type="text" v-model="password">
    </form>
  </div>
</template>

```

```

    </form>
  </div>
</template>
<script>
export default {
  data(){
    return{
      account: '',
      password: ''
    }
  },
  methods: {
    //方法1:收集表单数据,返回数据
    getValue() {
      return{
        account: this.account,
        password: this.password
      }
    },
    //方法2:重置表单
    resetValues (){
      this.account=''
      this.password=''
    }
  }
}
</script>
<style>
</style>

```

## Vue异步更新.\$nextTick

需求:编辑标题i,编辑框自动聚焦

1. 点击编辑,立刻获得焦点
2. 让编辑框,立刻获得焦点

`this.isShowEdit = true` //显示输入框

`this.$refs.inp.focus()` //获取焦点

问题:"显示之后",立刻获得焦点是不能成功的!

原因:Vue是异步更新DOM(提升性能)

`$nextTick`:等Dom更新后,才会触发执行此方法里的函数体

语法:`this.$nextTick(函数体)`

```

<template>
  <div>
    <div v-if="isShow">
      <input ref="input" v-model="editValue" type="text">
      <button @click="alertEdit">确认</button>
    </div>
    <div v-else>
      <span>{{title}}</span>
      <button @click="handleEdit">编辑</button>
    </div>
  </div>
</template>
<script>
export default {

```

```

data(){
  return{
    title: '大标题',
    editValue: '',
    isShow:false
  }
},
methods:{
  handleEdit(){
    //Vue异步Dom更新
    this.isShow = true
    this.$nextTick(()=>{
      this.$refs.input.focus()
    })
  },
  alertEdit(){
    alert('修改成功')
  }
}
}
</script>
<style>
</style>

```

## 自定义指令

自定义指令：自己定义的指令，可以封装一些dom操作，扩展额外功能

需求：当页面加载时，让元素将获得焦点（autofocus在safari里浏览器有兼容性）

操作Dom：dom元素.focus()

全局注册-语法 main.js

```

Vue.directive('指令名',{
  inserted(el){
    //可以对el标签扩展额外功能
    el.focus()
  }
})

```

局部注册-语法

```

directives:{
  "指令名":{
    inserted(){
      //可以对el标签,扩展额外功能
      el.focus()
    }
  }
}

```

### 全局注册

```

Vue.directive('focus', {
  //inserted 会在指令所在的元素被操作到页面中触发
  inserted(el) {
    el.focus()
  }
})

```

### 局部注册

```

directives: {
  Twofocus: {
    inserted(el) {
      el.focus()
    }
  }
}

```

## 自定义指令-指令的值

需求:实现一个Color指令-传入不同的颜色,给标签设置文字颜色

- 语法:再绑定指令时,可以通过"等号"的形式为指令 绑定具体的参数值  
v-color="color" , 通过等号可以绑定指令的值
- 通过binding.value可以拿到指令值,指令值修改会触发update函数
- 通过update钩子,可以监听指令值的变化,进行dom更新操作

```

<template>
  <div>
    <h1 v-color = "color1">指令的值</h1>
    <h1 v-color = "color2">指令的值</h1>
  </div>
</template>

```

```

<script>
export default {
  data(){
    return{
      color1: 'red',
      color2: 'blue'
    }
  },
  directives: {
    color: {
      inserted(el,d){
        el.style.color=d.value
      },
      update(el,d){
        alert("值修改了")
        el.style.color=d.value
      }
    }
  }
}
</script>
<style>
</style>

```

## 自定义指令-v-loading指令封装

场景：实际开发过程中，发送请求需要事件，在请求的数据未回来时，页面会处于空白状态=>用户体验不好  
需求：封装一个v-loading指令，实现加载中的效果

核心思路：

- (1)准备类名,通过伪元素提供遮罩层
- (2)添加或移除类名,实现loading蒙层的添加移除
- (3)利用指令语法,封装v-loading通用指令  
inserted钩子中2,binding,value判断指令

```
<template>
  <div class="box " v-loading="isLoading" >
    <ul>
      <li v-for="item in list" :key="item.id">
        {{item.title}}
        {{item.time}}
        
      </li>
    </ul>
  </div>
</template>
<script>
import axios from "axios";
export default {
  data(){
    return{
      list:[],
      isLoading:true
    }
  },
  async created() {
    const res = await axios.get('http://hmajax.itheima.net/api/news')
    setTimeout(() => {
      this.list=res.data.data
      this.isLoading=false
    },5000)
  },
  directives:{
    loading :{
      inserted(e1,b){
        b.value?e1.classList.add('loading'):e1.classList.remove('loading')
      },
      update(e1,b){
        b.value>0?e1.classList.add('loading'):e1.classList.remove('loading')
      }
    }
  }
}
</script>
<style>
.loading:before{
  content: '加载中';
  position: absolute;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  background: #fff url("assets/logo.png") no-repeat center;
```

```

}
.box{
  width: 800px;
  min-height: 500px;
  border: 3px solid orange;
  border-radius: 5px;
  position: relative;
}
</style>

```

## 插槽-默认插槽

作用: 让组件内的一些结构支持自定义

需求: 要在页面中显示一个对话框, 封装成一个组件

插槽的基本语法:

1. 组件内需要定制的部分, 改用<slot></slot>占位
2. 使用组件时, <MyDialog></MyDialog>标签内部, 传入结构替换slot

使用步骤

1. 现在组件内用slot占位
2. 使用组件时, 传入具体标签内容插入

```

<template>
  <div>
    <h3>友情提示</h3>

    <h4>
      <slot></slot>
    </h4>
    <button>确认</button>
    <button>取消</button>
  </div>

</template>

<script>
export default {

}
</script>

<style>
body{
  background-color: #4d4d4d;
}
</style>

```

```

<template>
<div>
<!--2. 在使用组件时, 在标签内填入内容-->
  <BaseSelect>
    你确认要删除嘛
  </BaseSelect>
  <BaseSelect>
    你确认要离开嘛
  </BaseSelect>

```

```

</div>
</template>

<script>
import BaseSelect from "@/components/BaseSelect.vue";
export default {
  components: {BaseSelect}
}
</script>

<style>

</style>

```

## 插槽-后备内容(默认值)

插槽的后备内容:封装组件时,可以为预留的<slot>提供插槽提供后备内容  
语法:

在<slot>标签内,放置内容,作为默认显示内容

```

<template>
  <div>
    <h3>友情提示</h3>
    <h4>
      <slot>你确定要删除我嘛</slot>
    </h4>
    <button>确认</button>
    <button>取消</button>
  </div>
</template>
<script>
export default {
}
</script>

<style>
body{
  background-color: #4d4d4d;
}
</style>

```

## 插槽-具名插槽

需求: 一个组件内有多处结构,需要外部传入标签,进行定制  
默认插槽: 一个的定制位置(多处结构需要定制的场景)  
语法:

1. 多个slot使用name属性区分名字  
 <slot name="名字"></slot>
2. template配合v-slot: 名字来分发对于标签  
 <template v-slot:'名字'>内容</template>
3. v-slot插槽名(简写)==#插槽名字

```

<template>
  <div>

```



```

    <h3>
      <slot name="head"></slot>
    </h3>

    <h4>
      <slot name="body">你确定要删除我嘛</slot>
    </h4>
    <button>确认</button>
    <button>取消</button>
  </div>

</template>

<script>
export default {
}
</script>

<style>
body{
  background-color: #4d4d4d;
}
</style>

```

```

<template>
<div>
<!--2. 在使用组件时, 在标签内填入内容-->
  <BaseSelect>
    <template v-slot:head>
      我是大标题
    </template>
    <template v-slot:body>
      你确定要删除我嘛
    </template>
  </BaseSelect>
</div>
</template>
<script>
import BaseSelect from "@/components/BaseSelect.vue";
export default {
  components: {BaseSelect}
}
</script>
<style>
</style>

```

## 插槽-作用域插槽

分类:默认插槽 具名插槽

作用域插槽:定义slot插槽的同时,是可以传值的.给插槽上可以绑定数据,将来使用组件时可以用.

场景:封装表格组件

1. 父传子,动态渲染内容
2. 利用默认插槽,定制操作列
3. 删除或查看都需要用到当前项的id,属于组件内部的数据

通过作用域插槽传值绑定,进而使用

基本使用步骤

1. 给slot标签,以添加属性的方式传值

```
<slot :id='item.id' msg='测试文本'></slot>
```

2. 所有添加的属性,都会被收集到一个对象中

{id:3,msg:'测试文本'}、

3. 在template中,通过'#插槽名="obj"接受,默认插槽名为Default'

```
<template>
  <table style="border: #00BE9A 1px solid">
    <thead >
      <tr >
        <th>序号</th>
        <th>姓名</th>
        <th>年纪</th>
        <th>操作</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="(item,index) in data" :key="item.id">
        <td>{{index + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.age}}</td>
        <td>
          <slot :row="item" msg="测试文本"></slot>
        </td>
      </tr>
    </tbody>
  </table>
</template>
<script>
export default {
  props:{
    data: Array
  }
}
</script>
<style>
</style>
```

```
<template>
<div>
<!--2. 在使用组件时,在标签内填入内容-->
  <BaseSelect :data="list1">
    <template #default="obj">
      <button @click="lookRow(obj.row)">删除</button>
    </template>
  </BaseSelect>
  <BaseSelect :data="list2">
    <template #default="obj">
```

```

        <button @click="del(obj.row.id)">查看</button>
      </template>
    </BaseSelect>
  </div>
</template>
<script>
import BaseSelect from "@components/BaseSelect.vue";
export default {
  methods: {
    del(id){
      alert(id)
    },
    lookRow(row){
      alert(`姓名: ${row.name};年龄: ${row.age};序号: ${row.id}`)
    }
  },
  components: {BaseSelect},
  data(){
    return{
      list1:[
        {id:1,name:'小黑',age:18},
        {id:2,name:'小吴',age:19},
        {id:3,name:'小席',age:17}
      ],
      list2:[
        {id:1,name:'赵小云',age:18},
        {id:2,name:'吴小超',age:19},
        {id:3,name:'席小田',age:17}
      ],
    }
  }
}
</script>
<style>
</style>
</style>

```

## 商品列表-项目实例

App.vue

```

<template>
  <div class="table-case">
    <Mytable :data="goods">
      <template #head>
        <th>编号</th>
        <th>图片</th>
        <th>名称</th>
        <th width="100px">标签</th>
      </template>
      <template #body="obj">
        <td>{{obj.index +1}}</td>
        <td></td>
        <td>{{obj.item.name}}</td>
        <td>
          <MyTag v-model="obj.item.tag"></MyTag>
        </td>
      </template>
    </Mytable>
  </div>
</template>

```

```

        </Mytable>
    </div>
</template>

<script>
import Mytable from "@/components/Mytable.vue";
import MyTag from "@/components/MyTag.vue";
export default {
  components: {
    MyTag,
    Mytable
  },
  data () {
    return {
      //测试组件功能的临时数据
      goods: [
        { id: 101, picture: 'https://yanxuan-item.nosdn.127.net/f8c37ffa41ab1eb84bff499e1f6acfc7.jpg', name: '梨皮朱泥三绝清代小品壶经典款紫砂壶', tag: '茶具' },
        { id: 102, picture: 'https://yanxuan-item.nosdn.127.net/221317c85274a188174352474b859d7b.jpg', name: '全防水HABU旋钮牛皮户外徒步鞋山宁泰抗菌', tag: '男鞋' },
        { id: 103, picture: 'https://yanxuan-item.nosdn.127.net/cd4b840751ef4f7505c85004f0bebc5.png', name: '毛茸茸小熊出没，儿童羊羔绒背心73-90cm', tag: '儿童服饰' },
        { id: 104, picture: 'https://yanxuan-item.nosdn.127.net/56eb25a38d7a630e76a608a9360eec6b.jpg', name: '基础百搭，儿童套头针织毛衣1-9岁', tag: '儿童服饰' },
      ]
    }
  }
}
</script>

<style lang="less" scoped>
.table-case {
  width: 1000px;
  margin: 50px auto;
  img {
    width: 100px;
    height: 100px;
    object-fit: contain;
    vertical-align: middle;
  }
}
</style>

```

MtTag.vue

```

<template>
  <div class="my-tag">
    <input
      v-focus
      v-if="isEdit"
      class="input"
      type="text"
      :value="value"
    >
  </div>
</template>

```

```

        placeholder="输入标签"
        @blur="isEdit = false"
        @keyup.enter="handleEnter"
      />
      <div class="text" v-else
        @dblclick="isEdit = true"
        >{{value}}</div>
    </div>
</template>

<script>
export default {
  props: {
    value: String
  },
  data () {
    return {
      isEdit: false
    }
  },
  methods: {
    handleEnter (e) {
      if (e.target.value === ''){
        this.isEdit = false
        alert('标签不能为空')
      }else {
        this.isEdit = false
        this.$emit('input', e.target.value)
      }
    }
  }
}
</script>

<style lang="less" scoped>
.my-tag {
  cursor: pointer;
  .input {
    appearance: none;
    outline: none;
    border: 1px solid #ccc;
    width: 100px;
    height: 40px;
    box-sizing: border-box;
    padding: 10px;
    color: #666;
    &::placeholder {
      color: #666;
    }
  }
}
}
</style>

```

Mytable.vue

```

<template>
  <table class="my-table">

```

```

    <thead>
    <tr>
      <slot name="head"></slot>
    </tr>
  </thead>
  <tbody>
    <tr v-for="(item,index) in data" :key="item.id">
      <slot name="body" :item="item" :index="index"></slot>
    </tr>
  </tbody>
</table>
</template>

<script>

export default {
  props:{
    data: {
      type: Array,
      required: true
    }
  }
}
</script>

<style lang="less" scoped>
.my-table {
  width: 100%;
  border-spacing: 0;
  img {
    width: 100px;
    height: 100px;
    object-fit: contain;
    vertical-align: middle;
  }
  th {
    background: #f5f5f5;
    border-bottom: 2px solid #069;
  }
  td {
    border-bottom: 1px dashed #ccc;
  }
  td,
  th {
    text-align: center;
    padding: 10px;
    transition: all .5s;
    &.red {
      color: red;
    }
  }
  .none {
    height: 100px;
    line-height: 100px;
    color: #999;
  }
}
</style>

```

## 路由介绍

单网页应用: 系统类网站/内部网站/文档类网站/移动端站点

多页面应用: 公司官网/电商类网站

单网页应用:

优点: 按需更新性能高, 开发效率高

缺点: 不利于SEO

路由的介绍:

路由是一种映射关系

生活中的路由: 设备和ip的映射关系 192.168.0.10 192.168.0.19 192.168.0.42

Vue中的路由: 路径和组件的映射关系 根据路由就能知道不同路径的, 应该匹配渲染那个组件

## VueRouter介绍

目标: 认识插件VueRouter, 掌握VueRouter的基本使用步骤

作用: 修改地址栏路径时, 切换显示匹配的组件

说明: Vue官方的一个路由插件

VueRouter的使用(5+2)

1. 下载: 下载VueRouter模块到当前工程版本3.6.5 eg: vue2 vue-router3.x vuex3.x vue3  
vue-router4.x vue4.x

```
npm add vue-router@3.6.5
```

2. 引入

```
import VueRouter from 'vue-router' main.js
```

3. 按照注册

```
Vue.use(VueRouter)
```

4. 创建路由对象

```
const router=new VueRouter()
```

5. 注入, 将路由对象注入到new Vue实例中, 建立关联

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

6. 创建需要的组件(vue目录), 配置路由规则

```
Find.vue My.vue Friend.vue  
const router=new VueRouter({  
  routes:[  
    {path: '/find' , component:Find},  
    {path: '/My' , component:My},  
    {path: '/Friend' , component:Friend}  
  ]  
})
```

7. 配置导航, 配置路由出口(路径匹配的组件显示的位置)

```
<a href="#/find">发现音乐</a>
```

```
<router-view></router-view>
```

```

Main.js
import Vue from 'vue'
import App from './App.vue'
import VueRouter from "vue-router";
Vue.use(VueRouter)
Vue.config.productionTip = false
const router=new VueRouter()
new Vue({
  render: h => h(App),
  router:router
}).$mount('#app')

```

## 组件存放目录问题

组件分类：**vue**文件分2类;页面组件和复用组件 注意：**.vue**文件本质无区别  
 分类开来 更易维护  
**src/views**文件夹  
     页面组件-页面展示-配合路由  
**src/components**文件夹  
     复用组件-展示数据-常用于复用

## 路由的封装抽离

目标：将路由模块抽离出来.好处:拆分模块,利于维护  
 不适合全部放在**main.js**  
 1.新建**router**下面的**index.js** 在**main**导入就行 在**router**写路由规则

```

//模块类
import Find from "@views/Find.vue";
import My from "@views/My.vue";
import Friend from "@views/Friend.vue";
import Vue from "vue";
import VueRouter from "vue-router";
//插件的初始化
Vue.use(VueRouter)
//创建路由对象
const router=new VueRouter({
  routes:[
    {path: '/find', component:Find},
    {path: '/my',component:My},
    {path: '/friend',component:Friend},
  ]
})
//导出router必写这个
export default router

```

## 声明式导航-导航链接

需求:实现导航高亮的效果  
**vue-router**提供了一个全局组件**router-link**(取代**a**标签)  
 1.能跳转,配置**to**属性指定路径(必须).本质还是**a**标签,**to**无需#  
 2.能高亮,默认就会提供高亮类名,可以直接设置高亮样式  
**<router-link to="/find">发现音乐</router-link>** **<a href="#/Friend">我的朋友</a>**



```

<template>
  <div>
    <div class="tag">
      <router-link to="/find">发现音乐</router-link>
      <router-link to="/my">我的音乐</router-link>
      <router-link to="/friend">我的朋友</router-link>
    </div>
    <div>
      <router-view></router-view>
    </div>
  </div>
</template>
<script>
export default {
}
</script>
<style>
body{
  margin: 0;
  padding: 0;
}
.tag{
  display: flex;
  width: 200px;
  height: 100px;
  color: white;
  background-color: black;
}
.tag a.router-link-active{
  background-color: #5EB69C;
}
</style>

```

## 声明式导航-两个类名

说明:我们发现router-link自动给当前导航添加了两个高亮的类名

router-link-exact-active

精确匹配 to="/my"仅可以匹配 /my

router-link-active

模糊匹配(用的多) to="/my" 可以匹配/my /my/a /my/b

说明: router-link的两个高亮类名太长了,我们希望能定制怎么办

```

//模块类
import Find from "@views/Find.vue";
import My from "@views/My.vue";
import Friend from "@views/Friend.vue";

import Vue from "vue";
import VueRouter from "vue-router";
//插件的初始化
Vue.use(VueRouter)
//创建路由对象
const router=new VueRouter({
  routes:[
    {path: '/find', component:Find},

```

```

    {path: '/my',component:My},
    {path: '/friend',component:Friend},
  ],
  //自定义高亮的类名
  linkActiveClass: 'active', //配置模糊匹配
  linkExactActiveClass: 'exact-active' //配置精确
})
//导出router
export default router

```

## 声明式导航-跳转传参

目标:在跳转路由时,进行传值

### 1. 查询参数传参

#### a. 语法如下

a.1 to="/path?参数名=值"

#### b. 对应页面组接收传递过来的值

b.1 \$route.query.参数名

### 2. 动态路由传参

#### a. 配置动态路由 index.js

```

path: '/saerch/:words'
component: Search

```

#### b. 配置导航链接

b.1 to="/path/参数值"

#### c. 对应页面组件接收传递过来的值

c.1 \$route.params.参数名

动态路由参数可选符

问题: 配了路由path:"/search/:words"为什么按下面步骤操作,会为匹配到组件,显示空白

原因:/search/:words 表示必须要传参数,如果不传参数,也希望匹配,可以加个'?'

/search/:words?

```

import Home from '@views/Home'
import Search from '@views/Search'
import Vue from 'vue'
import VueRouter from 'vue-router'
Vue.use(VueRouter) // VueRouter插件初始化
// 创建了一个路由对象
const router = new VueRouter({
  routes: [
    { path: '/', redirect: '/home' },
    { path: '/home', component: Home },
    { path: '/search/:words', component: Search }
  ]
})
export default router

```

```

<template>
  <div class="home">
    <div class="logo-box"></div>
    <div class="search-box">
      <input type="text">
      <button>搜索一下</button>
    </div>
    <div class="hot-link">
      热门搜索:
    </div>
  </div>
</template>

```

```

        <router-link to="/search/黑马程序员">黑马程序员</router-link>
        <router-link to="/search/前端培训">前端培训</router-link>
        <router-link to="/search/如何成为前端大牛">如何成为前端大牛</router-link>
    </div>
</div>
</template>

<script>
export default {
  name: 'FindMusic'
}
</script>

<style>
.logo-box {
  height: 150px;
  background: url('@/assets/logo.jpeg') no-repeat center;
}
.search-box {
  display: flex;
  justify-content: center;
}
.search-box input {
  width: 400px;
  height: 30px;
  line-height: 30px;
  border: 2px solid #c4c7ce;
  border-radius: 4px 0 0 4px;
  outline: none;
}
.search-box input:focus {
  border: 2px solid #ad2a26;
}
.search-box button {
  width: 100px;
  height: 36px;
  border: none;
  background-color: #ad2a26;
  color: #fff;
  position: relative;
  left: -2px;
  border-radius: 0 4px 4px 0;
}
.hot-link {
  width: 508px;
  height: 60px;
  line-height: 60px;
  margin: 0 auto;
}
.hot-link a {
  margin: 0 5px;
}
</style>

```

```

<template>
  <div class="search">
    <p>搜索关键字: {{ $route.params.words }} </p>

```

```

    <p>搜索结果: </p>
    <ul>
      <li>.....</li>
      <li>.....</li>
      <li>.....</li>
      <li>.....</li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'MyFriend',
  created () {
    // 在created中, 获取路由参数
    // this.$route.query.参数名 获取
    console.log(this.$route.query.key);
  }
}
</script>

<style>
.search {
  width: 400px;
  height: 240px;
  padding: 0 20px;
  margin: 0 auto;
  border: 2px solid #c4c7ce;
  border-radius: 5px;
}
</style>

```

```

<template>
  <div id="app">
    <div class="link">
      <router-link to="/home">首页</router-link>
      <router-link to="/search">搜索页</router-link>
    </div>

    <router-view></router-view>
  </div>
</template>

<script>
export default {};
</script>

<style scoped>
.link {
  height: 50px;
  line-height: 50px;
  background-color: #495150;
  display: flex;
  margin: -8px -8px 0 -8px;
  margin-bottom: 50px;
}
.link a {

```

```

display: block;
text-decoration: none;
background-color: #ad2a26;
width: 100px;
text-align: center;
margin-right: 5px;
color: #fff;
border-radius: 5px;
}
.link a.router-link-active{
  background-color: #5EB69C;
}
</style>

```

## Vue路由-重定向

问题:网页打开,url是默认/路径,未匹配到组件时,  
 说明:重定向->匹配path后,强制跳转到path路径  
 语法: {path: 匹配路径,redirect:重定向到路径}

```

import Home from '@/views/Home'
import Search from '@/views/Search'
import Vue from 'vue'

import VueRouter from 'vue-router'
Vue.use(VueRouter) // VueRouter插件初始化

// 创建了一个路由对象
const router = new VueRouter({
  routes: [
    { path: '/', redirect: '/home'},
    { path: '/home', component: Home },
    { path: '/search/:words', component: Search }
  ]
})
export default router

```

## Vue路由-404

作用: 当路径找不到匹配时,给个提示页面  
 位置: 配在路由最后  
 语法: path:"\*" (任意路径)->前面不匹配就命名最后这个  
 { path: '\*',component: NotFound}

## Vue路由-模式设置

问题: 路由的路径看起来不自然,有#,能否切换成真正路径形式  
 a. hash路由(默认) 例如: http: //localhost:8080/#/home  
 b. history(常用) 例如: http: //localhost:8080/home

```

import Home from '@/views/Home'
import Search from '@/views/Search'
import NotFound from "@/views/NotFound.vue";

```

```
import Vue from 'vue'

import VueRouter from 'vue-router'
Vue.use(VueRouter) // VueRouter插件初始化

// 创建了一个路由对象
const router = new VueRouter({
  mode: 'history',
  routes: [
    { path: '/', redirect: '/home' },
    { path: '/home', component: Home },
    { path: '/search/:words', component: Search },
    { path: '*', component: NotFound }
  ]
})
export default router
```

## 编程式导航-基本跳转

问题：点击按钮跳转如何实现  
 编程式导航：用JS代码来进行跳转  
 两种语法：

- 1.path路径跳转(简易方便)
 

```
this.$router.push('路由路径')
this.$router.push({
  path: '路由路径'
})
```
- 2.name命名路由跳转(适合path路径长的场景)
 

```
this.$router.push({
  name: '路由名'
})
{name : '路由名', path: '/path/xxx', component: xxx}
```

## 编程式导航-路由传参

问题：点击搜索按钮，跳转需要传参如何实现？  
 两种传参方式：查询参数+动态路由传参  
 两种跳转方式，对于两种传参方式都支持

- 1.path 路径跳转传参
 

path 路径跳转传参(query传参)('/path?参数名1=参数值&参数2=参数值')

```
this.$router.push({
  path: '/路径'
  query: {
    参数名1: '参数值1',
    参数名2: '参数值2',
  }
})
```

path 路径跳转传参(动态路由)

```
this.$router.push('/路径/参考值')
this.$router.push({
  path: '/路径.参数值'
})
```
- 2.name 命名路由传参
 

name 路径跳转传参(query传参)

```
this.$router.push({
```

```

name: '路由名字'
query:{
  参数名1: '参数值1',
  参数名2: '参数值2',
}
})
name 路径跳转传参(动态路由)
this.$router.push('/路径/参考值')
this.$router.push({
name: '路由名字'
parmas:{
  参数名:'参数值'
}
})

```

## 二级路由

```

{path: '/',
  component : Layout,
  children: [
    {path: '/article',component:Article},
    {path: '/collect',component:Collect},
    {path: '/like',component:Like},
    {path: '/user',component:User}
  ]},

```

## 组件缓存

原因: 路由跳转后,组件被销毁了,返回回来组件又被重建了,所以数据重新被加载了  
组件缓存keep-alive

### 1. keep-alive是什么

keep-alive是vue的内置组件,当他包裹动态组件时,会缓存不活动的组件实例,而不是销毁他们  
keep-alive时一个抽象组件,它自身不会渲染成一个DOM元素,也不会出现在父组件链中

### 2. keep-alive的优点

在组件切换过程中,把切换出去的组件保留在内存中,防止重复渲染DOM  
减少加载时间以及性能消耗,提高用户体验性

### 3. keep-alive的三个属性

- 1.include:组件名数组,只有匹配的组件才会被缓存 :include="['LayoutPage']"
- 2.exclude:组件名数组,任何匹配的组件都不会被缓存
- 3.max:最多可以缓存多少组件实例

### 4. keep-alive的使用会触发两个生命周期函数

activated当组件被激活(使用)的时候触发->进入页面触发  
deactivated当组件不被使用的时候触发->离开页面后触发

## 自定义创建项目

目标:基于vueCli自定义创建项目架子  
设置router css label Linter vue2.x VueRouter hash模式 CSS预处理Less Standard&Lint on Save

## ESlint代码规范

目标: 认识代码规范

代码规范: 一套写代码的约定规则。

规则的一小部分

字符串使用单引号 `'abc'`

无分号 `const name = 'zs'`

关键字后加空格

如果你的代码不符合`standard`的要求, `ESlint`会跳出来刀子嘴, 豆腐心地提示你

比如: 在`main.js`中随意做一些改动, 添加一些分号, 空行。

两种解决方案

### 1. 手动修正

根据错误提示来一项一项手动修改纠正

### 2. 自动修正

基于`vscode`插件`ESLint` 高亮错误 , 并通过配置自动帮助我们修复错误

## Vuex概述

目标: 明确`vuex`是什么, 应用场景, 优势

### 1. 是什么

`vuex`是一个`vue`的状态管理工具(状态就是工具)

大白话: `vuex`是一个插件, 可以帮助管理`vue`通用的数据(多组件共享的数据)

### 2. 场景

1. 某个状态在很多个组件来使用(个人信息)

2. 多个组件共同维护一份数据(购物车)

### 3. 优势

1. 共同维护一份数据, 数据集中化管理

2. 响应式变化

3. 操作简介(`vuex`提供了一些辅助函数)

## 构建vuex[多组件数据共享]环境

目标: 基于脚手架创建项目, 构建`vuex`多组件数据共享环境

目标: 创建一个空仓库

按照`vuex`插件, 初始化一个仓库

1. 安装`vuex` `yarn add vuex@3`

2. 新建`store/index.js` 专门存放`vuex`

3. `Vue.use(Vuex)` 创建仓库 `new Vuex.Store()`

// 这里面存放的就是 `vuex` 相关的核心代码

```
import Vue from 'vue'
```

```
import Vuex from 'vuex'
```

```
import user from './modules/user'
```

```
import setting from './modules/setting'
```

// 插件安装

```
Vue.use(Vuex)
```

// 创建仓库

```
const store = new Vuex.Store({
```

```
  // 严格模式 (有利于初学者, 检测不规范的代码 => 上线时需要关闭)
```

```
  strict: true,
```

```
  // 1. 通过 state 可以提供数据 (所有组件共享的数据)
```

```
  state: {
```

```
    title: '仓库大标题',
```



```

    count: 100,
    list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  },

  // 2. 通过 mutations 可以提供修改数据的方法
  mutations: {
    // 所有mutation函数, 第一个参数, 都是 state
    // 注意点: mutation参数有且只能有一个, 如果需要多个参数, 包装成一个对象
    addCount (state, obj) {
      console.log(obj)
      // 修改数据
      state.count += obj.count
    },
    subCount (state, n) {
      state.count -= n
    },
    changeCount (state, newCount) {
      state.count = newCount
    },
    changeTitle (state, newTitle) {
      state.title = newTitle
    }
  },

  // 3. actions 处理异步
  // 注意: 不能直接操作 state, 操作 state, 还是需要 commit mutation
  actions: {
    // context 上下文 (此处未分模块, 可以当成store仓库)
    // context.commit('mutation名字', 额外参数)
    changeCountAction (context, num) {
      // 这里是setTimeout模拟异步, 以后大部分场景是发请求
      setTimeout(() => {
        context.commit('changeCount', num)
      }, 1000)
    }
  },

  // 4. getters 类似于计算属性
  getters: {
    // 注意点:
    // 1. 形参第一个参数, 就是state
    // 2. 必须有返回值, 返回值就是getters的值
    filterList (state) {
      return state.list.filter(item => item > 5)
    }
  },

  // 5. modules 模块
  modules: {
    user,
    setting
  }
})

// 导出给main.js使用
export default store

```

## 核心状态-state状态

目标：明确如何给仓库提供数据,如何使用仓库的数据

### 1.提供数据

**State**提供唯一的公共数据项,所有共享的数据都要统一放到**store**中的**state**中存储  
在**State**对象中可以添加我们要共享的数据

### 2.使用数据

#### 1.通过store直接访问

#### 2.通过辅助函数

1.**this.\$store**

2.**import**导入**store**

模板中: `{{ $store.state.xxx }}`

组件逻辑中: `this.$store.state.xxx`

JS模块中: `store.state.xxx`

**mapState**是辅助函数,帮助我们把**store**中的数据自动映射到组件的计算属性中

```
import { mapState } from 'vuex'
```

```
mapState(['count'])
```

```
computed: { ...mapState(['count']) }
```

## 核心概念

目标:明确**vuex**同样遵循单向数据流,组件中不能直接修改仓库的数据  
通过**strict:true** 可以开启严格模式

## 核心概念-mutations

目标:掌握**mutations**的操作流程,来修改**state**数据(**state**数据的修改只能通过**mutations**)

### 1.定义Mutations对象,对象中存放修改state方法

### 2.组件中提交调用mutations

目标:掌握**mutations**传参语法

提交**mutation**函数(带参数-提交载荷payload)

```
addCount(state,n){
```

```
}
```

```
this.$store.commit('addCount',10)
```

如果需要多个参数,可以包装成一个对象

目标:实时输入,实时更新,巩固**mutations**传参语法

```
methods: {  
  handleAdd () {  
    this.$store.commit('addCount')  
  }  
}
```

```
mutations: {  
  addCount (state) {  
    state.count += 1  
  },  
  changeTitle (state) {  
  }  
}
```

```
<template>
  <div>
    <input :value="$store.state.count" type="text">
    <HelloWorld></HelloWorld>
  </div>
</template>
```

## 辅助函数:mapMutations

目标: 掌握辅助函数mapMutations, 映射方法

mapMutations和mapState很像,他是把位于Mutations中的方法提取了出来,映射到组件Methods中

```
mutations:{
  subCount (state, n){
    state.count -= n
  }
}
import { mapMutations } from 'vuex'
methods: {
  ...mapMutations(['subCount'])
}
//调用
this.subCount(10)
```

```
<script>
import { mapMutations } from 'vuex'
export default {
  methods: {
    ...mapMutations(['subCount']),
    handleSub (n) {
      this.$store.commit('subCount', n)
    }
  }
}
</script>
```

## 核心概念-actions

目标: 明确actions的基本语法,处理异步操作

需求:一秒钟之后,修改state的count成66

说明:mutations必须是同步的(便于监测数据变化,记录调试)

异步的话必需使用actions

```
handleChange () {
  this.$store.dispatch('setAsyncCount', 666)
}
```

```
actions: {
  setAsyncCount (context, num) {
    setTimeout(() => {
      context.commit('changeCount', num)
    }, 1000)
  }
}
```

## 辅助函数-mapActions

目标:掌握辅助函数mapActions, 映射出来  
mapActions是把位于actions的方法提取出来,映射到组件methods中

```
import { mapActions } from 'vuex'
methods:{
  ...mapActions(['setAsyncCount'])
}
```

```
<template>
  <div>
    vuex:{{$store.state.title}}
    vuex:{{$store.state.count}}
    <button @click="handleSub(5)">+5</button>
    <button @click="handleSub(15)">+15</button>
    <button @click="setAsyncCount(1020)">延时修改</button>
  </div>
</template>

<script>
import { mapMutations, mapActions } from 'vuex'

export default {
  methods: {
    //辅助函数
    ...mapMutations(['subCount']),
    ...mapActions(['setAsyncCount']),
    handleSub (n) {
      this.$store.commit('subCount', n)
    }
  }
}
</script>
<style>
</style>
```

## 核心概念-getters

目标:掌握核心概念getters的基本语法(类似于计算属性)

说明: 除了state之外,有时我们还需要从state中派生出一些状态,这些状态时依赖state的,此时会用到getters

例如: state定义了List ,为1-10的数组,组件中,需要显示所有大于5的数据

定义

```
getters:{
  filterList(state){
    return state.list.filter(item => >5)
  }
}
```

使用方式

a. 访问getters

通过store访问getters

```
{{ $store.getters.filterList }}
```

b. 通过辅助函数mapGetters映射

```
computed:{
  ...mapGetters([])
}
```

```

<template>
  <div>
    vuex:{{$store.state.title}}
    vuex:{{$store.state.count}}
    <hr>
    <div>
      {{$store.state.list}} <br>
      {{getCount}}
    </div>
    <button @click="handleSub(5)">+5</button>
    <button @click="handleSub(15)">+15</button>
    <button @click="setAsyncCount(1020)">延时修改</button>
  </div>
</template>

<script>
import { mapMutations, mapActions, mapGetters } from 'vuex'
export default {
  computed: {
    ...mapGetters(['getCount'])
  },
  methods: {
    ...mapMutations(['subCount']),
    ...mapActions(['setAsyncCount']),
    handlesub (n) {
      this.$store.commit('subCount', n)
    }
  }
}
</script>
<style>
</style>

```

```

getters: {
  getCount (state) {
    return state.list.filter(item => item > 5)
  }
}

```

## 核心概念-模块module(进阶语法)

目标:掌握核心概念module模块的创建

由于vuex使用单一状态树,应用的所有状态会集中到一个比较大的对象,当应用变得非常复杂时,store对象就有可能变得臃肿。

```

state: {
  title: '仓库大标题',
  count: 100,
  list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
模块拆分:
user模块:store/modules/user.js
const state={}
const motations={}
export default{

```

```

state,
mutations
}
import user from './modules/user'
const store=new Vuex.Store({
  modules: {
    user,
    setting
  }
})

```

目标:掌握模块中**state**的访问语法

尽管已经分模块了,单其实子模块的状态,还是会挂到根级别的**state**中,属性名就是模块名  
使用模块中的数据

1.直接通过模块名访问\$store.state.模块名.xxx

2.通过mapState映射

默认根级别的映射 mapState([''])

子模块的映射 mapState('模块名',['xxx']), 需要开启命名空间

```

export default{
  namespaced: true
  state,
  mutations
}

```

```

<template>
  <div>
    user: {{$store.state.user.userInfo.name}} <br>
    setting: {{$store.state.setting.theme}}
    <hr>
    user: {{user.userInfo.name}}<br>
    setting: {{setting.desc}}<br>
    userInfo: {{userInfo.name}}<br>
    setting: {{desc}}
  </div>
</template>
<script>
import { mapState } from 'vuex'

export default {
  computed: {
    ...mapState(['user', 'setting']),
    ...mapState('user', ['userInfo']),
    ...mapState('setting', ['desc'])
  }
}
</script>
<style>
</style>

```

目标:掌握模块中getters的访问语法

使用模块中的数据

1. 直接通过模块名访问 `$store.getters['模块名/xxx']`

2. 通过mapGetters映射

默认根级别的映射 `mapGetters(['xxx'])`

子模块的映射 `mapGetters('模块名',['xxx'])`, 需要开启命名空间

```
export default {
  namespaced: true,
  state,
  mutations
}
```

```
<template>
  <div>
    {{ $store.getters['user/UpperCaseName'] }}
    <br>
    {{ UpperCaseName }}
  </div>
</template>
<script>
import { mapGetters } from 'vuex'
export default {
  computed: {
    ...mapGetters('user', ['UpperCaseName'])
  }
}
</script>
<style>
</style>
```

目标:掌握模块中mutations的访问语法

注意: 默认模块中的Mutation和actions会被挂载到全局,需要开启命名空间,才会挂载到子模块调用子模块中Mutation:

1. 直接通过store调用 `$store.commit('模块名/xxx', 额外参数)`

2. 通过mapMutations映射

默认根级别的映射 `mapMutations(['xxx'])`

子模块的映射 `mapMutations('模块名',['xxx'])`-需要开启命名空间

```
export default {
  namespaced: true,
  state,
  mutations,
  actions,
  getters
}
```

```
<template>
<div>
  {{ $store.state.user.userInfo.name }}
  <button @click="updateUser">更新个人信息</button>
  {{ $store.state.setting.theme }}
  <button @click="updateSetting">更新主题配色</button>
  {{ $store.state.user.userInfo.name }}
  <button @click="setUser({ name: 'Su', age: 25})">更新个人信息</button>
  {{ $store.state.setting.theme }}
  {{ theme }}
</div>
</template>
```

```

      <button @click="setSetting('dark')">更新主题配色</button>
    </div>
  </template>

<script>
import { mapMutations, mapState } from 'vuex'

export default {
  computed: {
    ...mapState(['user', 'setting']),
    ...mapState('user', ['userInfo']),
    ...mapState('setting', ['theme'])
  },
  methods: {
    ...mapMutations('setting', ['setSetting']),
    ...mapMutations('user', ['setUser']),
    updateUser () {
      this.$store.commit('user/setUser', {
        name: '张三',
        age: 25
      })
    },
    updateSetting () {
      this.$store.commit('setting/setSetting', 'dark')
    }
  }
}
</script>
<style>
</style>

```

目标:掌握模块中action的调用语法(同理-直接类比mutation 即可)

注意: 默认模块中的Mutation和actions会被挂载到全局,需要开启命名空间,才会挂载到子模块  
调用子模块中action:

- 1.直接通过store调用 \$store.dispatch('模块名/xxx', 额外参数)
- 2.通过mapActions映射
  - 默认根级别的映射 mapActions(['xxx'])
  - 子模块的映射 mapActions('模块名', ['xxx'])-需要开启命名空间

```

export default {
  namespaced: true,
  state,
  mutations,
  actions,
  getters
}

```

```

<template>
  <div>
    {{ $store.state.user.userInfo.name }}
    <button @click="handleUser">更新个人信息</button>
    {{ $store.state.setting.theme }}
    <button @click="setSetting('green')">更新个人配色</button>
  </div>
</template>

<script>

```



```
import { mapActions } from 'vuex'

export default {
  methods: {
    ...mapActions('setting', ['setSetting']),
    handleUser () {
      this.$store.dispatch('user/setUserInfo', {
        name: '李四',
        age: 26
      })
    }
  }
}
</script>
<style>
</style>
```

## 默认的json-server

```
npm install -g json-server
创建db文件夹,创建index.js
启动
json-server --watch index.js
```

## vant组件库

目标：认识第三方Vue组件库vant-ui  
 组件库：第三方 封装好了很多很多的组件,整合到一起就是一个组件库。  
<https://youzan.github.io/vant/v2/#/zh-CN/intro>  
 目标：了解其它Vue组件库  
 vue的组件库并不是唯一的,vant-ui也仅仅是组件库的一种  
 1.pc端: element-ui ant-design-vue  
 2.移动端: vant-ui Mint ui Cube ui

## vant全部导入 和 按需导入

目标：明确 全部导入 和 按需导入 的区别  
 全部导入：  
 1.安装vant-ui  
 npm install vant@latest-v2 -S --force  
 2.main.js注册  
 import Vant from 'vant'  
 import 'vant/lib/index.css'  
 Vue.use(Vant)  
 3.使用测试  
 按需导入：  
 1.安装插件  
 npm i babel-plugin-import -D  
 2.babel.config.js 配置  
 module.exports = {  
 plugins: [  
 ['import', {  
 libraryName: 'vant',  
 libraryDirectory: 'es',

```

    style: true
  }, 'vant']
]
};
3.创建vant-ui.js 在utils
在main.js导入 import '@utils/vant-ui.js'

```

```

import Vue from 'vue'
import { Button, Icon, Tabbar, TabbarItem, NavBar } from 'vant'
Vue.use(NavBar)
Vue.use(Tabbar)
Vue.use(TabbarItem)
Vue.use(Icon)
Vue.use(Button)

```

## 项目中的vw适配

目标：基于postcss插件,实现项目vw适配

```

npm install postcss-px-to-viewport@1.1.1 -D

```

根目录创建postcss.config.js填入下面配置

```

module.exports = {
  plugins: {
    'postcss-px-to-viewport': {
      viewportwidth: 375,
    },
  },
};

```

## request模块-axios封装

目标：将axios请求方法,封装到request模块

使用axios来请求后端端口,一般都会对axios进行一些配置

所以项目开发中,d都会对axios进行基本的二次封装,单独封装到一个request模块中,便于维护使用

```

import axios from 'axios'
const instance = axios.create({
  baseURL: 'http://smart-shop.itheima.net/index.php?s=/api',
  timeout: 5000
})
// Add a request interceptor
instance.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
instance.interceptors.response.use(function (response) {
  // Any status code that lie within the range of 2xx cause this function to trigger

```

```

    // Do something with response data
    return response.data
  }, function (error) {
    // Any status codes that falls outside the range of 2xx cause this function to
    trigger
    // Do something with response error
    return Promise.reject(error)
  })
export default instance

```

## api接口模块-封装图片验证码接口

目标：将请求封装成方法，统一存放到api模块，与页面分离

API模块：存放封装好的请求函数（请求A函数，请求B函数） 方法：(Afn(), Bfn())

封装api模块的好处

1. 请求与页面逻辑分离
2. 相同的请求可以直接复用
3. 请求进行了统一管理

新建请求模块 => 封装请求函数 => 页面中导入调用

```

import request from '@utils/request'
export const getPicCode = () => {
  return request.get('/captcha/image')
}

```

## Toast轻提示

目标：阅读文档，掌握toast轻提示

1. 注册安装
2. 使用 通过

```
import { toast } from 'vant'
```

本质：将方法，注册挂载到了vue原型上

```
this.$toast('提示内容')
```

## 响应拦截器统一处理错误提示

目标：通过响应拦截器，统一处理接口的错误提示

说明：响应拦截器是咱们拿到数据的第一个数据流转站，可以在里面统一处理错误

```

instance.interceptors.response.use(function (response) {
  // Any status code that lie within the range of 2xx cause this function to
  trigger
  // Do something with response data
  const res = response.data
  if (res.status !== 200) {
    // 给提示
    Toast.fail(res.message)
    // 抛出一个错误的promise
    return Promise.reject(new Error(res.message))
  }
  return response.data
}, function (error) {

```

```
// Any status codes that falls outside the range of 2xx cause this function to trigger
// Do something with response error
return Promise.reject(error)
})
```

## 登录权证信息存储

目标：vuex构建user模块存储登录权证

补充说明：

1.token存入vuex的好处,容易获取,响应式

2.vuex需要分模块=>user模块

构建user模块 => 挂载到vuex => 提供mutations => 页面中Commit调用

## storage存储模块-vuex持久化处理

目标：封装storage存储模块,利用本地存储,进行vuex持久处理

```
localStorage.setItem('hm_shopping_info', JSON.stringify(xxx))
```

## 添加Loading效果

目标:统一在每次请求后台时,添加loading效果

背景: 有时候因为网络原因,一次请求的结果可能需要一段时间后才能回来

此时,需要给用户添加loading提示

添加loading提示的好处

1.节流处理 防止用户在一次请求还没回来之前,多次进行点击,发送无效请求

2.友好提示:告知用户,目前是在加载中,请耐心等待,用户体验更好

实现步骤:

1.请求拦截器中,每次请求,打开loading

2.响应,每次响应,关闭loading

```
import axios from 'axios'
import { Toast } from 'vant'
const instance = axios.create({
  baseURL: 'http://smart-shop.itheima.net/index.php?s=/api',
  timeout: 5000
})
// Add a request interceptor
instance.interceptors.request.use(function (config) {
  // Do something before request is sent
  Toast.loading({
    message: '加载中...',
    forbidClick: true,
    duration: 0
  })
  return config
}, function (error) {
  // Do something with request error
  return Promise.reject(error)
})
```

```
// Add a response interceptor
instance.interceptors.response.use(function (response) {
  // Any status code that lie within the range of 2xx cause this function to trigger
  // Do something with response data
  const res = response.data
  if (res.status !== 200) {
    // 给提示
    Toast.fail(res.message)
    // 抛出一个错误的promise
    return Promise.reject(new Error(res.message))
  } else {
    Toast.clear()
  }
  return res
}, function (error) {
  // Any status codes that falls outside the range of 2xx cause this function to trigger
  // Do something with response error
  return Promise.reject(error)
})
export default instance
```

## 页面访问拦截

目标：基于全局前置守卫,进行页面访问拦截处理

路由导航守卫：全局前置守卫

1. 所有的路由一旦被匹配到,都会经过全局前置守卫
2. 只有全局前置守卫放心,才会真正解析渲染组件,才能看到页面内容

**to:**到哪里去

**from:**从哪里来

**next()**放行

**next(路径)**进行拦截

```
const authUrls = ['/order', '/pay']
router.beforeEach((to, from, next) => {
  if (!authUrls.includes(to.path)) {
    next()
    return
  }
  const token = store.getters.token
  if (token) {
    next()
  } else {
    next('/login')
  }
})
```

## 打包

```
vue.config.js 加上
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
  publicPath: './',
  transpileDependencies: true
})
```

# Vue3

## 认识create-vue 创建项目

1. 前提环境条件  
已安装node.js
2. 创建一个vue应用  
npm init vue@latest
3. 除了ESLint都选no

## 项目结构

vite.config.js 项目配置文件 基于vite配置  
package.json 项目包文件 核心依赖项变成vue3 和 vite  
main.js 入口文件  
app.vue 根组件  
index.html 单页入口 根据id为app的挂载点

## 组合式API-setup

比beforeCreate函数更早  
在setup中获取不到this  
数据和函数 需要在Setup最后return 才能在模板中被使用

```
<script setup>
  const message = 'Hello Vue3'
</script>
```

经过语法糖的封装更简单使用封装api

## 组合式APIreactive()和ref

reactive()  
作用: 接受对象类型数据的参数传入并返回一个响应式对象  
核心步骤:

1. 从vue包中导入reactive函数
2. 在中执行reactive函数并传入类型为对象的初始值, 并使用变量接受返回值

```
<script setup>
import { reactive } from "vue";
const state = reactive({
  count: 100
})
</script>
```

ref()  
作用: 接受简单类型或者对象类型的数据传入并返回一个响应式对象  
核心步骤:

1. 从vue包中导入ref函数
2. 在中执行ref函数并传入类型为对象的初始值, 并使用变量接受返回值

```
<script setup>
import { ref } from "vue";
const state = ref(0)
const setCount = () => {
  state.value++
}
</script>
推荐使用ref()
```

## 组合式APIcomputed()

计算属性基本思想和vue2的完全一致,组合式api下的计算属性只是修改了写法  
核心步骤

1. 导入computed函数
2. 执行函数 在回调函数中return基于响应式数据做计算的值,用变量接受

## 组合式APIwatch

作用: 侦听一个或者多个数据的变化,数据变化时执行回调函数  
两个额外参数: 1.immediate(立即执行) 2.deep(深度监听)

基础使用:

1. 导入watch函数
2. 执行watch函数

```
<script setup>
import {ref, watch} from "vue";

const count = ref(0)
const nickName = '张三'
const chageCount=()=>{
  count.value++
}

// 单个对象
watch(count, (newValue, oldValue)=>{
  alert(newValue+oldValue)
})

// 多个对象
watch([count, nickName], (newValue, oldValue)=>{
  console.log(newValue, oldValue)
})
</script>

deep(深度监视), 默认watch进行的是浅层监视
const ref1 = ref(复杂类型)
const setUserInfo = () =>{
  userInfo.value = {name: '张三', age: 121}
}

精确侦听对象的某个属性
需求: 再不开启deep的前提下, 侦听age的变化, 只有age变化时才执行回调
watch(() => userInfo.value.age, (newValue, oldValue) => {
  console.log(newValue, oldValue)
})
```

## 组合式API生命周期函数

选项式API	组合式API
beforeCreate/created	setup
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeUnmount	onBeforeUnmount
unmounted	onUnmounted

```
beforeCreate/created 一律在setup中执行
onMounted(() => {
  console.log('mounted生命周期函数')
})
```

## 组合式API-父子通信

父传子

基本思想：

1. 父组件给子组件绑定属性
2. 子组件内部通过props选项接受 需要借助于编辑器宏函数接受子组件传递的数据

**defineProps**原理：就是编译阶段的一个标识,实际编译器解析

```
const props = defineProps({
  car: String, // 静态传递
  money: Number // :money='money' 动态传递
})
```

```
<script setup>
const props = defineProps({
  car: String,
  money: Number
})
</script>
<template>
  <div class="son">我是子组件</div>
  {{props.car}}
  {{props.money}}
</template>

<style scoped>
.son{
  color: red;
  font-size: 10px;
  border: 1px solid red;
  width: 100px;
  height: 20px;
}
</style>
```



```

<template>
  <div>
    <SonCom car="宝马车" :money="money"></SonCom>
  </div>
</template>

<script setup>
import {ref} from 'vue'
const money = ref(100)
import SonCom from "@components/son-com.vue";
</script>

<style>

</style>

```

子传父

基本思想:

1. 父组件中给予组件标签通过@绑定事件
2. 子组件内部通过emit方法触发事件

```

<script setup>
const props = defineProps({
  car: String,
  money: Number
})
const emit = defineEmits(['changeMoney'])
const buy = () => {
  emit('changeMoney',5)
}
</script>

<template>
  <div class="son">我是子组件</div>
  {{props.car}}
  {{props.money}}
  <button @click="buy">花钱</button>
</template>

<style scoped>
.son{
  color: red;
  font-size: 10px;
  border: 1px solid red;
  width: 100px;
  height: 20px;
}
</style>

```

```

<template>
  <div>
    <SonCom car="宝马车" :money="money" @changeMoney="changeFn"></SonCom>
  </div>
</template>

```

```

<script setup>
import {ref} from 'vue'
const money = ref(100)
import SonCom from "@/components/son-com.vue";
const changeFn = (val) => {
  money.value = val
}
</script>

<style>

</style>

```

## 组合式API-模板引用

通过ref标识获取真实的Dom对象或者组件实例对象  
如何使用

- 1.调用ref函数生成一个ref对象
- 2.通过ref标识绑定ref对象到标签

defineExpose()

默认情况下在<script setup>语法糖下组件内部的属性和方法是不开放给父组件访问的  
可以通过defineExpose编译宏指定哪些属性和方法允许访问

```

<template>
  <div>
    <p>测试组件</p>
  </div>
</template>

<script setup>
defineExpose({
  const,
  sayHi
})
</script>

<style>

</style>

```

```

<template>
  <div>
    <input ref="inp" type="text">
    <SonCom ref="testRef"></SonCom>
  </div>
</template>

<script setup>
import SonCom from "@/components/son-com.vue";
import {onMounted, ref} from "vue";
const inp = ref(null)
const testRef = ref(null)
onMounted(() => {
  inp.value.focus()
})

```

```
}  
</script>  
  
<style>  
  
</style>
```

## 组合式API-provide和inject

作用和场景

顶层组件向任意的底层组件传递数据和方法,实现跨层组件通信

顶层组件: `provide('key', 顶层组件中的数据)`

底层组件: `const message = inject('key')`

## vue3.3新特性defineOptions

```
defineOptions({  
  name : 'loginIndex'  
})
```

## Pinia

### 简介

Pinia是Vue的最新状态管理工具,是Vuex的替代品

1. 提供更加简单的API(去掉了Mutation)
2. 提供符合, 组合式风格的API(和Vue3新语法统一)
3. 去掉了modules的概念, 每一个store都是一个独立的模块
4. 配合TypeScript更加友好, 提供可靠的类型判断

## Element Plus

1. 安装  
`pnpm install element-plus`
2. 配置
3. 直接使用