# VUE随堂笔记

## Vue.js框架是什么，为什么学习它

VueJs作为国人开发的一款前端框架，不仅有强大的社区支持，很多API都是中文的，学习难度相对小；近几年来，得益于手机设备的普及和性能的提升，移动端的web需求大量增加，用户的体验也越来越大，功能越来越复杂，交互也越来越酷炫；
此外，接收用户输入的同时，很可能要及时更新视图，比如用户输入不同的内容，页面就会相对应进行更新，点击不同的选项，就会显示不同的状态等等交互效果。一旦这种交互多了，要手动地进行操作，代码就容易变得复杂和难以维护;

像有些页面元素非常多，结构很庞大的网页中，数据和视图如果全部混杂在一起，像传统开发一样全部混合在HTML中，那么要对它们进行处理会十分的费劲，并且如果其中有几个结构之间存在藕断丝连的关系，那么会导致代码上出现更大的问题；
是否还记得你当初写JQuery的时候，有$('#xxx').parent().parent().parent()这种代码呢？当第一次写的时候，你觉得页面元素不多，不就是找这个元素的爸爸的爸爸的爸爸吗，大不了在注释里面写清楚这个元素的爸爸的爸爸的爸爸不就好了。但是万一过几天之后项目组长或者产品经理突然对做的网页提出修改要求，这个修改要求将会影响页面的结构，也就是DOM的关联与嵌套层次要发生改变，

那么$('#xxx').parent().parent().parent()可能就会变成
$('#xxx').parent().parent().parent().parent().parent()了。
这还不算什么，等以后产品迭代越来越快，修改越来越多，而且页面中类似的关联和嵌套DOM元素不止一个，那么修改起来将非常费劲。而且JQuery选择器查找页面元素以及DOM操作本身也是有性能损失的，可能到时候打开这个页面，会变得越来越卡，会变得无从下手，无法维护。。。
我们把HTML中的DOM就可以与其他的部分独立开来划分出一个层次，这个层次就叫做视图层。
Vue 的核心库只关注视图层

## 什么是MVVM

M model 模型
V view 视图
C controller 控制器
viewModel  监控者

## Vue.Js的核心

Vue官网对它的描述：通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。
**1、Vue的数据驱动：**

数据改变驱动了视图的自动更新，传统的做法你得手动改变DOM来改变视图，vuejs只需要改变数据，就会自动改变视图，一个字：爽。再也不用你去操心DOM的更新了，这就是MVVM思想的实现
**2、视图组件化：**

把整一个网页的拆分成一个个区块，每个区块我们可以看作成一个组件。网页由多个组件拼接或者嵌套组成

## Vue-CLi是啥

它是一个vueJs的脚手架工具。说白了就是一个自动帮你生成好项目目录，配置好Webpack，以及各种依赖包的工具;

**为啥要用咧?**

可以帮助你快速开始一个vue项目，给你一套文件结构，包含基础的依赖库，你只需要npm install一下就可以安装，让你不需要为编译或其他琐碎的事情而浪费时间，而且不会限制到你的发挥；

# Vue.js与Webpack有什么关系?

Webpack是一个前端打包和构建工具。如果你之前一直是手写HTML，CSS，JavaScript，并且通过link标签将CSS引入你的HTML文件，以及通过Script标签的src属性引入外部的JS脚本，那么你肯定会对这个工具感到陌生。
WebPack可以看做是模块打包机：它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。
1、热加载 Webpack不止这点功能，它还可以通过安装各种插件来扩展，比如说热加载技术，就是解放键盘的F5键。让我们修改代码，并且按Ctrl+S保存之后，浏览器页面自动刷新变化，不需要我们去手动刷新
2、模块化，让我们可以把复杂的程序细化为小的文件;
3、scss，less等CSS预处理器

# 常用指令

## 什么是指令

指令：是带有v-前缀的特殊属性，通过属性来操作元素

### v-for

定义：根据变量的值来循环渲染元素

```
data:{
     arr:['a','b','c'],
     obj:{name:'abc',id:1},
     lists:[{name:'fangfang',id:1},{name:'fangfang2',id:2},
}    {name:'fangfang3',id:3}]
```

```
<ul>
        <!--遍历数组 v value i index-->
        <li v-for="(v,i) in arr">{{v}}{{i}}</li>
</ul>
<ul>
        <!-- 遍历对象 v value k key-->
        <li v-for="(v,k) in obj">{{v}}==={{k}}</li>
</ul>
<ul>
        <!--v value i index-->
        <li v-for="(v,i) in lists">{{v.name}}==={{i}}</li>
</ul>
```

### v-text | v-html

v-text定义：在元素中插入值

v-html定义：在元素中不仅可以插入文本，还可以插入标签

```
data:{
     msg:'fangfang',
     html:'<span>123</span>'
}
```

```
<!-- v-cloak 解决{{}}浏览器闪烁 -->
<div v-cloak>{{msg}}</div>
<div v-text="msg"></div>
<div v-html="html"></div>
注意:
<style>
    /*必须配置CSS样式，否则不生效*/
    [v-cloak]{
        display:none;
    }
</style>
```

## v-if | v-else

定义：根据表达式的真假值来动态插入和移除元素

```
data:{
     age:21,
     flag:true
}
```

```
<div v-if="age>=30">已经30岁喽</div>
<div v-else-if="age>=20">已经20岁喽</div>
<div v-else>{{age}}</div>
<div v-if="flag?age=18:age=20">{{age}}</div>
```

## v-show

定义：根据表达式的真假值显示和隐藏元素

```
data:{
     flag:true
}
```

```
<div v-show="flag">我显示啦</div>
```

## v-on

定义：监听元素事件，并执行相应的操作

```
data:{
     age:21,
     flag:true
}
methods:{   //方法
    play:function(){
        this.age = 40;
    },
    play2:function(i){
        this.age = i;
    }
}
```

```
<!--事件-->
<button @click="age = 30">click1</button>
<button @click="play">click2</button>
<button @click="play2(10)">click3</button>

<button @mouseover="play2(10)" @mouseout="play2(20)">mouseover</button>

<button @dblclick="play2(50)">dblclick</button>

键盘事件
<button @keydown.enter="play2(13)">keydown</button>
<button @keydown.a="play2(65)">keydown</button>
```

## v-bind

定义：绑定元素的属性并执行相应的操作

```
data:{
     d1:'d1',
     d2:'d2',
     check:true,  //复选框
     backgreen:{background:'green'},
     blue:{color:'blue'},
     red:'red',
     url:'https://www.baidu.com/img/bd_logo1.png'
}
```

```
<style>
    .d1{
        color:#f00;
    }
    .d2{
        border:1px solid #666;
    }
    .green{
        color:green;
    }
    .blue {
        color:blue;
    }
</style>
```

```
<!-- 单个引用 -->
<div class="d1">红色</div>
<!-- 单个引用 'd1'这个表示样式名  d1表示变量，需要在data中定义-->
<div v-bind:class="'d1'">红色</div>
<div :class="d1">红色</div>
```

```
<!-- 多个引用 -->
<div :class="['d1','d2']">红色</div>
<div :class="[d1,d2]">红色</div>
<div :class="[d1,check?d2:'blue']">红色</div>
```

```
<!-- 条件判断 -->
<!-- 三目运算 -->
<input type="checkbox" v-model="check" />{{check}}
<div :class="check ? d1 :'blue'">红色</div>
```

```
<!-- style -->
<!-- 单个引用 -->
<div style="color:red">红色</div>
<div style="color:#f00">红色</div>
```

```
<!-- 变量 -->
<div :style="backgreen">红色</div>
<div :style="{background:'#f60',fontSize:'30px',marginTop:'20px'}">红色</div>
<div :style="{background:red}">红色</div>
<div :style="{background:'blue'}">红色</div>
```

```
<!-- 动态图片 -->
<img :src="url" />
```

## v-model

定义：把input的值和变量绑定，实现数据和视图的双向绑定

```
data:{
    msg:'fangfang',
value:'',
    flag:true,
    flagN:['1'],   //多选框
    flagN2:['跳舞'],    //多选框value的值
    radioP:'1',   //单选框
    items:[{text:'老师',value:'1'},{text:'学生',value:'2'},{text:'家
长',value:'3'}],
    selected:'2',
    selected2:'家长',
    selected3:'梨子',   //下拉框默认值
    selected4:'2',
    num:20
}
```

```
<!-- input输入a则禁用 -->
<input type="text" :disabled="value=='a'" v-model="value"/>
<input type="text" :disabled="flag" v-model="value"/>
```

```
<!-- 复选框   v-model 默认当前的状态true false-->
<input type="checkbox" v-model="flag" />{{flag}}
<!-- 复选框 组 当我们初始化数据的数组里赋予上面的value值时，所对应的checkbox便会默认选中-->
<input type="checkbox" v-model="flagN2" value="跳舞"/>跳舞
<input type="checkbox" v-model="flagN2" value="唱歌"/>唱歌
<input type="checkbox" v-model="flagN2" value="打游戏"/>打游戏
<span>{{flagN2}}</span>
<input type="checkbox" v-model="flagN" value="0"/>跳舞
<input type="checkbox" v-model="flagN" value="1"/>唱歌
<input type="checkbox" v-model="flagN" value="2"/>打游戏
<span>{{flagN}}</span>
```

```
<!-- 单选框 组-->
<input type="radio" v-model="radioP"  value="0"/>男{{radioP}}
<div>
<input type="radio" v-model="radioP" name='test' value="0"/>男
<input type="radio" v-model="radioP" name='test' value="1"/>女
<span>{{radioP}}</span>
</div>
```

```
<!-- 下拉框 获取 当前'梨子'-->
<div>
    <select v-model="selected3">
        <option>苹果</option>
        <option>香焦</option>
        <option>梨子</option>
    </select>
    <span>{{selected3}}</span>
</div>
<!-- 下拉框 获取 当前'0' value值-->
<div>
    <select v-model="selected4">
        <option value='0'>苹果</option>
        <option value='1'>香焦</option>
        <option value='2'>梨子</option>
    </select>
    <span>{{selected4}}</span>
</div>
```

```
<!-- 修饰符 -->
lay：在改变后才触发（也就是说只有光标离开input输入框的时候值才会改变）
<input v-model.lazy="msg" >

trim：自动过滤用户输入的首尾空格
<input type="text" v-model.trim='msg'/> {{msg}}

number：将输出字符串转为Number类型
<input v-model.number='num' type="number"/>  {{num}}
```

# 常用小实例扩展

1）tab切换

2）商城数据动态数据处理

3）二级导航目录

4）邀请好友

5）VUE实现全选

6）监听购物车数量变化（模糊查询）

7）表格的增删改查

# 脚手架的搭建

随着vue.js越来越火爆，更多的项目都用到vue进行开发，在实际的开发项目中如何搭建开发脚手架呢，今天跟大家分享一下：

首先需要了解的知识

Html

Css

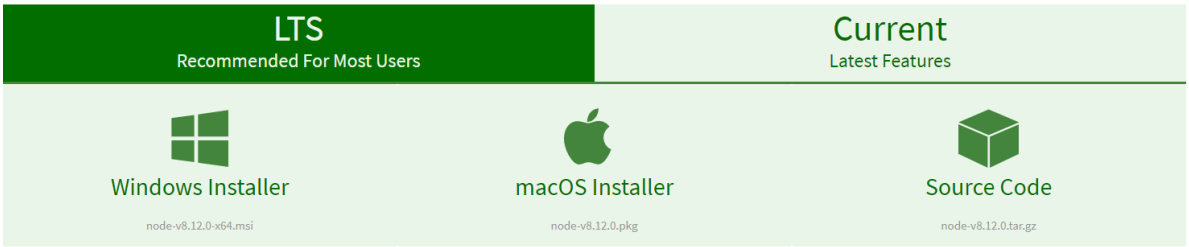
Javascript

Node.js 环境（npm包管理工具）

Webpack 自动化构建工具

## node安装

进入官网下载node.js




| | | | |
|---|---|---|---|
| Windows Installer (.msi) | 32-bit | 64-bit | |
| Windows Binary (.zip) | 32-bit | 64-bit | |
| macOS Installer (.pkg) | 64-bit | | |
| macOS Binary (.tar.gz) | 64-bit | | |
| Linux Binaries (x86/x64) | 32-bit | 64-bit | |
| Linux Binaries (ARM) | ARMv6 | ARMv7 | ARMv8 |
| Source Code | node-v8.12.0.tar.gz | | |

node版本必须在10.0.0以上才支持

## 安装cnpm

1、说明：npm（node package manager）是nodejs的包管理器，用于node插件管理（包括安装、卸载、管理依赖等）；

2、使用npm安装插件：命令提示符执行npm install

3、选装 cnpm 因为npm安装插件是从国外服务器下载，受网络影响大，可能出现异常，如果npm的服务器在中国就好了，所以我们乐于分享的淘宝团队干了这事！来自官网：“这是一个完整 npmjs.org 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟 一次以保证尽量与官方服务同步。”

安装：npm install -g cnpm --registry=https://registry.npm.taobao.org

输入cnpm -v，可以查看当前cnpm版本

PS:

yarn是个包管理器，是facebook发布的一款取代npm的包管理工具

//npm安装yarn

npm install -g yarn

## 安装vue-cli脚手架构建工具

vue-cli 提供一个官方命令行工具，可用于快速搭建大型单页应用。

1、cnpm install -g @vue/cli

```
FF@DESKTOP-C9RNFVR MINGW64 ~/Desktop
$ cnpm install -g @vue/cli
Downloading @vue/cli to C:\Users\FF\AppData\Roaming\npm\node_modules\@vue\cli_tm
p
Copying C:\Users\FF\AppData\Roaming\npm\node_modules\@vue\cli_tmp\_@vue_cli@4.1.
1@@vue\cli to C:\Users\FF\AppData\Roaming\npm\node_modules\@vue\cli
Installing @vue/cli's dependencies to C:\Users\FF\AppData\Roaming\npm\node_modul
es\@vue\cli\node_modules
[1/36] deepmerge@^3.2.0 installed at node_modules\_deepmerge@3.3.0@deepmerge
[2/36] commander@^2.20.0 installed at node_modules\_commander@2.20.3@commander
[3/36] @vue/cli-ui-addon-widgets@^4.1.1 installed at node_modules\_@vue_cli-ui-a
ddon-widgets@4.1.1@@vue\cli-ui-addon-widgets
[4/36] @vue/cli-ui-addon-webpack@^4.1.1 installed at node_modules\_@vue_cli-ui-a
ddon-webpack@4.1.1@@vue\cli-ui-addon-webpack
```

## 查看版本

vue -V

```
λ vue -V
@vue/cli 4.5.6
```

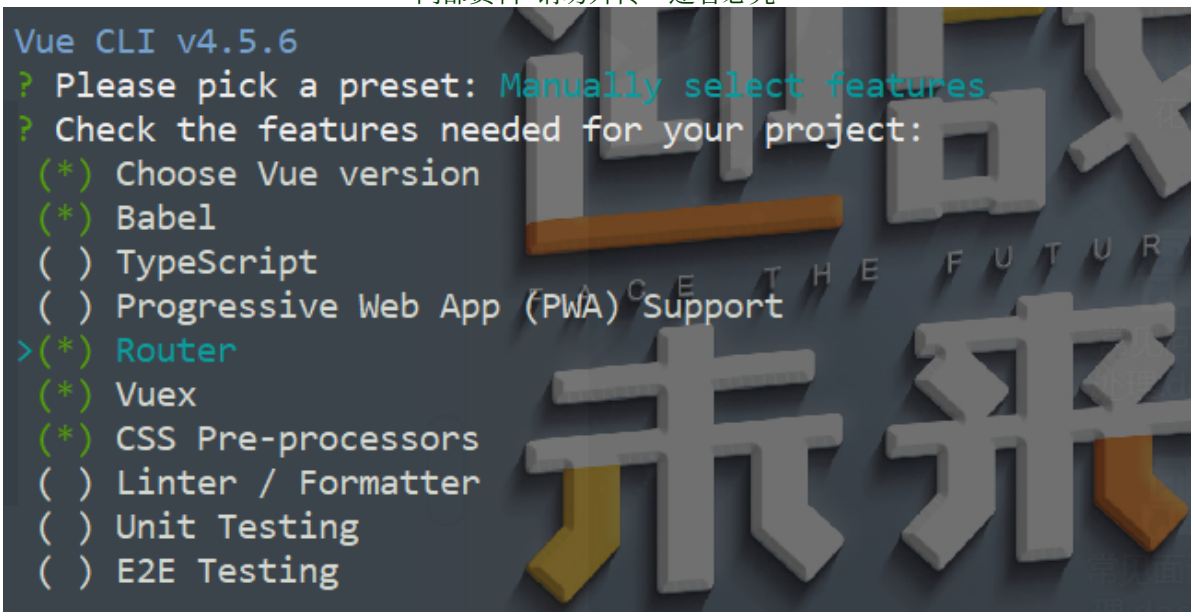
## 创建项目

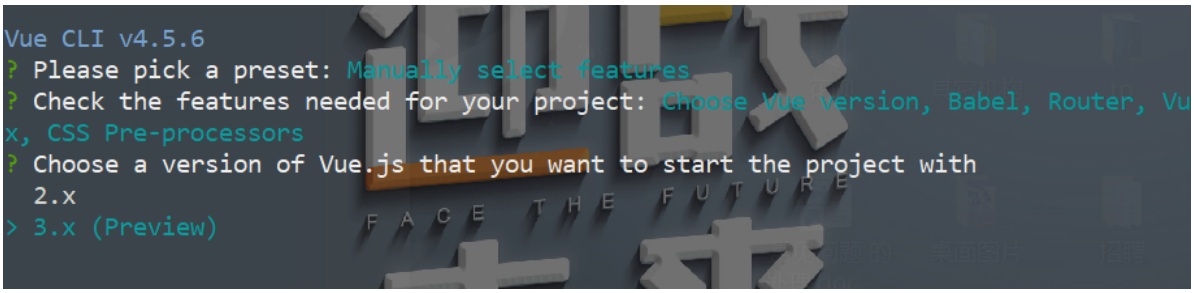
vue create 项目名称

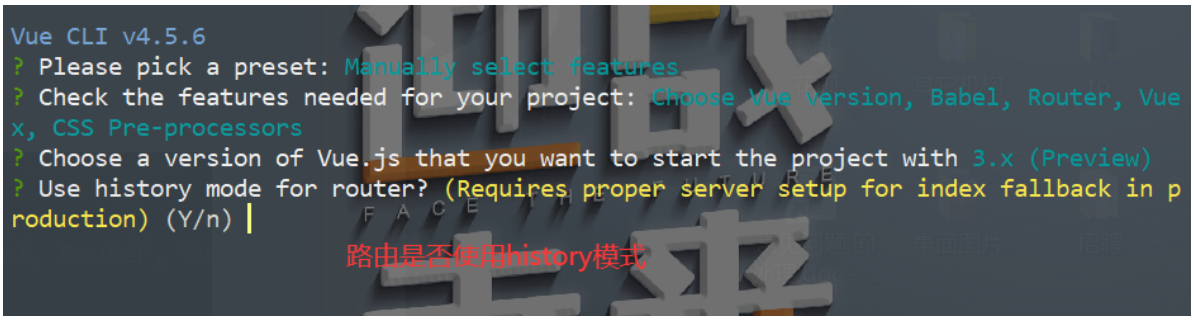
```
λ vue create vue3-project
```

手动选择vue2.0或vue3.0

![(.\img\vue-4.png)


Vue CLI v4.5.6
? Please pick a preset: Manually select features
? Check the features needed for your project:
(*) Choose Vue version
(*) Babel
( ) TypeScript
( ) Progressive Web App (PWA) Support
>(*) Router
(*) Vuex
(*) CSS Pre-processors
( ) Linter / Formatter
( ) Unit Testing
( ) E2E Testing


Vue CLI v4.5.6
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vu
x, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with
2.x
> 3.x (Preview)

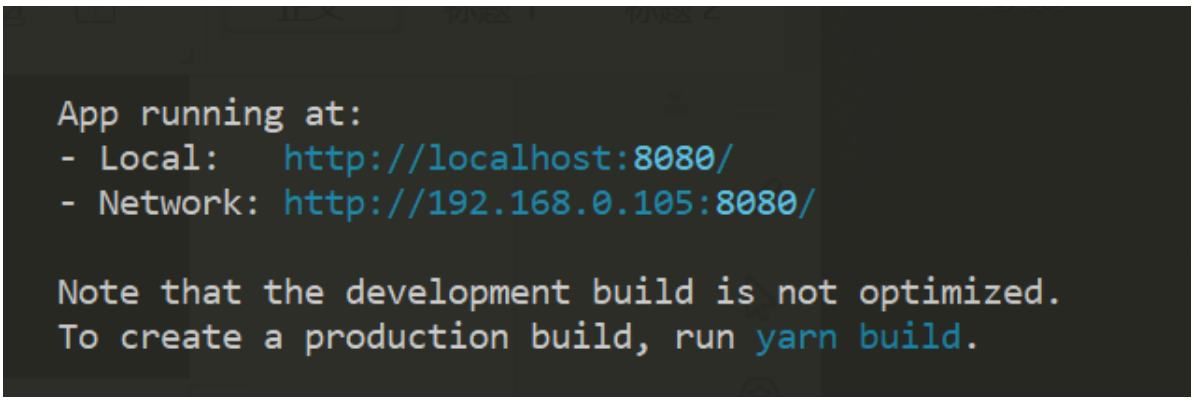

Vue CLI v4.5.6
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vue
x, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use history mode for router? (Requires proper server setup for index fallback in p
roduction) (Y/n)
路由是否使用history模式


Vue CLI v4.5.6
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, Router, Vue
x, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use history mode for router? (Requires proper server setup for index fallback in p
roduction) No
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by d
efault): Sass/SCSS (with node-sass)
? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config fi
les
? Save this as a preset for future projects? (y/N)
将此保存为将来项目的预设

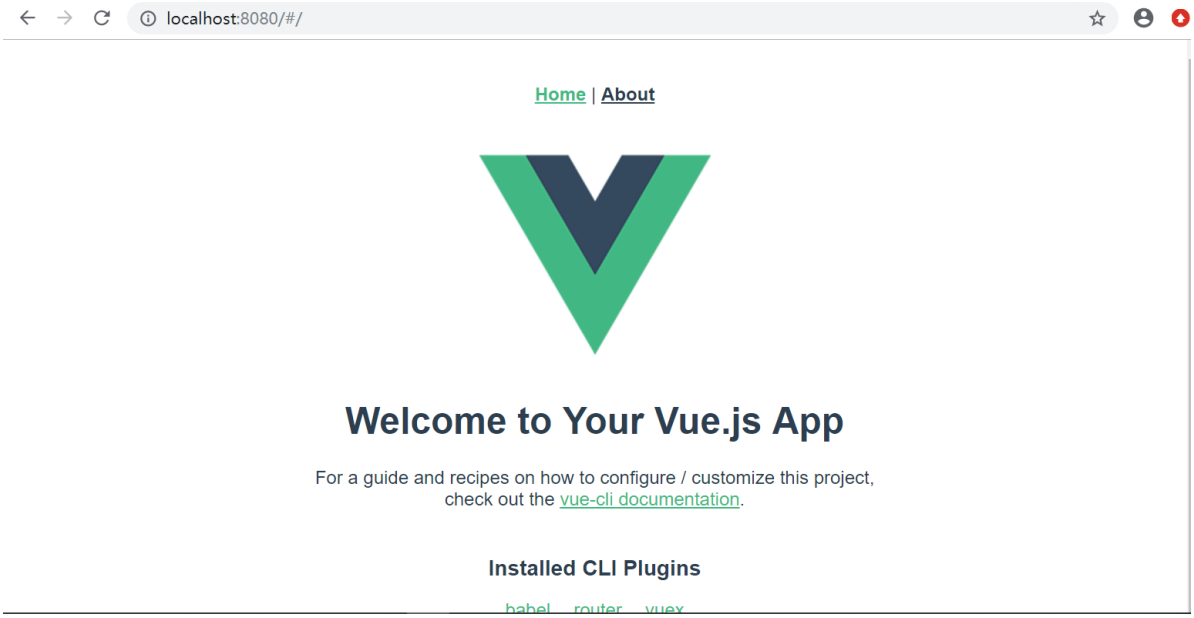

info fsevents@1.2.13: The platform "win32" is incompatible with this module.
info "fsevents@1.2.13" is an optional dependency and failed compatibility check. Exc
luding it from installation.
[3/4] Linking dependencies...
[4/4] Building fresh packages...
success Saved lockfile.
Done in 5.51s.
Running completion hooks...
Generating README.md...
Successfully created project vue3-demo.
Get started with the following commands:
$ cd vue3-demo
$ yarn serve
WARN Skipped git commit due to missing username and email in git config, or failed
to sign commit.
You will need to perform the initial commit yourself.

**项目的启动**


E:\XF\public\vue3
λ cd vue3-demo
E:\XF\public\vue3\vue3-demo (master -> origin)
λ npm run serve


App running at:
- Local: http://localhost:8080/
- Network: http://192.168.0.105:8080/
Note that the development build is not optimized.
To create a production build, run yarn build.

**项目访问**




# 模块化

## 定义

模块通常是指编程语言所提供的代码组织机制，利用此机制可将程序拆解为独立且通用的代码单元。所谓模块化主要是解决代码分割、作用域隔离、模块之间的依赖管理以及发布到生产环境时的自动化打包与处理等多个方面。

简单的说：模块化就是将变量和函数 放入不同的文件中；

## 模块的优点

1.可维护性。 因为模块是独立的，一个设计良好的模块会让外面的代码对自己的依赖越少越好，这样自己就可以独立去更新和改进。

2.命名空间。 在 JavaScript 里面，如果一个变量在最顶级的函数之外声明，它就直接变成全局可用。因此，常常不小心出现命名冲突的情况。使用模块化开发来封装变量，可以避免污染全局环境。

3.重用代码。 我们有时候会喜欢从之前写过的项目中拷贝代码到新的项目，这没有问题，但是更好的方法是，通过模块引用的方式，来避免重复的代码库。

## CommonJS AMD/CMD

CommonJS是一种规范，其内容有很多种，NodeJS是这种规范的实现。SeaJS是模块加载器，是用CMD规范
AMD/CMD是从 CommonJS 讨论中诞生的，
RequireJS 遵循 AMD（异步模块定义）规范，
Sea.js 遵循 CMD （通用模块定义）规范；
规范的不同，导致了两者 API 不同。

AMD 提前执行：提前异步并行加载
优点：尽早执行依赖可以尽早发现错误；
缺点：容易产生浪费

CMD 延迟执行：延迟按需加载
优点：减少资源浪费
缺点：等待时间长、出错时间延后

扩展：

RequireJS 是一个前端的模块化管理的工具库，遵循AMD规范,通过一个函数来将所有所需要的或者说所依赖的模块实现装载进来，然后返回一个新的函数（模块），我们所有的关于新模块的业务代码都在这个函数内部操作，其内部也可无限制的使用已经加载进来的以来的模块。

main.js

```
require.config({
    //baseUrl: "../lib",
    shim: {
        'bootstrap': {
            deps: ['jquery'],
            exports: 'bootstrap'
        }
    },
    paths: {
        /*"jquery": [
            'http://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min',
            '../lib/jquery/dist/jquery.min'
        ],*/
        "jquery": "../lib/jquery/dist/jquery.min",
        "bootstrap": "../lib/bootstrap/dist/js/bootstrap.min",
        "index":"index"
    }
});
require(['jquery','bootstrap','index'], function ($,bootstrap,index){
   // $(".p1").text("hello world");
    console.log(index.web.add(11));
    $(".p1").text(index.web.add(11));

});
```

CMD

```
    //CMD    依赖就近   要用到哪个加载
define(function (require, exports, module){    //定义模块
    var demo = require('demo');
    var y = demo.demo();
    var web = {
        add:function(x) {
            return x +y;
        }
    };
    return {
        web:web
    };

});
```

## ES6模块(module)

模块Module
一个模块，就是一个对其他模块暴露自己的属性或者方法的文件。
导出Export
作为一个模块，它可以选择性地给其他模块暴露（提供）自己的属性和方法，供其他模块使用。

### 导出export

```
var name = 'fang';
var phone = '18040505058;
var hobby = '购物';
export {name, phone, hobby};

导入
import {name, phone, hobby} from './views/moduleA'
```

### 导出export 别名

```
var name = 'fangfang';
var age = 18;
var hobby = '购物';
export { name as v1, age as v2, hobby as v3};

导入
import {v1, v2, v3} from './views/moduleA';
```

### 导出export default 仅有一个

```
export default {name:'abc',hobby:'games'}

导入
import obj from './views/moduleD'
```

### 定义公共数据

```
//性别
export const sexData = [
    { key: 'man', value: '男' },
    { key: 'woman', value: '女' }
  ];
  //用户
export const userData = [
    { id: '1', value: 'name1' },
    { id: '2', value: 'name2' }
  ];

导入
import {sexData,userData} from './views/moduleB'
```

**图片的导入**

1、页面直接引入

```
<img  src='@/assets/img/1.jpg' />
```

2、通过import的方式

```
import img1 from '@/assets/img/1.jpg'  //导入图片
<img  :src="img1" /></div>
```

3、通过require的方式

```
data () {
    return {
      img:[require("@/assets/img/1.jpg"),
           require("@/assets/img/2.jpg")]

    }
  },
```

## common.js和ES6模块区别

node使用的是commonjs 在使用模块的时候是运行时同步加载的 拷贝模块中的对象

模块可以多次加载，但只会在第一次加载 之后会被缓存 引入的是缓存中的值

1.commonjs输出的，是一个值的拷贝，而es6输出的是值的引用；

2.commonjs是运行时加载，es6是编译时输出接口；

# 组件

定义：组件是vue里面最强的功能，可以扩展html，封装重用的代码

vue中的核心之一就是组件，所以有页面都是通过组件来管理，当组件过多时，如何进行参数的传递

## 父组件传参到子组件

为什么组件需要传入参数呢?

定义按钮组件

```
<button>修改</button>
```

不同的页面中有多个按钮，有的名称是'新增'，'提交'，'删除'，'退出'，'修改'等，也有可能会有不同的样式，以及不同的函数...能不能写一个组件，传入不同来实现

1、传入不同的名称

```
index.vue
<Btn :title="'修改'" />


Btn.vue
<button>{{title}}</button>
1、对象的方式(1)  直接定义
```

```
 props: {
    title: String
 }
 对象的方式(2)  添加类型和默认值
 props: {
    title: {
        type: String,
        default:'按钮'
    }
 }
2、数组的方式
props:['title']
```

2、传入不同的样式

```
index.vue
<Btn :title="'修改'" :color="'red'" />


Btn.vue
<button :style="{color:color}">{{title}}</button>
1、对象的方式
 props: {
    title: String,
    color: String
 }
2、数组的方式
props:['title','color']
```

3、传入一个对象

```
index.vue
<Btn :obj = "obj" />
data(){
    return{
      obj:{title:'查看详情',color:'green',padding:'10px'}
    }
}


Btn.vue
<button :style="{color:obj.color,padding:obj.padding}">{{obj.title}}</button>
接收父的参数:
1、对象的方式
 props: {
    obj:{
      title: String,
      color: String,
      padding: String
    }
 }
2、数组的方式
props:['obj']
```

## 父组件操作子组件

ref被用来给DOM元素或子组件注册引用信息，引用信息会根据父组件的$refs对象进行注册。

如果在普通DOM元素上使用，引用信息是元素，如果在子组件上，引用信息就是组件实例

简单理解：

1. 如果ref用在子组件上，指向的是组件实例，可以理解为对子组件的索引，通过$ref可能获取到在子组件里定义的属性和方法。
2. 如果ref在普通的 DOM 元素上使用，引用指向的就是 DOM 元素，通过$ref可能获取到该DOM 的属性集合，轻松访问到DOM元素，作用与JQ选择器类似

```
index.vue
<!-- 父操作子-->
<Btn :obj = "obj"  ref='btn'/>
<button @click="change()">改变</button>

data(){
    return{
      obj:{title:'查看详情',color:'green',padding:'10px'}
    }
}
methods:{
    change(){
      //操作子组件的属性和方法
      console.log(this.$refs.btn.sub);
      console.log(this.$refs.btn.action());
    }
}


Btn.vue
<button :style="{color:obj.color}">{{obj.title}}</button>
接收父的参数:
props:['obj']
data(){
    return{
      sub:'我是子组件按钮',
    }
},
methods:{
    action(){
       console.log('我是子组件按钮')
    }
}
```

## 子组件操作父组件

$emit 实现子组件向父组件通信

```
vm.$emit( event, arg )
```

$emit 绑定一个自定义事件event，当这个这个语句被执行到的时候，就会将参数arg传递给父组件，父组件通过@event监听并接收参数。

1、子组件传参父组件

```
index.vue
<!-- @e-child 监听并接收参数-->
<Btn :obj = "obj"  @e-child="acceptSon" />

data(){
    return{
      obj:{title:'查看详情',color:'green',padding:'10px'}
    }
}
methods:{
    acceptSon(value){
        //value就是子组件传过来的值
    }
}


Btn.vue
<button :style="{color:obj.color}" @click="send()">{{obj.title}}</button>
接收父的参数:
props:['obj']

methods:{
    send(){
       this.$emit('e-child','子要传给父的值')     //发射一个事件 把子的值传给父
    }
}
```

2、子组件操作父组件

方法一:

```
index.vue
<!-- @e-child 监听并接收参数-->
<Btn :obj = "obj" :event="action" @e-child="acceptSon"/>

data(){
    return{
      obj:{title:'查看详情',color:'green',padding:'10px'}
    }
}
methods:{
    action(){  //父组件的方法
      this.name = '父组件的值';
    },
    acceptSon(res){  //
        //res()就是action函数
        res()
    }
}


Btn.vue
<button :style="{color:obj.color}" @click="send()">{{obj.title}}</button>
接收父的参数:
props:['obj','event']
```

```
methods:{
    send(){
       this.$emit('e-child',this.event)     //子操作父的方法  把父的方法传入到子中处理
    }
}
```

## 兄弟组件传参

场景：testA.vue和testB.vue二个子组件，testA.vue获取到testB.vue的值

```
index.vue
<div>
  <h1>父组件：{{title}}</h1>
  <testA :title="title"/>
  <testB @e-child="acceptSon"/>
</div>
methods:{
    acceptSon(res){  //接受testB子组件信息，同时传给testA
       this.title = res
    }
}

testA.vue
A组件：<button>{{title}}</button>
props:{
    title:String
}

testB.vue
B组件：<input type="text" v-model="title" @input="send()" />
data(){
    return{
      title:'testB',

    }
},
methods:{
    send(){
        this.$emit('e-child',this.title)    //发射一个事件    把子的值传给父
    }
}
```

## 动态模板

当在这些组件之间切换的时候，有时会想保持这些组件的状态，以避免反复重渲染导致的性能问题

### 1、通过条件匹配来显示

```
<button @click="flag='testA'">A组件</button>
<button @click="flag='testB'">B组件</button>
```

```
<testA v-if="flag=='testA'"></testA>
<testB v-if="flag=='testB'"></testB>
```

**2、动态模板**

vue中提供了一个动态模板，可以在任意模板中切换，就是用vue中用:is来挂载不同的组件。

```
<component :is="flag"></component>

import testA from './testA.vue'; //导入
import testB from './testB.vue'; //导入

data(){
    return{
      flag:'testA',
    }
}
```

**3、异步组件**

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，Vue 允许你以一个工厂函数的方式定义你的组件，这个工厂函数会异步解析你的组件定义

```
非异步组件加载，提前加载
import testA from './testA';
import testB from './testB';

components: {
    testA,
    testB
},
```

```
异步组件
components: {
    //不会提前加载，在需要用到的时候才会加载组件
    //异步解析组件  当使用局部注册的时候，你也可以直接提供一个返回 Promise 的函数
    'testB': () => import('./testB')
},
```

## 全局组件定义

在我们项目开发中，经常需要import或者export各种模块，那么有没有什么办法可以简化这种引入或者导出操作呢

```
import A from 'components/A'
import B from 'components/B'
import C from 'components/C'
import D from 'components/D'
......
```

这样很头疼，因为每加一个组件，都要写这么一句，有规律的事，是否可以通过自动化完成呢

定义全局组件的方式：

```
Vue.component('myComponent', {render(){return <h1>hello world</h1>}})
```

webpack中require.context

```
require.context(directory, useSubdirectories, regExp)
```

用法：

```
require.context('./', true, /\.js$/);
```

定义全局自定义组件

```
import Vue from 'vue'
const componentsContext = require.context('./', true, /\.js$/);

componentsContext.keys().forEach(component => {
    const componentConfig = componentsContext(component)
    // 兼容import export和require module.export两种规范
    const ctrl = componentConfig.default || componentConfig;
    // 加载全局组件
    if (ctrl && ctrl.name) {
        Vue.component(ctrl.name, ctrl);
    }
})
```

代码分析：

```
const componentsContext = require.context('./', true, /\.js$/);

//返回的是webpackContext方法
webpackContext(req) {
    var id = webpackContextResolve(req);
    return __webpack_require__(id);
}
```

```
componentsContext.keys()

//当前目录下所有的js文件
0: "./index.js"
1: "./my-banner/index.js"
2: "./my-button/index.js"
3: "./my-button2/index.js"
```

这样其它页面就可以直接引用了

```
<my-button />
```

## 组件缓存 keep-alive

每次切换新标签的时候，Vue 都创建了一个新的实例。

重新创建动态组件的行为通常是非常有用的，如果希望那些标签的组件实例能够被在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 `<keep-alive>` 元素将其动态组件包裹起来。

keep-alive是Vue提供的一个抽象组件，用来对组件进行缓存，从而节省性能，由于是一个抽象组件，

所以在页面渲染完毕后不会被渲染成一个DOM元素

include: 只有匹配的组件才会缓存,符合条件：字符串/正则

exclude: 任何组件都不会缓存，符合条件：字符串/正则

```
<button @click="flag='testA'">a</button>
<button @click="flag='testB'">b</button>
<button @click="flag='testC'">c</button>

import testA from './testA.vue'; //导入
import testB from './testB.vue'; //导入
import testC from './testC.vue'; //导入

data(){
    return{
      flag:'testA'
    }
  },
```

```
只有组件testA才会缓存
<keep-alive include='testA'>
        <component :is="flag"></component>
</keep-alive>
```

```
组件testA，testB才会缓存
<keep-alive include='testA,testB'>
    <component :is="flag"></component>
</keep-alive>
```

```
除了组件testA，testC缓存
<keep-alive exclude='testA,testC'>
    <component :is="flag"></component>
</keep-alive>
```

对于路由切换后想缓存组件的状态，这种如何处理咧?

```
<keep-alive>
    <!-- 这里是会被缓存的视图组件 -->
    <router-view v-if="$route.meta.keepAlive" />
</keep-alive>


路由中的定义
const routes = [
  {
    path: '/',
    redirect:'/about'
  },
  {
    path: '/about',
    name: 'About',
    component: () => import('../views/About.vue')
  },
  {
    path: '/tab',
    name: 'tab',
    meta: {
```

```
      keepAlive: true // 需要被缓存
    },
    component: () => import('../views/Tab.vue')
  },
```


当进入到TAB切换时会缓存当前是TAB选项的状态，再次进入还是'标题4'

# 路由

## 定义

Vue路由是指根据url分配到对应的处理程序；作用就是解析URL，调用对应的控制器（的方法，并传递参数）。Vue路由有助于在浏览器的URL或历史记录与Vue组件之间建立链接，从而允许某些路径渲染与之关联的任何一个视图；

简而言之：所谓"路由"，是指把数据从一个地方传送到另一个地方的行为和动作

## 用法

页面定义：

```
<div>
    <!-- 使用 router-link 组件来导航. -->
    <!-- 通过传入 `to` 属性指定链接. -->
    <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
    <router-link to="/home">home</router-link>
    <router-link to="/news">news</router-link>
</div>
<!-- 路由出口 -->
<!-- 路由匹配到的组件将渲染在这里 -->
<router-view></router-view>
```

1、定义（路由）组件

```
var Home = {
    template: '#home'
}
var News = {
    template: '#news'
}

对应的模块内容：
<template id="home">
    <div>
        <h3>组件home</h3>
    </div>
</template>
```

```
<template id="news">
    <div>
        <h3>组件news</h3>
    </div>
</template>
```

2、定义路由

```
const routes = [
    {path: '/home', component: Home},
    {path: '/news', component: News},
    {path: '/', redirect: '/home'}
];
```

路由命名

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 `routes` 配置中给某个路由设置名称。

```
routes: [
    {
      path: '/user/:id',
      name: 'user',
      component: User
    }
  ]
```

3、创建 router 实例

```
const router = new VueRouter({
    routes,// (缩写) 相当于 routes: routes
    linkActiveClass: 'active'
});
```

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
window.onload=function(){
    new Vue({
        el:'#my',
        router
    });
}
```

## 设置路由导航的两种方法

声明式

```
 <router-link :to="/home">
```

编程式

```
router.push('/home')
```

**声明式的常见方式**

```
<router-link to="/home">home</router-link>

对象
<router-link :to="{path:'/home'}">home</router-link>

路由通过名称
<router-link :to="{name: 'homename'}">home</router-link>

直接路由带查询参数query，地址栏变成 /home?id=10
<router-link :to="{path: '/home', query: {id: 10 }}">home</router-link>

命名路由带查询参数query，地址栏变成/home?id=10
<router-link :to="{name: 'homename', query: {id: 10 }}">home</router-link>
```

**编程式的常见方式**

```
字符串
router.push('/home')

对象
router.push({path:'/home'})

路由通过名称
router.push({name: 'homename'})

直接路由带查询参数query，地址栏变成 /home?id=10
router.push({path: 'home', query: {id: 10 }})

命名路由带查询参数query，地址栏变成/home?id=10
router.push({name: 'homename', query: {id: 10 }})
```

## 路由传参

1、http://localhost:8080/user/10

传入参数的方式：

```
<router-link :to="'/user/'+id">user</router-link>
```

路由配置：

```
const routes = [
    {path: '/home', component: Home},
    {path: '/news', component: News},
    {path: '/user/:id', component: User},
    //路由中定义http://localhost:8080/#/user/10  需要定义ID
];
```

2、http://localhost:8080/home?id=10

传入参数的方式：

```
<router-link :to="{path:'/home',query: {id: id}}">test</router-link>
```

路由中定义：user?id=10 不需要在路由配置中定义参数

## 常见路由对象

在使用了 vue-router 的应用中，路由对象会被注入每个组件中，赋值为 this.$route ，并且当路由切换时，路由对象会被更新，路由对象暴露了以下属性

1.$route.path
字符串，等于当前路由对象的路径，会被解析为绝对路径，如 "/home/news" 。

2.$route.params
对象，包含路由中的动态片段和全匹配片段的键值对。3.$route.query
对象，包含路由中查询参数的键值对。例如，对于 /home/news/detail/01?favorite=yes ，会得到 $route.query.favorite == 'yes' 。
4.$route.router
路由规则所属的路由器（以及其所属的组件）。

5.$route.matched
数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。

6.$route.name
当前路径的名字，如果没有使用具名路径，则名字为空。

7.$route.meta

路由元信息

8.$route.fullPath

完成解析后的 URL，包含查询参数和hash的完整路径。

## 路由中的props属性

有时需要在路由中定义一些值来供对应的组件使用，props除了父子传参外，路由可以用来传递参数吗？答案是肯定的

路由中的定义方式

```
const routes = [
    {path: '/home', component: Home},
    {path: '/news', component: News},
    {
    path: '/part1',
    name: 'part1',
    props: {   //可以自定义路由参数
      data:{
        a: 'a',
        b: 'b'
      }
    },
    component: () => import('../views/part/part1.vue')
```

```
  },
];
```

组件的获取方式：

```
props:['data'],  //获取到路由中的props

mounted(){
    this.title = this.data.a;
},

{{title}}
```

## 二级路由

### 路由配置

```
const routes = [
    {path: '/home', component: Home},
    {
        path: '/news',
        component: News,
        children:[{      //二级路由
            path:'login',     ///news/login
            component:Login
            },{
            path:'regist/:name/:pwd',   ///news/regist/abc/123
            component:Regist
        }]
    },
    {path: '/', redirect: '/home'}   //重定向
];
```

### 二级路由渲染

```
<div id="my">
        <router-link to="/home">Home</router-link>
        <router-link to="/news">News</router-link>
        <div>
            <!-- 一级路由出口 -->
            <router-view></router-view>
        </div>
</div>
```

定义组件

```
var Home = {
    template: '#home'
}
var News = {
    template: '#news'
}
```

```
var Login={
    template: '<h3>Login--获取参数:{{$route.query.name}}</h3>'
}
var Regist={
    template: '<h3>Regist--参数:{{$route.params.name}}</h3>'
}

对应的模块内容:
<template id="home">
    <div>
        <h3>组件home</h3>
    </div>
</template>

<template id="news">
    <div>
        <h3>组件news</h3>
        <ul>
            <li><router-link to="/news/login">用户登录</router-link></li>
            <li><router-link :to="'/news/regist/'+name+'/'+id">用户注册</router-
link></li>
        </ul>
        <!-- 二级路由出口 -->
        <router-view></router-view>
    </div>
</template>
```

## 路由拦截

定义：路由拦截就是路由在发生变化时需要进行的拦截处理，比如跳转到某个页面要判断是否有登录等;

写法:

```
路由拦截    /*在跳转之前执行*/
beforeEach函数有三个参数:
 to:router即将进入的路由对象
 from:当前导航即将离开的路由
 next:Function,进行管道中的一个钩子，如果执行完了，则导航的状态就是 confirmed （确认的）；否
则为false，终止导航。
afterEach函数不用传next()函数
router.beforeEach(function(to, from, next) {
  next()
})
```

主要是对进入页面的限制；比如判断有没有登录，没有就不能进入某些页面，只有登录了之后才有权限查看页面

```
router.beforeEach(function(to, from, next) {
  if (!localStorage.getItem("username")) {
      if (to.path !== '/login') {
          next('/login')
      }
  };
  next()
})
```

```
/*在跳转之后判断*/
会在每次路由切换成功进入激活阶段时被调用。
Vue.afterEach(function(to,form){
    console.log('成功浏览到: ' + to.path)
})
```

## slot插槽

插槽，也就是槽，是组件的一块HTML模板，这块模板显示不显示，以及怎样显示由父组件来决定。

插槽分为：匿名插槽| 具名插槽| 作用域插槽

### 匿名插槽

它不用设置名称属性，可以放置在组件的任意位置；可以理解为父传入样式及内容，子负责展示

```
index.vue  父组件
<slot2 :data="" color="" >
      <!--匿名插槽-->
      <ul>
          <li>1</li>
          <li>2</li>
      </ul>
      <ul slot>
          <li>1</li>
          <li>2</li>
      </ul>
</slot2>

slot2.vue   子组件
<!--匿名插槽-->
<slot></slot>
```

### 具名插槽

插槽加了名称属性，就变成了具名插槽。具名插槽可以在一个组件中出现N次，出现在不同的位置

```
index.vue  父组件
<slot2>
    <div slot="n1">
        <h2>具名插槽n1</h2>
    </div>
    <!--具名插槽 传入父的值-->
    <div slot="n2">
        <h2>具名插槽n2</h2>
        <ul>
            <li v-for="(v,i) in arr"  :key="i">{{v}}</li>
        </ul>
    </div>
</slot2>

slot2.vue   子组件
<!--具名插槽-->
<slot name="n1"></slot>
<slot name="n2"></slot>
```

### 作用域插槽

官方叫它作用域插槽，实际上，对比前面两种插槽，我们可以叫它带数据的插槽。

什么意思呢，就是前面两种，都是在组件的模板里面写

作用域插槽跟单个插槽和具名插槽的区别，因为单个插槽和具名插槽不绑定数据，所以父组件提供的模板一般要既包括样式又包括内容， 而作用域插槽，父组件只需要提供一套样式（在确实用作用域插槽绑定的数据的前提下）相当于父组件提供一套样式，数据都是子组件的。

```
index.vue  父组件
<slot2>
    <div slot-scope="scope">
        {{scope.title}}
    </div>
</slot2>

slot2.vue   子组件
<!-- 作用域插槽 -->
<slot :title="title"></slot>

子组件中传入数组
index.vue  父组件
<slot2>
    <div slot-scope="scope">
        <ul>
            <li v-for="(v,i) in scope.arr" :key="i">{{v}}</li>
        </ul>
    </div>
</slot2>
```

## axios

页面引入及调用:

```
axios({
        method: 'get',
        url: 'http://localhost:3000/map/get'
    }).then(response=>{
        console.log('请求成功:' + response);
    }).catch(error => {
        console.log('请求失败:' + error);
    });
```

get请求,传入参数 http://localhost:3333/get_table/?id=1&name=jindu

```
axios.get('http://localhost:3333/get_table/', {
 params:{
    name:'jindu',
    id:1
 }
params:this.user
})
.then(resp => {
    console.log(resp);
}).catch(err => {
    console.log(err);
})
```

post请求

```
axios({
    method:'post',
    url:'http://localhost:3000/map/add1',
    data: {}
}).then(function(response){
    console.log(response)
}).catch(function(error){
    console.log(error);
})
```

简写:

```
axios.post('http://localhost:3000/map/add1',{})
.then(function(response){
    console.log(response)
}).catch(function(error){
    console.log(error);
})
```

## axios全局引入

```
main.js
import axios from 'axios'

Vue.prototype.$http= axios;
axios.defaults.baseURL = 'http://127.0.0.1:3333/'
```

组件中请求的方式为

```
this.$http({
    method: 'get',
    url: 'map/get'
}).then(response=>{
    console.log('请求成功:' + response);
}).catch(error => {
    console.log('请求失败:' + error);
});
```

### axios请求拦截封装

全局处理请求、响应拦截的处理，常见处理请求动画，错误码等

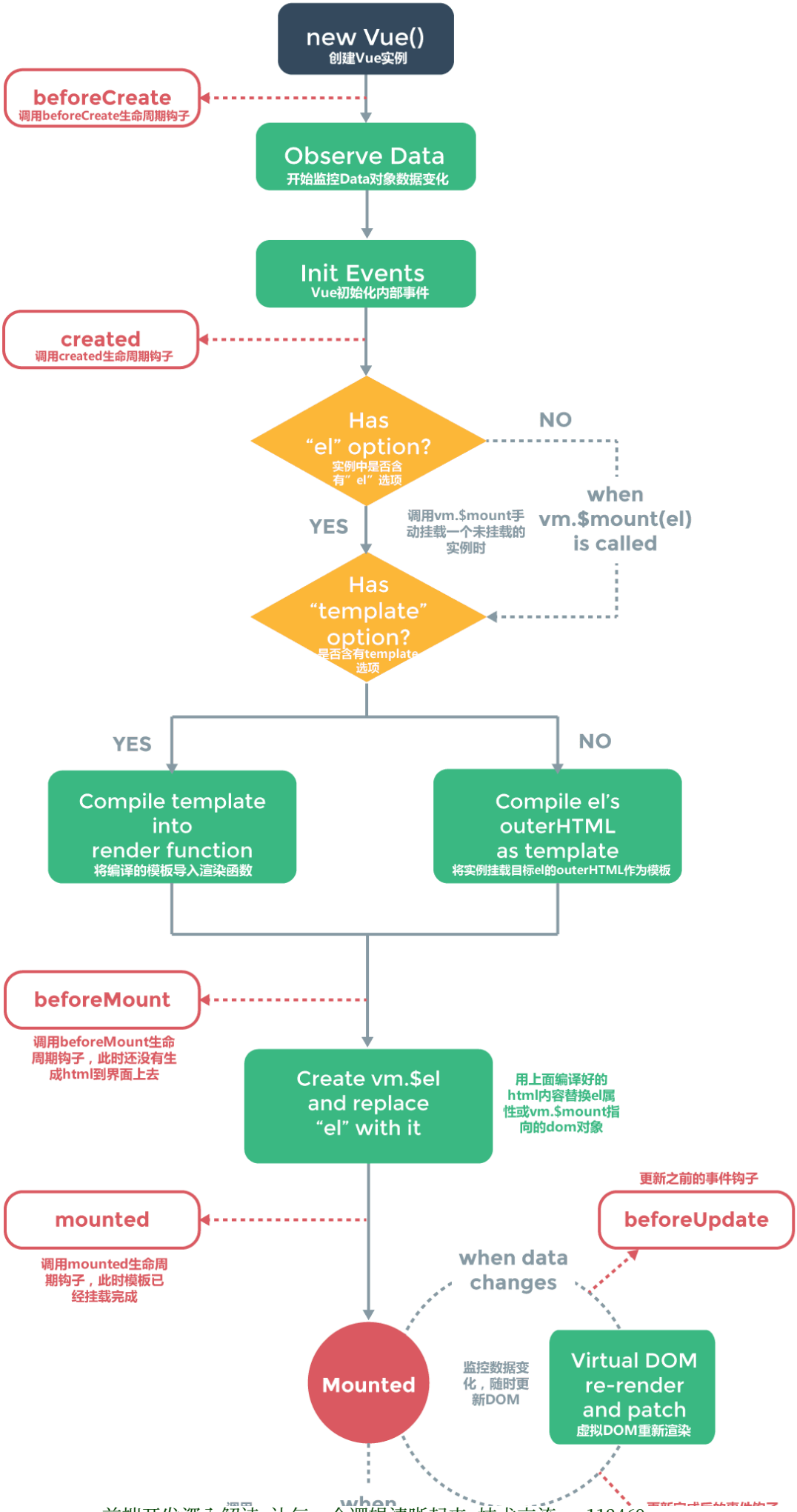
```
import axios from 'axios'

axios.defaults.baseURL = `http://127.0.0.1:3333`;
// 添加请求拦截器
// 在发送请求之前做些什么
axios.interceptors.request.use((config)=>{
    return config;
})
// 添加响应拦截器
axios.interceptors.response.use((response)=>{
    // 对响应数据做点什么
    return response
}, err=>{
    // 对响应错误做点什么
    return Promise.reject(err);
})

export default axios
```

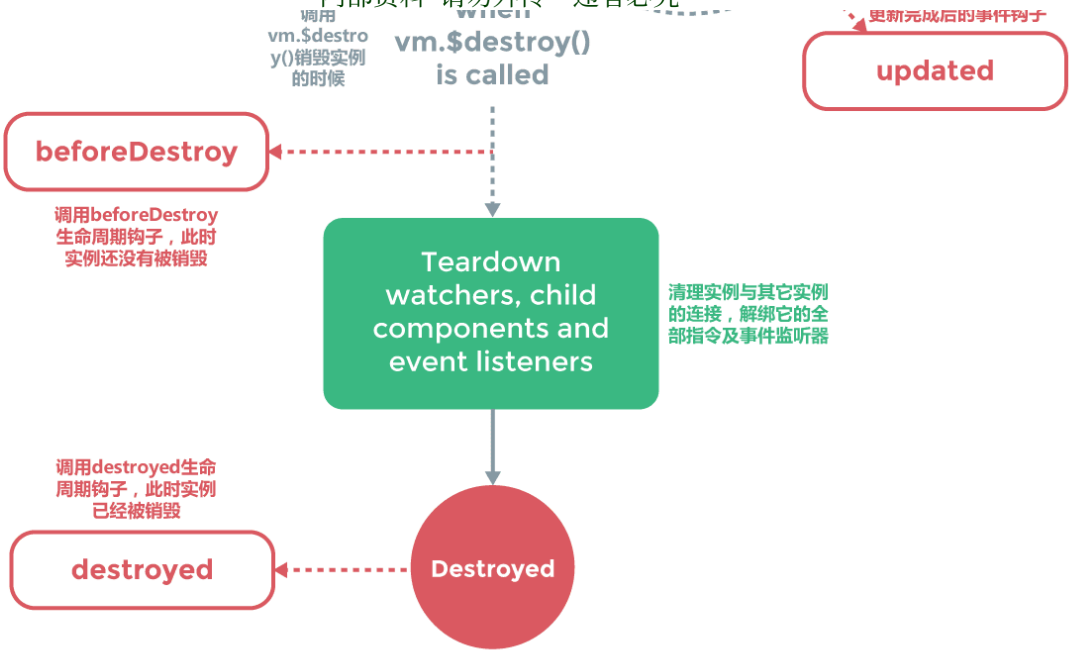
页面调用：

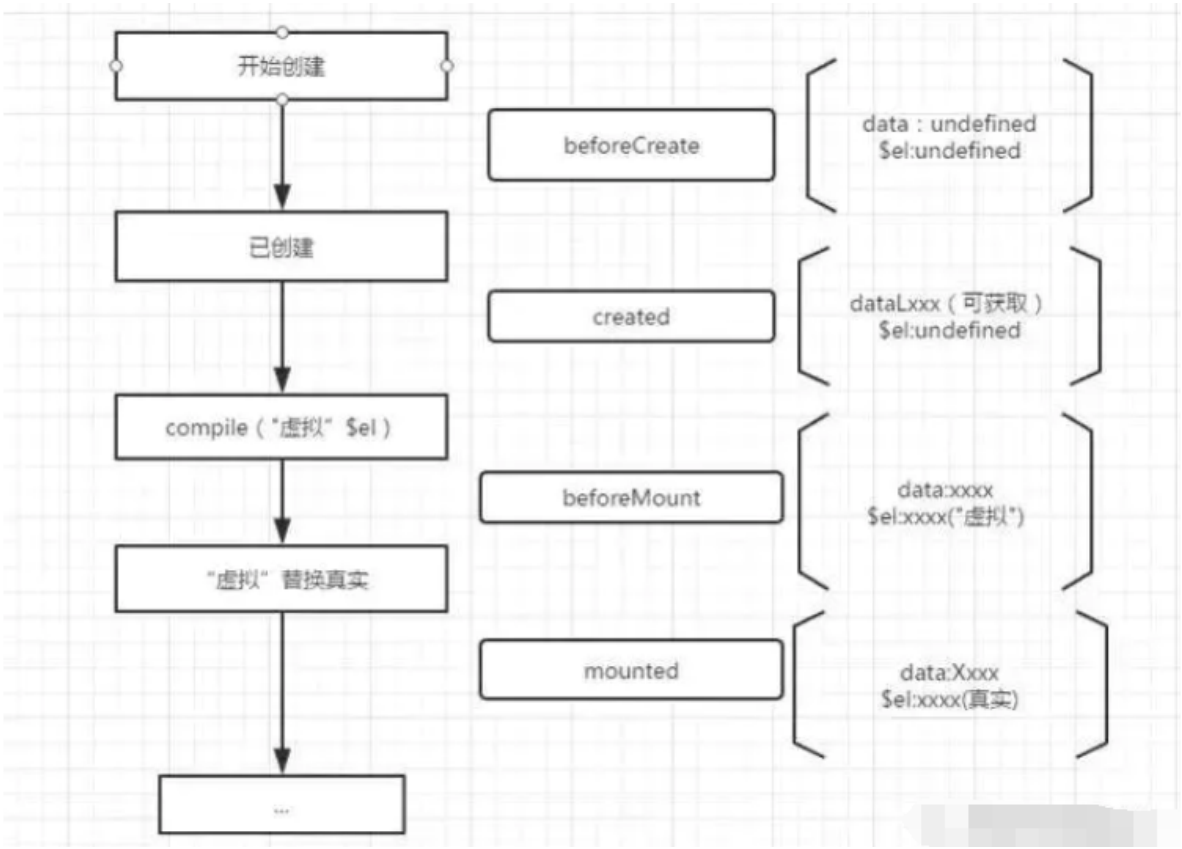
```
import axios from '@/api/index'  //引入方式

axios({
    method:'post',
    url:'/map/add1',
    data: {}
}).then(function(response){
    console.log(response)
}).catch(function(error){
    console.log(error);
})
```

## 生命周期

Vue实例有一个完整的生命周期，也就是说从开始创建、初始化数据、编译模板、挂在DOM、渲染-更新-渲染、卸载等一系列过程，我们成为Vue 实例的生命周期，钩子就是在某个阶段给你一个做某些处理的机会。






beforeCreate( 创建前 )

在实例初始化之后，数据观测和事件配置之前被调用，此时组件的选项对象还未创建，el 和 data 并未初始化，因此无法访问methods， data， computed等上的方法和数据。

created ( 创建后 )

实例已经创建完成之后被调用，在这一步，实例已完成以下配置：数据观测、属性和方法的运算，watch/event事件回调，完成了data 数据的初始化，el没有。 然而，挂在阶段还没有开始, $el属性目前不可见，这是一个常用的生命周期，因为你可以调用methods中的方法，改变data中的数据，并且修改可以通过vue的响应式绑定体现在页面上，获取computed中的计算属性等等，

beforeMount

挂在开始之前被调用，相关的render函数首次被调用（虚拟DOM），实例已完成以下的配置： 编译模板，把data里面的数据和模板生成html，完成了el和data 初始化，注意此时还没有挂在html到页面上。

mounted

挂在完成，也就是模板中的HTML渲染到HTML页面中，此时一般可以做一些ajax操作，mounted只会执行一次。

beforeUpdate

在数据更新之前被调用，发生在虚拟DOM重新渲染和打补丁之前，可以在该钩子中进一步地更改状态，不会触发附加地重渲染过程

updated（更新后）

在由于数据更改导致地虚拟DOM重新渲染和打补丁只会调用，调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作，然后在大多是情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环，该钩子在服务器端渲染期间不被调用

beforeDestroy（销毁前）

在实例销毁之前调用，实例仍然完全可用，

1. 这一步还可以用this来获取实例，
2. 一般在这一步做一些重置的操作，比如清除掉组件中的定时器 和 监听的dom事件

destroyed（销毁后）

在实例销毁之后调用，调用后，所以的事件监听器会被移出，所有的子实例也会被销毁，该钩子在服务器端渲染期间不被调用

# 计算属性

定义：计算属性就是当其依赖属性的值发生变化时，这个属性的值会自动更新，与之相关的DOM部分也会同步自动更新。

```
computed:{ //计算属性
   getN:function(){
        return this.n-1;
   }
}
```

## 计算属性传参数

```
{{total(2)}}

computed:{ //计算属性
    total(){
        return function(i){
            return i+1
        }
}
```

## watch

定义：watch的作用可以监控一个值的变换,并调用因为变化需要执行的方法

```
watch:{
    //几种写法 example为监听的名称
    example(curVal,oldVal){
        console.log(curVal,oldVal);
    },
    'example'(curVal,oldVal){
        console.log(curVal,oldVal);
    },
    example:'a',//a表示为methods的方法名
    example:{
    //注意：当观察的数据为对象或数组时，curVal和oldVal是相等的，因为这两个形参指向的是同一个
数据对象
    handler(curVal,oldVal){
        console.log(curVal,oldVal)
    },
    deep:true  //对象内部的属性监听，也叫深度监听
    }

},
```

## nextTick

将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新。$nextTick它跟全局方法 Vue.nextTick 一样，不同的是回调的 this 自动绑定到调用它的实例上。

常用的场景是在进行获取数据后，需要对新视图进行下一步操作或者其他操作时，发现获取不到dom。因为赋值操作只完成了数据模型的改变并没有完成视图更新

```
<template>
    <div>
      <button  @click="textChange()" ref="btn">{{text}}</button>
    </div>
</template>

<script>
export default {
  data () {
    return {
      text:"原始值",
    }
  },
  methods:{
    textChange:function(){
      this.text="修改值";
      //this.$refs.btn获取指定DOM，输出：原始值
      console.log(this.$refs.btn.innerText);
      //Vue 实现响应式并不是数据发生变化之后 DOM 立即变化，而是按一定的策略进行 DOM 的更新
      this.$nextTick(function(){
        console.log(this.$refs.btn.innerText);  //输出：修改值
      });
```

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

## 什么时候用nextTick

Vue生命周期的**created()钩子函数进行的DOM操作一定要放在Vue.nextTick()的回调函数中**，原因是在created()钩子函数执行的时候DOM 其实并未进行任何渲染，而此时进行DOM操作无异于徒劳，所以此处一定要将DOM操作的js代码放进Vue.nextTick()的回调函数中。与之对应的就是mounted钩子函数，因为该钩子函数执行时所有的DOM挂载已完成。

```
<template>
    <div ref="d">
      <button  @click="textChange()" ref="btn">{{text}}</button>
    </div>
</template>

<script>
export default {
  data () {
    return {
      text:"原始值",
    }
  },
  created(){
   //由于DOM还没有生成，不通过this.$nextTick会报错
    this.$nextTick(function(){
        var span = document.createElement('span'); //1、创建元素
        span.innerHTML='span新元素';
        this.$refs.d.appendChild(span); //2、找到父在末尾添加元素
    });
  },
  methods:{
    textChange:function(){
      this.text="修改值";
      console.log(this.$refs.btn.innerText);   //this.$refs.btn获取指定DOM，输出：原始值
      this.$nextTick(function(){
        console.log(this.$refs.btn.innerText);  //输出：修改值
      });

    }
  }
}
</script>
```

如果是在dom生成后操作，可以直接

```
mounted(){
    var span = document.createElement('span'); //1、创建元素
    span.innerHTML='span新元素';
    this.$refs.d.appendChild(span); //2、找到父在末尾添加元素
},
```

### Vue.nextTick(callback) 使用原理

Vue是异步执行dom更新的，一旦观察到数据变化，Vue就会开启一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。如果这个watcher被触发多次，只会被推送到队列一次。这种缓冲行为可以有效的去掉重复数据造成的不必要的计算和DOm操作。而在下一个事件循环时，Vue会清空队列，并进行必要的DOM更新。
当设置 vm.someData = 'new value'，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新时才会进行必要的DOM更新。如果此时你想要根据更新的 DOM 状态去做某些事情，就会出现问题。。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用。

## VUEX

### vuex是什么?

Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式**。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。

### 什么是“状态管理模式”?

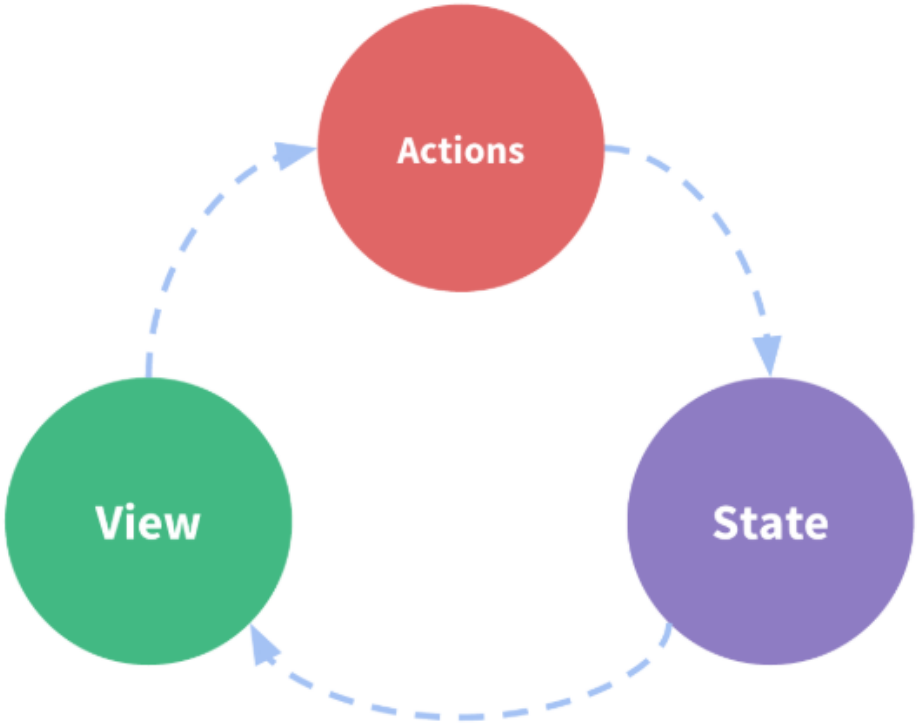
让我们从一个简单的 Vue 计数应用开始：

```
new Vue({
  // state
  data () {
    return {
      count: 0
    }
  },
  // view
  template: `
    <div>{{ count }}</div>
  `,
  // actions
  methods: {
    increment () {
      this.count++
    }
  }
})
```

这个状态自管理应用包含以下几个部分：

- **state**，驱动应用的数据源；
- **view**，以声明方式将 **state** 映射到视图；
- **actions**，响应在 **view** 上的用户输入导致的状态变化。

以下是一个表示“单向数据流”理念的简单示意：




但是，当我们的应用遇到**多个组件共享状态**时，单向数据流的简洁性很容易被破坏：

- 多个视图依赖于同一状态。
- 来自不同视图的行为需要变更同一状态。

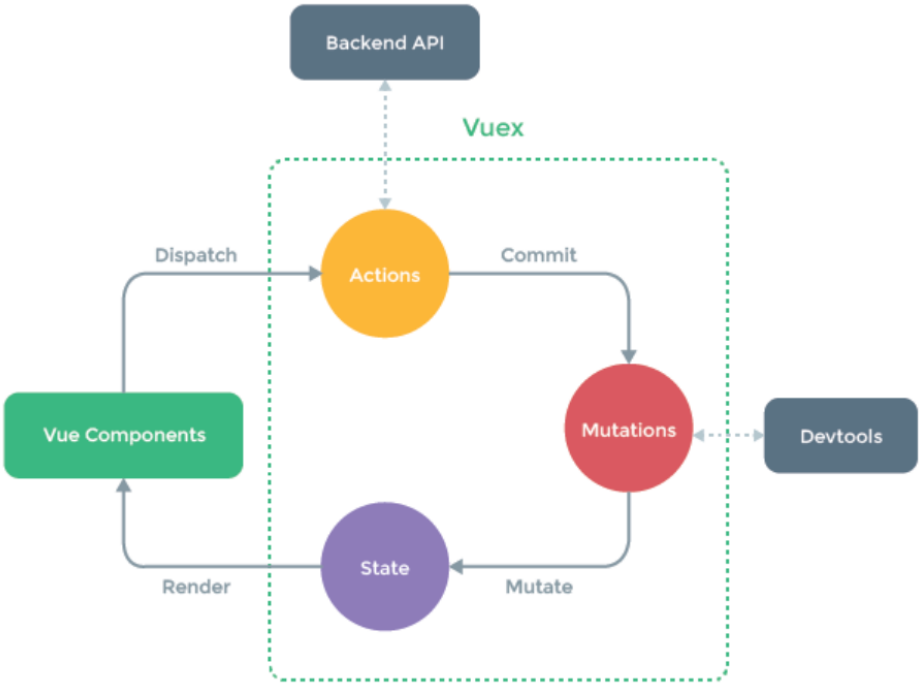
对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。

对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

因此，我们为什么不把组件的共享状态抽取出来，以一个全局单例模式管理呢？在这种模式下，我们的组件树构成了一个巨大的“视图”，不管在树的哪个位置，任何组件都能获取状态或者触发行为！

通过定义和隔离状态管理中的各种概念并通过强制规则维持视图和状态间的独立性，我们的代码将会变得更结构化且易维护。

这就是 Vuex 背后的基本思想,Vuex 是专门为 Vue.js 设计的状态管理库，以利用 Vue.js 的细粒度数据响应机制来进行高效的状态更新。




## 什么情况下我应该使用 Vuex?

Vuex 可以帮助我们管理共享状态，并附带了更多的概念和框架。这需要对短期和长期效益进行权衡。

如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的。确实是如此——如果您的应用够简单，您最好不要使用 Vuex。一个简单的 store模式就足够您所需了。但是，如果您需要构建一个中大型单页应用，您很可能会考虑如何更好地在组件外部管理状态，Vuex 将会成为自然而然的选择。引用 Redux 的作者 Dan Abramov 的话说就是：

> Flux 架构就像眼镜：您自会知道什么时候需要它。

每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的**状态 (state)**。Vuex 和单纯的全局对象有以下两点不同：

1. Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。
2. 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地**提交 (commit) mutation**。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

## 核心概念

### State

单一状态树

Vuex 使用**单一状态树**——用一个对象就包含了全部的应用层级状态。至此它便作为一个“唯一数据源 而存在。这也意味着，每个应用将仅仅包含一个 store 实例。单一状态树让我们能够直接地定位任一特定的状态片段，在调试的过程中也能轻易地取得整个当前应用状态的快照。

**Getter**

Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。就像计算属性一样，getter 的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

**Mutation**

更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutation 非常类似于事件：每个 mutation 都有一个字符串的 **事件类型 (type)** 和 一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数，

不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 `increment` 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 **store.commit** 方法

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

**Action**

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。

Action 通过 `store.dispatch` 方法触发：

```
store.dispatch('increment')
```

乍一眼看上去感觉多此一举，我们直接分发 mutation 岂不更方便？实际上并非如此，还记得 **mutation 必须同步执行**这个限制么？Action 就不受约束！我们可以在 action 内部执行**异步**操作

**Module**

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

为了解决以上问题，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、甚至是嵌套子模块——从上至下进行同样方式的分割

```
const moduleA = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态
```

## 实操运用

main.js引入

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store/index'

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

vuex配置

```
store/index.js
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)

//创建store对象
```

```
const store = new Vuex.Store({
    state : {
        count: 10
    },
    mutations:{
        add(state){
            state.count+=1;
        }
    },
    actions:{
        increment(context){
            context.commit('add');
        },
        //另一种写法
        increment({commit, state}){
            commit('add');
        }
    }
})

//导出store对象
export default store;
```

页面获取vuex值

```
<template>
  <div>
    <h1>{{$store.state.count}}</h1>
  </div>
</template>
```

修改vuex中state的值

```
<template>
  <div>
    <h1>{{$store.state.count}}</h1>
    <button @click="increment">加</button>
  </div>
</template>

methods:{
    increment(){
      this.$store.dispatch('increment');
    },
```

## 辅助函数

### mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性：

```
<template>
  <div id="app">
    <router-view />
    <NavBottom  v-show="showNav"/>
```

```
  </div>
</template>

import {mapGetters} from 'vuex'  //辅助函数

export default {
  data(){
    return {
    }
  },
  computed:mapGetters(['showNav']),
}
```

**mapActions辅助函数**

使用 mapActions 辅助函数将组件的 methods 映射为 store.dispatch调用

1、没有使用mapActions 辅助函数的调用方式

```
<template>
    <div>
      <h1>{{$store.state.count}}</h1>
      <button @click="add()">add</button>
      <button @click="del()">del</button>
    </div>
</template>

export default {
  data () {
    return {
      text:"find",
    }
  },
  methods:{
    add(){
      this.$store.dispatch('ADD');
    },
    del(){
      this.$store.dispatch('DEL');
    }
  }
}
```

2、使用mapActions 辅助函数的处理

```
<template>
    <div>
      <h1>{{$store.state.count}}</h1>
      <button @click="add()">add</button>
      <button @click="del()">del</button>
      <button @click="ADD">ADD</button>
      <button @click="DEL">DEL</button>
    </div>
</template>

import {mapActions} from 'vuex'  //辅助函数
```

```
export default {
  data () {
    return {
      text:"find",
    }
  },
   methods:mapActions(['ADD','DEL'])
}
```

扩展练习

vuex数据共享：
1、底部导航显隐
2、购物车的状态及数据处理