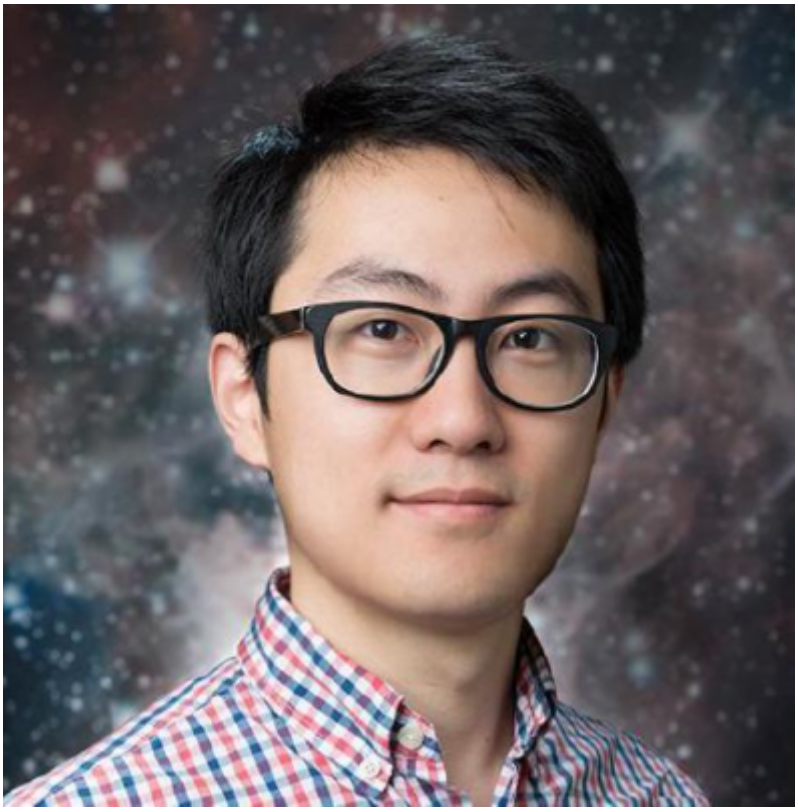
1.认识Vue

官网地址：https://cn.vuejs.org/v2/guide/

Vue.js 是一套构建用户界面的渐进式框架。

Vue 2 是在2016年发布使用，2020是 vue3 才刚发布，时隔一年左右就已经将 vue3 作为了默认版本

> 尤雨溪，Vue.js和Vite的作者，HTML5版Clear的打造人，独立开源开发者。
>
> 曾就职于Google Creative Labs和Meteor Development Group。
>
> 由于工作中大量接触开源的JavaScript项目，最后自己也走上了开源之路，现全职开发和维护Vue.js。
>
> 尤雨溪毕业于上海复旦附中，在美国完成大学学业
>
> 尤雨溪大学专业并非是计算机专业，在大学期间他学习专业是室内艺术和艺术史



为什么要使用vue?

- 虚拟DOM： 性能高
  - 数据驱动试图
- 组件化:
- 数据与视图分离

2.基本语法

```
<div id="app">
  {{ message }}
</div>
<script>
    var app = new Vue({
      el: '#app',
      data: {
        message: 'Hello Vue!'
      }
    })
</script>
```

双大括号插值表达式：双大括号其实提供了一个js执行环境，可以写任意js表达式，不识别html结构

一个元素中可以有多个插值表达式

数据统一存放在data中

3.指令

注意：所有指令后面的引号是提供一个js执行环境的，不是字符串的引号

1.v-html

把一段html结构渲染到它所绑定的元素中

如果元素本身内部有内容，会被指令中的内容覆盖掉

2.v-text

把一段文本内容渲染到它所绑定的元素中

3.v-bind

动态绑定属性

```
v-bind:属性名="数据"
:属性名="数据"
```

v-bind的特殊情况

- class
  - 可以绑定一个字符串，字符串名就是class名
  - 可以绑定一个对象，对象的属性名就是class名，对象的属性值是布尔，代表是否有这个class
  - 可以绑定一个数组，数组里是变量名，变量值就是class名字
- style
  - 绑定到一个对象，对象就是样式对象
  - 绑定到一个数组，就是绑定到多个样式对象

4.v-on

```
v-on:事件名="函数名"      缩写为 @事件名="函数"
```

函数如果有参数，在指令里面直接加括号就行了，没有参数就不需要加括号

关于事件对象：

```
v-on:事件名="(e)=>{fn(e,arg)}"
```

按键修饰符：

在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 v-on 在监听键盘事件时添加按键

```
<input v-on:keyup.enter="submit">
<input v-on:keyup.page-down="onPageDown">
```

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

- `.enter`
- `.tab`
- `.delete` (捕获“删除”和“退格”键)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

5.v-if v-else

控制整个元素是否渲染，不是基于样式的， 是真正的条件渲染

```
<h1 v-if=" grades=='A' ">优秀</h1>
<h1 v-else-if="grades=='B'">良好</h1>
<h1 v-else-if="grades=='C'">及格</h1>
<h1 v-else>不及格</h1>
```

注意，使用v-if的时候

- v-if的元素是和v-else必须是同级
- v-if的元素和v-else的元素必须紧挨着，中间不允许夹杂其他元素

6.v-show

基于样式的切换

v-if 与 v-show的性能对比

> v-if 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。
>
> v-if 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。
>
> 相比之下，v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

> 一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。

7.v-for

1.基本用法

- 可以循环数组 value index
- 可以循环对象 value key index
- 可以循环整数 item

> 建议尽可能在使用 `v-for` 时提供 `key` attribute
>
> 除非遍历输出的 DOM 内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。

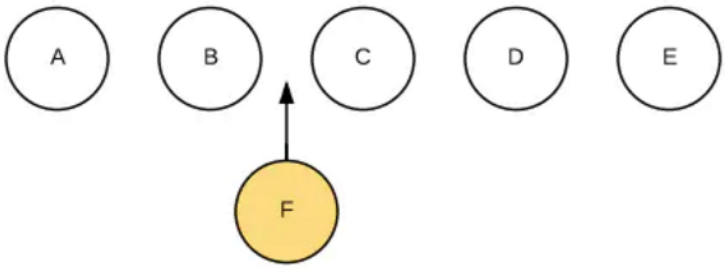
注意：v-if和v-for尽量避免同时使用：

因为v-for的优先级比v-if更高

2.key的作用

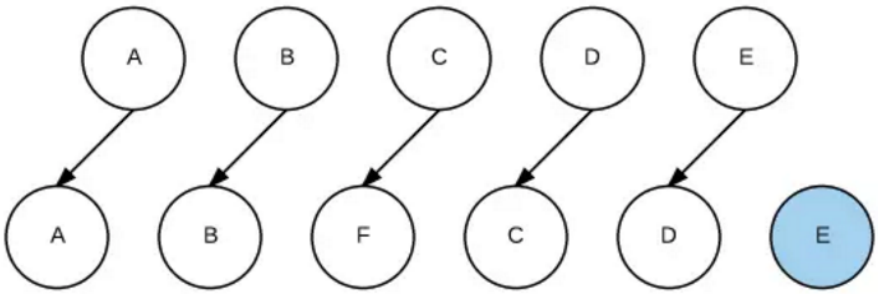
为什么需要key?

比如一下这个情况：



image

我们希望可以在B和C之间加一个F，Diff算法默认执行起来是这样的：



image

即把C更新成F，D更新成C，E更新成D，最后再插入E，是不是很没有效率？
所以我们需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点，找到正确的位置区插入新的节点。

```
<div id="app">
  <input type="text" v-if="see"/>
  <input type="password" v-else />
    <button @click="change">更改</button>
</div>

<script>
new Vue({
  el: '#app',
  data: {
    see:true
  },
    methods:{
        change(){
            this.see=false
        }
    }
})
</script>
```

> 尽量不要使用索引值index作key值，一定要用唯一标识的值，如id等。因为若用数组索引index为key，当向数组中指定位置插入一个新元素后，因为这时候会重新更新index索引，对应着后面的虚拟DOM的key值全部更新了，这个时候还是会做不必要的更新，就像没有加key一样，因此index虽然能够解决key不冲突的问题，但是并不能解决复用的情况。如果是静态数据，用索引号index做key值是没有问题的。

3.响应式原理

由于 JavaScript 的限制，Vue **不能检测**数组和对象的变化

对于对象来说:

property 必须在 data 对象上存在才能让 Vue 将它转换为响应式的

对于已经创建的实例，Vue 不允许动态添加根级别的响应式 property

```
//可以使用如下方法嵌套对象添加响应式 property
Vue.set(object, propertyName, value)
```

对于数组来说:

直接通过角标赋值是无法设置为响应式的

直接修改长度也是无法设置为响应式的

```
//可以使用如下方法为数组添加响应式 property
Vue.set(vm.items, indexOfItem, newValue)
```

4.数组更新检测

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

8.v-model

1.基本使用

表单的双向绑定

v-model默认绑定到表单的value属性

- 多选框： 单个多选框，绑定到布尔值，多个复选框，绑定的是数组
- 单选框：值直接绑定到值
- 下拉选择框：直接绑定到值

双向绑定也可以绑定倒数组的某一项或者对象的某一项

2.修饰符

```
<!-- 在“change”时而非“input”时更新 -->
<input v-model.lazy="msg">
<!--如果想自动将用户的输入值转为数值类型，可以给 v-model 添加 number 修饰符：-->
<input v-model.number="num">
<!--如果要自动过滤用户输入的首尾空白字符，可以给 v-model 添加 trim 修饰符：-->
<input v-model.trim="msg">
```

4.methods

方法的集合，鼓励使用es6的形式

5.computed

1. 计算属性是个函数，要求一定有返回值
2. 不能和data中的数据重名、
3. 计算属性是基于缓存的，只有在他计算依赖的项有变化的时候他才会重新运算，否则不运算
    1. 依赖：此函数中用到的数据有变化，那么计算属性都会重新运行。
4. 计算属性最终的结果是个属性，不能加括号
5. 计算属性也是可以传参的，只需要让返回值是个函数即可

> 计算属性跟函数的区别?
>
> 1.计算属性一定要返回值，函数不一定
>
> 2.函数需要主动调用，计算属性依赖的项变化的时候才执行
>
> 3.计算属性比函数性能高，因为是基于缓存的

6.watch

> 监听某个数据，只要该数据变了，就会执行函数

1.函数名必须是一个已存在的数据名，函数名即是你所监听的数据名

2.首次加载不执行，只有在后续数据变化的时候才会运行

3.检测的数据可以是data中的，也可以是computed

computed和watch的异同点?

1.计算属性是返回一个值，watch是监听一个值做某件事情的。

2.计算属性是值变化的时候重新运算，watch值变化的时候执行函数

3.都不需要主动调用

7.组件

1.组件的两种注册方式

- 全局注册
- 局部注册

注意：1.组件标签只能有一个根节点，最外层要由一个大的div包裹

2.组件名在js中是使用驼峰式，在html中改为以横线连接的小写字母

3.组件中的data必须是函数，并且必须要return一个对象，就是我们的数据对象

```
//全局注册
Vue.component("组件名",{template:"html代码", data(){},methods:{},computed:
{},watch:{})
//局部注册
var head = {
            template: `<div><h1 >{{msg}}</h1></div>`,
            data: function () { return { msg: "我是头部" } },
        }

new Vue({
            el: "#app",
            data: {},
            components: {
                myhead:head,
            }
        })
```

2.组件之间的传值

> 1.父- 子 props
>
> 2.子 -父 自定义事件
>
> 3.同级：先子给父，再父给子

组件之间传值的三种情况

1.父-子：props(属性)

注意：子组件不允许直接修改父组件传入的属性，如果需要更新(更改)这个属性，只能让父组件重新传

用法：

1.子组件内部要接收属性并规定属性的使用场景

2.父组件中使用子组件要给属性传实际的数据

> 父子传值案例

```
    <div id="app">
        <my-com :a="msg" :b="change" :c="arr"></my-com>
    </div>
```

```
    <script>
        const myCom={
            props:["a","b","c"],
            template:`<div>
                        <h1 @click="b">我是组件：{{a}}</h1>
                        <ul>
                            <li v-for="item in c">{{item}}</li>
                        </ul>
                      </div>`,
        }
        new Vue({
            el:"#app",
            data:{
                box:"welcome",
                msg:"我是父组件的数据",
                info:"哈哈",
                arr:[1,2,3,4]
            },
            methods:{
                change(){
                    console.log(this)
                }
            },
            components:{
               //组件名:组件配置项
               myCom
            }

        })
```

2.子-父：自定义事件

> 子父传值案例

```
 <div id="app">
        <h1 >我是父组件{{msg}}</h1>
       <child @abc="fn"></child>
    </div>
<script>
const child={
                data(){
                    return{
                        info:"我是子组件的数据"
                    }
                },
                template:`<h1 @click="change">我是子组件</h1>`,
                methods:{
                    change(){
                        //$emit用于主动触发绑定在本组件标签上的自定义事件
                        //$emit(自定义事件名,参与方法的参数)
                        this.$emit("abc",this.info)
                    }
                }
            }
            new Vue({
                el:"#app",
```

```
                data:{
                    msg:""
                },
                components:{
                    child
                },
                methods:{
                    fn(m){
                      this.msg=m
                    }
                }
            })
</script>
```

8.插槽

插槽：其实就是给组件标签中的内容(可以是多个标签)预留的位置

如果组件模板中有多个插槽，那么组件标签的内容将会复制，插入到多个插槽中

1.普通插槽

```
    <div id="app" >
        <my-head>
            <h1>hello</h1>
        </my-head>
    </div>
    <script>
        var head = {
            template: `<div>
                            <slot></slot>
                            <h1 >{{msg}}</h1>
                       </div>`,
            data: function () { return { msg: "我是头部" } },
        }
        new Vue({
            el:"#app",//挂载点
            components:{
                myHead:head
            }
        })
    </script>
```

2.具名插槽

```
<body>
    <div id="app" >
        <my-head>
            <template v-slot:before> <!--v-slot可以缩写为#-->
```

```
                <h1>hello</h1>
            </template>
            <template v-slot:after>
                <h1 >world</h1>
            </template>

        </my-head>
    </div>

    <script>
        var head = {
            template: `<div>
                            <slot name="before"></slot>
                            <h1 >{{msg}}</h1>
                            <slot name="after"></slot>
                        </div>`,
            data: function () { return { msg: "我是头部" } },
        }
        new Vue({
            el:"#app",//挂载点
            components:{
                myHead:head
            }
        })
    </script>
</body>
```

3.编译作用域

> 如果插槽模板中想使用该组件内部的数据，首先需要在模板的插槽中定义v-bind:属性="数据"
>
> 组件调用的地方插槽通过 v-slot:属性="prop" prop代表所有属性的集合

```
<body>
    <div id="app">
        <mycom>

            <template #before="{abc}">
                <h1>{{abc}}</h1>
            </template>
            <template #after>
                <p>123456</p>
            </template>
            <template #middle>
                <h3>就是开心</h3>
            </template>

        </mycom>
    </div>
    <script>
        const mycom = {
            data(){
                return{
                    msg:"我是子组件的数据"
                }
```

```
            },
            template: `
                <div>
                    <h1>我是组件</h1>
                    <slot name="before" v-bind:abc="msg" ></slot>
                    <ul>
                        <slot name="middle"></slot>
                        <li>1</li>
                        <li>2</li>
                    </ul>
                    <slot name="after"></slot>
                </div>
            `
        }
        new Vue({
            el: "#app",
            components: {
                mycom
            },
            data:{
                msg:"我是父组件"
            }
        })
    </script>
</body>
```

9.组件的生命周期

生命周期：组件从创建到销毁的完成过程

生命周期(钩子)函数：在特定的时间节点会自动触发的函数

```
  beforeCreate(){
    console.log("组件创建之前")
  },
  created(){
    console.log("组件创建完毕")
  },
  beforeMount(){
    console.log("组件渲染前")
  },
  mounted(){
    console.log("组件挂载完毕")
  },
   //1.页面重新渲染的时候触发
   //2.子组件用到的属性渲染到了视图中
  beforeUpdate(){
    console.log("组件更新前")
  },
  updated(){
    console.log("组件更新后")
  },
  beforeDestroy(){
    console.log("组件销毁前")
```

```
  },
  destroyed(){
    console.log("组件销毁后")
  },
```

10.动态组件

```
<div id="app">
        <component v-bind:is="title"></component>
        <button @click="change(2)">按钮</button>
    </div>
    <script>
        var coma = {
            template: `<div><h1>我是第一个组件</h1></div>`,
        }
        var coma2 = {
            template: `<div><h1>我是第二个组件</h1></div>`,
        }
        var coma3 = {
            template: `<div><h1>我是第三个组件</h1></div>`,
        }
        var app = new Vue({
            el: "#app",
            data: {
                tab: "mycom",
                title:"mycom"
            },
            components: {
                mycom: coma,
                mycom2: coma2,
                mycom3: coma3,
            },
            methods:{
                change(n){
                    this.title="mycom"+n
                }
            }
        })
    </script>
```

11.Vue脚手架

> 什么是脚手架?

Vue脚手架是Vue官方提供的标准化开发工具(开发平台),它提供命令行和UI界面,方便创建vue工程、配置第三方依赖、编译vue工程。

> 如果没有脚手架，我们要如何搭建项目?

- 通过npm init初始化项目
- 安装webpack和脚手架
    - npm install webpack webpack-cli -D

- 配置ES6/7/8转ES5代码
  - npm install babel-loader @babel/core @babel/preset-env -D
- 创建webpack.config.js文件，配置webpack
- 安装html-webpack-plugin依赖
  - npm install html-webpack-plugin -D
- 安装vue-loader
  - npm install vue-loader
- 安装 vue-template-compiler
  - npm install vue-template-compiler
- 安装 cache-loader 用于缓存loader编译的结果
  - npm install cache-loader
- 安装less loader 安装sass-loader 安装style-loader file-loader url-loader postcss-loader autoprefixer
- 安装webpack-dev-server
- 安装 vue-router vuex axios
- 配置webpack

> 脚手架的安装

```
npm install -g @vue/cli
```

> 利用脚手架创建项目

```
vue create 项目名字
```

> 项目文件解读

node_modules:核心模块

public：存放静态文件

src：最重要的文件，我们要编写的代码基本都位于src目录下

gitignore：git上传需要忽略的文件格式

babel.config.js:主要用于在当前和较旧的浏览器或环境中将ES6代码转化为向后兼容的版本

jsconfig.json: -jsconfig.json文件指定根目录和JavaScript服务提供的功能选项。

package-lock.json:锁定安装模块的版本号

package.json:模块基本信息，依赖的模块，名称，版本号

README.md:对项目的主要信息进行描述

main.js:是项目的入口文件

关于render函数

render: h => h(App)的原理

等同于 :h=>render{return h(App)}

等同于 render:function(h){return h(App)}

等同于render:function(createElement){return createElement(App)}

12.单文件组件

单文件组件：一个文件就是一个组件，都是以.vue结尾的，允许正常的编写html,支持模块化的样式

如果没有单文件组件的话：

1.多个组件写在一个页面中，乱

2.没有html提示，没有语法高亮

3.样式的模块化很明显被遗漏了

13.vue-router

1.安装

```
npm install vue-router
//安装指定版本
npm install vue-router@3.2.0
```

2.使用步骤

1.引入vue、 vue-router

2.注册VueRouter

3.定义路由配置项

4.创建VueRouter实例，传入配置项

3.路由的两个核心

（1）定义路由配置项

（2）定义路由视图的显示位置(router-view)

> 路径定义规则：路径以斜线打头
>
> 页面与路由一定是一一对应的，一个页面一定要对应路由，否则找不到

路由的懒加载：

```
component: () => import('../views/About.vue')
```

4.关于路径的写法

```
import Abc from "./demo"
```

1.如果当前目录下有demo文件夹，那么是去引入的demo文件夹下的index.js
2.如果不存在demo文件夹，会有选去找当前目录下的demo.vue文件
3.如果不存在.vue文件，那么会优先查找当前目录下的demo.js文件

路径中@代表的是src目录，本质上是因为webpack内部配置的

5.嵌套路由

每一级路由都必须要有该级路由的router-view

哪个组件还有子路由，那么需要在该组件内部定义router-view，代表该组件的子路由显示在该位置上

子路由的配置，定义在父路由的children属性下，配置内容同父路由

子路由如果不加斜线，那么默认在上级路由上叠加，如果加了斜线，那么前面必须带上 上级路由

6.路由的跳转

> **1.js形式的跳转**

如果看到了某个方法是以$打头的，基本可以断定，该方法是Vue实例的方法

```
this.$router.push(字符串路径)
router.push({ path: 'home' })
router.push({ name: 'user'})
```

> 2.标签形式的跳转

```
<router-link to="/about"> </router-link>
// to 可以是字符串路径      还可以是  对象     {path:"路径"}  {name:"名字"}
```

> 3.浏览器的跳转

```
// 在浏览器记录中前进一步，等同于 history.forward()
router.go(1)

// 后退一步记录，等同于 history.back()
router.go(-1)

// 前进 3 步记录
router.go(3)

// 如果 history 记录不够用，那就默默地失败呗
router.go(-100)
router.go(100)
```

7.动态路由

多个路由匹配同一个页面

动态路由匹配 在路径后面加/:名字

如:

```
{ path: '/users/:id', component: User },
 //可以同时存在多个动态路由
{ path: '/users/:id/:no', component: User },
```

> 注意只改后面动态路径参数是不会重新触发组件生命周期的
>
> 但是如果页面中用到了动态路由的数据，那么切换动态路径的时候，是会触发组件更新的生命周期的

动态路由的跳转:

```
1.router.push({ name: 'user', params: { userId: '123' }})
2.router.push({ path: '/user/123' })
2.router.push( '/user/123')
//下面的写法是错误的
router.push({ path: '/user', params: { userId: '123' }})//如果提供了 path，params 会被忽略
```

同样的规则适用于router-link标签

8.查询参数

用于跨组件传递数据

跳转的时候直接在地址后面拼接?key=value

```
router.push({ path: '/register', query: { plan: 'private' }})
router.push({ path: '/register?key=value'}})
router.push("/register?key=value")
```

注意 name和params配套

path和query 配套

8.捕获所有路由

```
{
  // 会匹配所有路径
  path: '*'
}
{
  // 会匹配以 `/user-` 开头的任意路径
  path: '/user-*'
}
```

> 当使用*通配符*路由时，请确保路由的顺序是正确的，也就是说含有*通配符*的路由应该放在最后
>
> 新版本不限制 通配符的存放位置，也就是说就算是放在最开始，其他路由也是能匹配上的，不过不推荐

9.组件路由对象

在组件中通过this.$route拿到当前组件的路由信息

```
{name:"组件的名字",fullPath:"完整路径",path:"路径",matched:[匹配上的所有路由],meta:"路由元信息", params:"动态路径参数",query:"查询参数"}
```

10.重定向和别名

重定向：“重定向”的意思是，当用户访问 /a时，URL 将会被替换成 /b，然后匹配路由为 /b

redirect:"要跳转的路径"

重定向的目标也可以是一个命名的路由：

```
const router = new VueRouter({
  routes: [
    { path: '/a', redirect: { name: 'foo' }}
  ]
})
```

别名：/a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样

```
const router = new VueRouter({
  routes: [
    { path: '/a', component: A, alias: '/b' }
  ]
})
```

> “别名”的功能让你可以自由地将 UI 结构映射到任意的 URL，而不是受限于配置的嵌套路由结构。

11.路由组件传参

可以将params直接映射到组件的属性中

之前的做法：

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
const router = new VueRouter({
  routes: [{ path: '/user/:id', component: User }]
})
```

用了组件传参后

```
const User = {
  props: ['id',"num"],
  template: '<div>User {{ id }} {{ num }}</div>'
}
const router = new VueRouter({
  routes: [
    { path: '/user/:id/:num', component: User, props: true },
  ]
})
```

12.路由模式

hash模式和history模式

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

nginx

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

13.导航守卫

1.全局前置守卫

```
const router = new VueRouter({ ... })

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

- `to: Route`: 即将要进入的目标 路由对象
- `from: Route`: 当前导航正要离开的路由

- `next: Function`：一定要调用该方法来 **resolve** 这个钩子。执行效果依赖 `next` 方法的调用参数。
  - `next()`：进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 **confirmed** (确认的)。
  - `next(false)`：中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 `from` 路由对应的地址。
  - `next('/')` **或者** `next({ path: '/' })`：跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 `next` 传递任意位置对象，且允许设置诸如 `replace: true`、`name: 'home'` 之类的选项以及任何用在 `router-link` 的 `to` prop 或 `router.push` 中的选项。
  - `next(error)`：(2.4.0+) 如果传入 `next` 的参数是一个 `Error` 实例，则导航会被终止且该错误会被传递给 `router.onError()` 注册过的回调。

注意避免死循环，下面的写法是死循环

```
router.beforeEach((to,from,next)=>{
  if(sessionStorage.getItem('loginData')){
    Toast('跳转成功');
    next();
  }else {
    //没有登录，去跳转登录页
    next({
      path:'/login'
    });
  }
});
```

加个验证就ok了

```
router.beforeEach((to,from,next)=>{
  // console.log(to);
  // console.log(from);
  if(sessionStorage.getItem('loginData')){
    Toast('跳转成功');
    next();
  }else {
    //没有登录，去跳转登录页
    if(to.path === '/login'){
      next();
    }else {
      next({
        path:'/login'
      });
    }
  }
});
```

2.全局解析守卫

在 2.5.0+ 你可以用 router.beforeResolve 注册一个全局守卫。这和 router.beforeEach 类似，区别是在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用。

3.全局后置钩子

```
//和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身：
router.afterEach((to, from) => {
  // ...
})
```

4.路由独享的守卫

```
    export default {
        mounted(){
            console.log(this.$route)
        },
        beforeRouteEnter(to, from, next) {
        // 在渲染该组件的对应路由被 confirm 前调用
        // 不！能！获取组件实例 `this`
        // 因为当守卫执行前，组件实例还没被创建
      },
      beforeRouteUpdate(to, from, next) {
        // 在当前路由改变，但是该组件被复用时调用
        // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的
时候，
        // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调
用。
        // 可以访问组件实例 `this`
      },
      beforeRouteLeave(to, from, next) {
```

```
        // 导航离开该组件的对应路由时调用
        // 可以访问组件实例 `this`
      }
    }
```

5.完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用 `beforeRouteLeave` 守卫。
3. 调用全局的 `beforeEach` 守卫。
4. 在重用的组件里调用 `beforeRouteUpdate` 守卫 (2.2+)。
5. 在路由配置里调用 `beforeEnter`。
6. 解析异步路由组件。
7. 在被激活的组件里调用 `beforeRouteEnter`。
8. 调用全局的 `beforeResolve` 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 `afterEach` 钩子。
11. 触发 DOM 更新。
12. 调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数，创建好的组件实例会作为回调函数的参数传入。

14.路由元信息

定义路由的时候可以配置 meta 字段

meta可以设置一些跟当前路由相关的自定义信息，比如页面标题，面包屑导航，页面权限等。

15.滚动行为

```
const router = new VueRouter({
  scrollBehavior (to, from, savedPosition) {
      //to from是路由对象

    // return 期望滚动到哪个的位置    x是离左边距离   y是离顶部距离
    //有滚动条的前提下才生效，没有滚动条不生效
    return  { x: 0, y: 1000 }
  }
})
```

14.Vuex

> **注意:**
>
> Vuex 3.x版本是与Vue2.0版本配套
>
> Vuex4.x版本是与Vue3.0版本配套

> **概念:**
>
> Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式 + 库。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化
>
> 这么说吧，将vue想作是一个js文件、组件是函数，那么vuex就是一个全局变量，只是这个“全局变量”包含了一些特定的规则而已。

1.使用

```
npm install vuex
```

注意，安装的时候如果有以下报错,说明是安装版本过高导致，安装低版本即可解决

```
npm ERR! Could not resolve dependency:
npm ERR! peer vue@"^3.0.2" from vuex@4.0.2
npm ERR! node_modules/vuex
npm ERR!   vuex@"*" from the root project
npm ERR!
npm ERR! Fix the upstream dependency conflict, or retry
npm ERR! this command with --force, or --legacy-peer-deps
npm ERR! to accept an incorrect (and potentially broken) dependency resolution.
npm ERR!
npm ERR! See C:\Users\Administrator\AppData\Local\npm-cache\eresolve-report.txt for a full report.
```

解决方案:

```
npm install vuex@3.6.2
```

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  }
})
//main.js
const app = new Vue({
  el: '#app',
  // 把 store 对象提供给 “store” 选项，这可以把 store 的实例注入所有的子组件
  store,
})
```

2.五大核心概念

1.state

用来存放共享数据的

```
const store = new Vuex.Store({
  state: {
    count: 10,
    like:["篮球","足球"],
    movie:{"name":"速度与激情",actor:"保罗沃克"}
  }
})
```

2.getters

类似于计算属性 ，默认接受第一个参数，该参数是所有的state，其他的用法跟computed是一模一样的

第二个参数是所有的getters

1.基本用法

`index.js`

```
const store = new Vuex.Store({
  state: {
    count: 10
  },
  getters:{
    money(state){
       return state.count+"元"
    }
  }
})
```

`App.vue`

```
 mounted(){
    console.log(this.$store.getters.money) //10元
  },
```

2.getters也可以传参

```
 getters:{
    meiyuan:state=>(rate)=>{
      return state.rmb/rate
    }
  },
```

3.getters的第二个参数:所有getters

```
  getters:{
    money(state,getters){
       return getters.info+"元"
    },
    info(state){
        return state.count*2
    }
  }
```

3.mutations

1.修改state数据的唯一办法就是提交mutation

2.mutations中修改的数据如果是引用类型，一定注意要完整的赋值（不能单独修改某一项）

1.基本用法

就是你想做的事情的列表，mutation要通过commit提交

```
const store = new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment (state) {
      // 变更状态
      state.count++
    }
  }
})

export default store
```

调用

```
store.commit('increment')
```

2.载荷（payload）

可以向 store.commit 传入额外的参数，即 mutation 的载荷（payload）:

```
mutations: {
  increment (state, n) {
    state.count += n
  }
}

//调用
store.commit('increment', 10)
```

大多数情况下，载荷应该是一个对象，这样可以包含多个字段并且记录的 mutation 会更易读

```
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
//调用
mutations: {
  increment (state, payload) {
    state.count += payload.amount
  }
}
```

3.对象风格的提交方式

```
mutations: {
  increment (state, payload) { //此时payload是包含着type的
    state.count += payload.amount
  }
}
//调用
store.commit({
  type: 'increment',
  amount: 10
})
```

4.常量替代 Mutation 事件类型

`index.js`

```
import Vue from 'vue'
import Vuex from 'vuex'
import {ADD,MINUS} from "./mutation-type"
Vue.use(Vuex)
const store = new Vuex.Store({
  state: {
    count: 10
  },
  mutations:{
    [ADD](state){
        return state.count++
    },
    [MINUS](state){
        return state.count--
    }
  }
})

export default store
```

`mutation-type.js`

```
export const ADD = 'add';
export const MINUS = 'minus';
```

App.vue

```
import {ADD} from "./store/mutation-type"
export default{
  data(){
    return{
      msg:"我是App组件"
    }
  },
  mounted(){
   this.$store.commit(ADD)
    console.log(this.$store.state.count)//11
  },
```

> **5.Mutation 必须是同步函数**

为何mutation不能包含异步操作?

使用层面：代码更高效易维护, 逻辑清晰（规范，而不是逻辑的不允许）；
具体原因：为了让devtools 工具能够追踪数据变化;
具体原因详解
每个mutation执行完成后都会对应到一个新的状态变更，这样devtools就可以打个快照存下来（每次状态的改变都会生产一个全新的 state 对象），然后就可以实现 “time-travel” 了。如果mutation支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难。

注： vue-devtools 的时间旅行 - time travel
Vuex 借鉴 Flux 单向数据流思想，采用集中式统一管理保存状态。但是这些状态不会随时间改变而变化。为了使状态，可以被捕获、重播或重现。vue-devtools工具，可以帮助我们的 Vue 应用可以实现这种时间旅行!
每次状态的改变都会生产一个全新的 state 对象，当你想展现什么时间段的状态，只需要切换到那个时间段的 state 对象，所以vuex原则上只能通过mutation 并且非异步更改状态，否则无法实现state修改的记录保留或无法正确记录。

4.actions

> Action 类似于 mutation，不同在于：
>
> Action 提交的是 mutation，而不是直接变更状态。
> Action 可以包含任意异步操作。

演示代码：

```
const store = new Vuex.Store({
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
```

```
      context.commit('increment')
    }
  }
})
```

Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。

分发action：

```
store.dispatch('increment')
//带参数的形式

// 以载荷形式分发
store.dispatch('incrementAsync', {
  amount: 10
})

// 以对象形式分发
store.dispatch({
  type: 'incrementAsync',
  amount: 10
})
```

组合action：

> 1.dispatch函数返回值是promise

```
actions: {
  actionA ({ commit }) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        commit('someMutation')
        resolve()
      }, 1000)
    })
  }
}
```

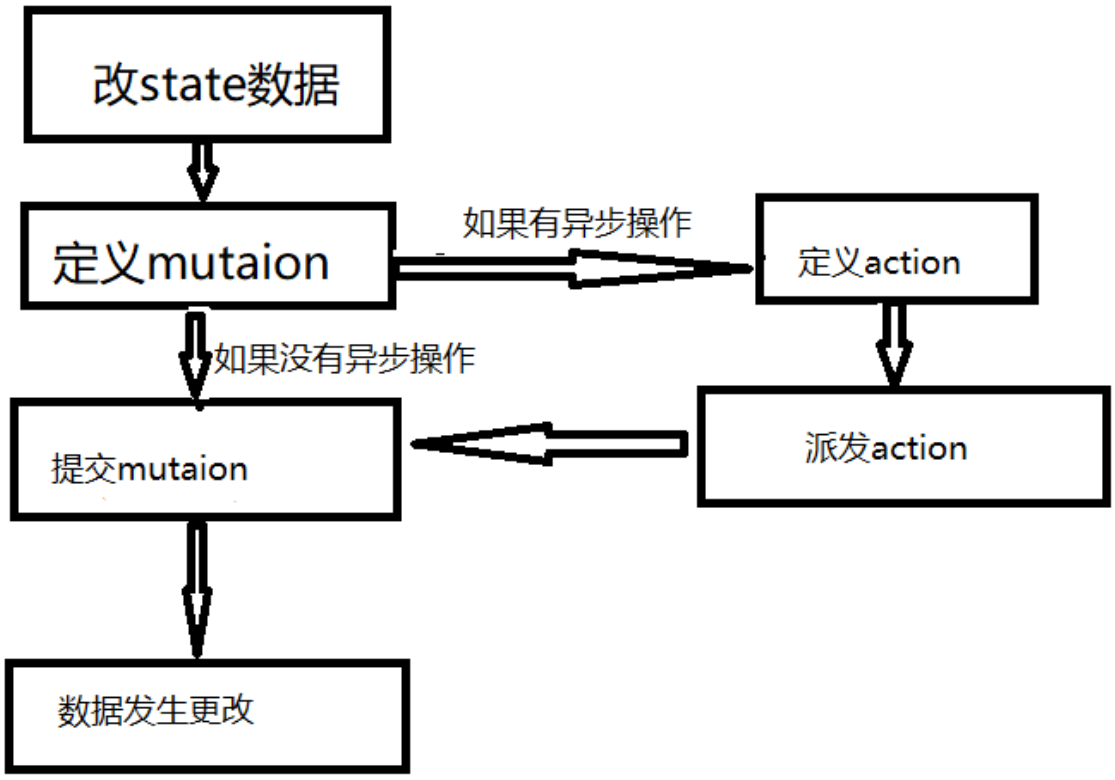
```
store.dispatch('actionA').then(() => {
  // ...
})
```

```
actions: {
  // ...
  actionB ({ dispatch, commit }) {
    return dispatch('actionA').then(() => {
      commit('someOtherMutation')
    })
  }
}
```

```
// 假设 getData() 和 getOtherData() 返回的是 Promise

actions: {
  async actionA ({ commit }) {
    commit('gotData', await getData())
  },
  async actionB ({ dispatch, commit }) {
    await dispatch('actionA') // 等待 actionA 完成
    commit('gotOtherData', await getOtherData())
  }
}
```

5.mutaions actions流程



6.modules

1.基本用法

> Vuex 允许我们将 store 分割成模块（module）。每个模块拥有自己的 state、mutation、action、getter

单独定义一个模块跟我们之前定义store是一样的

`moduleA.js`

```
export default{
    state:{
        text:"我是moduleA的数据"
    },
    getters:{},
    mutations:{},
    actions:{}
}
```

`moduleB.js`

```
export default{
    state:{
        text:"我是moduleB的数据"
    },
    getters:{},
    mutations:{},
    actions:{}
}
```

`store-index.js`

```
import Vue from "vue";
import Vuex from "vuex";
import moduleA from "./moduleA"
import moduleB from "./moduleB"
Vue.use(Vuex);
const store=new Vuex.Store({
    state:{
    },
    modules:{
       // 模块名:模块对象
        moduleA,moduleB
    }
})

export default store
```

读取数据语法：

```
this.$store.state.模块名.数据名
//例如
this.$store.state.moduleB.text
```

2.多层嵌套

每一个子模块依旧可以有自己的子模块，比如moduleA里添加一个moduleC

`moduleA.js`

```
import moduleC from "./moduleC"
export default{
    state:{
        text:"我是moduleA的数据"
    },
    getters:{},
    mutations:{},
    actions:{},
    modules:{
        moduleC
    }
}
```

`moduleC.js`

```
export default{
    state:{
        text:"我是moduleC的数据"
    },
    getters:{},
    mutations:{},
    actions:{}
}
```

访问：

```
this.$store.state.模块名.子模块名.数据名
//例如
this.$store.state.moduleA.moduleC.text
```

3.命名空间

> 默认情况下，模块内部的 action、mutation 和 getter 是注册在全局命名空间的

比如：moduleA和moduleB都定义了一个叫changeText的mutation

`moduleA.js`

```
import moduleC from "./moduleC"
export default{
    state:{
        text:"我是moduleA的数据"
    },
    getters:{},
    mutations:{
        changeText(state){
            state.text="moduleA的数据改了"
        }
    },
    actions:{},
    modules:{
        moduleC
    }
}
```

moduleB.js

```
export default{
    state:{
        text:"我是moduleB的数据"
    },
    getters:{},
    mutations:{
        changeText(state){
            state.text="moduleB的数据改了"
        }
    },
    actions:{}
}
```

MyHome.vue

```
 computed:{
       textA(){
            console.log(this.$store.state)
             return this.$store.state.moduleA.text
       },
       textB(){
            console.log(this.$store.state)
             return this.$store.state.moduleB.text
       }
        },
      methods:{
          //我们会发现我们直接提交mutation不需要加模块名字，A和B的mutation都能触发
          changeText(){
              this.$store.commit("changeText")
          }
      }
```

getters也是同理，多个模块都有同一个getter，默认读取的是根store的getters，如果根store中没有，则按照模块注册顺序读取

> **开启命名空间：**
>
> 可以通过添加 namespaced: true 的方式使其成为带命名空间的模块。
>
> 当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名

示例：

moduleA.js

```
import moduleC from "./moduleC"
export default{
    namespaced: true, //开启命名空间
    state:{
        text:"我是moduleA的数据"
    },
    getters:{
```

```
        info(){
            return "哈哈"
        }
    },
    mutations:{
        changeText(state){
            state.text="moduleA的数据改了"
        }
    },
    actions:{},
    modules:{
        moduleC
    }
}
```

此时修改数据需要加上模块名

```
this.$store.commit("moduleA/changeText")//代表提交moduleA里的changeText
//对于嵌套模块
this.$store.commit("moduleA/moduleC/changeText")//提交moduleA里的moduleC里的
changeText
//对于getters
this.$store.getters["moduleA/info"]
//对于actions
this.$store.dispatch("moduleA/actionA")
```

4.访问全局内容

在开启了命名空间的模块内部，也是可以访问到全局数据的

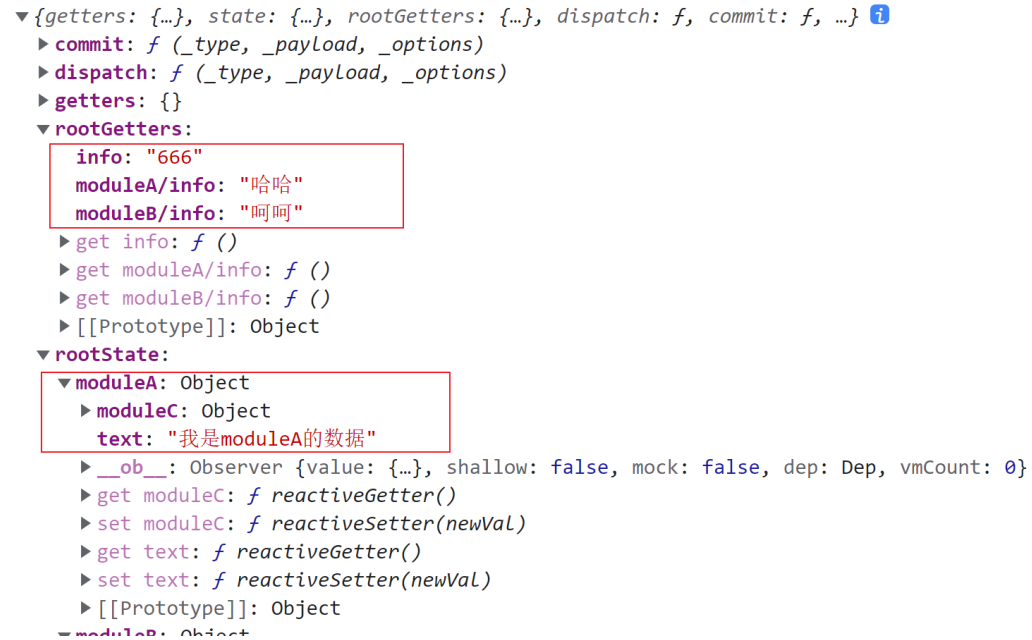
> getters中可以拿到全局的state和getters

```
someGetter (state, getters, rootState, rootGetters) {
    getters.someOtherGetter // -> 'foo/someOtherGetter'
    rootGetters.someOtherGetter // -> 'someOtherGetter'
 },
```

> actions中可以拿到所有的state和getters

```
 actions:{
        actionA(context){
            console.log(context)
        }
    },
```

输出内容:



> 在模块内部提交全局的action或者mutation
>
> 将 `{ root: true }` 作为第三参数传给 `dispatch` 或 `commit` 即可。

```
dispatch('someOtherAction') // -> 'foo/someOtherAction'
dispatch('someOtherAction', null, { root: true }) // -> 'someOtherAction'

commit('someMutation') // -> 'foo/someMutation'
commit('someMutation', null, { root: true }) // -> 'someMutation'
```

5.带命名空间的辅助函数

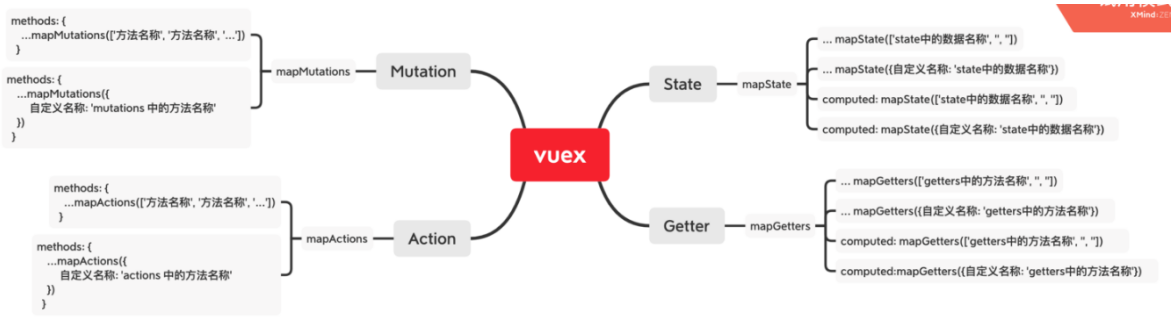
```
...mapMutations("moduleA",["changeText"])
...mapState("moduleA/moduleC",["text"]),
```

还可以通过使用 `createNamespacedHelpers` 创建基于某个命名空间辅助函数。它返回一个对象，对象里有新的绑定在给定命名空间值上的组件绑定辅助函数：

```
import { createNamespacedHelpers  } from 'vuex'
const { mapState } = createNamespacedHelpers('moduleA/moduleC')

  computed:{
            ...mapState(["text"]),//这样text默认就是在moduleA/moduleC下的
  }
```

3.辅助函数



> 使用:

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

mapState函数返回值是一个对象

初步优化: （公司里肯定不会这么写）

```
computed:mapState({
    money:state=>state.money
    a:state=>state.a
})
```

二次优化（99%不会这么写）

```
 computed:mapState({
            //前面是组件中计算属性的名字 后面是vuex中的名字
            money:"money",
            a:"a",
            b:"b",
            c:"c",
        })
```

最终优化（工作中推荐）

```
 computed:mapState(["money","a","b","c"])
```

```
//使用扩展运算符将辅助函数和其他计算属性混合到一起，组成一个对象
        data(){
            return{
                count:100,
                gender:1
            }
        },
        //mapState
        computed:{
           ...mapState(["money","a","b","c"]),
            sex(){
                return this.gender==1?"男":"女"
            }
        },
```