Vue是轻量级框架，适用于手机端的开发。

创建vlue来实现，这个负责链接view和model

//vuejs的核心是一个允许采用简洁的模板语法来声明式的将数据渲染进DOM

// 通过创建一个 Vue实例来实现， 这个vue实例负责连接view和model

// 所以经常称这个对象为viewModel

注意：

1. vue中的实例必须定义负责会报错；
2. 同步数据中直接更改长度与直接更改下标不可用。
   1. 例：由于 JavaScript 的限制，Vue 不能检测以下变动的数组：
   2. 当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue
   3. 当你修改数组的长度时，例如：vm.items.length = newLength

## 开始

<!--id为app的元素就是view-->可以新建多个  
<div id="app">

New vue({

el:”app”, 定义区域名称

data:{ //每个vue实例都会代理其data对象里的所有属性

msg:”hello,vue!”

})

## 指令

1. v-html 插值语法会将数据解析为纯文件 ，而html代码不能正常解析需要用该指令解析。

使用方法：<div v-html=”<h1>你好</h1>”></div>

1. v-bind 因为插值语法不能在html属性中使用，在属性上绑定动态值需使用v-bind指令。
   1. 使用方法：<span v-bind:title=”msg”>世界</span>
   2. 简化语法：缩成冒号：
   3. 例：<span :title=”msg”>世界</span>
2. v-model 双向绑定，使用方法与angular相同
3. v-for 遍历指令，用法与angular相同，当需要获取索引时用数组加入值

例：v-for=”(user,index) in users” user为遍历后内容，index为自定义存储下标容器，users为数组

1. v-show用法与angular相同，值内可写表达式
2. v-if用法与angular中相同，值内也可写表达式。
3. v-else可与v-if一起使用类似于if…else..
4. v-else-if可多次套用，与if..elseif..elseif..else…相同的用法，类似于js。

v-if的效果与v-show相同但是机制不同：

* 1. v-show是通过控制display属性来更改状态。
  2. v-if是通过加载或不加载来控制元素。
  3. v-if有更高的切换开销，而v-show有更高的初始渲染开销，如果需要非常频繁切换用v-show如果切换机会不多用v-if。

1. v-bind:class=”{bg:hasBg，’col-white’:hascolor}”，控制class的命令，可以写入表达式，可简写为：class
   1. 键值对写法：可以将判断写至后台，用对象形式控制，在页面写入对象名。
   2. 数组写法：可以在后台写成数组形式，通过控制数组中的内容来控制class，数组名写入页面，写入数组内的class直接生效。
2. :style=”{fontSize:myfontsize+”px”}”控制样式，若有横线的css属性则按照js方法写入，可拼接字符。
3. :class=”{class:条件}” 若条件成立则添加该class，也可不加大括号为直接添加该class但是可以通过传入函数的形式控制class ，写运算用【】。

## 事件

事件之前添加v-on:

例：点击：<button v-on:click=”showInfo()”>THET<button>

简化为：@字符

<button @click=”showInfo()”>THET</button>

## 属性

Vue的data中属性会被vue继承，在data中统一定义属性。

例data: { //每个vue实例都会代理其data对象里的所有属性  
 msg: "hello, vue"  
},

## 行为

在methods中统一定义方法，methods内部的行为若需对vue内部属性操作则需要用this使用。引用时需要用使用函数的方法，可在双括号内直接调用。

例：methods: { //自定义方法都写在这里  
 showInfo: **function** () {  
 console.log("test!!!!");  
 }  
}

## 计算属性

方法名：computed,使用methods方法同样可以实现定义函数,但是每次返回值都会重新计算。调用方式为属性名方式。

不同的是计算属性是基于它们的依赖进行缓存的。  
计算属性只有在它的相关依赖发生改变时才会重新求值。  
这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果

例：computed: { //计算属性  
 welcomeUsername: **function**(){  
 **return** "hello, " + **this**.username + "," + Math.random();  
 }  
}

## 动画过渡

1. 添加自定义标签，定义唯一标识name

例： <transition name="fade"> 用自定义标签包裹住过渡的标签

<p v-if="show">hello</p>

</transition>

1. 定义所用的class并加上所定义的标识
2. fade-enter-active: 定义过渡的状态。在元素整个过渡过程中作用，在元素被插入时生效，在 transition/animation 完成之后移除。 这个类可以被用来定义过渡的过程时间，延迟和曲线函数。
3. fade-enter进入过渡的结束状态
4. fade-leave-to:  定义离开过渡的结束状态。在离开过渡被触发一帧后生效（于此同时 v-leave 被删除），在 transition/animation 完成之后移除。
5. fade-leave-active:  定义离开结束状态的过渡结束。

注：也可以通过添加动画来更改，也可直接添加过渡事件transform.

## 组件

组件 (Component) 是 Vue.js 最强大的功能之一。组件可以扩展 HTML 元素，封装可重用的代码。  
(自定义标签) 若定义的标签名由多个字母组成则后面首字母为小写并加上横线。  
<home></home>

#### 全局方式注册

（定义）一个组件(component) ，全部vue都可以使用该组件。

Vue.component("home", { 引用Vue方式，定义自定义标签名  
 template: '<div><h3>hello, {{homeData}}!</h3> <button @click="testHome">testHome</button></div>', 定义显示插入的内容  
 data: **function**(){ //组件中data必须定义为一个方法，这个方法返回一个对象  
 **return** { 这样做是为了避免vue报错  
 homeData: "home!!!"   
 };  
 },  
 methods: {  
 testHome: **function** () {  
 alert("home!!");  
 }  
 }  
});

#### 局部方式注册

通过变量存储定义的组件，并在区域内部使用components方法调用该组件。

使用变量存储定义的组件。  
**var** Home = { //其实就定义了一个普通的js对象  
 template: "<h3>hello, {{info}}</h3>",  
 data: **function**(){  
 **return** {  
 info: "home!!!"  
 };  
 }  
};

**new** Vue({  
 el: "#app",  
 data: { //vue中data定义为一个对象  
 msg: "hello, vue"  
 },  
 components: { 在区域内部调用并改名。  
 home: Home, //home即为组件名称，Home是home这个组件的具体内容，习惯组件名以小写开始，  
 // 如果多个单词连缀，建议使用驼峰命名法则  
 myHeader: MyHeader 使用的标签名为左边的，右边为传入的变量。  
 }  
});

### 父组件对子组件传递数据数据

父组件就是整个Vue对象，而子组件是所创建的自定义标签内容。

组件实例的作用域是孤立的。这意味着不能 (也不应该) 在子组件的模板内直接引用父组件的数据。

要让子组件使用父组件的数据，我们需要通过子组件的 props 选项。

prop 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来。  
这是为了防止子组件无意修改了父组件的状态——这会让应用的数据流难以理解。

1，从页面中专获取父组件内容：<home :msg="welcome"></home>

2，定义从页面中的那个属性获取值

**var** home={  
 template:"<h1>你好{{msg}}{{aa}}</h1>", 输出获取的值  
 props:["aa"], 定义从页面获取的父组件的值  
 data:**function** () {  
 **return**{  
 msg:"世界"  
 }  
 }  
};

### 子组件对父组件传递数据

需要使用Vue的自定义事件方法——$emit

在子组件定义的变量中对行为中定义检测的方法

1. 子组件的定义变量中定义行为

template:"<div><h3>hello,{{name}}</h3><button @click='sendData'>点击</button></div>", 控制自定义事件可以在子组件之中设置

methods:{   
 sendData:**function** () { 定义为可控的函数名称  
 **this**.$emit("receive",**this**.name); 通过$emit来创建事件，this代表子组件内容  
 /\*这句话一执行，home上的监听就检测到会执行后面的处理程序\*/可用多种方式控制自定义事件。  
 }  
}

1. 在Vue父组件的行为中定义函数

methods:{  
 showInfo:**function** (data) { 自定义行为触发后需要执行的函数  
 console.log(data) data为事件指定的内容，例：上面改的name  
 **this**.username=data; 将从函数中获取子组件的值赋值到父组件之中。  
 }  
},

1. 页面应用

<home @receive="showInfo"></home> 自定义属性，当触发后才会获取值。

!--监听receive事件的发生，如果发生就执行showInfo方法-->

### 兄弟组件之间传递数据

该方法仅适用于简单工程

新建一个bus的js文件，创建实例、导出

**import** Vue **from** 'vue'  
**export default new** Vue()

然后在兄弟组件之间导入该文件，

在传输的事件页面定义函数并应用

<div class="click" @click="addCart('12356')"></div> 传入参数内容为点击后传出的

methods:{  
 addCart(num) {  
 Bus.$emit('getTarget', num); num为传出的数据  
 console.log(num) 可以传入一个值或者数组都行。  
 }  
}

在显示层接受

created() {  
 Bus.$on('getTarget', target => { 在钩子函数内监听事件，该函数获取的为传来的值  
 console.log(target);  
 });  
}

## 监听

使用watch方法监听数组或者对象的变化，当变化后获取改变后的内容。

watch: {  
 buyNum: { 监听的对象可以数组或者对象  
 handler: **function** (val, oldVal) { 处理器调用，得到改变的值  
 可以对改变后的值进行操作  
 },  
 deep: **true 进行深层次的监听，监听属性值的内容改变** }  
},

监听事件的方法

bus.$on("getBuyNum", target => { 第一个值为事件的名称，后面的为回调函数，  
对事件发生后得到的内容进行处理}  
});

## 实例生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如，实例需要配置数据观测(data observer)、编译模版、挂载实例到 DOM ，然后在数据变化时更新 DOM 。在这个过程中，实例也会调用一些 **生命周期钩子** ，这就给我们提供了执行自定义逻辑的机会。例如，**[created](https://cn.vuejs.org/v2/api/" \l "created)** 这个钩子在实例被创建之后被调用：

也有一些其它的钩子，在实例生命周期的不同阶段调用，如 [**mounted**](https://cn.vuejs.org/v2/api/#mounted)、**[updated](https://cn.vuejs.org/v2/api/" \l "updated)**、**[destroyed](https://cn.vuejs.org/v2/api/" \l "destroyed)**。钩子的 this 指向调用它的 Vue 实例。一些用户可能会问 Vue.js 是否有“控制器”的概念？答案是，没有。组件的自定义逻辑可以分布在这些钩子中。

生命周期图示



## vue路由

用 Vue.js + vue-router 创建单页应用，是非常简单的。使用 Vue.js ，我们已经可以通过组合组件来组成应用程序，当你要把 vue-router 添加进来，我们需要做的是，将组件(components)映射到路由(routes)，然后告诉 vue-router 在哪里渲染它们。

### 安装

1. NPM: npm install vue-router –save-dev

2. 如果你想使用最新的开发版，就得从 GitHub 上直接 clone，然后自己 build 一个 vue-router。

### 使用

1．HTML中标签写入

<router-link>使用 router-link 组件来导航，通过传入 `to` 属性指定链接，默认会被渲染成一个 `<a>` 标签

例： <router-link to="/foo">Go to Foo</router-link>

路由出口：路由匹配到的组件将渲染在这里

例： <router-view></router-view>

**1. 配置JS路由**

1. JS中配置

配置组件，创建对象建立路由规则，一个路由配置一个组件。组件内可以继续嵌套组件路由。把对象new出来，作为参数new给vueRouter这个对象，作为它的构造函数的参数传递进去。最终在new的根实例中配置的是new出来的对象。总结为：配置的路由规则是为了作为参数传递给new的Vuerouter这个对象，而造出来这个对象是为了用在vue根实例上。也可以把创建路由规则直接写在对象内进行创建。

例： 1. 定义（路由）组件。可以从其他文件 import 进来

const Foo = { template: '<div>foo</div>' }

const Bar = { template: '<div>bar</div>' }

2). 定义路由

每个路由应该映射一个组件。 其中”component” 可以是通过 Vue.extend() 创建的组件构造器，

或者，只是一个组件配置对象。我们晚点再讨论嵌套路由。

const routes = [

{ path: ‘/foo’, component: Foo },

{ path: ‘/bar’, component: Bar }

]

3). 创建 router 实例，然后传 `routes` 配置你还可以传别的配置参数, 不过先这么简单着吧。

const router = new VueRouter({

routes // （缩写）相当于 routes: routes

})

4). 创建和挂载根实例。

记得要通过 router 配置参数注入路由，

从而让整个应用都有路由功能

const app = new Vue({

router

}).$mount('#app')

**2. 模块化组件**

模块化引入在main.js中引入vue-router，引入需要插入的文件，创建对象引入路由。

例：**import** VueRouter **from** 'vue-router';

**import** Home **from**'./components/home.vue';  
**import** About **from**'./components/about.vue'; //用es6引入vue-router  
Vue.*use*(VueRouter);  
//如果在一个模块化工程中使用它，必须要通过Vue.use() 明确的安装路由功能  
**var** myRouter = **new** VueRouter({ //创建new对象  
 routes: [ //直接以数组的方式批量创建路由组件  
 {path: "/home", component: Home},  
 {path: "/product", component: product}  
 {path: "/", redirect: "/home"}//重定向方式配置默认路由，redirect：重定向功能，写到最后。  
 ]  
});

## 获取后台数据

引入插件vue-resource.js，它体积小，功能性插件，仅为实现获取数据的功能。

### Get

在methods中定义行为函数，

在定义的方法函数中使用this.$http.get(“接口地址”)

.then(function(data){第一个返回函数为成功函数},function(err){第二个返回函数为访问失败函数})

例：methods: {  
 getData: **function** () {  
 **this**.$http.get("http://localhost:3000/IndexInfo")  
 .then(**function** (data) {  
 console.log(data)  
 }, **function** (err) {  
 console.log(err)  
 })  
 }  
}

### Post请求

同样在methods中定义自定义行为函数：

需要传输的数据需要与前台数据映射，所以要先对前台数据进行初始。该动作在data中进行。

在函数中使用$http.post(“接口地址”,{需要传入的数据与data中的数据进行映射}，{因为需要设置使用Json模式传递数据，所以添加命令emulataJSON:true})

.then(function(data){},function(err){})

最后调用的时候可以在页面使用事件来运行该自定义行为函数。

例： data: {  
 name: ""  
 },//初始化数据  
 methods: {  
 regUser: **function** () {  
 **this**.$http.post("http://localhost:3000/user/reg", {  
 name: **this**.name   
 }, {emulateJSON: **true**}).then(**function** (data) {  
// /\*需要特殊配置支持JSON格式\*/  
 console.log(data, "成功")  
 }, **function** (err) {  
 console.log(err)  
 })  
 }  
 }

### 脚手架工具

Vue-cli工具

工作中不会js直接引入，而是会把js，可以帮前端人员快速构建一个足以支撑实际项目开发的vue环境。他的存在降低了项目初始化工作的复杂度。自动安装的有及时更改等功能。

思想：组件作为独立的业务单元，认为组件的相关内容（布局，逻辑，样式）不应该被过分拆分，应该就近编辑，就近维护。

1. 安装vue-cli工具
   1. Vue是一个npm安装包
   2. 在全局模式下安装vue-cli工具，cnpm install vue-cli -g
2. 进入需要安装的文件夹，使用vue-cli创建一个原始模板工程，
   1. Vue-init模板名称，工程名称。例：vue init webpack-simple vue-todos

模板一般使用选择webpack或webpack-simple

* + 1. webpack的自带包有三十多个，
    2. webpack-simple的自带包十多个

b) 安装时的选项大多时提示文件信息与是否安装其他插件。

1. 用IDE（webstorm）打开创建的工程

打开终端🡪cnpm install 安装packgjson中的所有配置文件。

1. 运行工程

num run dev

1. 根据自己的业务修改脚手架创建的工程 App.vue

**<template>**

**<script>**组件代码，核心页面数据来源。可以通过es6的import命令导入外部文件例css文件

**例：import** './assets/site.less'//用引入文件的方式必须要能解析该文件，公用文件可以用导入的方式。

在js中配置默认模块

**export default** {//导出默认模块  
 name: 'app', //模块名，可省略  
/\* data () {  
 return {  
 msg: 'Welcome to Your doudou' 该方法为es6的写法  
 }  
 }\*/  
 data:**function** () { vue组件中的data必须声明为一个函数，而这个函数需要返回一个对象。  
 **return** {  
 title:"vue-todos",  
 todos:[ //事情数组,自定义的  
 {value:"阅读一篇关于前端的技术文章",done:**false**},//done标识状态，false表示没有做，对状态不同的内容可以用不同的显示方法。  
 ]  
 }  
 }  
}

**注：**若对vue进行操作应写在该标签的export default内，不用再次创建vue。

**<style>**需要使用less来进行预处理(参考less中文网)，扩展了css语言，增加了变量，Mixin、函数等特性，使Css更易维护和扩展，可以运行在node或浏览器。

安装包 style-loader css-loader less less-loader（其中css-loader已经安装过，不需要再次安装。

cnpm install style-loader less less-loader --save-dev 直接在本地安装包

1. 在vue文件中的style标签上添加 lang="less" rel="stylesheet" 表明内部语言为less，区域独有的样式可以写在其中。
2. 也可使用script标签引用，不但要引入文件还要对webpack.config.js进行更改配置，添加module属性中的rules方法，添加上转译less文件的配置。

例：{  
 test:/\.less$/,//转译后缀为less的文件  
 loaders:['style-loader','css-loader','less-loader']

//需要三个一起才能解析  
}

定义变量例：@base:#fg38ab;若在多出使用只需要吧变量名引用过去。

样式写法为包裹的关系，解析后的类似于：#app #mode

&符号，有&时解析的是同一个元素或者此元素的伪类，没有&解析是后代元素。

运算及函数。

—————————————————————————————————————————————————

在src目录下可以看到有.vue结尾的文件，这是vuejs特有的文件格式，也是vue的一大特点，被称之为“单页式组件”，一个vue文件为一个组件。\*，.vue可以同时承载三类内容：

<template> 视图模板

<script> 组件代码

<style> 样式定义思想：组件作为独立的业务单元，认为组件的相关内容（布局，逻辑，样式）不应该被过分拆分，应该就近编辑，就近维护。

render:h=>h(App)

//渲染：将App.vue文件传给vue实例

注意

使用webstrom编辑可安装vue插件，便于更改文件，并将语言更改为ES6.

#### 使用webpack安装模板：

1，创建工程，vue init webpack 工程名

其中的插件都不需要，当需要时自己安装。简单修改后即可自己写。

2，修改后台文件，该文件自带的有后端文件，稍加改动后即可用于前端。

将json文件，dev server文件进行覆盖，

### vue中的滑动联动

vue用ref来获取Dom元素。

probetype:3表示能获取值。

click:true 表示可以点击，默认是不可点击的。

ES6中拼接字符与字符变量，例：’ 配送${xx}元 ’

使用better-scroll插件来完成手机页面的滑动联动。

**首先**

导入better-scroll，对商品列表和菜单列表添加overflow: hidden;并且都设置脱离标准流的定位。

页面属性

<div class="goods" ref="goods"><!--最外围框-->

<!--ref是vue用来找到dom元素的方式-->

<!----------------------------------菜单页框------------------------------ -->

<div class="menu-wrapper" ref="menuWrapper">

<ul>  
 <li v-for="(item,index) in goods" class="menu-item menu-wrapper-hook"  
 :class="{current: currentIndex==index}"  
 @click="selectMenu(index)">

在所遍历的数据中添加class 为menu-wrapper-hook，该class是为了寻找每一层的dom元素。绑定的class是为了在联动时触发高亮效果。

-------------------------------------商品页----------------------------------

<div class="goods-wrapper" ref="goodsWrapper"><!--商品页展示框-->  
 <ul>  
 <li v-for="item in goods" class="food-list goods-wrapper-hook">  
 <!-- goods-wrapper-hook只给js用，获取商品每层高度

**js属性**

**首先在data中定义相关属性**

listHeight: [], 右侧商品高度的数组  
scrollY: 0, 右侧商品块的滚动高度  
menuScrollY: 0, 左侧菜单块滚动高度  
visiHeight: 0,  
menuHeight: [] 左侧菜单每层高度的数组

**在methods中定义滚动效果行为**

initScroll() { 定义滚动行为  
 **this**.menuScroll = **new** BetterScroll(**this**.$refs.menuWrapper, {

用vue的方法获取dom元素  
 click: **true**, /\*因为better-scroll会取消默认点击\*/  
 probeType: 3 设置制定的数值类型，不清楚啥类型  
 });  
 **this**.menuScroll.on("scroll", (pos) => { 检测滚动事件  
 **this**.menuScrollY = Math.abs(Math.round(pos.y)); 将滚动的距离取整后赋值  
 });  
 **this**.goodsScroll = **new** BetterScroll(**this**.$refs.goodsWrapper, { 商品块的滚动  
 probeType: 3, //相当于探针，实时告知滚动位置  
 click: **true**,  
 });  
 **this**.goodsScroll.on("scroll", (pos) => { 获取商品列表的滚动距离并保存  
 **this**.scrollY = Math.abs(Math.round(pos.y));  
 });  
},

**在methods中保存菜单栏和商品栏的每层高度**

calcHeight() {//定义行为  
 let goodsList = **this**.$refs.goodsWrapper.getElementsByClassName("goods-wrapper-hook");  
 //保存用vue获取的class的dom元素  
 let height = 0;  
 //初始化变量  
 **this**.listHeight.push(height);  
 //将每层商品的高度保存在vue属性内的数组中  
 **for** (let i = 0; i < goodsList.length; i++) {//对每层商品进行遍历。  
 let item = goodsList[i]; //将每层不同的高度保存在属性中。  
 height += item.clientHeight;  
 **this**.listHeight.push(height);  
 }  
 //计算菜单条高度  
 let menuList = **this**.$refs.menuWrapper.getElementsByClassName("menu-wrapper-hook");  
 let height2 = 0; //获取菜单栏的高度，初始化高度  
 **this**.menuHeight.push(height2); //将初始化的保存在菜单属性的数组中  
 **for** (let i = 0; i < menuList.length; i++) {//同样将每层高度保存在属性的数组中  
 let item = menuList[i];  
 height2 += item.clientHeight;  
 **this**.menuHeight.push(height2);  
 }  
 **this**.visiHeight = **this**.$refs.goods.clientHeight;//获取ref属性的高度  
},

**引用函数来改变右侧商品块距顶部距离。**  
selectMenu(index) {  
 let goodsList = **this**.$refs.goodsWrapper.getElementsByClassName("goods-wrapper-hook");  
 let el = goodsList[index];  
 **this**.goodsScroll.scrollToElement(el, 500); //该方法传入的是一个dom元素，跳转到dom元素所在的顶部，后跟的为过渡时间。  
}

计算属性computed中定义函数判断右侧商品列表的高度位置来反射左侧菜单栏的样式变化

currentIndex() {//定义计算属性  
 **for** (let i = 0; i < **this**.listHeight.length; i++) { 对每层的高度进行遍历  
 let height1 = **this**.listHeight[i]; 获得相邻两层的高度  
 let height2 = **this**.listHeight[i + 1];  
 **if** (!height2 || (**this**.scrollY >= height1 && **this**.scrollY < height2)) {

如果商品的距离顶部距离大于当前小于第二个或者没有下一个高度  
 let menuList = **this**.$refs.menuWrapper.getElementsByClassName("menu-wrapper-hook"); 保存页面的菜单dom数组  
 let el = menuList[i];  
 **if** (**this**.menuHeight[i + 1] >= **this**.visiHeight || Math.abs(**this**.menuScrollY) >= **this**.menuHeight[i]) { 如果下一个的高度大于视口高度或者当前菜单块的距离顶部距离大于显示的菜单块高度  
 //超出可视区

调用滚动属性，调整左侧菜单位置  
 **this**.menuScroll.scrollToElement(el,500);  
 }

如果可以显示则直接返回i来更改左侧的样式高亮  
 **return** i;  
 }  
 }

如果存在下一个并且不大于当前的高度则为第一个，返回0  
 **return** 0;

}

在created得钩子函数中完成数据获取后调用

**this**.$nextTick(() => { 该方法在dom渲染完成之后才会执行函数  
 **this**.initScroll(); 初始化调用函数  
 **this**.calcHeight();  
});

### 动态路由配置

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用『动态路径参数』（dynamic segment）来达到这个效果：

传送门：<https://router.vuejs.org/zh-cn/essentials/dynamic-matching.html>

例：定义路径

routes: [

// 动态路径参数 以冒号开头

{ path: '/user/:id', component: User }

]

**被复用的组件不会再重复出发周期函数，需要使用监听来确定页面的重开。**

提醒一下，当使用路由参数时，例如从 /user/foo 导航到 /user/bar，**原来的组件实例会被复用**。因为两个路由都渲染同个组件，比起销毁再创建，复用则显得更加高效。**不过，这也意味着组件的生命周期钩子不会再被调用**。

复用组件时，想对路由参数的变化作出响应的话，你可以简单地 watch（监测变化） $route 对象：

**例：**<router-link :to="{name:'goodMsg',params:{id:1},query:{name:'dxc'}}">购买商品</router-link>

**Params为路径id，query为页面传值内容，两个不相同，但是提供了path则params会被忽略所以需要设置name，写name路径则正常使用。**