支持vue3.0脚手架工具 安装的命令

npm i -g @vue/cli

创建vue

vue create 项目名

组件相关

组件分类： 页面级组件、业务上可重复的基础组件、与业务无关的独立组件

vue组件创建方法：

<script>

const app = Vue.**createApp**({

**data**(){

    return{

    }

  }

})

*// 注册全局组件*

*// 全局组件可在当前实例中任意位置使用*

app.**component**("Mybox",{

  template:`

    <div style="width:150px;height:100px;background-color:#faf"></div>

  `

})

*// 另一个全局组件*

*// 如果两个全局组件想要相互只用，需要在对应的组件的模板中调用*

app.**component**("box1",{

  template:`

    <div style="width:350px;height:200px;background-color:#f00">

      <Mybox></Mybox>

      </div>

  `

})

app.**mount**('#app')

vue的局部组件只能用在注册其的页面上

eg:有两个子组件被vue实例引用了，这个vue实例可以用这两个子组件，但是这两个子组件不能相互使用，如果a子组件想用b子组件，那么需要在a中注册一下

let b={

  template:`

    <div style="width:350px;height:200px;background-color:#f00">

      <a></a>

      </div>

  `,

  components:{

    a

  }

}

**组件传参：**

注意:组件传参的属性名不能使用驼峰式，因为浏览器都会将其解析成小写

**父→子**

父组件中：

在调用子组件的时候 通过动态绑定属性的方式，将数据传过去，属性名和数据名最好一样

 <div *id*="app">

    <h1>父传子</h1>

    <box *:colleges*="msg2" *:branch*="branch"></box>

  </div>

子组件中

在子组件中通过prop来接收父组件传过来的数据，prop里是父组件传过来的属性名，要加引号

 const Box = {

      template: "#box",

*// 接收父祖件的数据*

      props: ["colleges", "branch"]

    }

**prop的类型**

1.只指定名称

 props: ["colleges", "branch"]

2.指定名称和类型

注意此时的prop已经是对象了

 props:{

        colleges:**String**,

        branch:**Number**

      }

3.指定名称/类型/必要性/默认值

props:{

        colleges:{type:**String**,required:true,default:xxx},

        branch:{type:**Number**,required:true,default:xxx},

      }

prop中数据的类型有

String/Number/Boolean/Array/Object/Date/Function/Symbol/自定义类型

**子→父**

使用场景，子组件更改某个操作希望通知父组件，让父组件做出操作、或子组件像父组件传递数据

子组件中

创建子组件，当子组件里的按钮被点了触发btnClick事件，然后通过**$emit**会发射一个事件

    const box = {

      template: "#box",

      methods:{

**btnClick**(){

**alert**("你是🐕🐕\*10")

*// 发射自定义事件*

*//****这里发射事件的名称不能是驼峰命名***

*// 两个参数，第一个是事件名称，第二个是数据*

*// 多个数据的传递一般采用对象或者数组的方式*

*// this.$emit("send-click","我是恁爹的数据","1","2")*

*// 可以发射多个不同的自定义事件*

          this.**$emit**("send-click",{name:"张三",age:13})

        }

      }

    }

父组件中

父组件绑定子组件发射的自定义事件，当子组件一发射这个事件，那么就会执行绑定的函数

<div *id*="app">

    <box *@send-click*="clickFun"></box>

</div>

//methods中

  methods:{

*// 接收父组件自定义事件是没有事件对象的*

*// 子组件自定义函数中的形参不是事件对象，而是子组件传来的值*

**clickFun**(val){

           console.**log**("子组件的事件被点击了");

           console.**log**(val);

        },

      }

**组件间的访问**

父组件访问子组件

在调用子组件的时候需要为其设置一个ref名称(可在任意标签中设置，用this.$refs.ref名访问即可)，代表该子组件

父组件：

<div *id*="app">

*<!-- 这里需要给子组件绑定一个ref，代表当前组件的名称 -->*

  <lk-box *ref*="box1"></lk-box>

  <lk-box *ref*="box2"></lk-box>

  <lk-box *ref*="box3"></lk-box>

  <button *@click*="getChildrenComponent">click me </button>

</div>

在调用该子组件的父组件中通过this.$refs.ref名访问

父组件：

methods:{

**getChildrenComponent**(){

*// $refs每个实例都会有*

*// this.$refs.box1表示拿到box1组件了，然后可以其经行操作*

     console.**log**(this.$refs.box1.msg);

*//  this.$refs.box1.btnClick()*

     this.$refs.box2.msg="我偷偷的更改了信息呢"

    }

  }

子组件访问父组件(不建议使用，破坏了子组件的封闭性)

设有两个组件a和b，组件a引用了组件b，此时组件a就是组件b的父组件，如果组件b想要访问组件a，直接**this.$parent（表示父组件a）**即可，对其内容 **.** 操作即可。

访问根组件：**this.$root**

  const Btn = {

      template: `

      <button @click="plus">点击了{{count}}</button>

    `,

**data**() {

        return {

          count: 0,

        }

      },

      methods: {

**plus**() {

          this.count++;

*// this.$parent代表引用当前子组件的父组件*

          console.**log**(this.$parent.a);*//哈哈哈*

*// console.log(this.$parent.$parent.msg);//郑工学院*

*// this.$root表示访问根组件*

          console.**log**(this.$root.msg);

        }

      }

    }

**插槽 v-slot（匿名插槽和具名插槽）**

插槽是组件里面的一个知识点，所谓插槽就是在子组件中预留一个位置，然后父组件在调用子组件的时候，可以在这块位置经行操作。(比如说，一个页头好多地方用但是里面的细节不同需要单独设置，但是整个结构是不变的)

**匿名插槽**

子组件：

在子组件中用<slot></slot>占位，slot标签中可以写东西，当父组件没有插入东西时，就使用默认的

 <template *id*="box">

    <div *style*="width: 200px;height:150px;background-color: #f00;">

      <p>淘！淘你喜欢！</p>

*<!-- <button>淘一下</button> -->*

*<!-- 预留插槽 -->*

      <slot>插槽的默认内容</slot>

    </div>

  </template>

  <script>

    const hhBox={

      template:"#box",

    }

  </script>

父组件：

父组件在调用子组件的时候，在标签内部写东西，就会替换掉子组件的slot标签

<div *id*="app" >

  <hhbox>

    <button>淘一下</button>

  </hhbox>

  <hhbox>

    <input *type*="text" *placeholder*="请输入内容">

  </hhbox>

  <hhbox>

</div>

**具名插槽**

子组件

将每个插槽命名

 <template *id*="box">

*<!-- 头部 -->*

    <header>

      <slot *name*="header">头部插槽</slot>

    </header>

*<!-- 主体 -->*

*<!-- 此插槽是匿名插槽 -->*

    <main>

      <slot >主体插槽</slot>

    </main>

*<!-- 脚部 -->*

    <footer>

      <slot *name*="footer">脚部插槽</slot>

    </footer>

    <p>======================================</p>

  </template>

  <script>

    const MyBox = {

      template: "#box",

    }

父组件

使用v-slot:插槽名即可将对应的内容替换到插槽中，template标签会被内容替换掉不占用位置

*匿名插槽和具名插槽的相互使用：*

*在子组件中，有匿名插槽也有具名插槽，在父组件中，具名插槽用v-slot：插槽名占用，用v-slot:default的模板会自动替换到匿名插槽中*

<div *id*="app">

    <mybox>

      <template *v-slot:header*>

        <button>我是头部</button>

      </template>

*<!-- 使用v-slot:default可以将此模板替换为子组件的匿名插槽 -->*

      <template *v-slot:default*>

        <input *type*="text" *placeholder*="我是主体">

      </template>

      <template *v-slot:footer*>

        <img *src*="1.jpg">

      </template>

    </mybox>

  </div>

**渲染作用域**

结论：父级模板里的所有内容都是在父级作用域中编译的

 子级模板的所有内容都是在子级作用域中编译的

<div *id*="app">

*<!-- div会显示，说明用的数据是父组件里面的 -->*

*//box是子组件 子组件和父组件中都有*isShow，但是会调用父组件中的数据

  <box *v-show*="isShow"></box>

</div>

**作用域插槽：**

子组件在预留插槽的时候，通过绑定属性的方式将子组的数据暂存到当前插槽中，父组件在使用插槽的时候通过变量接住数据，即可实现父组件中可以调用子组件的数据（直接在视图层经行操作），$refs必须实在js中经行操作

简而言之：数据是子组件的，由父组件传结构过来

子组件

通过绑定的属性的方式将数据绑定到插槽中

<template *id*="box">

*<!-- data:属性名，可自定义，即父组件接收数据对象的键名，nameArr是值， -->*

  <slot *:data*="nameArr"></slot>

</template>

<script>

 const box = {

   template:"#box",

**data**(){

     return{

       nameArr:["张三","李四","王五","赵六"]

     }

   }

 }

父组件

在调用子组件的时候，通过具名或者匿名插槽的方式，在其后面给个变量接住传过来的值，变量名可自定义,这个变量是一个对象，接住了子组件传来的数据

<box>

//v-slot:default表示调用匿名插槽，也可以用具名插槽

    <template *v-slot:default*="slotProps">

*<!-- slotProps代表子组件插槽传来的数据对象 -->*

*<!-- { "data": [ "张三", "李四", "王五", "赵六" ] } -->*

      <div> {{slotProps}}</div>

    </template>

  </box>

**动态组件**

当组件之间，切换的时候，有时候会想保持这些组件的状态，以避免反复重复渲染导致的性能问题

根据数据的变化，结合component标签，来动态切换组件的显示

用is属性动态绑定组件，

eg：按钮组件和输入框组件只能显示其一

首先创建两个全局组件

 app.**component**("c-button", {

      template: `

    <button>百度一下</button>

  `,

    });

    app.**component**("c-input", {

      template: `

  <input type="text" placeholder="请输入内容"/>

  `,

    })

然后在调用该组件

通过component标签和is属性，is后面的变量名称与哪个组件名相同，就会显示哪个组件

keep-alive标签：失活组件可以缓存

失活：比如切换输入框显示/隐藏的时候切换前输入的内容会没有，用来这个标签就不会出现这种情况

<div *id*="app">

    <keep-alive>

      <component *:is*="cItem"></component>

    </keep-alive>

    <p>-----------</p>

    <button *@click*="isShow">切换</button>

  </div>

//vue实例中

**data**() {

        return {

          cItem: "c-button",

        }

      },

**异步组件**

在大型应用中，我们可能将应用分割成一些小块的代码，并且只在需要的时候才从服务器加载一个模块

Vue中通过Vue.defineAsyncComponent来创建里面必须用Promise语法

   app.**component**("async-item", Vue.**defineAsyncComponent**(() => {

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

**setTimeout**(() => {

**resolve**({

            template: `

                <div style="background-color:#faf;">这是一个从服务器端异步拿过来的全局组件的数据</div>

              `,

          })

        }, 2000)

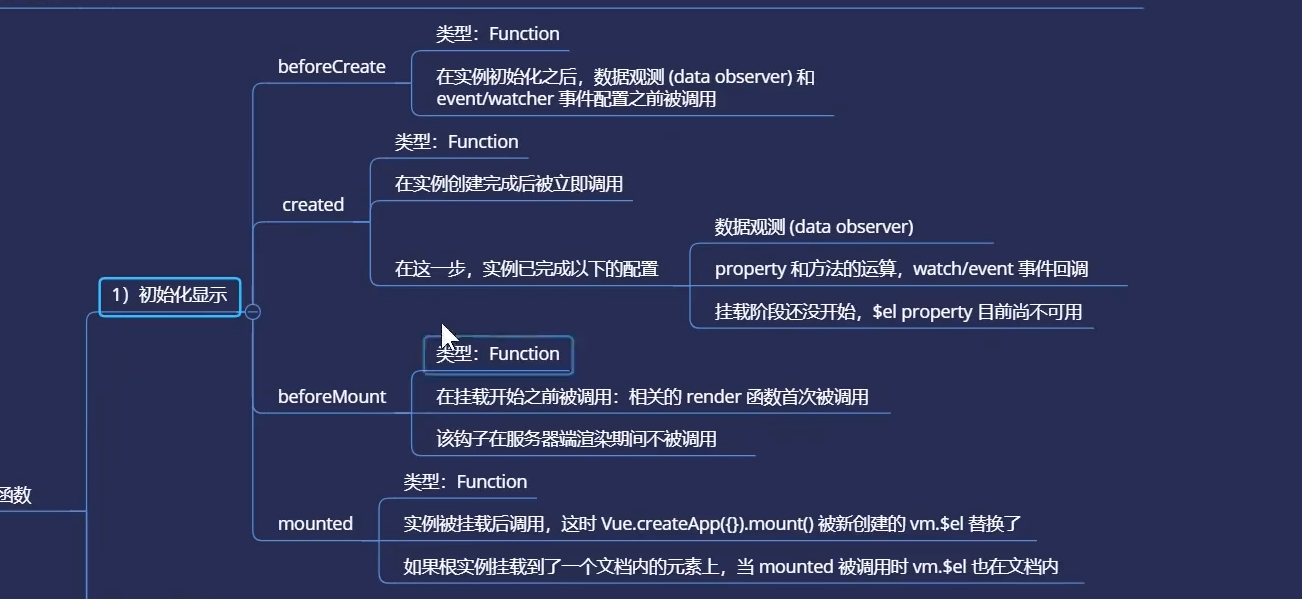
      })

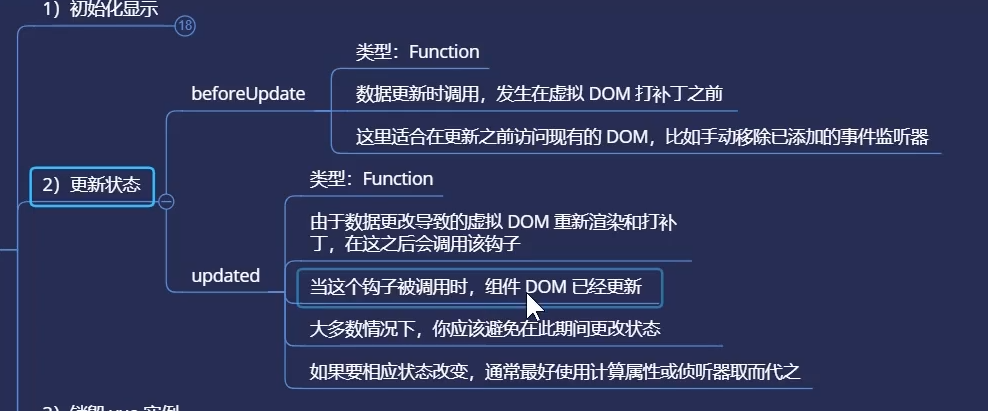
    }

    ))

**生命周期函数**

还有一些没写出来，官方手册查询





**侦听器watch**

注意：要监听的数据和函数要同名，

const app = Vue.**createApp**({

**data**(){

    return{

        price:5,

    }

  },

  watch:{

*// 监听属性有两个参数 current：现在的值 prev之前的值*

**price**(current,prev){

      console.**log**(current,prev);

    }

  },

计算属性和监听器的区别：

0.计算属性底层是靠watch实现的

1.计算属性在调用时需要在模板中渲染

2.计算属性默认深度依赖，watch默认浅度依赖

3.计算属性适合做筛选，不可异步，watch适合执行异步或开销较大的操作

**单项数据流**

vue中组件传参是单向数据

单项数据流：在Vue中都是组件化开发，比如父组件的数据要给子组件用，也要给孙组件以及更深层次的子组件，这时要保证子组件不能修改数据，否则其他的数据都会变，这就是数据单向流

**组件传参的快捷语法**

注意：传参的时候以对象的方式，对象的键名要与接收参数的组件的props里面的名称相对应，然后在要调用的子组件标签以v-bind的方式传入

 <div *id*="app">

    <p>键名必须和子组件中props中的名称对应</p>

    <box *v-bind*="numObj"></box>

*<!-- 使用简洁语法底层解析是这样的 -->*

*<!-- <box :num1="numObj.num1" :num2="numObj.num2" :num3="numObj.num3"></box> -->*

  </div>

  <script>

    const box = {

    props:["num1","num2","num3"],

      template: `

      <div>数据1:{{num1}}</div>

      <div>数据2:{{num2}}</div>

      <div>数据3:{{num3}}</div>

    `,

    }

    const app = Vue.**createApp**({

**data**() {

        return {

          numObj:{

            num1:100,

            num2:200,

            num3:300

          }

        }

      },

      components: {

        box

      }

**no-props**

正统的数据传递，父组件在调用子组件的时候，通过属性绑定的方式将数据传递给子组件，子组件通过props接收即可，但是当想要传递一个样式或者class类的时候，用no-props传递比较简单，就是子组件内不用props接收

父组件：

<div *id*="app">

  <box *msg*="我是" *style*="background-color:red"></box>

</div>

创建全局子组件：

1.父组件传值过来，子组件不用props接收，那么就会子组件的模板的元素中添加对应的属性

2.一般用no-props给子组件传递样式或class,如果子组件没有props接收数据，这时就会多出来样式属性

<div msg="我是" style="background-color:red">我是全局子组件</div>

3.默认如果子组件内部只有一层包裹就会作用到爹身上，如果有多个平级元素no-props不生效

4.如果子组件的模板中有多个同一级别的标签，在每个标签上v-bind="$attrs",使得当前标签no-props生效

5.如果父组件传来多个属性，只需要个别，使用 :自定义名称="$attrs.想要的传来的属性名" eg: :style="$attrs.style"

6.$attrs其实就是一个对象，用来接收传来的所有属性

7.使用inheritAttrs:false,不接受父组件传来的属性

app.**component**("box",{

*// props:["msg"],*

*// 不接受no-props属性*

*// inheritAttrs:false,*

  template:`

  <div :s1="$attrs.msg">我是全局子组件</div>

  <div  v-bind="$attrs">我是全局子组件</div>

  `,

})

动画

**自定义动画**

vue中使用自定义动画和过渡跟css中差不多，也是在style中定义动画或过渡，然后通过动态绑定class的方式调用

<div *id*="app">

    <div>

      <div *class*="box" *:class*="animates">

        {{msg}}

      </div>

      <button *@click*="animates.animate=!animates.animate">开始/结束</button>

    </div>

  </div>

**Vue内部封装好的动画**

想使用Vue内部封装的动画，必须用<transition></transition>包裹住要使用动画的元素

因为Vue封装的动画会作用在transition标签内，如果有多个此标签将会全部都应用，为了避免这个，给transition标签加上name属性，对应的v-xxx也要改为 自定义名称-xxx

eg:lk-enter-active

**动画/过渡动画**

进场动画类

.v-enter-from{…} 进来的开始

.v-enter-active{…} 进来的中间时刻

.v-enter-to{…} 进来的结束

离场动画类

.v-leave-from{…} 离开的开始

.v-leave-active{…} 离开的中间时刻

.v-leave-to{…} 离开的结束

以上两大类需要哪个就调用哪个即可

<div *id*="app">

    <div>

*<!-- 使用transition标签包裹，动画会作用内部，但是多个此标签，希望有不同的效果 -->*

*<!-- 给transition标签添加一个name属性 -->*

*<!-- 那么相应的将v-enter-xxx,前面的v改为name对应的值  eg:lk-enter-active -->*

      <transition *name*="lk">

        <div *class*="box" *v-show*="flag">{{msg}}</div>

      </transition>

      <button *@click*="flag=!flag">显示/隐藏</button>

    </div>

  </div>

动画和过渡结合使用

如果动画和过渡结合使用，要保证两个的时长不能差太多，如果是想以某一个为主，（动画时长4秒，过渡时长1秒，系统会自动综合两者，这时如果想以过渡为主，过渡一结束，动画就结束，那么就在transition标签加type属性值为transition反之加animation）

此外transition还有一个:duration=”毫秒数”属性 值是毫秒数，规定了动画和过渡的时长

:duration的值还可以更加的详细

:duration={enter:1000,leave:3000}

表示：动画和过渡的时长进入是1秒，离开是3秒

第三方动画库的使用

这里使用的是<https://animate.style/>的动画库

这里transition标签又多出来一些属性—作用跟进场离场动画类一样

 <transition

*enter-from-class*="类名"

*enter-active-class*="类名"

*enter-to-class*="类名"

*leave-from-class*="类名"

*leave-active-class*="类名"

*leave-to-class*="类名"

    >

    </transition>

**js的方法实现动画/过渡**

Vue在transition标签中提供了js钩子，每个钩子对应一个函数，在函数中对元素经行操作,含义参考上面进场/退场动画类

@before-enter="beforeEnter"

@enter="enter"

@after-enter="afterEnter"

@before-leave="beforeLeave"

@leave="leave"

@after-leave="afterLeave"

:css=false 禁用css3动画/过渡

每个钩子函数都有一个参数，表示当前调用动画/过渡的元素

注意：其中enter和leave有两个参数，第二个参数必须调用才能触发@after-enter/@after-leave

<transition

*@before-enter*="berforeEnter"

*@enter*="enter"

*@after-enter*="afterEnter"

*:css*=false>

      <div *v-show*="flag" *class*="box">{{msg}}</div>

    </transition>

 methods: {

**berforeEnter**(el) {

*// el 表示当前调用动画的标签*

        },

**enter**(el, done) {

*// done函数必须调用才会执行@afterEnter方法*

      done()

},

**afterEnter**(el) {

        }

      }

**多组元素之间的过渡/动画**

设transition标签中有两个元素，一个是按钮，一个是文本框，点击切换按钮，让一个隐藏一个显示，但是两个动画/过渡会重复，

解决：在transition添加mode属性

取值：

in-out: 新元素先进行过渡，完成之后当前元素过渡离开。

out-in: 当前元素先进行过渡，完成之后新元素过渡进入

appear：表示页面一加载就会实现动画效果

<div *id*="app">

<div>

*<!-- appear : 页面加载就会实现一次动画 -->*

      <transition *mode*="out-in" appear>

        <button *v-if*="flag">click me </button>

        <p *v-else*>我是MT</p>

      </transition>

      <p>-------------</p>

      <button *@click*="change">切换</button>

    </div>

  </div>

**多个组件之间的过渡**

这个跟多个元素过渡一样的

<div *id*="app">

    <div>

      <transition *mode*="out-in" *appear* *:duration*="1000">

        <component *:is*="componentId"></component>

      </transition>

      <p>-------------</p>

      <button *@click*="change">切换</button>

</div>

**列表的过渡**

设，有一个ul,需要添加或者删除li，那么让这个添加或者删除的过程实现动画/过渡效果

将要调用的标签用<transition-group></transition-group>包裹起来，

style中定义的动画效果会在该标签内部实现

  <style>

    .v-enter-from {

      opacity: 0;

      transform: **translateY**(20px);

    }

   ……………

    .v-leave-to {

      opacity: 1;

      transform: **translateY**(-20px);

    }

  </style>

<div *id*="app">

    <div>

      <ul>

        <transition-group>

          <li *v-for*="(item , i ) in list" *:key*="i" *class*="list-item">{{item}}</li>

        </transition-group>

      </ul>

      <p>--------------------------</p>

      <button *@click*="add">增加</button>

      <button *@click*="del">删除</button>

    </div>

  </div>

Vue3高阶

**Mixin**

**在Vue3x版本中用的不多了**

mixin来分发Vue中可重复使用的部分，比如说，在一个实例中的data，methods等另外一个实例也要用，那么就可将要复用的东西拎出来，单独放在一个对象中，在实例中用mixins接住即可，mixins是一个数组，可以放多个mixin对象

<body>

  <div *id*="app">

    <div>

      <h1>{{msg}}</h1>

      <button *@click*="hello">打个招呼</button>

    </div>

  </div>

  <script>

*// 创建一个mixin对象*

    const myMixin = {

**data**() {

        return {

          msg: "你好"

        }

      },

      methods: {

**hello**() {

**alert**("你好")

        }

      }

    }

    const app = Vue.**createApp**({

      mixins:[myMixin]

    }).**mount**('#app')

**mixin细节点**

1.实例中的data，methods之类的选项，优先级高于mixin中定义的

eg:如果mixin对象的数据中有msg变量,当前实例中也有msg变量,那么会优先采用实例中的msg

2.mixin对象中的数据会和实例中的数据混用,

eg:假设mixin对象data中有msg变量,实例中有一个site变量，那么实例在调用这两个变量的时候都会显示

3.如果mixin对象和实例中都有生命周期钩子，两者都会执行，但是会先执行mixin对象中的钩子

**mixin-自定义属性**

有的时候我们会在mixin对象中传入自定义的属性,这些属性不会直接存在data中，所以不能直接使用，而是要在$options中调用

$options 当一个实例加载完毕后，里面的所有内容都会挂载到$options对象中

注意：如果实例中也有一个同名的属性，首先会用实例中的

此外还可以通过配置，优先使用mixin中的属性，具体见一下代码

注意：mixin还可以全局配置，然后当前实例的组件就不需要用mixins接收了，

具体见下面代码

 <div *id*="app">

    <div>

      <h1>{{msg}}</h1>

      <h1>{{$options.age}}</h1>

    </div>

  </div>

  <script>

*// 创建一个mixin对象,里面是自定义属性*

    const myMixin = {

      age:30

    }

    const app = Vue.**createApp**({

      mixins: [myMixin],

**data**() {

        return {

          msg: "你好",

        }

      },

      age:89,

    })

*// 如果说mixin和实例中有同名的属性，默认是先使用实例中的，如果想改变的话，使用一下配置*

    app.config.optionMergeStrategies.**age** = (mixinVal, appVal) => {

      return mixinVal || appVal

    }

*// mixin还可以全局配置，在整个实例中的所有组件都可以用*

*// app.mixin({*

*//   data(){*

*//     return {*

*//       age:100*

*//     }*

*//   }*

*// })*

    app.**mount**('#app')

  </script>

**自定义指令**

可以在全局创建，也可以在局部创建

指令里面的钩子，具体可以去官网手册查询

mounted - 元素插入父 DOM 后发生。

每个钩子里面的参数含义

mounted(el,binding,vnode){…}

el:代表当前使用该指令的元素

binding：指令传来的值

vnode：当前元素节点相关

在调用的时候可以:eg:v-fiexd=”xxx”传递值

全局定义

app.**directive**("focus",{

**mounted**(el) {

    el.**focus**()

  },

})

还可以这样定义，不在钩子函数中使用，直接使用

app.**directive**("focus",(el,binding,vnode)=>{

el.focus()

}

})

局部定义

只能被挂载的组件使用

需要在使用的实例中挂载

 <script>

*// 创建一个局部的自定义指令*

    const myDirective = {

      focus:{

**mounted**(el) {

          el.**focus**()

        },

      }

    }

  </script>

const app = Vue.**createApp**({

  directives:myDirective,

})

调用的时候

v-指令名即可,可传参可不传递(传参的话需要创建的时候接收一下)

<div *id*="app">

  <h1>自定义指令</h1>

  <input *type*="text" *placeholder*="请输入内容" *v-focus*>

</div>

**teleport 传送门**

比如说，父组件中包裹一个子组件，不想让这个子组件显示到父组件上，想让子组件显示到其他地方，这时候就可以用teleport了

将要传送的标签或者组件，用<teleport></teleport>包裹

to属性取值为选择器，

eg：to=”body” to=”#box” to=”.box”

  <div *id*="app">

    <div *class*="box1"></div>

    <div *class*="box">

*<!-- 让蒙版展示到id为box1的元素中 -->*

      <button *@click*="btn">蒙版</button>

      <teleport *to*=".box1">

        <div *v-if*="isShow" *class*="mask">hahah</div>

      </teleport>

    </div>

  </div>

**组合式API** (composition API)

在vue2x的版本中，是是由分散式的API(options API)，就是用到哪个就写哪个，比如用到data，将data写进实例，用到methods，将其写入，一旦代码量多了不容易维护

vue3x中采用了组合式的API，所有相关的数据、方法等等都放在一个函数中，用到哪个函数，通过setup钩子调用即可，增加了代码的可维护性

**setup**

从生命周期的角度来看，steup是取代了2x版本的BeforeCreate(创建前)和create函数

注意点：setup函数里面是不能使用this相关的东西，和实例中的选项因为setup没被挂载，也不能使用生命周期函数

**ref**

让基础类的数据具备响应式，

在composition API中的数据是不具备相应式的，修改数据，视图层不会改变。

在创建数据的时候用ref包裹一下即可

const app = Vue.**createApp**({

**setup**(props,context) {

*// 在composition API 中更改数据，页面中不会变化，不具备响应式*

*// ref ：让基础类的数据具备响应式*

*// 被包裹的数据 底层通过proxy代理，让其具备响应式-->proxy({value:"你好呀！"})*

*// 1.引入ref*

    const {ref} = Vue

*// 用ref将数据包裹*

    let msg = **ref**("你好呀！")

*// 因为被包裹的数据，被proxy代理的成了一个对象，value指向这个对象的值，所以要修改value*

*// 两秒后修改*

**setTimeout**(()=>{

      msg.value="hahah"

    },2000)

    return{

      msg

    }

  }

}).**mount**('#app')

**reactive**

让引用类数据具备响应式

用法同ref

   const app = Vue.**createApp**({

**setup**(props, context) {

*// reactive：让引用类的数据具备响应式*

*// 被包裹的数据 底层通过proxy代理，让其具备响应式-->proxy({name:"张三",gender:"男"})*

*// 1.引入reactive*

        const { reactive } = Vue

*// 用reactive将数据包裹*

        let pObj = **reactive**({ name: "张三", gender: "男" })

        let pArr = **reactive**(["Vue", "React"])

*// 这里直接修改即可，因为地址一样*

**setTimeout**(() => {

          pObj.name = "张某某"

          pArr[0] = "javaScript"

        }, 2000)

        return {

          pObj,

          pArr

        }

      }

    }).**mount**('#app')

**readonly**

让数据只读，不能修改

在composition API 中数据是可以修改的，这违背了Vue单项数据流的初心，为了防止子组件中修改父组件的数据，用readonly将数据包裹即可，

let pObj = readOnly({name:”张三”,age:12})

**toRefs**

让解构的变量也具备响应式，

我们正常解构出来的变量是不具备响应式的，修改数据视图层不会改变，通过toRefs包裹要解构的数据，即可将解构出来的变量具备响应式，（即使解构的对象没有被reactive包裹，当被toRefs包裹后，原对象也会具有reactive被reactive包裹的功能,不建议这样）

**setup**(props, context) {

        const { reactive, toRefs } = Vue

*//* *let pObj = reactive({ name: "张三", gender: "男" })*

let pObj ={ name: "张三", gender: "男" }

**setTimeout**(() => {

          pObj.name = "张某某"

        }, 2000)

*// 将数据从变量结构出来*

*// 将数据从变量结构出来,用toRefs将原对象包裹*

*// 原理，会将对象中的每个属性拿出来在代理包装一下，此时变量就具备响应式了*

*/\**

*proxy({ name: "张三", gender: "男" })---->*

*name:proxy({value:"张三"})*

*gender:proxy({value:"男"})*

*const { name } 🡪 const { proxy({value:"张三"}):name }*

*此时name就变成响应式的了*

*\*/*

        const {name, gender} = **toRefs**(pObj)

        return {

          name,

          gender

        }

      }

**toRef**

有时候不确定传来的数据对象中是否包含某个属性，

eg:针对此类情况如果pObj对象中有age就拿出来赋值给age变量，如果没有就强行添加一个值为undefined的属性age

 let age = **toRef**(pObj, "age")

**context**

在setup中有两个参数，其中一个参数是context，代表上下文对象里面有attrs,slots,emit

attrs:里面存放的是no-props相关的东西

slots：里面存放的是插槽相关的东西

emit:里面存放的是子组件发射的自定义事件

context-attrs

这个里面是no-props相关的东西，比如父组件在调用子组件的时候传入了一个属性，那么通过context.attrs.属性名，即可获取到传入的属性

父组件给子组件传参：

<box s1=”666”></box>

子组件中

setup(props,context){

console.log(context.attrs.s1)

//结果:666

}

context-slots

这个里面存放的是插槽相关的东西，比如子组件中有一个具名插槽box和一个匿名插槽，父组件在具名插槽box使用button按钮占位，匿名插槽使用输入框占位，在子组件的setup函数中，如果想获得匿名插槽相关的东西，就调用**context.slots.default()**,如果是具名插槽，**context.slots.具名插槽名()**

**获得数据是数组，访问里面内容的时候记得加下标**

 const box = {

      template: "#box",

**setup**(props, context) {

        const { slots } = context

        console.**log**(slots.default());//获取的是匿名插槽的内容

console.**log**(slots.box())//获取的是具名插槽box的内容

      },

    }

**context-emit**

主要解决在composition API 中，不能使用this.$emit发射事件也不能用。

只需要将emit 从context中解构出来，用emit代替this.$emit发射事件即可

 function **handel**() {

**alert**("子组件点击")

*// 在composition API中，因为不能用this，所以直接用context中的emit代替this.$emit发射事件*

**emit**("my-son", "我是信息")

        }

**计算属性(computed)的新用法：**

在composition API 中computed用法也发生了改变，仅在创建的时候发生了改变，其他的都没变，创建前 需要从Vue中引入

const app = Vue.**createApp**({

**setup**(props, context) {

*// 引入*

        const { ref, computed } = Vue;

        let num1 = **ref**(10);

*// 计算属性*

        let num2 = **computed**(() => {

          return num1.value \* 10

        })

*// add*

        let **add** = () => {

          num1.value += 10;

        }

*// 将数据抛出*

        return {

          num1, num2, add

        }

      }

    }).**mount**('#app')

此外计算属性的参数也可以是set和get，而且也可以对引用数据类型的值经行修改

 <script>

    const app = Vue.**createApp**({

**setup**(props, context) {

*// 引入*

        const { reactive, computed } = Vue;

*// 值为引用类*

        let num1 = **reactive**({ count: 100 });

*// 计算属性*

        let num2 = **computed**({

**get**() {

            return num1.count \* 10

          },

**set**(res) {

            console.**log**(res);

            num1.count = res / 10

          }

        })

*// 定时器*

**setTimeout**(() => {

          num2.value = 2000

        }, 2000)

*// add*

        let **add** = () => {

          num1.count += 10;

        }

*// 将数据抛出*

        return {

          num1,

          num2,

          add

        }

      }

    }).**mount**('#app')

  </script>

**侦听器的新用法watch**

在composition API 中，侦听器也需要改变一下用法，

代码中包含监听一个属性或者监听多个属性的用法

既可以侦听基础类型的数据也可以侦听引用类型的数据

侦听器的特性：

1. 具有惰性（页面加载的时候不触发，必须在数据变动的情况下才会触发）
2. 更加具体，
3. 可以访问属性改变之前的值
4. 可配置（非惰性、深度监视…）
5. 惰性：true:不等到监听的数据变化才调用（自动触发一次）

深度监视：加入监听的是一个对象，对象本身变化才会触发（eg：添加一个属性），对象的属性变化不会触发(浅度监听) ，开启深度监听，对象的属性变化也会触发

侦听器回调函数的options参数：

watch(source,cb,{

immediate:true 是否非惰性，默认false

deep:true 是否深度监听：默认false

})

侦听基础类型数据：

 const app = Vue.**createApp**({

**setup**(props, context) {

        const { ref, watch } = Vue

        let brand = **ref**("")

        let site = **ref**("")

*// 侦听器*

*// watch(source,cb,option)*

*// 第一个参数表示侦听谁  第二个是回调函数 第三个是用来配置深度监视还是浅度监视*

*// 监听一个属性*

*/\* watch(brand, (currentVal, preVal) => {*

*// 相应业务*

*console.log("现在的值：" + currentVal);*

*console.log("之前的值：" + preVal);*

*}) \*/*

*// 侦听多个属性*

**watch**([brand, site], ([currentBrand, currentSite], [preBrand, preSite]) => {

          console.**log**("现在的值：" + currentBrand, currentSite);

          console.**log**("之前的值：" + preBrand, preSite);

        })

        return {

          brand,

          site

        }

      },

    }).**mount**('#app')

侦听引用类型数据：

如果要侦听对象中的某个属性，只需要将对象的属性包裹在函数中return出去即可

**setup**(props, context) {

        const { reactive, watch, toRefs } = Vue

        let brandObj = **reactive**({ brand: "" })

*// 侦听器*

*// watch(source,cb,option)s*

*// 第一个参数表示侦听谁  第二个是回调函数 第三个是用来配置深度监视还是浅度监视*

*// 监听引用数据类型属性，*

*// ()=>{return brandObj.brand }:简写*

**watch**(() => brandObj.brand, (currentVal, preVal) => {

*// 相应业务*

          console.**log**("现在的值：" + currentVal);

          console.**log**("之前的值：" + preVal);

        })

        const { brand } = **toRefs**(brandObj)

        return {

          brand,

        }

      },

**watchEffect**

副作用函数，监听整个实例中的数据，只要当前实例中任何数据发生了改变就会触发。(watchEffect会自动检测回调函数体中依赖的数据是否发生变化，发生变化，会执行)

特点：

1. 非惰性
2. 更加抽象
3. 不可以访问属性之前的值

  const app = Vue.**createApp**({

**setup**(props, context) {

        const { ref, watch, watchEffect } = Vue

        let brand = **ref**("")

        let site = **ref**("")

*// watchEffect*

**watchEffect**(()=>{

          console.**log**("开始侦听了");

          console.**log**(brand.value);

          console.**log**(site.value);

        })

        return {

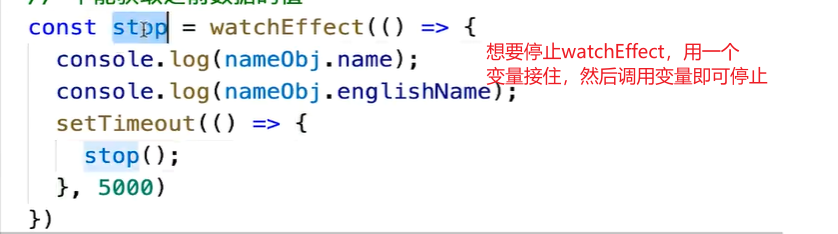
          brand,

          site

        }

      },

**停止watchEffect 和 watch**



**provide和inject**

发射和接收数据，当然也可以发射函数

有的时候组件的层级太深了，正常的组件传参会很麻烦，或者兄弟组件没法传参，这时候就用用到provide和inject。

用法：在要发射数据的组件中使用provide发射数据，在要接收数据的组件中使用inject接收数据。

子组件中：

 <script>

    const myson = {

      template: `

        <div style="width:150px;height:150px;background-color:#f00">

          {{sMsg}}-------{{sCollege}}

          <button @click="changeMsg">修改</button>

        </div>

      `,

**setup**(props, context) {

        const { inject } = Vue;

*// 通过inject接收别的组件发射的值，参数一是发射过来的值，参数二是默认值，可选参数*

        let sMsg = **inject**("msg", "默认值")

        let sCollege = **inject**("college", "默认值")

        let changeName = **inject**("changeName")

*// 创建修改数据的方法*

        let **changeMsg** = () => {

**changeName**("我不是一般的好！！")

        }

        return {

          sMsg,

          sCollege,

          changeMsg

        }

      }

    }

父组件中：

<script>

    const app = Vue.**createApp**({

**setup**(props, context) {

        const { ref, reactive, provide, readonly } = Vue

        let msg = **ref**("你好");

        let college = **reactive**(["数学", "语文", "英语"])

        let **changeName** = (val) => {

          msg.value = val

        }

*// 发射*

*// 通过provide发射数据，参数一是发射的数据的名称(自定义),参数二是值*

*// 在composition API中数据没有数据单向流，所以要在传递数据的时候设置一下*

**provide**("msg", **readonly**(msg))

**provide**("college", **readonly**(college))

*// 如果想要在子组件中修改父组件的数据，应当遵循单项数据流的原则，在源头修改，*

*// 发射一个数据的方法*

**provide**("changeName",changeName)

        return {

        }

      },

      components: {

        "my-son": myson

      }

    }).**mount**('#app')

  </script>

**生命周期钩子的新写法**

在composition API 中，因为setup代替了beforeCreate和created，所以这两个钩子失效，其他的只需要在Vue中引入，然后前面加一个on即可。



<script>

    const app = Vue.**createApp**({

**setup**(props, context) {

        const { onBeforeMount, onRenderTracked, onRenderTriggered, ref } = Vue

*//  生命周期钩子的使用*

**onBeforeMount**(() => {

          console.**log**("onBeforeMount");

        })

*// 每次渲染后重新收集响应式依赖，当视图层更新就会触发调用*

**onRenderTracked**(() => {

          console.**log**("onRenderTracked()");

        })

*// 每次触发页面重新渲染自动执行，当控制层的数据发生了变化调用*

**onRenderTriggered**(() => {

          console.**log**("onRenderTriggered()");

        })

        let msg = **ref**("魑魅魍魉")

        let **handel** = ()=>{

          msg.value = "哈哈"

        }

        return {

          msg,

          handel

        }

      }

    }).**mount**('#app')

  </script>

**ref(获取真实DOM)**

不推荐使用，除非真的需要

<div *id*="app">

    <div *ref*="box">哈哈</div>

    <div *ref*="aaa">嘿嘿</div>

  </div>

  <script>

    const app = Vue.**createApp**({

**setup**(props, context) {

        const { onMounted, ref } = Vue

*// ref 这里ref指的是获取真实dom*

*// 创建一个和视图层标签同名的ref*

        const box = **ref**(null)

        const aaa = **ref**(null)

*//*

**onMounted**(()=>{

          console.**log**(box.value);

          console.**log**(aaa.value);

        })

        return{

          box,

          aaa

        }

  }

    }).**mount**('#app')

  </script>

Vue-Router

**使用步骤**

**第一步：**

yarn add vue-router@4 下载

**第二步：**

创建一个路由对象

涉及到一个路由从定向的知识点：

// 一旦匹配到"/",就会从定向"/home"中

{path:"/",redirect:"/home"},

*// 引入*

import { createRouter, createWebHashHistory } from "vue-router"

import Home from "../components/Home.vue"

import Mine from "../components/Mine.vue"

*// 2.创建路由对象*

const routes = [

*// 路由重定向*

*// 一旦匹配到"/",就会从定向"/home"中*

  {path:"/",redirect:"/home"},

*// 路由*

  {path:"/home",component:Home},

  {path:"/mine",component:Mine},

]

const router = **createRouter**({

*// 路由模式 history模式和hash模式*

  history:**createWebHashHistory**(),

*// 路由配置表*

*// routes:routes*

  routes

})

*// 3.导出路由对象*

export default router

**第三步：**

在main.js配置

import { createApp } from 'vue'

import App from './App.vue'

*// 引入路由对象*

import router from "./router/index"

const app = **createApp**(App)

*// 使用路由*

app.**use**(router)

*// 挂载*

app.**mount**('#app')

**第四步：**

在主页面中(App.vue)

通过<router-view></router-view>来占位，当有组件跳转到此，会自动替换到有此标签的位置，<router-view>的位置可以任意放

<template>

<div>

*<!-- 导航 -->*

  <div *class*="nav">

    <router-link *to*="/home">首页</router-link>

    <router-link *to*="/mine">我的</router-link>

  </div>

*<!-- 路由出口 --占位-->*

  <router-view></router-view>

</div>

</template>

**知识点**

**路由的两种模式的区别：**

具体见思维导图

hash模式(常用):

1. 兼容到IE8

history模式:

1. 兼容到IE10
2. 用户在浏览器中直接访问嵌套路由的时候会出现404报错

**路由跳转**

**页面中跳转：**

使用<router-link to=”/组件地址”></router-link>

底层会调用location.hash(),来经行跳转

eg:

 <router-link *to*="/home">首页</router-link>

**router-link常用的属性：**

**1.选中的时候的样式**

**① 直接在标签中添加此属性**

**active-class="类名"**

**eg:**

<router-link *to*="/mine" *active-class*="current">我的</router-link>

**② css中有一个默认的类，会直接作用于该标签上，不用添加属性**

<router-link *to*="/home">首页</router-link>

<style>

*/\* 默认的选中样式类 \*/*

.router-link-active{

  color: aqua;

}

</style>

**③ 直接在路由对象中配置，将该类的样式作用在router-link标签上**

const router = **createRouter**({

  routes,

  linkActiveClass:'类名'

}

**2.replace**

(导航后不会留下历史记录)

禁止后退，当在当前组件跳转到其他组件的时候，点击浏览的后退可以回去，不希望有此功能，直接添加一个replace属性即可

<router-link *to*="/mine" *replace*>我的</router-link>

**js操作跳转**

首先先从vue-router中获取，

useRouter是获取整个路由对象

useRoute是获取当前活动的路由对象

然后将useRouter赋值给一个变量获得整个路由对象，然后使用里面的push方法

当然也可以直接将路由对象文件导入，就不需要useRouter了

<script>

import {useRouter} from "vue-router"

*// import router from "./router/index"*

export default {

**setup**(props, context) {

*// 获取全局router对象*

    let router = **useRouter**()

*// 跳转到首页*

    let **home** = () => {

     router.**push**("/home")

    };

*// 跳转到我的*

    let **mine** = () => {

     router.**push**("/mine")

    };

    return {

**home**,

**mine**,

    };

  },

};

</script>

**动态路由**

使用场景：有些场景下，一个页面的path路径可能是不确定的(一个新闻详情页中有不同的子新闻，news/001,news/002,无论后面的值多少都会跳转到news组件中，然后以此组件为新闻的根组件，然后显示不同的新闻)

使用方法：在路由后动态绑定一个id属性即可

**路由中：**

const routes = [

*// 动态路由*

  {path:"/news/:id",component:News}

]

**页面中：**

 <router-link :*to*="'/news/' + newsId">新闻</router-link>

**如何获得动态路由的id**

**页面中获取：**

在页面中可以通过$route直接获取到当前路由对象

<p>当前新闻的id：{{$route.params.id}}</p>

**js中获取：**

引入useRoute对象，调用这个函数，拿到具体的当前路由对象，然后在里面获取

<script>

import {useRoute} from "vue-router"

import {ref} from "vue"

export default {

**setup**(props) {

    let route = **useRoute**()

    let newID = **ref** (route.params.id)

    return{

      newID

    }

  }

}

</script>

**路由懒加载**

原因：当打包构建应用时，JavaScript包会变得非常大，影响页面加载，这时需要把不同路由对应的组件分割成不同代码块，当路由被访问的时候才加载对应组件，这样会更加高效

结果：使用懒加载后，首次加载只加载首页，当跳转到别的组件才会加载别的组件相关的东西，而且在项目打包的时候，会将每个组件的css和js单独抽出来，根据不同的页面调取不同的组件，其实也是利用函数只有调用才会触发的原理

使用方法：

在路由文件中，引入组件的方式发生了变化，其他的都没变

const **Home** = ()=>import ("./../components/Home.vue")

const **Mine** = ()=>import ("./../components/Mine.vue")

const **News** = ()=>import ("./../components/News.vue")

当然也可以这种方式引入

 {path:"/home",**component**:()=>import ("./../components/Home.vue")},

你也可以给这些懒加载的组件在打包的时候起个名字

 {path:"/home",**component**:()=>**import** (*/\* webpackChunkName: "自定义名称" \*/*"./../components/Home.vue")},

**路由嵌套**

原因：一些应用程序的UI由多层嵌套的组件组成，比如“我的”页面中，有我的订单和我的消息(/mine/order和/mine/msg)

**使用方法**，

在**路由文件中：**

在一级路由添加一个children数组，操作同一级路由差不多只不过不需要加 /

二级路由添加路由重定向见代码，

如果在二级路由里面需要嵌套三级路由，做法和二级一样

 {

    path:"/mine",

**component**:**Mine**,

    children:[

*// 重定向*

      {path:"/mine",redirect:"/mine/msg"},

      {path:"msg",**component**:**MineMsg**},

      {path:"order",**component**:**MineOrder**},

    ]

  },

**页面里**：

to后面跟的地址要以一级路由开头

<router-link *to*="/mine/msg">我的消息</router-link>

**路由传参**

**分为params和query**

1. **params的用法见上”动态路由”**
2. **query**

在创建路由的时候没啥变化，但是在调用的时候发生变化，

to的值是一个对象，query中的键值对为浏览器导航解析的内容，自定义的，

浏览器解析时候会自动添加一个？

“http://localhost:3000/#/circle?name=百度site=baidu.com”

 <router-link

        :*to*="{ path: '/circle', query: { name: '百度', site: 'baidu.com' } }"

        >圈子</router-link

**获取query传参的值**

页面中直接通过$route.query获得

<p>{{ $route.query }}</p>

js中通过route对象获得

 let route = **useRoute**();

    console.**log**(route.query);

**在js中跳转路由并传参**

*// 新闻,params传参*

    let **news** = ()=>{

      router.**push**("/news/"+newsId.**value**)

    }

*// 圈子，query传参*

      let **circle** = ()=>{

      router.**push**({ path: '/circle', query: { name: '百度', site: 'baidu.com' } })

    }

**导航守卫**

通过路由守卫可以对每次刷新或进入的路由界面进行权限验证，相当于Vue中的全局中间件

分类：分为全局前置守卫**(beforeEach**)和全局后置守卫**(afterEach)**

使用方法：

在导出路由对象之前，导航守卫是router对象中的一个方法

三个参数：

to:从哪里来，这是一个对象，里面包含好多东西

from:到哪里去,这也是一个对象

next:这是一个放行函数，只有调用才能跳转到相应的路由

前置守卫**(beforeEach**):

router.**beforeEach**((to,from,**next**)=>{

*// console.log('从哪里来',from);*

  console.**log**('到哪里去',to);

  document.title = to.meta.title

*// 放行*

**next**()

})

此外，我们还可以在定义路由的时候通过**meta**属性，来绑定自定义的内容，然后可以在to或者from对象通过**.meta**的方式获取其中的内容(比如更改标题)，meta不仅可以作用一级路由还可以作用二级路由…

 {path:"/home",**component**:**Home**,meta:{title:"首页"}},

后置守卫**(afterEach)：**

用法和前置守卫一样，只不过没有next参数；

**路由独享守卫**（几乎不会用到了解即可）

在定义路由的时候定义守卫，单独为这一个路由服务

  {path:"/circle",**component**:**Circle**,meta:{title:"圈子"},**beforeEach**:(to,from,next)=>{console.**log**(1);}},

**keep-alive**

在路由中，keep-alive的作用是用来缓存组件的状态的。(比如,当我们进入一个网页，加载完毕后，我们又跳转到别的组件，在跳转回来，那么又会重新加载，使用keep-alive来解决)

使用方法：

在占位的地方更改

<router-view *v-slot*="{ Component }">

      <keep-alive>

        <component :*is*="Component" />

      </keep-alive>

</router-view>

注意，当用keep-alive包裹组件的时候会触发两个生命周期钩子,一旦组件之间跳转就会触发

onActivated和onDeactivated

**keep-alive---常用属性：**

include 值可以是：string/RegExp/Array 只有名称匹配的组件才会被缓存

exclude 值可以是：string/RegExp/Array 任何被匹配的组件都不会被缓存

max --number/string 最多可以缓存多少个实例

注意：这里匹配的名子，是在组件实例创建的时候的name属性

 <router-view *v-slot*="{ Component }">

*<!-- <keep-alive exclude="home,news"> -->*

<keep-alive *exclude*="[“home”,”news”]">

        <component :*is*="Component" />

      </keep-alive>

    </router-view>

axios

get请求

*// 带参数的请求*

  axios

    .**get**("http://demo.itlike.com/web/xlmc/api/homeApi/categoriesdetail/lk001", {

      params: { id: 1 },

    })

    .**then**((res) => {

      getData.data = res.data.data;

    })

    .**catch**((err) => console.**log**(err));

};

post请求

let **postReq** = () => {

**axios**({

    method: "post",

    url: "http://demo.itlike.com/web/xlmc/api/homeApi/categoriesdetail/lk001",

    data: {

      id: '002',

    },

  })

    .**then**((res) => {

      console.**log**(res);

    })

    .**catch**((err) => console.**log**(err));

};

并发请求

*// 并发请求*

let **allReq** = () => {

  axios

    .**all**([

**axios**({

        url:"http://demo.itlike.com/web/xlmc/api/homeApi/categoriesdetail/lk001"

      }),

**axios**({

        url:"http://demo.itlike.com/web/xlmc/api/homeApi/categoriesdetail/lk002"

      })

    ])

    .**then**((res) => {

      console.**log**(res);

    })

    .**catch**((err) => {

      console.**log**(err);

    });

};

可通过spread方法将并发请求的结果拆开

.**then**(

*// 有几个值就写几个参数*

      axios.**spread**((res1, res2) => {

        console.**log**(res1);

        console.**log**(res2);

      })

    )

封装axios

项目中，一般都是将axios封装到一个js文件中，然后再引入到页面中，这样只需要更改页js一处即可

示例：

*// 此文件用来配置axios的请求的*

*// 1.引入*

import axios from "axios"

*// 配置基础路径*

axios.defaults.baseURL = "http://demo.itlike.com/web/xlmc/api";

*// 配置请求时间,如果超出时间就取消请求，单位ms*

axios.defaults.timeout = 5000;

*// 配置请求的头文件*

axios.defaults.headers.post['Content-Type']='application/x-www-form-urlencoded'

export default function **ajax**({ url = "", params = "", type = "get" }) {

  return new **Promise**(

    (**resolve**, **reject**) => {

*// 0.定义一个变量接收数据*

      let promise;

*// 1.判断请求类型*

      if (type.**toUpperCase**() === "GET") {

        promise = **axios**({

          url,

          params

        })

      } else if (type.**toUpperCase**() === "POST") {

        promise = **axios**({

          method: "post",

          url,

          data: params

        })

      }

*// 3.处理返回*

*// 此时promise的值为axios返回的值，是promise类型的，可以用then和catch*

      promise.**then**((res) => {

**resolve**(res)

      }).**catch**((error) => {

**reject**(error)

      })

    }

  )

}

axios中的请求拦截与响应拦截

*// 配置请求拦截*

axios.interceptors.request.**use**(

  (config)=>{

    config.abc="我是大哥"

    console.**log**(config);

*// 拦截之后，操作以后再将请求发出*

    return config

  },

  (error)=>{

    return **Promise**.**error**()

  }

)

*// 配置响应拦截*

axios.interceptors.response.**use**(

  (response)=>{

*// config.abc="我是大哥"*

    console.**log**(response);

*// 拦截之后，操作以后再将请求发出*

    return response

  },

  (error)=>{

    return **Promise**.**error**()

  }

)

vuex

使用：先去下载，然后再一个单独的js文件中配置

*// 1.引入*

import { createStore } from "vuex"

*// 2.创建一个store对象*

let store = **createStore**({

*// 使用严格模式--只要状态的变更不是通过mutation触发都会抛出错误*

*// strict:true,*

  strict: process.env.NODE\_ENV !== 'production',

*// 全局共享的状态(数据)存放*

  state: {

    counter: 10,

  },

*//  同步提交状态(修改、处理)*

  mutations: {

*// 1.加一*

**INCREMENT**(state) {

      state.counter++

    },

*// 3.两秒后加1999*

**INCREMENT1999**(state,num) {

      state.counter += num

    }

  },

*// 提交操作给mutations,(同步/异步操作)*

  actions: {

*// 加一*

**increment**({ **commit** }) {

**commit**("INCREMENT")

    },

*// 2秒加1999*

**increment1999**({ **commit** },num) {

**setTimeout**(() => {

**commit**("INCREMENT1999",num)

      }, 2000)

    }

  },

*// 相当于计算属性的作用*

  getters: {}

})

*// 3.导出*

export default store

调用的时候在需要使用的文件中引入

如果修改的内容为同步操作，可以使用actions🡪mutations🡪state,也可使用mutions🡪state

如果为异步的话则必须使用actions🡪mutations🡪state

异步的调用：store.dispatch()

同步可使用：store.commit()

<script>

import { useStore } from "vuex";

export default {

**setup**(props) {

    let store = **useStore**();

*// 下面方法中有两种不同的方式*

*// 一种是走全流程*

*// 一种是直接通过同步提交*

*// 加一*

    let **plus** = () => {

*// actions-->mutations-->--->state*

*// store.dispatch("increment");*

*// mutations-->sate*

      store.**commit**("INCREMENT");

    };

*// 减一*

    let **sub** = () => {

*// actions-->mutations-->state*

*// store.dispatch("decrement");*

*// mutations-->state*

      store.**commit**("DECREMENT");

    };

*// 两秒后加1999*

    let **addSum** = () => {

      store.**dispatch**("increment1999",1999);

    };

    return {

**plus**,

**sub**,

**addSum**,

    };

  },

  components: {

    HelloWorld,

  },

};

</script>

也可以简化：

**setup**(props) {

    let store = **useStore**();

    return {

      counter: **computed**(() => store.state.counter),

**increment**: () => store.**dispatch**("increment"),

**decrement**: () => store.**commit**("DECREMENT"),

**increment1999**: () => store.**dispatch**("increment1999",1999),

    };

  },

视图层获取

<h1>首页的:{{ $store.state.counter }}</h1>

suspense--异步请求的好帮手

如果想用suspense，那么需要再setup中返回一个用promise包裹的，

子组件

<template>

  <h1>{{result}}</h1>

//因为定时器返回了result,所以这里可直接使用

</template>

<script *lang*="ts">

import { defineComponent } from 'vue'

export default **defineComponent**({

**setup**() {

    return new **Promise**((**reslove**)=>{

**setTimeout**(()=>{

        return **reslove**({

          result:404

        })

      },3000)

    })

  },

})

</script>

也可采用es7的async await来定义

  async **setup**() {

    const rawData = await axios.**get**("https://dog.ceo/api/breeds/image/random")

    return {

      result:rawData.data

    }

  },

父组件或别处使用的时候：

可以用匿名插槽放异步请求后的内容，

剧名插槽放还没请求回来的内容占位

 <**Suspense**>

#v-slot简写:#

      <template *v-slot:default*>

        <**AsyncShow**/>

      </template>

      <template #*fallback*>

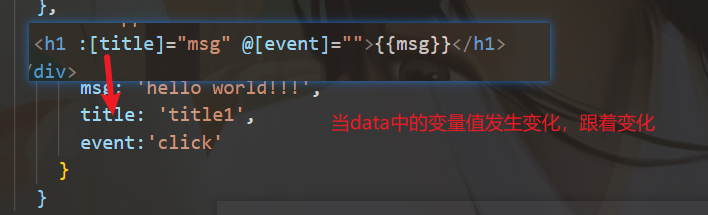
        <h1>Loading ! ...</h1>

      </template>

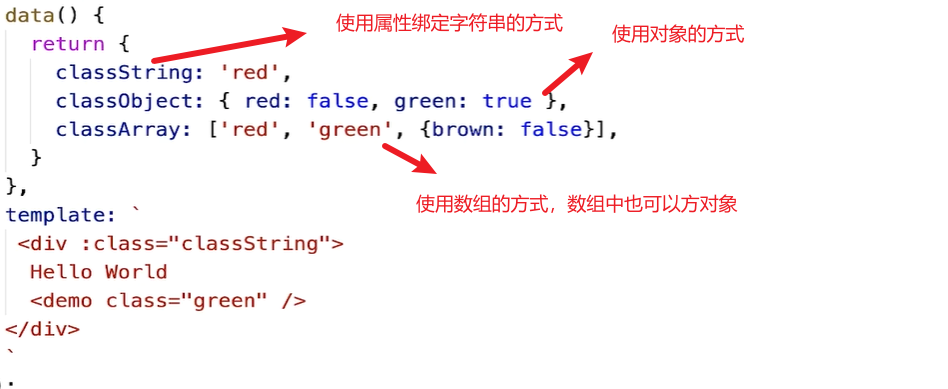
    </**Suspense**>

补充知识点

### 动态属性



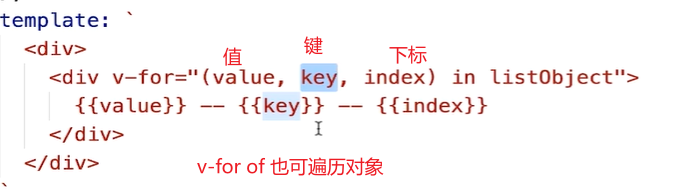
### 更改样式



### 更改内联样式



### v-for遍历对象



### 一个事件触发多个函数

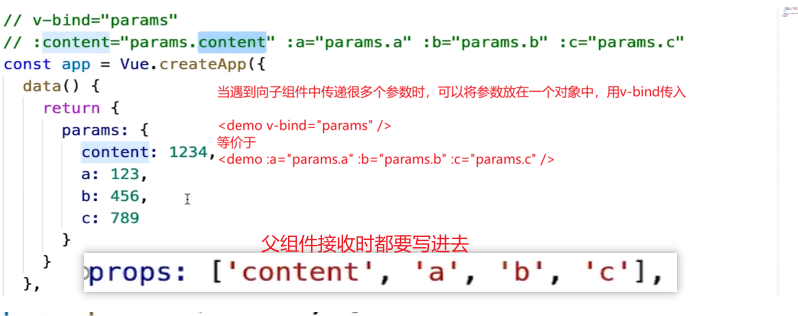


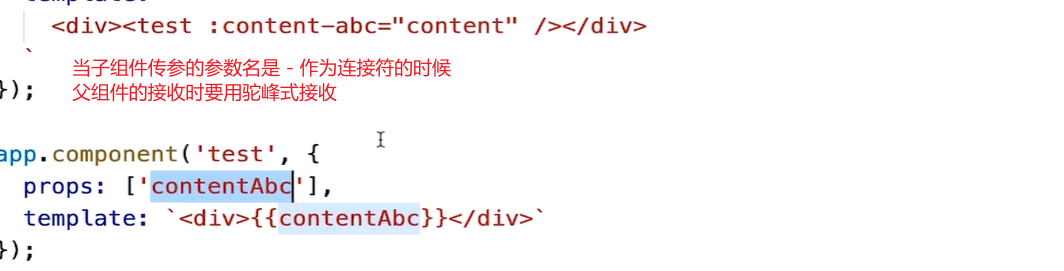
### checkbox的高级用法



#### Props



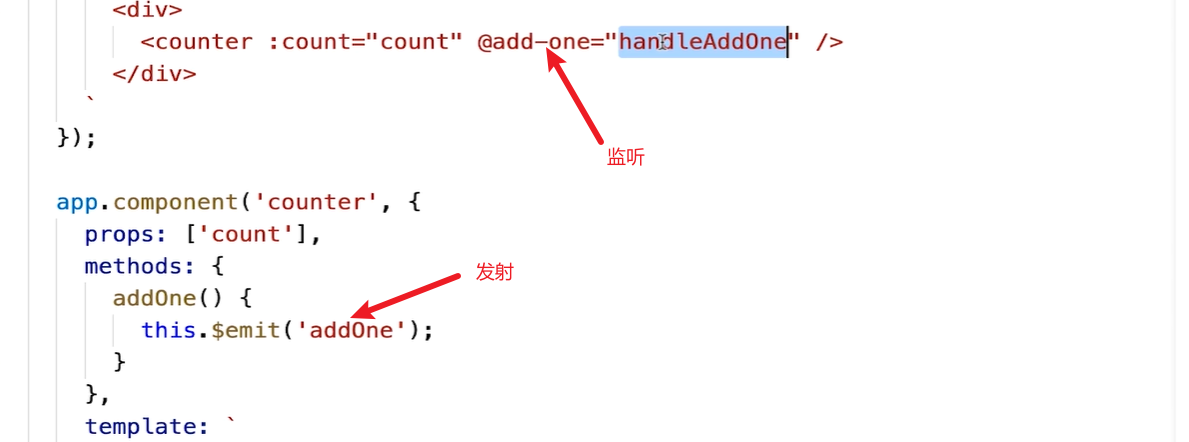




#### emit

注意

emit发射事件的时候 采用驼峰命名 监听事件的时候采用 ‘-’连接符命名



v-model在子组件中的用法

如果希望子组件和父组件的data建立联系的时候 可以使用这个，代码比$emit写法简洁



自定义名字



**同时使用多个v-model在子组件中**



**自定义组件修饰符**

