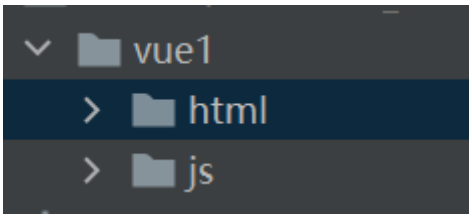
Vue Vue

0.开发工具vue devtools的使用

`Vue.config.devtools = true;`

1.建立Hello Vue

建立一个文件夹，里面加入html文件和vue-min.js文件



在html文件中建立一个Hello Vue语句，通过建立Vue实例来接管html元素

```
<div id="root">
    <h1>Hello {{msg.toUpperCase()}}</h1>
</div>
<script>
    Vue.config.productionTip=false
    new Vue({
        el:"#root",
        data:{
            msg:"vue",
        }
    })
</script>S
```

这里的el就是element的意思，里面的参数就是css选择器，data中建立了一个msg实例，当然其中还可建立其他的实例，通过前端的`{{msg}}`的读取返回在了页面上，这里一个vue实例管理一个html元素，不得有其他的vue实例来干涉

el与data的两种写法

```
Vue.config.devtools = true;
new Vue({
    data:{
        msg:"vue",
        school:{
            url:"https://www.baidu.com",
        },
    }
}).$mount("#root")
```

```
Vue.config.devtools = true;
const v = new Vue({
    el:'#root',
    data:function(){
        return{
            msg:"java",
        }
    }
})
```

2.模板语法

- 插值语法

在vue实例中建立的实例数据直接可以通过`{{}}`来获取显示在前端的界面上

- 指令语法

1.v-bind ===> :

这个标签可以给html元素的属性动态赋值

```
<a :href="school.url">点击</a>
```

```
Vue.config.devtools = true;
new Vue({
    el:"#root",
    data:{
        msg:"vue",
        school:{
            url:"https://www.baidu.com",
        },
    }
})
```

这个标签是单向绑定的，就是在devtool中修改了之后会在页面上显示，但是页面上修改后，就无法将devtool中的值修改

2.v-model　　v-model:value ===> v-model

这个标签是双向的，就是你修改了页面上的值，devtool上的值也会修改，而且这个标签只能用于表单中

```
<input type="text" v-bind:value="msg">
<input type="text" v-model:value="msg">
```

3.v-on　===> @

- 使用`v-on:xxx`或`@xxx`绑定事件，其中xxx是时间名字
- 事件回调函数需要配置在Vue实例对象中的methods属性上
- methods中配置的函数，不要使用箭头函数，否则函数所属对象的this就不是对象本身了
- methods中配置的函数，都是被Vue所管理的函数，this的指向是vm或组件实例对象
- `@click="demo"`和`@click="demo($event)"`的效果是一样的，但是后者可以传递参数

```
<div id="root">
    <button v-on:click="showInfo($event,55)">点击这里</button>
</div>
<script>
    let data ={
        name:"feng",
        address:"杭州"
    }
    const vm = new Vue({
        el:"#root",
        data:data,
        methods:{
            showInfo(e,number){
                console.log("Hello"+number)
            }
        }
    })
</script>
```

3.数据代理

1.Object.defineProperty()

通过这个函数可以给js对象动态地添加属性

```
<script>
    let person ={
        name:"张三",
        sex:"男"
    }
    Object.defineProperty(person,'age',{
        value:18,
        enumerable:true,
        writable:true,
        configurable:true,
    })
```

其中的属性更是提供了很大的动态管理

- enumerable:使这个元素加到js对象中，使得这个属性在js对象中可枚举
- writeable:使这个元素可以被修改

- configurable:控制这个元素是否可以删除

通过这个函数的get和set函数可以实现对这个元素的动态修改

```
get(){
    return number
},
set(v) {
    number = v
}
```

当获取对象中的新加属性的时候，会自动调用get的方法，将age属性值返回

当修改对象中的新加属性的时候，会自动调用set的方法，将age的属性值修改

2.数据代理

通过一个对象来代理管理另一个对象的属性

```
let obj1 = {
    name:"张三"
}
let obj2 = {
    name:"李四"
}
Object.defineProperty(obj2,"y",{
    get() {
        return obj1.name
    },
    set(v){
        obj1.name = v
    }
})
```

Vue的实例中有个属性叫做：`_data`，这个属性就是数据代理的例子，这里data就是Vue实例中的_data属性

```
let data ={
    name:"feng",
    address:"杭州"
}
const vm = new Vue({
    el:"#root",
    data:data
})
script>
```

4.事件处理

1.Vue中的事件修饰符

这些键可以实现组合使用，也就是链式编程

- prevent　阻止html标签的默认事件的发生

```
n" @click.prevent="showInfo">点
```

- stop　　阻止事件冒泡　　从底层标签将相同的方法向上逐一调用

```
<div id="root">
    <div id="div1" @click="showInfo">
        <div id="div2">
            <button @click.stop="showInfo">点我2</button>
        </div>
    </div>
</div>
```

- once　　事件只触发一次

```
<div id="root">
    <button @click.once="showInfo">点击我</button>
</div>
```

- capture　直接先使用事件的捕获模式　　**先捕获再冒泡**　　从上级标签向下调用相同的方法

```
<div id="div1" @click.capture="showInfo1">
    <div id="div2">
        <button @click="showInfo2">点我2</button>
    </div>
</div>
```

- self　　只有event.target是当前的操作的元素才触发事件

```
<div id="div1" @click.self="showInfo1">
    <div id="div2">
        <button @click="showInfo1">点我2</button>
    </div>
</div>
```

- passive　事件的默认行为立即执行，无需等待函数的回调执行完毕　**先执行函数体，在执行标签的默认行为**

```
<div id="root">
    <a href="https://www.baidu.com" @click.passive="showInfo1">点我</a>
</div>
```

2.键盘事件

1.Vue中常用的键盘按键别名

- 回车　`enter`
- 删除　`delete`　　捕获"delete"和”backspace“按键
- 退出　`esc`
- 空格　`space`
- 换行　`tab`　　这个按键十分的特殊，这个要配合@Keydown的绑定去使用
- 上　`up`
- 下　`down`
- 左　`left`
- 右　`right`

```
<div id="root">
    <input type="text" id="input1" name="按我测试" value="" @keyup.enter="showInfo2">
</div>
```

2.系统修饰键(用法特殊) `ctrl` `alt` `shift` `meta`　(这个meta键就是windows电脑上的win键)

> a. 配和`keyup`使用：按下修饰键的同时，按下其他的按键，再释放其他的按键，事件才会被触发
>
> 组合键的使用：实例：只能同时按下ctrl和y的时候触发事件　@keyup.ctrl.y
>
> ```
> 
>     <input type="text" id="input1" name="按我测试" value="" @keyup.ctrl.enter="showInfo2">
> 
> ```

> b.配和`keydown`使用正常

```
<input type="text" id="input1" name="按我测试" value="" @keydown.ctrl="showInfo2">
<input type="text" id="input2" name="按我测试" value="" @keyup.ctrl="showInfo2">
```

3.`Vue.config.keyCodes.自定义键名 = 键码`，可以实现自定义键名

```
<input type="text" id="input1" name="按我测试" value="" @keyup.huice="showInfo2">
/div>
script>
Vue.config.keyCodes.huice = 13
```

4.函数的写法

这里的e是键盘事件，这个`e.target.value`是指的是`input`中的输入值

```
new Vue({
    el:'#root',
    methods:{
        alter(e){
            console.log(e.target.value)
        }
    }
})
```

5.计算属性

1.实现名和字之间用-链接并可动态实现

1.`{{}}`的使用

这个在需要对字符串做一些复杂的操作的时候会显得十分的累赘，繁琐

```
<div id="root">
    姓：<input type="text" v-model:name="firstName" value=""><br/><br/>
    名：<input type="text" v-model:name="lastName" value=""><br/><br/>
    <span>{{firstName}}-{{lastName}}</span>
</div>
<script>
    new Vue({
        el:"#root",
        data:{
            firstName:"张",
            lastName:"三"
        }
    })
```

2.methods的使用

这个通过一个函数来实现效果，但是当用户量多了的时候，每次的查询的时候就会不断地调用该方法，导致冗余的情况发生

```
<div id="root">
    姓：<input type="text" v-model:name="firstName" value=""><br/><br/>
    名：<input type="text" v-model:name="lastName" value=""><br/><br/>
    <span>{{fullName()}}</span>
</div>
<script>
    new Vue({
        el:"#root",
        data:{
            firstName:"张",
            lastName:"三"
        },
        methods:{
            fullName(){
                return this.firstName+"-"+this.lastName
            }
        }
    })
```

3.计算属性 `computed`

1.定义：这个需要在Vue实例中的属性来进行计算

2.原理：底层使用了`Objcet.defineproperty()`方法提供的`getter`和`setter`

3.`get`函数什么时候执行

- 最初的时候会执行一次
- **当依赖的数据发生改变时会被再次调用**

4.优势：与`methods`的实现相比，computed中有缓存机制，效率更高，调试方便

5.备注

- 计算属性最终会出现在Vue的实例对象中，直接使用就行
- 如果计算属性要被修改，那么必须写`set`函数去相应修改，且`set`中要引起计算时相应的依赖数据发生改变，也就是要改变原数据，这里直接修改原数据就行了
- 简写computed属性，当仅仅是只需要获得值的时候就直接简写就行了

```
<div id="root">
    姓：<input type="text" v-model:name="firstName" value=""><br/><br/>
    名：<input type="text" v-model:name="lastName" value=""><br/><br/>
    <span>{{fullName}}</span>
</div>
<script>
    const vm = new Vue({
        el:"#root",
        data:{
            firstName:'张',
            lastName:'三'
        },
        computed:{
            fullName:{
                get(){
                    return this.firstName+'-'+this.lastName
                },
                set(value){
                    const split = value.split('-')
                    this.firstName = split[0]
                    this.lastName = split[1]
                }
            }
        }
    })
```

6.数据监听

1.实现天气切换

直接给button的click属性加上一个bool值切换的方法

```
<div id="root">
    <h1>今天天气很{{isHot?"炎热":"凉快"}}</h1>
    <button @click="isHot=!isHot">点我切换天气</button>
</div>
```

同理可以通过methods的方法加一个计算属性来动态切换天气值

```
methods:{
    changeWeather(){
        this.isHot = !this.isHot
    }
},
computed:{
    info(){
        return this.isHot?"炎热":"凉快"
    }
}
```

2.监听属性 `watch`

- **当被监听的属性变化时，回调函数自动调用，进行相关操作**
- 监视的属性必须存在于Vue的实例中，才能进行监视，既可以监视data中的属性，还可以监听computed的属性
- 配置项属性`immediate:false`，改为true，则初始化的时候调用一次`handle(newValue,oldValue)`
- 监视有两种写法

> a.创建`vue`的时候加上watch:{}的配置
>
> b.通过`vm(vue的实例)vm.$watch()`来进行监视

第一种方式：

```
watch:{
    isHot:{
        immediate:true,
        handler(newValue,oldValue){
            console.log(newValue,oldValue,"isHot被改变了")
        }
    }
}
```

第二种方式：

```
vm.$watch('isHot',{
    immediate:true,
    handler(newValue,oldValue){
        console.log(newValue,oldValue,"isHot被改变了")
    }
})
```

3.深度监听

- `Vue` 中的 `watch` 默认不监听对象内部值的改变(一层)
- 在 `watch` 中配置 `deep:true` 可以检测对象内部值的改变(多层)

注意:

- `Vue` 自身可以检测对象内部值的改变，但是 `Vue` 提供的watch默认不提供
- 使用 `watch` 时根据监视数据的具体结构，决定是否采用深度监视

```
data:{
    isHot:false,
    number:{
        a:3
    }
},
```

vue默认的属性监视就是通过字符串实现的，这个就是深度的属性监听

```
watch:{
    'number.a':{
        immediate:true,
        handler(newValue,oldValue){
            console.log(newValue,oldValue,"isHot被改变了")
        }
    }
}
```

通过设置属性 `deep` 为true来引导深层监听的开启

```
watch:{
    number:{
        immediate:true,
        deep:true,
        handler(newValue,oldValue){
            console.log(newValue,oldValue,"isHot被改变了")
        }
    }
}
```

```
vm.$watch('isHot',{
    immediate:true,
    deep:true,
    handler(newValue,oldValue){
        console.log(newValue,oldValue,"isHot被改变了")
    }
})
```

监听属性简写，在这个简写中无法写更多的属性配置

```
watch:{
    isHot(newValue,oldValue){
        console.log(newValue,oldValue,"isHot被改变了")
    }
}
```

4.计算属性VS侦听属性

`computed`和`watch`间的区别

- `computed`能完成的功能，`watch`都能完成
- `watch`能完成的功能，`computed`不一定能完成，例如`watch`可以进行异步操作

两个重要的小原则

- 所有`vue`管理的函数，最好写成普通函数，这样**this**的指向才是`vm`或者`组件实例对象`
- 所有不被`vue`管理的函数(`定时器的回调函数`，`ajax的回调函数`等，`promise的回调函数`)，最好写成`箭头函数`，这样**this**的指向才是`vm`或`组件实例对象`，否则指向的对象就是window对象了

watch的异步实现和箭头函数的使用

```
watch:{
    isHot(newValue,oldValue){
        setTimeout(()=>{
            console.log(newValue,oldValue,"isHot被改变了")
        },2000)
    }
}
```

7.Class和Style属性绑定

绑定样式：

- 写法：`:class="xxx"`，xxx可以是字符串，数组，对象
- `:style="[a,b]"`，其中a,b是样式对象
- `:style="{fontSize:xxx}"`其中xxx是动态值
    - 字符串写法适合于：类名不确定，要动态获取
    - 数组写法适用于：要绑定多个样式，个数不确定，名字也不确定
    - 对象写法适用于：要绑定多个样式，个数确定，名字也确定，但不确定用不用

这里的数组有`shift`将数组的第一个元素删除 `push`将元素的加入到数组末尾中

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>HelloVue</title>
        <script src="../../js/vue.min.js"></script>
        <style>
            .basic {width: 300px;height: 50px;border: 1px solid black;}
            .happy {border: 3px solid red;background-color: rgba(255, 255, 0,
0.644);
                background: linear-gradient(30deg, yellow, pink, orange,
yellow);}
            .sad {border: 4px dashed rgb(2, 197, 2);background-color: skyblue;}
            .normal {background-color: #bfa;}
            .atguigu1 {background-color: yellowgreen;}
            .atguigu2 {font-size: 20px;text-shadow: 2px 2px 10px red;}
            .atguigu3 {border-radius: 20px;}
        </style>

    </head>
    <body>
        <div id="root">
            <!-- 绑定class样式--字符串写法，适用于：样式的类名不确定，需要动态指定 -->
            <div class="basic" :class="mood" @click="changeMood">{{name}}</div>
<br/><br/>

            <!-- 绑定class样式--数组写法，适用于：要绑定的样式个数不确定、名字也不确定 -->
            <div class="basic" :class="classArr">{{name}}</div><br/><br/>

            <!-- 绑定class样式--对象写法，适用于：要绑定的样式个数确定、名字也确定，但要动态
决定用不用 -->
```

```
            <div class="basic" :class="classObj">{{name}}</div><br/><br/>

            <!-- 绑定style样式--对象写法 -->
            <div class="basic" :style="styleObj">{{name}}</div><br/><br/>

            <!-- 绑定style样式--数组写法 -->
            <div class="basic" :style="styleArr">{{name}}</div>
        </div>

        <script type="text/javascript">
            Vue.config.productionTip = false

            const vm = new Vue({
                el: '#root',
                data: {
                    name: '尚硅谷',
                    mood: 'normal',
                    classArr: ['atguigu1', 'atguigu2', 'atguigu3'],
                    classObj: {
                        //通过boolean来控制class是否显示出来
                        atguigu1: false,
                        atguigu2: false,
                    },
                    styleObj: {
                        //字符串直接在devtools中修改
                        fontSize: '40px',
                        color: 'red',
                    },
                    styleObj2: {
                        backgroundColor: 'orange'
                    },
                    styleArr: [
                        {
                            fontSize: '40px',
                            color: 'blue',
                        },
                        {
                            backgroundColor: 'gray'
                        }
                    ]
                },
                methods: {
                    changeMood() {
                        const arr = ['happy', 'sad', 'normal']
                        const index = Math.floor(Math.random() * 3)
                        this.mood = arr[index]
                    }
                },
            })
        </script>
    </body>
</html>
```

8.条件渲染

`v-if`

- 写法和`if-- else--`的语法类似

> v-if="布尔表达式"
>
> v-else-if="布尔表达式"
>
> v-else

- 适用于：切换频率较低的场景，因为不展示的DOM元素直接被移除了
- 注意：`v-if`可以和`v-else-if` `v-else` 一起使用，但要求结构不能被打断

`v-show`

- 写法：`v-show="表达式"`
- 适用于：切换频率较高的场景
- 特点：不展示的DOM为移除，仅仅是使用样式隐藏掉了`display:none`

备注:不适用`v-if`的时候，元素可能无法获取到，而使用`v-show`一定可以获取到

template标签不影响结构，页面的`html`中也不会有这个标签，但只能配合`v-if`，不能配合`v-show`

```
<div id="root">
    <h1 v-show="false">你好</h1>
</div>
```

```
<div id="root">
    <h1 v-if="false">你好</h1>
</div>
```

```
<div id="root">
    <button @click="n++">点我n++{{n}}</button>
    <h1 v-if="n===1">你好</h1>
    <h1 v-else-if="n===2">吃了没</h1>
    <h1 v-else="n===3">再见</h1>
</div>
```

9.列表渲染

`v-for`语句

- 用于展示列表数据
- 语法`<li v-for="(item,index) in items" :key="index">{{item}}</li>`，这里的`key`可以是index，更好的是遍历对象的唯一标识
- 可遍历：数组，对象，字符串，指定次数

```
    <ul>
        <li v-for="(item,index) in 5" :key="index">{{index}}--{{item}}</li>
    </ul>
/div>
script type="text/javascript">
    Vue.config.productionTip = false
    Vue.config.devtools = true
    new Vue({
        el:"#root",
        data:{
            persons:[
                {id:"001",name:"张三",age:13},
                {id:"001",name:"李四",age:13},
                {id:"001",name:"王五",age:13},
            ],
            car:{
                name:"五菱宏光",
                price:"4万",
                color:"black",
            },
            str:"hello",
        },
```

面试题

react和vue中的key有什么作用？(key的内部原理)

- 虚拟DOM中key的作用：key是虚拟DOM中对象的标识，当数据发生变化时，Vue会根据新数据生成的新的DOM的虚拟DOM，随后Vue进行新虚拟DOM与旧虚拟DOM的差异比较，比较规则如下：
- **对比规则：**
  a.旧的虚拟DOM中找到了与新的虚拟DOM相同的key

  > **1.若虚拟DOM中的内容没变，直接使用之前的真实DOM**
  >
  > **2.若虚拟DOM中的内容变了，则生成新的真实DOM，随后替换掉页面中之前的真实 DOM**

  b.旧虚拟DOM中未找到与新的虚拟DOM相同的key创建新的真实的DOM，随后渲染到页面

- 用index作为key可能发生的问题
  - 若对数据进行逆向添加，逆序删除等破坏顺序操作，会产生没有必要的真实DOM 效率慢
  - 若结构中还包含输入类的DOM:会产生错误的DOM更新，导致界面出错
- 开发中如何选择key
  - **最好选择使用每条数据的每一标识作为key，比如id,手机号等**
  - 如果不存在对数据的逆序添加，逆序删除等破坏顺序的操作，仅用于渲染列表，使用index作为key是没有问题的

10.列表过滤

1.实现列表的查询功能

1.基于watch的实现

这里有个输入框的限制，所以可以通过一个watch的属性进行监视，注意一个细节，就是开始的时候要进行一次查询，因为原来的filterPersons中是什么都没有的，所以要先查询一次

这里是通过检测n和keyword的变化来实现功能，注意html的不等号和等号的区别，这里是通过一个副本来实现操作

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>HelloVue</title>
        <script src="../../js/vue.min.js"></script>
    </head>
    <body>
        <div id="root">
            <input type="text" placeholder="请输入关键词" v-model:value="keyWord"/>
            <button @click="n=2">升序排序</button>
            <button @click="n=1">降序排序</button>
            <button @click='n=0'>原顺序</button>
            <ul>
                <li v-for="(item,index) in filterPersons" :key="item.id">
{{item.name}}--{{item.age}}--{{item.sex}}</li>
            </ul>
        </div>
    </body>
    <script type="text/javascript">
        Vue.config.devtools=true
        new Vue({
            el:'#root',
            data:{
                keyWord:'',
                n:0,
                persons:[
                    {id:"001",name:'马冬梅',age:13,sex:"女"},
                    {id:'002',name:'周冬雨',age:24,sex:"女"},
                    {id:'003',name:'周杰伦',age:15,sex:"男"},
                    {id:'004',name:'文绍伦',age:19,sex:"男"},
                ],
                filterPersons:[]
            },
            watch:{
                keyWord: {
                    immediate:true,
                    handler(newValue){
                        this.filterPersons = this.persons.filter((p)=>{
                            return p.name.indexOf(newValue)!==-1
                        })
                    }
                },
                n:{
                    immediate: true,
                    handler(newValue) {
```

```
                        if(newValue){
                            this.filterPersons.sort((p1,p2)=>{
                                return newValue===1?p1.age-p2.age:p2.age-p1.age
                            })
                        }
                        else{
                            this.filterPersons = this.persons.filter((p)=>{
                                return p.name.indexOf(this.keyWord)!==-1
                            })
                        }
                    }
                }
            }
        })
    </script>
</html>
```

2.基于computed的实现

这里的keyWord和n值的改变是通过函数或者标签固有的方式进行动态的变化，从而使得`filterPersons`的变化

```
computed:{
    filterPersons:{
        get(){
            const arr = this.persons.filter((p)=>{
                return p.name.indexOf(this.keyWord)!==-1
            })
            arr.sort((p1,p2)=>{
                if(this.n){
                    return this.n===1?p1.age-p2.age:p2.age-p1.age
                }
            })
            return arr
        }
    }
}
```

11.数据监测原理

对象和数组分清

1.重现vue的data的get和set的方法重现

这个是一个单层对象实例，无法检测到深层的数据检测

```
let data={
    name:'feng',
    address:'杭州',
}
function Observer(obj){
    const keys = Object.keys(data)
    //foreach的参数是一个函数
```

```
    keys.forEach((k)=>{
        //k是index的参数，这里的this就是data的实体
        Object.defineProperty(this,k,{
            get(){
                return obj[k]
            },
            set(val){
                obj[k] = val
            }
        })
    })
}
console.log(new Observer(data))
```

2.原理：

- vue会检测data中的所有数据，深层的检测数据 重新渲染页面的效果
- 如何检测对象中的数据

  通过setter实现检测，且要在new vue时候就要传入要检测的数据
  - 对象创建后追加的属性，vue默认不做响应式处理
  - 如需给后面添加的属性做响应式，使用下面的API

    > Vue.set(target,propertyName/index,value)
    >
    > vm.$set(target,propertyName/index,value)

注意 Vue.set和this.$set的方法不能给vm或者vm的数据对象(data等)添加属性

通过控制v-if标签来动态显示数据的出现

```
methods:{
    addProperty(){
        // Vue.set(this.student,'sex','男')
        this.$set(this.student,'sex','男')
    }
}
```

如何检测数组中的数据

- 通过包裹数组更新元素的方法实现，本质就是做了两件事
  - 通过原生对应的方法对数组进行更新
  - 重新解析模板，进而更新页面

在vue修改数组中的某个元素一定要用如下方法 [数组操作]

> push() pop() unshift() shift() **splice()** sort() reverse() 这几个方法被Vue重写了

两种方法改变数组中的值

```
addProperty(){
    this.student.friends.unshift({name:'jerry',age:13})
   Vue.set(this.student.friends,this.student.friends.length,{name:'jerry',age:13})
},
```

通过filter来过滤掉相关的属性

```
removeSmoke(){
    this.student.hobby = this.student.hobby.filter((h)=>{
        return h != '抽烟'
    })
}
```

12.收集表单数据

收集表单数据

- 若 `<input type="text"/>`，则 `v-model` 收集的是 `value` 的值，用户输入的内容就是 `value` 的值
- 若 `<input type="radio"/>`，则 `v-model` 收集的是 `value` 的值，且要给标签配置 `value` 的属性
- 若 `<input type="checkbox"/>`
    - 没有配置 `value` 属性，那么收集的是 `checked` 的属性(勾选or未勾选，是布尔值)
    - 配置了 `value` 属性
        - `v-model` 的初始值是非数组，那么收集的就是 `checked` (勾选or未勾选，是布尔值)
        - `v-model` 的初始值是数组，那么收集到的就是 `value` 组成的数组

`v-model` 的三个修饰符

- `lazy` 是失去焦点后在收集数据
- `number` 输入字符串转为有效的数字　这里可以配置**input的type="number"**的属性来使用
- `trim` 输入的字符串首位去掉空格

```
<body>
    <div id="root">
        账号：<input type="text" v-model:value.trim="username"><br/><br/>
        密码：<input type="text" v-model:value="password"><br/><br/>
        年龄：<input type="number" v-model:value.number="age"><br/><br/>
        性别：
        男：<input type="radio" v-model:name="sex" value="male">
        女：<input type="radio" v-model:name="sex" value="female"><br/><br/>
        爱好：
        学习：<input type="checkbox" v-model:name="hobby" value="study">
        游戏：<input type="checkbox" v-model:name="hobby" value="game">
        程序：<input type="checkbox" v-model:name="hobby" value="programming">
<br/><br/>
        所属校区：
        <select v-model:name="school">
            <option>请选择校区</option>
            <option value="beijing">北京</option>
            <option value="shanghai">上海</option>
            <option value="hangzhou">杭州</option>
        </select><br/><br/>
        其他信息：
        <textarea v-model:name.lazy="userInfo"></textarea><br/><br/>
        <input type="checkbox" v-model:name="agree">接受<br/><br/>
        <button>提交</button>

    </div>
```

```
</body>
<script type="text/javascript">
    Vue.config.devtools = true
    new Vue({
        el:"#root",
        data:{
            username:'',
            password:'',
            age:'',
            sex:'male',
            hobby:[],
            school:'请选择校区',
            userInfo:'',
            agree:'',
        }
    })
</script>
```

13.过滤器 (Vue3废除)

定义:对要显示的数据进行特定格式化后再进行显示(适用于简答的逻辑处理)

注册过滤器:

- `Vue.filter(name,callback)` 这里的callback是一个函数　全局过滤器
- `new Vue{filters:{}}`　局部过滤器

使用过滤器: `{{xxx | 过滤器名}}` 或 `v-bind:属性="xxx | 过滤器名"`

备注:

- 过滤器可以接受额外参数，多个过滤器也可以串联
- 并没有改变原本的数据，而是产生新的对应的数据

```
<body>
    <div id="root">
        <p>{{msg | MySlice | Mys}}</p>
    </div>
    <div id="root1">
        <p>{{msg | Mys}}</p>
    </div>
</body>
<script type="text/javascript">
    Vue.config.devtools = true
    Vue.filter('Mys',(val)=>{
        return val.slice(0,1)
    })
    new Vue({
        el:'#root',
        data:{
            msg:"hello"
        },
        filters:{
            MySlice(val){
                return val.slice(0,3)
            }
        }
    })
```

```
    new Vue({
        el:'#root1',
        data:{
            msg:"hello"
        },
    })
</script>
```

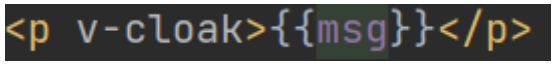
14.内置指令

1. `v-text` `v-html`

- 作用：向其所在的节点中渲染文本内容
- 与插值语法的区别：`v-text`会替换掉节点中的内容，{{XXX}}则不会，`v-html`会使html语法生效，这个有安全问题

2. `v-cloak`

- 本质就是一个特殊属性，Vue实例创建完毕或并接管容器后，会删除`v-cloak`属性
- 使用css配合`v-cloak`可以解决网速慢时页面展示{{XXX}}的问题



3. `v-once`

- `v-once`所在的节点在初次动态渲染后，就视为静态属性了
- 以后数据的改变不会引起`v-once`所在结构的更新，可以视为优化性能

4. `v-pre`

- 跳过其所在的节点的编译过程
- 可利用他跳过，没有使用指令语法，没有使用插值语法的节点，会加快编译

```
<p v-pre>{{n}}</p>
```

5.自定义指令

- 函数式　element是绑定的html元素　binding是绑定的vue的数据

1.指令与元素成功绑定时

2.指令所在元素被插入到页面时

3.指令所在的模板被重新解析时

```
<p v-big:value="n"></p>
<p v-small:value="n"></p>
```

```
directives:{
    big:function (element,binding){
        element.innerHTML = binding.value*10
    },
    small(element,binding){
        element.innerHTML = binding.value*3
    }
}
```

- 对象式

3个回调函数

- bind(element,binding)　指令与元素成功绑定的时候
- inserted(element,binding)　指令所在元素被插入页面时调用
- update(element,binding)　指令所在模板结构被重新解析时候调用

`element`就是`DOM`元素，`binding`就是要绑定的对象，他包含以下属性：`name` `value` `oldValue` `expression` `arg` `modifiers`

```
<input type="text" v-fmodel:value="n">
```

```
fmodel:{
    bind(element,binding){
        element.value = binding.value*3
    },
    inserted(element,binding){
        element.focus()
    },
    update(element,binding){
        element.value = binding.value*3
    }
}
```

备注:

- 指令定义时不加`v-`，但使用时候要加上`v-`
- 指令名如果时多个单词，要使用`kebab-case`命名方式，不要用`camelCase`命名

1. `kebab-case`

```
<br/><br/>
<input type="text" v-f-model:value="n">
```

```
'f-model':{
    bind(element,binding){
        element.value = binding.value*3
    },
    inserted(element,binding){
        element.focus()
    },
    update(element,binding){
        element.value = binding.value*3
    }
}
```

2.全局指令函数

```
Vue.directive('small',function (element,binding){
    element.innerHTML = binding.value*3
})
```

```
Vue.directive('small',{
    bind(element, binding) {
        element.innerHTML = binding.value*3
    },
    inserted(element,binding){
    },
    update(element,binding){
        element.innerHTML = binding.value*3
    }
})
```

15.Vue生命周期

1.定义

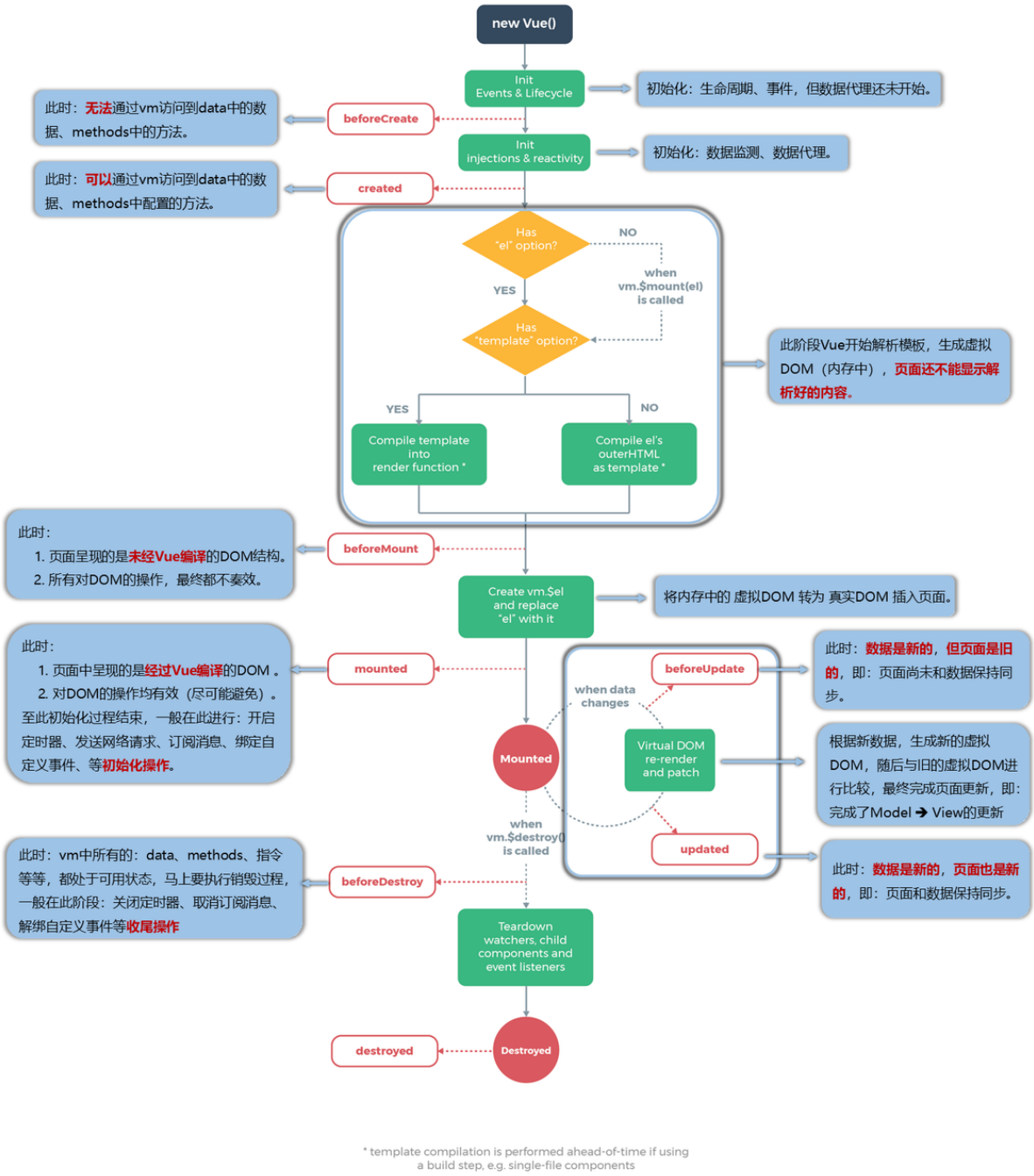
- 又名：生命周期回调函数、生命周期函数、生命周期钩子
- 是什么：Vue在关键时期帮我们调用的一些特殊名称的函数
- 生命周期函数的名字不可更改，但是函数的具体内容是程序员根据需求编号的
- 生命周期函数中的`this`的指向是`vm`或`组件实例对象`

2.实现透明度的动态切换

```
<body>
    <div id="root">
        <h1 v-bind:style="{opacity: opacity}">杭州你好</h1>
    </div>
</body>
<script type="text/javascript">
    Vue.config.devtools = true
    new Vue({
        el:'#root',
        data:{
            opacity:1,
        },
        mounted(){
            setInterval(()=>{
                this.opacity -= 0.1
                if(this.opacity<=0) this.opacity = 1
            },100)
        }
    })
```

```
</script>
```

3.生命周期图




```
<body>
    <div id="root" v-bind:x="n">
        <h1 v-bind:style="{opacity: opacity}">杭州你好</h1>
        <h2 v-text="n"></h2>
        <h2>当前的n的值是：{{n}}</h2>
        <button @click="add">点我n++</button>
        <button @click="bye">点我去掉vm实例</button>
    </div>
</body>
<script type="text/javascript">
```

```
    Vue.config.devtools = true
    new Vue({
        el:'#root',
        data:{
            opacity:1,
            n:1,
        },
        methods:{
            add(){
                this.n++;
            },
            bye(){
                this.$destroy()
            }
        },
        watch:{
            n:function (){
                console.log("我被修改了")
            }
        },
        mounted(){
            console.log("mounted")
            this.id = setInterval(()=>{
                this.opacity -= 0.1
                if(this.opacity<=0) this.opacity = 1
            },100)
        },
        beforeCreate(){
            console.log("beforeCreate")
        },
        created(){
            console.log("created")
        },
        beforeMount(){
            console.log("beforeMount")
        },
        beforeUpdate(){
            //联动整个Vue的实例，只要其中一个属性变化之后，这个函数就调用
            console.log("beforeUpdate")
        },
        updated(){
            console.log("updated")
        },
        beforeDestroy(){
            clearInterval(this.id)
            console.log("beforeDestroy")
        },
        destroyed(){
            console.log("destroyed")
        }
    })
</script>
```

控制台上动态控制主体元素

```
beforeCreate
created
super_copy_cracked:false
> vm.$mount('#root')
beforeMount
< ▶ Cn {_uid: 0, _isVue:
>
```

返回主体的html元素

 image-20220323180821344

用new Vue中的template向root中添加元素，注意这里一定要有`<div></div>`包裹

```
const vm = new Vue({
    el:'#root',
    data:{
        opacity:1,
        n:1,
    },
    template:`
      <div>
      <h2 v-text="n"></h2>
      <h2>当前的n的值是：{{n}}</h2>
      <button @click="add">点我n++</button>
      <button @click="bye">点我去掉vm实例</button>
      </div>
    `,
```

总结：

常用的生命周期钩子

- `mounted`发送ajax请求，启动定时器，绑定自定义事件，订阅消息等初始化操作
- `beforedestroy`清除定时器，解除自定义事件，取消订阅消息等收尾工作

关于销毁`Vue`实例

- 销毁后借助`Vue`开发者工具看不到任何消息
- 销毁后自定义事件会失效，但是原生DOM事件依然有效
- 一般不会在`beforeDestory`操作数据，因为即使操作数据，也不会再触发更新流程了

16.Vue组件化编程

1.概念

1.模块

- 理解：向外提供特定功能的js程序，一般就是一个js文件
- 为什么：js文件很多很复杂
- 作用：简化，复用js的编写，提高js的运行速率

2.组件

- 定义：用来实现局部功能的代码和资源的集合
- 为什么：一个界面的功能很复杂
- 作用：复用代码，简化项目的代码，提高效率

3.模块化

- 当应用的js都是以模块来编写的，那么这个应用就是一个模块化的应用

4.组件化

- 当应用中的功能都是多组件的方式来编写的，那么这个应用就是一个组件化的应用

4.非单体文件组件

- 非单体文件组件：一个文件中有多个组件
- 单体文件组件：一个文间中仅包含一个组件

5.基本使用

定义组件

使用`Vue.extend(options)`创建，其中`options`和`new Vue(options)`时传入的`options`几乎一样，但也有区别

- `el`不要写，因为最终所有的组件都要经过一个`vm`的管理，有`vm`中的`el`决定服务那个容器
- `data`必须要写成函数，避免组件被复用时，数据存在引用关系，使得数据独立起来

注册组件

- 局部注册：`new Vue()`的时候options传入`components`选项
- 全局注册：`Vue.component('组件名',组件)`

使用组件

- 编写组件标签，这个标签就是你在注册时候的名字

```
<body>
    <div id="root">
        <h1>{{msg}}</h1>
        <student></student>
        <hr/>
        <School></School>
        <student></student>
    </div>
    <div id="root2">
        <hello></hello>
    </div>
</body>
<script type="text/javascript">
```

```
    Vue.config.devtools = true
    const hello = Vue.extend({
        template:`
<div>
<h1>{{name}}</h1>
    </div>
`,
        data(){
            return {
                name:'feng',
            }
        }
    })
    Vue.component('hello',hello)
    const student = Vue.extend({
        template:`
<div>
<h1>{{name}}</h1>
<h1>{{age}}</h1>
    </div>
`,
        data(){
            return {
                name:'feng',
                age:17,
            }
        }
    })
    const school = Vue.extend({
        template:`
<div>
<h1>{{name}}</h1>
<h1>{{address}}</h1>
    </div>
`,
        data(){
            return {
                name:'中国计量大学',
                address:'杭州',
            }
        }
    })
    new Vue({
        el:'#root',
        data:{
            msg:'Hello',
        },
        components:{
            school:school,
            student:student,
        }
    })
    new Vue({
        el:'#root2',
    })
</script>
```

```
function data(){
    return {
        a:1,
        b:1
    }
}
let c = data()
let d = data()
```

上述函数实现代码复用的效果，且对象之间没有影响

2.组件的注意事项

关于组件名

- 一个单词组成
  - 第一种(首字母小写)：school
  - 第二种(首字母大写)：School （推荐)
- 多个单词组成
  - 第一种(`kebab-case`命名)：my-school
  - 第二种(`CamelCase`命名)：MySchool(需要Vue的脚手架)
- 备注：
  - 组件名尽可能回避`html`中已有的元素名称
  - 可以使用`name`配置项自定组件再开发者工具中呈现的名字

关于组件标签

- 第一种写法：`<school></school>`
- 第二种写法：`<school/>`
- 备注：不使用脚手架的时候。`<school/>`会导致后续组件不能渲染

一种简写方式：`const school = Vue.extend(options)`可简写为`const school = options`，因为父组件`components`引入的时候会自动创建

3.组件的嵌套

直接在对应的component中定义components的options的属性，在template中使用

```
const hello = {
    name:'你好',
    template:`
        <div>
        <student></student>
          <h1>{{name}}</h1>
        </div>
    `,
    data(){
        return {
            name:'feng',
        }
    },
    components:{
        student
    }
}
```

4.VueComponent

- `school`组件本质就是一个名为`VueComponent`的构造函数，且不是程序员定义的，而是`Vue.extend()`生成的
- 我们只需要`<school/>`或`<school></school>`，`Vue`解析时会帮助我们建立school组件的实例对象，即`Vue`帮我们执行的`new VueComponent(options)`
- 每次调用Vue.extend,返回的都是一个全新的`VueComponent`,即不同组件是不同的对象
- 关于this的指向：
  - 组件配置中的`data`函数，`methods`中的函数，`watch`中的函数，`computed`中的函数他们的**this**均是`VueComponent`的实例对象
  - `new Vue(options)`配置中：`data`函数，`methods`函数，`watch`函数，`computed`函数，他们的**this**均是`Vue实例对象`
- `VueComponent`的实例对象，以后简称`vc(组件实例对象)`，`Vue的实例对象`，简称`vm`

5.一个重要的内置关系

这里的prototype指的是原型函数，这里所有的原型都是**Object**

- 一个重要的内置关系：`VueComponent.prototype.__proto__ === Vue.prototype`
- 这个关系可以使实例对象`vc`可以访问到`Vue原型`(这个是内置的原型)上的属性，方法

6.单文件组件

0.模板

```
<template>

</template>
<script>

</script>
<style>

</style>
```

1.Student.vue

```
<template>
<div>
    <h2>学生姓名：{{name}}</h2>
    <h2>学生年龄：{{age}}</h2>
    </div>
</template>

<script>
    export default {
        name:'Student',
        data() {
            return {
                name:'cess',
                age:20
            }
        },
    }
</script>
```

2.School

```
<template>
<div id='Demo'>
    <h2>学校名称：{{name}}</h2>
    <h2>学校地址：{{address}}</h2>
    <button @click="showName">点我提示学校名</button>
    </div>
</template>

<script>
    export default {
        name:'School',
        data() {
            return {
                name:'UESTC',
                address:'成都'
            }
        },
        methods: {
            showName(){
                alert(this.name)
            }
```

```
        },
    }
</script>

<style>
    #Demo{
        background: orange;
    }
</style>
```

3.App.vue

```
<template>
<div>
    <School></School>
    <Student></Student>
    </div>
</template>

<script>
    import School from './School.vue'
    import Student from './Student.vue'

    export default {
        name:'App',
        components:{
            School,
            Student
        }
    }
</script>
```

4.main.js

```
import App from './App.vue'

new Vue({
    template:`<App></App>`,
    el:'#root',
    components:{App}
})
```

5.index.html

```
<!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>单文件组件练习</title>
    </head>
    <body>
        <div id="root"></div>
        <script src="../../js/vue.js"></script>
        <script src="./main.js"></script>
```

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

Vue导出文件的方法

```
export school
export {school}
export default school
```

vue导入文件的方法

```
import name from "相对路径"
```

17.Vue-cli

0.项目的绝对路径，内置的

`<%= BASE_URL %>    //public的绝对路径`

1.导入Vue-cli

- 配置淘宝镜像：`npm config set registry http://registry.npm.taobao.org`
- 全局安装：`npm install -g @vue/cli`
- 切换到建立项目的目录，使用语句`vue create 项目名字`
- 选择vue的版本
- 启动vue的项目`npm run serve`
- 打包项目`npm run bulid`

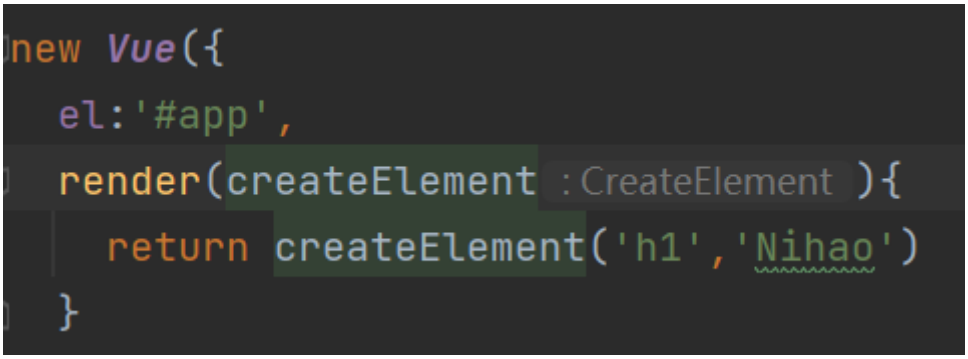
> Vue脚手架隐藏了所有webpack的相关配置，想要看的话就是用语句`vue inspect>output.js`

2.main.js

其中默认导入的vue文件夹中的js文件是一个残缺的vue.js文件，其中没有vue的渲染工具，要想使用这个工具，向上面一样配置文件，就需要使用完整版的vue.js

`import Vue from 'vue/dist/vue'`

其中有个函数叫做hender，这个函数是在渲染框架函数



3.关于不同版本的vue.js

- vue.js与vue.runtime.xxx.js的区别
  - vue.js是完整版的vue,包含核心功能+模板解析器
  - vue.runtime.js是运行版的vue，只有核心功能，没有模板解析器
- 因为vue.runtime.xxx.js中没有模板解析器，所以不能使用`template`配置项，需要使用`render`函数接受到的`createElement`函数去指定具体内容

4.vue.config.js的配置　**最好就不要修改**

`vue inspect>output.js`可以查看到vue脚手架的默认配置

使用`vue.config.js`可以对脚手架进行个性化定制，和`package.json`同级目录，这个可以在官网看见配置，在下面的这个模板修改

```
const { defineConfig } = require('@vue/cli-service')
module.exports = defineConfig({
    transpileDependencies: true,
    lintOnSave:false,
    pages:{
        index:{
            entry:'src/feng.js'
        }
    }
})
```

18.ref属性

ref属性被用来给元素或子组件注册引用信息(id的替代者)

- 应用在`html`标签上获取的是真实的`DOM`元素，应用在组件标签上获取的是组件实例对象`vc`
- 使用方式
  - 打标识：`<h1 ref='XXX'></h1>` 或 `<School ref='xxx'/>`
- 获取ref中的值是通过`this.$refs.xxx`

19.props配置项

props让组件接收外部传过来的数据

- 传递数据`<Demo name='xxx' :age="18"/>`，这里传递的数据就只能是字符串，这里的`age`前加的`:`，通过v-bind使得里面18是数字，而不是字符串的类型
- 接受数据
  - 第一中方法：`props:['name','age']`
  - 第二种方法：`props:{name:String,age:Number}`
  - 第三种方式(限制类型，限制必要性，指定默认值)(type,required,default)

```
props:{
    name:{
      type:String,
      required:true,
    },
    age:{
      type:Number,
      default:18,
    }
 }
```

备注：`props`是只读的，`Vue`的底层会检测你对`props`的修改，如果进行了修改，就会发出警告，若业务需求确实要修改，那么请复制到`props`的内容到`data`中，然后去修改`data`中的数据

20.mixin配置项

1.功能：可以把多个组件共用的配置提取成一个混入对象

2.使用方法：

- 定义混入：

定义mixin.js

```
export const mixin  = {
    methods:{
        addAge(){
            this.myAge++
        }
    },
}
```

在相关组件中引入mixin项

```
import {mixin} from "../mixin"
mixins:[mixin]     //在vue的实例中定义属性mixin
```

- 全局混入

在main.js中引入，全部的vc中都有这个方法

```
import {mixin} from "./mixin"
Vue.mixin(mixin)
```

备注

- 组件和混入对象含有同名选项时候，这些选项将以恰当的方式合并，在发生冲突时以组件优先
- 同名生命周期钩子将合并为一个数组，因此都将被调用。另外，混入对象的钩子将在组件自身钩子之前调用

20.plugin配置项

1.功能：用于加强`Vue`

2.本质：包含`install`方法的一个对象，`install`的第一个参数就是`Vue`，第二个以后的参数时插件使用者传递的参数，这里可以传递很多的参数

3.定义插件

```
export const plugin = {
    install(Vue){
        Vue.filter('mySlice',(value)=>{
            return value.slice(0,1)
        })
        Vue.mixin({
            methods:{
                addAge(){
                    this.myAge++
                }
            },
        })
    }
}
```

4.使用插件

`Vue.use()`在main.js中使用

```
import {plugin} from "./plugin";    //最好就是加上这个大括号
Vue.use(plugin)
```

21.scoped样式

1.作用：让样式子在局部生效，防止冲突

2.写法：`<style scoped></style>`

22.ToDo-list案例

组件化编程流程

1.拆分静态组件：组件要按照功能点拆分，命名不要和`html`中的元素冲突

2.实现动态组件：考虑好数据的存放的位置，数据是一个组件在用，还是一些组件在

- 一个组件在用：放在组件自身即可
- 一些组件在用：放在他们共同的父组件上(状态提升)

3.实现交互

`props`适用于

- 父组件===>子组件的通信
- 子组件===>父组件的通信 (要求父组件先给子组件一个函数)

使用`v-model`时要切记：`v-model`绑定的值不能是`props`传过来的值，因为`props`是不可修改的

`props`传过来的若是对象类型的值，修改对象中的属性时`Vue`不会报错，但是不推荐这样做

成功的项目在github上找　todolist　这里可以使用浏览器中的储存来存储数据

23.本地存储

1.WebStorage(js本地存储)

存储内容大小一般支持5MB左右(不同浏览器可能还不一样)

浏览器通过`Window.sessionStorage`和`Window.localStorage`属性来实现本地存储机制

相关的API

- `xxxStorage.setItem('key','value')`该方法接受一个键值对作为参数，会把键值对添加到存储中，如果键名存在，则更新器对应的值
- `xxxStorage.getItem('key')`该方法接受一个键名作为参数，返回键名对应的值
- `xxxStorage.removeItem('key')`该方法接受一个键名作为参数，并把该键名从存储中删除
- `xxxStorage.clear()`该方法会清空存储中的所有数据

备注:

- `SessionStorage`存储的东西会随着浏览器窗口的关闭而消失
- `localStorage`存储的东西，需要手动清除才会消失
- `xxxStorage.getItem(xxx)`，如果xxx的对应的value获取不到，那么`getItem()`的返回值事`null`
- `JSON.parse(null)`的结果依然事`null`

24.组件的自定义事件

- 一种组件间通讯的方式，适用于`子组件====>父组件`
- 使用场景：自组件想给的父组件床底数据，那么就要在父组件给组件绑定自定义事件(事件的回调函数)
- 绑定自定义事件
  - 第一方式：在<父组件中`<Demo @事件名='方法'/>`或`<Demo v-on='事件名'='方法'/>`
  - 第二种方式：在父组件中`ths.$ref.demo.$on('事件名','方法')`
  - 若想让自定义事件只能触发一次，可以使用`.once`修饰符(就像@click.once)，或`$once()`方法
- 触发自定义事件`this.$emit('事件名',数据)`　**...params是可以接受多个参数**
- 解绑自定义事件`this.$off('事件名')`，解绑多个事件可以使用数组，没参数的时候全部事件解绑
- 组件上也可以绑定原生`DOM`事件，需要使用`native`修饰符，`@click='方法'`，上面绑定了自定义事件，即使绑定的是原生事件也会被认为是自定义的，需要加`native`，加上后就将此事件给组件的根元素，就可以使用了原生的**DOM**事件
- 注意：通过`this.$refs.xxx.$on('事件名',回调函数)`绑定自定义事件时，回调函数要么配置在`methods`中，要么使用箭头函数，否则`this`的指向会出问题，谁触发绑定事件，this就是谁，箭头函数没有自己的实例对象

父组件中使用自定义事件

```
<!-- 通过父组件给子组件绑定一个自定义事件实现子给父传递数据
<Student @feng="getStudentName" @demo="m1"/>
```

子组件通过`$emit`触发身上绑定的事件

```
this.$emit( event: 'feng',this.name,666,888,900)
```

父组件的函数

```
getStudentName(name, ...params) {
  console.log('App收到了学生名：', name, params)
  this.studentName = name
},
```

通过ref的实现

```
mounted() {
  this.$refs.student.$on( event: 'feng', this.getStudentName)
```

销毁vc组件，这里的this就是vc组件本身

```
this.$destroy()
```

25.全局事件总线 important

_是参数的占位符

定义：一种可以在任意组件间通信的方式，本质就是一个对象，他必须满足一下条件

- 所有的组件对象都必须能看见他
- 这个对象必须能够使用`$on`,`$emit`,`$off`方法去绑定，触发和解绑事件

满足以上条件有`vc`和`vm`的实例对象

使用步骤

1.定义全局事件总线

定义一个js文件，里面建立一个Vue的实例，因为这个Vue的实例中有相关的API使用

```
import Vue from "vue";
const eventBus = new Vue()
export default eventBus
```

2.绑定组件事件

绑定事件最好在事件执行`mounted`方法的时候执行进行绑定事件，箭头函数这样的this就是**vm实例**

```
import eventBus from "@/event-bus";
mounted() {
    eventBus.$on("Hello", (data) => {
        console.log("我是School组件，收到了数据", data);
    });
},
```

3.触发绑定事件

```
import eventBus from "@/event-bus";
methods: {
    sendStudentName(){
        eventBus.$emit('Hello', this.name)
    }
}
```

4.最后记得解绑组件事件

```
beforeDestroy() {
    eventBus.$off("Hello");
},
```

26.$nextTick important

这个是一个生命周期钩子

方式：`this.$nextTick(回调函数)`在下一次`DOM`更新结束后执行其指定的回调函数

什么时候用：当改变数据后，要基于更新后的新`DOM`进行某些操作时，要在`nextTick`所指定的回调函数中执行

27.动画效果

28.Ajax的跨域问题

1.Vue脚手架配置代理

本案例需要下载`Axios`库，在文件的终端下输入`npm install axios`

这里开启vue-cli的代理服务器，需要参考文档

`Vue.config.js`是一个可选的配置文件，如果项目的(和`package.json`同级的)根目录中存在这个文件，那么他会被`@vue/cli-service`自动加载。你也可以使用`package.json`中的`Vue`字段，但是注意这种写法需要你严格遵照JSON的格式来写

方法一：在`Vue.config.js`中配置如下

```
devServer:{
    proxy:'http://localhost:5000'
}
```

App.vue

```
methods:{
    getStudents() {
      axios.get("http://localhost:8080/students").then(
          response=>{
            console.log("请求成功了",response.data)
          },
          error=>{
            console.log("请求失败了",error.message)
          }
      )
    },
}
```

方式二：

在这个方法中用了一个虚拟路径来实现路径的转换，这个方式可以配置多个代理服务器

- `pathRewrite`：表示这个路径进行重写
- `ws`：用于支持websocket,默认值为true
- `changeOrigin`：用于控制请求头中的host值，默认值为true　改变之后就是target的路径

```
devServer:{
    proxy:{
        '/feng':{
            target:"http://localhost:5000",
            pathRewrite:{'/feng':''},//重写的路径
            ws:true,
            changeOrigin:true
        },
    }
}
```

App.Vue

```
methods:{
    getStudents() {
      axios.get("http://localhost:8080/feng/students").then(
          response=>{
            console.log("请求成功了",response.data)
          },
          error=>{
            console.log("请求失败了",error.message)
          }
      )
    },
}
```

在这个vue中引入了axios库，来实现请求和跨域的解决方法

29.github案例

这个就是一个按照名字来动态搜索用户

源码在github

30.slot插槽

`<slot>`插槽：让父组件可以向子组件指定位置插入`HTML`结构，也是一种组件间通信的方式，适用于**父组件=>子组件**

在这节中最好就套上一个`<template></template>`的标签来使用

1.默认插槽

```
父组件：
<CateGory title="美食">
    <ul>
        <li v-for="(item,index) in foods" :key="index">{{item}}</li>
    </ul>
</CateGory>
子组件：
<slot></slot>
```

2.具名插槽

在`template`中还可以使用`v-slot:center`的写法

```
父组件：
<CateGory title="美食">
    <ul slot="center">
        <li v-for="(item,index) in foods" :key="index">{{item}}</li>
    </ul>
</CateGory>
子组件：
<slot name="center"></slot>
```

3.作用域插槽 `slot-scope 或 scope`

`scope`用于父组件往子组件插槽放的`html`结构接收子组件的数据

理解：数据在子组件的自身，但是根据数据生成的结构需要组件的使用者(父组件)来决定

```
父组件：
<CateGory title="美食">
    <template slot-scope="feng">
        <ul>
        <li v-for="(item,index) in feng.foods" :key="index">{{item}}</li>
        </ul>
    </template>
</CateGory>
子组件：     数据在子组件中
<slot :foods="foods"></slot>
```

31.vuex

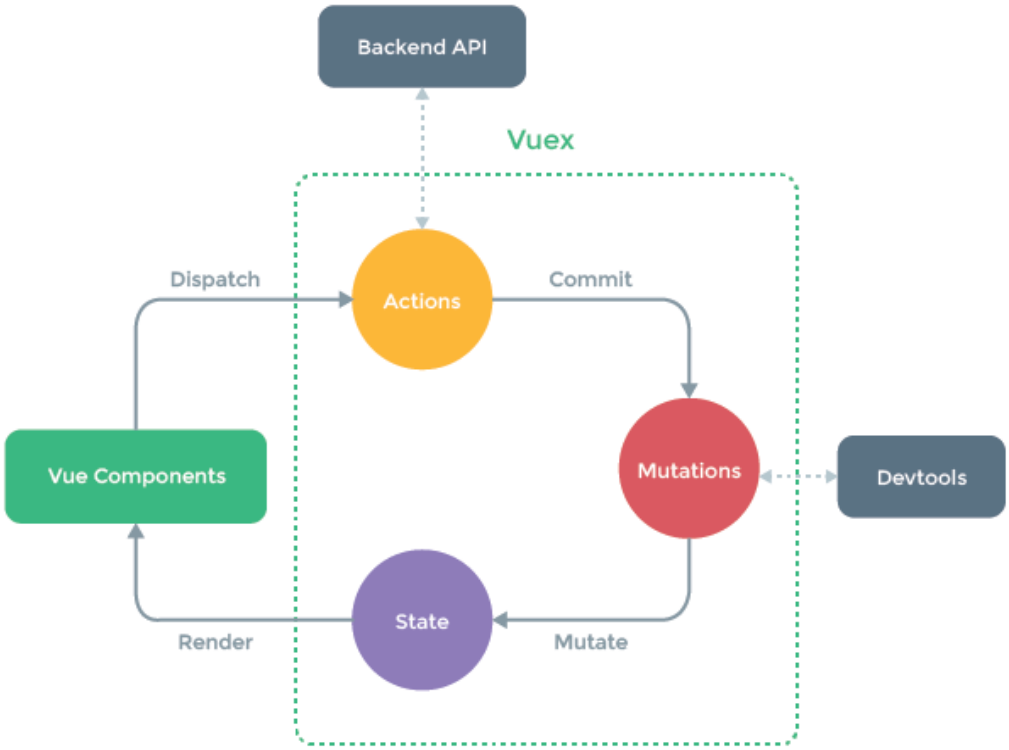
0.搭建vuex的环境

`npm i vuex@3`在`vue2`中是这样的，但是在`vue3`时是`npm i vuex@4`

1.概念

专门在Vue中实现集中式管理数据的一个插件，也是一种组件间的通信的方式，且适用于任意组件间的通信

2.工作原理



3.简单的使用

`store/index.js`

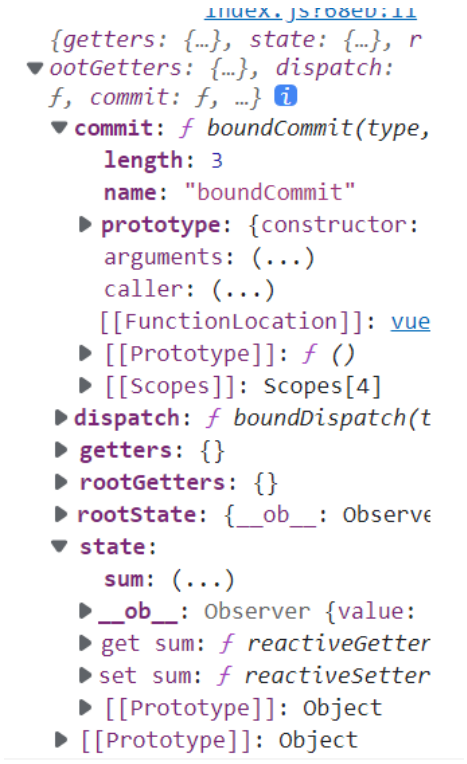
```
import Vuex from 'vuex'
import Vue from "vue";
Vue.use(Vuex)
const actions = {
    addOdd(context,value){
        if(context.state.sum%2){
            context.commit('ADD',value)
        }
    },
    addWait(context,value){
        setTimeout(()=>{
            context.commit('SUB',value)
        },1000)
    }
}
const mutations = {
    ADD(state,value){
        console.log("nia")
        state.sum += value
    },
    SUB(state,value){
        state.sum -= value
```

```
    }
}
const state = {
    sum:0
}
const store = new Vuex.Store({
    actions:actions,
    mutations:mutations,
    state:state
})
export default store
```

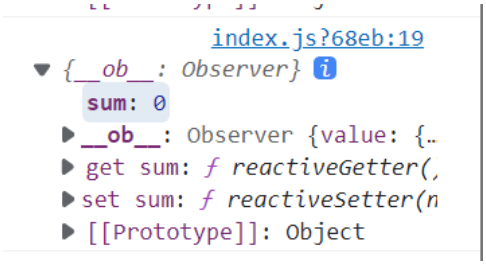
`count.vue`

```
methods:{
    add(){
        this.$store.commit('ADD',this.now)
    },
    sub(){
        this.$store.commit('SUB',this.now)
    },
    oddAdd(){
        this.$store.dispatch('addOdd',this.now)
    },
    addWait(){
        this.$store.dispatch('addWait',this.now)
    }
}
```

`context` 可以链式编程



`state` === data



4.getters配置项 === computed

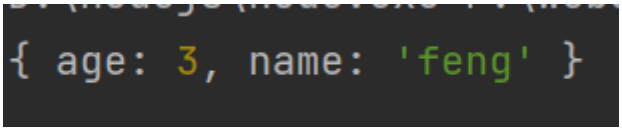
```
index.js
const getters={
    bigSum(state){
        return state.sum*10
    }
}
组件中
{{$store.getters.bigSum}}
```

5.mapState与mapGetters

- mapState:用于帮助映射`state`中的数据为计算属性
- mapGetters:用于帮助映射`getters`中的数据为计算属性

这里的...的作用是es6的语法

```
let person = {age:3,name:'feng'}
let persons = {
    ...person
}
console.log(persons)
```




```
computed: {
    ...mapState(['sum']),    //数组写法
    ...mapState({sum:'sum'}),  //对象写法
    ...mapGetters(['bigSum']),
    ...mapGetters({bigSum:'bigSum'})
},
```

6.mapActions与mapMutations

- mapActions用于帮助生成与actions对话的方法，即包含$store.dispatch(xxx)的函数
- mapMutations用于帮助生成与mutations对话的方法，即包含$store.commit(xxx)的函数

方法调用时的参数，入口传递参数，在绑定事件的时候传递参数

```
<button @click="ADD(now)">+</button>
<button @click="SUB(now)">-</button>
<button @click="addOdd(now)">当前求和为奇数再加</button>
<button @click="addWait(now)">等一等再加</button>
```

```
methods:{
    ...mapMutations({add:'ADD',sub:'SUB'}),    //对象
    ...mapMutations(['ADD','SUB']),    //数组
    ...mapActions({oddAdd:'addOdd',addWait:'addWait'}),
    ...mapActions(['addOdd','addWait'])
},
```

7.分模块编程

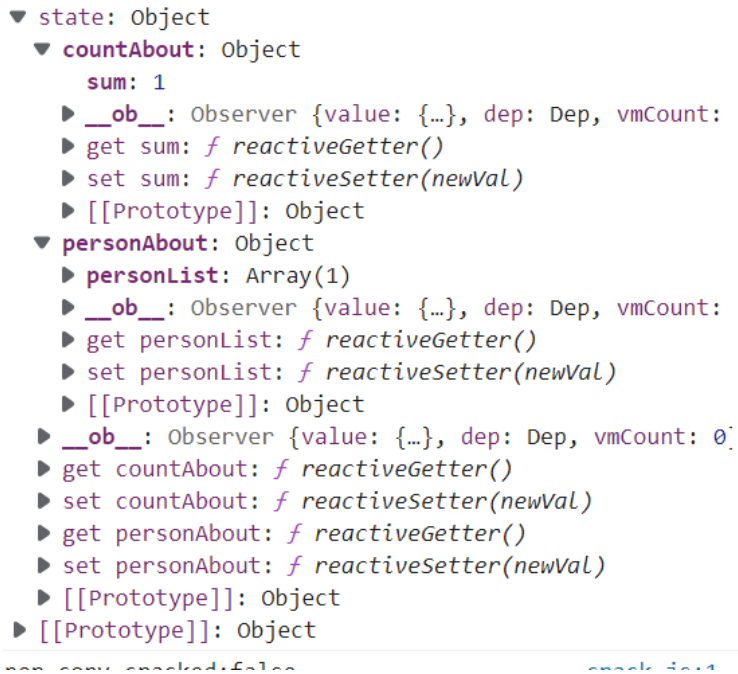
1.目的：让代码更好维护，让多种数据分类更加明确

2.修改`index.js`，为了解决不同模块命名冲突，将不同模块的`namespaced:true`，之后再不同页面中引入`getter` `actions` `mutations`时，需要加上所属的模块名，这里还可以分文件编写

```
const countAbout = {
  namespaced: true, // 开启命名空间
  state: {x:1},
  mutations: { ... },
  actions: { ... },
  getters: {
    bigSum(state){ return state.sum * 10 }
  }
}

const personAbout = {
  namespaced: true, // 开启命名空间
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    countAbout:countAbout,
    personAbout:personAbout
  }
})
```



读取`state`中的数据

```
// 方式一：自己直接读取
this.$store.state.personAbout.list
// 方式二：借助mapState读取：
...mapState('countAbout',['sum','school','subject']),
```

读取`getters`中的数据

```
//方式一：自己直接读取
this.$store.getters['personAbout/firstPersonName']
//方式二：借助mapGetters读取：
...mapGetters('countAbout',['bigSum'])
```

调用`dispatch`

```
//方式一：自己直接dispatch
this.$store.dispatch('personAbout/addPersonWang',person)
//方式二：借助mapActions：
...mapActions('countAbout',{incrementOdd:'jiaOdd',incrementWait:'jiaWait'})
```

调用`commit`

```
//方式一：自己直接commit
this.$store.commit('personAbout/ADD_PERSON',person)
//方式二：借助mapMutations：
...mapMutations('countAbout',{increment:'JIA',decrement:'JIAN'}),
```

32.路由

1.基本路由

- 导入`vue-router` `npm i vue-router@3`在vue2中，在vue3中是`npm i vue-router@4`
- 应用插件`Vue.use(VueRouter)` 在main.js中使用
- 编写`router`的配置项

```
import VueRouter from "vue-router";
import AbOut from "@/components/AbOut";
import HoMe from "@/components/HoMe";
export default new VueRouter({
    routes:[
        {
            path:'/about',
            component:AbOut
        },
        {
            path:'/home',
            component:HoMe
        }
    ]
})
```

- 实现切换

    - `<router-link></router-link>`浏览器会被替换成`a`标签
    - `active-class`可配置高亮样式 **注意这里的active一定是class的选择器的样式**
    - `<router-link style="text-decoration: none" active-class="active" to="/about">About</router-link>`
- 指定展示位置`<router-view></router-view>` 这个会替换掉成组件中的内容

2.几个注意点

- 路由组件通常放在`pages`文件夹中，一般组件通常存放在`components`文件夹中
- 通过切换，'隐藏'了路由组件，默认是被销毁的，需要的时候再去生成
- 每个组件都有自己的`$route`属性，里面存储者自己的路由信息
- 整个应用只有一个`router`，可以通过组件的`$router`属性获取到

3.多级路由

- 配置路由规则，使用`children`配置项

```
{
    path:'/home',
        component:HoMe,
            children:[    //配置子路由
                {
                    path:'news',      //不加斜杠，写成'/news'
                    component:NeWs
                },
                {
                    path:'message',
                    component:MessAge
                }
            ]
},
```

- 跳转的路径要写完整

```
<router-link style="text-decoration: none" active-class="active"
to="/home/news">News</router-link>
```

4.路由传参

- 传递参数 (注意这个冒号的使用)

```
方式一：  字符串写法
<router-link :to="`/home/messages/detail?id=${item.id}&name=${item.name}`"
active-class="active">{{item.name}}</router-link>
方式二：  对象写法
<router-link :to="{path:'/home/news/detail',query:{id:item.id,name:item.name}}"
active-class="active">{{item.name}}</router-link>
```

- 接受参数

```
{{$route.query.id}}
{{$route.query.name}}
```

5.命名路由

- 作用：可以简化路由的旋转
- 如何使用：
  - 给路由命名

```
{
    name:'about',
        path:'/about',
            component:AbOut
},
```

  - 简化跳转 (注意这个冒号的使用)

```
<router-link active-class="active" :to="{name:'about'}">About</router-link>
```

6.params参数 restful风格

- 配置路由，声明接受 `params` 参数

```
{
    path:'messages',
        component:MessAge,
            children:[
                {
                    path:'detail/:id/:name',  //占位符
                    component: MessageDetail
                }
            ]
}
```

- 传递参数

```
方式一：    //字符串拼接
<router-link :to="`/home/messages/detail/${item.id}/${item.name}`" active-
class="active">{{item.name}}</router-link>
方式二：   //对象传递参数
<router-link :to="{name:'newsDetail',params:{id:item.id,name:item.name}}"
active-class="active">{{item.name}}</router-link>
```

注意：路由携带 `params` 参数时，若使用 `to` 的对象写法，则不能使用 `path` 配置项，必须使用 `name` 配置项

- 接受参数

```
{{$route.params.id}}
{{$route.params.name}}
```

7.路由的props配置

- props作用：让路由组件更方便的收到参数

```
方式一：    该对象中的所有的key-value的组合最终都会通过props传给组件，死数据
props:{a:900}
方式二：    该值为true时，则把路由收到的所有params参数通过props传给组件，query不行
props:true
方式三：    props是函数。该函数返回的对象中每一组key-value都会通过props传给组件，都行
props(route){
    return{
        id:route.params.id,
        name:route.params.name
    }
}
```

- 组件接受配置

```
props:['id','name']
```

8.router-link的replace的属性 (栈记录历史) 默认属性是push

- 作用：控制路由跳转时操作浏览器历史记录的模式
- 浏览器的历史记录有两种写入方式：`push`和`replace`
  - `push`是追加记录
  - `replace`是替换当前记录，路由跳转时候默认是`push`方式
- 开启`replace`模式 直接顶掉前面一个记录，加入自己的记录
  - `<router-link :replace='true'></router-link>`
  - 简写为`<router-link replace ></router-link>`

总结:浏览器记录本质就是一个栈，默认是`push`，点开新页面就会再栈顶追加一个地址，后退，栈顶指针向下移动，改为`replace`就是不追加，而将栈顶地址替换

9.编程式路由导航

- 作用:不借助`<router-link>`实现路由跳转，让路由跳转更加灵活

函数:

- this.$router.`push({})` 内传递的对象是与对象写法的`<router-link>`中的to是一样的
- this.$router.`replace({})` 和上面的一样
- this.$router.`forward()` 前进
- this.$router.`back()` 后退
- this.$router.`go(n)` 可前进也可后退 n为正数前进n，为负数后退n

```
this.$router.push({
    name:'newsDetail',
    params:{
        id:item.id,
        name:item.name
    }
})
this.$router.replace({
    name:'newsDetail',
    params:{
```

```
        id:item.id,
        name:item.name
    }
})
this.$router.back()
this.$router.forward()
this.$router.go(n)
```

10.缓存路由组件 (不生效) 等待实验

- 作用：让不展示的路由组件保持挂载，不被销毁，include的参数是组件名字，多个直接逗号分隔
- 写法：`<keep-alive include=""><router->view></router-view></keep-alive>`

```
<keep-alive include="NeWs">
    <router-view></router-view>
</keep-alive>
<keep-alive exclude="NeWs">
    <router-view></router-view>
</keep-alive>
```

11.activated和deactivated Vue-router的生命周期钩子

- `activated`和`deactivated`是路由组件所独有的两个钩子，用于捕获路由组件的激活状态
- 具体使用：
    - `activated`：路由组件被激活时触发
    - `deactivated`：路由组件失活时触发

12.路由守卫

1.作用：对路由进行权限控制

2.分类：全局守卫，独享守卫，组件内守卫

1.全局守卫

在对应的router的js文件中配置的对应的路由信息，就在meta中配置`meta:{key:value}`，在对应的route上

1.`router.beforeEach((to,from,next)=>{})`

全局前置守卫：初始化时，每次路由切换前执行

参数1：是当前url要跳转的去向

参数2：是当前url的路径

参数3：将路由跳转进行下去

```
router.beforeEach((to,from,next)=>{
    if(to.meta.isAuth){
        if(localStorage.getItem("name")==='feng1'){
            next()
        }else{
            alert("没有权限")
        }
    }else{
        next()
    }
})
```

2. `router.afterEach((to,from)=>{})`

全局后置守卫，初始化时，每次路由切换后执行

参数1：是当前url要跳转的去向

参数2：是当前url的路径

```
router.afterEach((to)=>{
    document.title=to.meta.title||'枫'
})
```

2.独享路由守卫

1. `beforeEnter(to,from,next){}` 这个写在meta的位置处

```
beforeEnter(to,from,next){
    if(localStorage.getItem('name')==="feng1"){
        next()
    }else{
        alert("没有权限")
    }
},
```

3.组件内守卫

在对应的vue组件中书写

1. `beforeRouterEnter(to,from,next){}`

进入守卫，通过路由规则，进入该组件时被调用

```
beforeRouteEnter(to,from,next){
    if(localStorage.getItem('name')==='feng'){
        next()
    }else{
        alert("没有权限")
    }
},
```

2. `beforeRouteLeave(to,from,next){}`

离开守卫，通过路由规则，离开该组件时被调用

```
beforeRouteLeave(to,from,next){
    console.log("再见")
    next()
}
```

13.路由器的两种工作模式

1.对于一个`url`来说，什么是`hash`值?

`#`及其后面的内容就是`hash`值

2.`hash`值 不会包含在`HTTP`请求中，即：`hash`值不会带给服务器

3.`hash`模式

- 地址中永远带着`#`号，不美观
- 若以后将地址通过第三方手机app分享，若手机app校验严格，则地址会被标记为不合格
- 兼容性好

4.`history`模式

- 地址干净，美观
- 兼容性和`hash`模式相比略差
- 应用部署上线时需要后端人员支持，解决刷新页面服务器404的问题

```
const router =  new VueRouter({
    mode:'history',
    routes:[...]
            })
    export default router
```

通过`npm run build`生成对应的js,css,html界面，在项目的dist文件夹中

33.Vue UI组件库

- Element UI　　Element-UI

1.引入element-ui组件库

`npm i element-ui -S`

2.在main.js中引入element-ui组件库

```
import Vue from 'vue'
import ElementUI from 'element-ui'
import 'element-ui/lib/theme-chalk/index.css'
Vue.use(ElementUI)
```

3.element-ui按需引入

- 安装babel-plugin-component `npm i babel-plugin-component -D`
- 修改`babel-config-js`文件

```
module.exports = {
  presets: [
    '@vue/cli-plugin-babel/preset',
    ["@babel/preset-env", { "modules": false }]
```

```
  ],
  plugins: [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "theme-chalk"
      }
    ]
  ]
}
```

- `main.js`中

```
import Vue from 'vue'
import App from './App.vue'
import Router from "@/router";
import {Button,Row} from 'element-ui'
Vue.config.productionTip = false
Vue.use(Button)  //Vue.component(Button.name,Button)
Vue.use(Row)     //Vue.component(Row.name,Row)
new Vue({
  el:'#app',
  render:h=>h(App),
  router:Router,
})
```