Vue 开发必须知道的 36 个技巧

前言

Vue基本用法很容易上手,但是有很多优化的写法你就不一定知道了,本文从列举了 36 个 vue 开发技巧;

1.require.context()

1.场景:如页面需要导入多个组件,原始写法:

1.场景:如页面需要导入多个组件,原始写法:

```
import titleCom from '@/components/home/titleCom'
import bannerCom from '@/components/home/bannerCom'
import cellCom from '@/components/home/cellCom'
components:{titleCom,bannerCom,cellCom}
```

2.这样就写了大量重复的代码,利用 require.context 可以写成

```
const path = require('path')
const files = require.context('@/components/home', false, /\.vue$/)
const modules = {}
files.keys().forEach(key => {
  const name = path.basename(key, '.vue')
  modules[name] = files(key).default || files(key)
})
components:modules
```

这样不管页面引入多少组件,都可以使用这个方法

3.API 方法

```
实际上是 webpack 的方法,vue 工程一般基于 webpack,所以可以使用
require.context(directory,useSubdirectories,regExp)
接收三个参数:
directory：说明需要检索的目录
useSubdirectories：是否检索子目录
regExp: 匹配文件的正则表达式,一般是文件名
```

2.watch

2.1 常用用法

1.场景:表格初始进来需要调查询接口 getList(),然后input 改变会重新查询

```
created(){
  this.getList()
},
watch: {
  inpVal(){
    this.getList()
  }
}
```

2.2 立即执行

2.可以直接利用 watch 的immediate和handler属性简写

```
watch: {
  inpVal:{
    handler: 'getList',
      immediate: true
  }
}
```

2.3 深度监听

3.watch 的 deep 属性,深度监听,也就是监听复杂数据类型

```
watch:{
  inpValObj:{
    handler(newVal,oldVal){
      console.log(newVal)
      console.log(oldVal)
    },
    deep:true
  }
}
```

此时发现oldVal和 newVal 值一样; 因为它们索引同一个对象/数组,Vue 不会保留修改之前值的副本; 所以深度监听虽然可以监听到对象的变化,但是无法监听到具体对象里面那个属性的变化

3.14种组件通讯

3.1 props

这个应该非常属性,就是父传子的属性; props 值可以是一个数组或对象;

```
// 数组:不建议使用
props:[]

// 对象
props:{
 inpVal:{
  type:Number, //传入值限定类型
  // type 值可为String,Number,Boolean,Array,Object,Date,Function,Symbol
  // type 还可以是一个自定义的构造函数，并且通过 instanceof 来进行检查确认
```

```
  required: true, //是否必传
  default:200,  //默认值,对象或数组默认值必须从一个工厂函数获取如 default:()=>[]
  validator:(value) {
    // 这个值必须匹配下列字符串中的一个
    return ['success', 'warning', 'danger'].indexOf(value) !== -1
  }
 }
}
```

3.2 $emit

这个也应该非常常见,触发子组件触发父组件给自己绑定的事件,其实就是子传父的方法

```
// 父组件
<home @title="title">
// 子组件
this.$emit('title',[{title:'这是title'}])
```

3.3 vuex

1.这个也是很常用的,vuex 是一个状态管理器 2.是一个独立的插件,适合数据共享多的项目里面,因为如果只是简单的通讯,使用起来会比较重 3.API

```
state:定义存贮数据的仓库 ,可通过this.$store.state 或mapState访问
getter:获取 store 值,可认为是 store 的计算属性,可通过this.$store.getter 或
       mapGetters访问
mutation:同步改变 store 值,为什么会设计成同步,因为mutation是直接改变 store 值,
         vue 对操作进行了记录,如果是异步无法追踪改变.可通过mapMutations调用
action:异步调用函数执行mutation,进而改变 store 值,可通过 this.$dispatch或mapActions
       访问
modules:模块,如果状态过多,可以拆分成模块,最后在入口通过...解构引入
```

3.4 和listeners

2.4.0 新增 这两个是不常用属性,但是高级用法很常见; 1.场景如果父传子有很多值那么在子组件需要定义多个解决 attrs获取子传父中未在 props 定义的值

```
// 父组件
<home title="这是标题" width="80" height="80" imgUrl="imgUrl"/>

// 子组件
mounted() {
  console.log(this.$attrs) //{title: "这是标题", width: "80", height: "80", imgUrl: "imgUrl"}
},
```

相对应的如果子组件定义了 props,打印的值就是剔除定义的属性

```
props: {
  width: {
    type: String,
    default: ''
  }
},
mounted() {
  console.log(this.$attrs) //{title: "这是标题", height: "80", imgUrl: "imgUrl"}
},
```

2.场景子组件需要调用父组件的方法解决父组件的方法可以通过listeners" 传入内部组件——在创建更高层次的组件时非常有用

```
// 父组件
<home @change="change"/>

// 子组件
mounted() {
  console.log(this.$listeners) //即可拿到 change 事件
}
```

如果是孙组件要访问父组件的属性和调用方法,直接一级一级传下去就可以

3.inheritAttrs

```
// 父组件
<home title="这是标题" width="80" height="80" imgUrl="imgUrl"/>

// 子组件
mounted() {
  console.log(this.$attrs) //{title: "这是标题", width: "80", height: "80", imgUrl: "imgUrl"}
},

inheritAttrs默认值为true，true的意思是将父组件中除了props外的属性添加到子组件的根节点上(说明，即使设置为true，子组件仍然可以通过$attr获取到props意外的属性)
将inheritAttrs:false后,属性就不会显示在根节点上了
```

3.5 provide和inject

2.2.0 新增 描述: provide 和 inject 主要为高阶插件/组件库提供用例。并不推荐直接用于应用程序代码中; 并且这对选项需要一起使用; 以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效。

```
//父组件:
provide: { //provide 是一个对象,提供一个属性或方法
  foo: '这是 foo',
  fooMethod:()=>{
    console.log('父组件 fooMethod 被调用')
  }
},
```

```
// 子或者孙子组件
inject: ['foo','fooMethod'], //数组或者对象,注入到子组件
mounted() {
  this.fooMethod()
  console.log(this.foo)
}
//在父组件下面所有的子组件都可以利用inject
```

provide 和 inject 绑定并不是可响应的。这是官方刻意为之的。 然而，如果你传入了一个可监听的对象，那么其对象的属性还是可响应的,对象是因为是引用类型

```
//父组件:
provide: {
  foo: '这是 foo'
},
mounted(){
  this.foo='这是新的 foo'
}

// 子或者孙子组件
inject: ['foo'],
mounted() {
  console.log(this.foo) //子组件打印的还是'这是 foo'
}
```

3.6 和children

父实例children:子实例

```
//父组件
mounted(){
  console.log(this.$children)
  //可以拿到 一级子组件的属性和方法
  //所以就可以直接改变 data,或者调用 methods 方法
}

//子组件
mounted(){
  console.log(this.$parent) //可以拿到 parent 的属性和方法
}
```

和parent 并不保证顺序，也不是响应式的 只能拿到一级父组件和子组件

3.7 $refs

```
// 父组件
<home ref="home"/>

mounted(){
  console.log(this.$refs.home) //即可拿到子组件的实例,就可以直接操作 data 和 methods
}
```

3.8 $root

```
// 父组件
mounted(){
  console.log(this.$root) //获取根实例,最后所有组件都是挂载到根实例上
  console.log(this.$root.$children[0]) //获取根实例的一级子组件
  console.log(this.$root.$children[0].$children[0]) //获取根实例的二级子组件
}
```

3.9 .sync

在 vue@1.x 的时候曾作为双向绑定功能存在，即子组件可以修改父组件中的值; 在 vue@2.0 的由于违背单项数据流的设计被干掉了; 在 vue@2.3.0+ 以上版本又重新引入了这个 .sync 修饰符;

```
// 父组件
<home :title.sync="title" />
//编译时会被扩展为
<home :title="title"  @update:title="val => title = val"/>

// 子组件
// 所以子组件可以通过$emit 触发 update 方法改变
mounted(){
  this.$emit("update:title", '这是新的title')
}
```

3.10 v-slot

2.6.0 新增 1.slot,slot-cope,scope 在 2.6.0 中都被废弃,但未被移除 2.作用就是将父组件的 template 传入子组件 3.插槽分类: A.匿名插槽(也叫默认插槽): 没有命名,有且只有一个;

```
// 父组件
<todo-list>
    <template v-slot:default>
       任意内容
       <p>我是匿名插槽 </p>
    </template>
</todo-list>

// 子组件
<slot>我是默认值</slot>
//v-slot:default写上感觉和具名写法比较统一,容易理解,也可以不用写
```

B.具名插槽: 相对匿名插槽组件slot标签带name命名的;

```
// 父组件
<todo-list>
    <template v-slot:todo>
        任意内容
        <p>我是匿名插槽 </p>
    </template>
</todo-list>

//子组件
<slot name="todo">我是默认值</slot>
```

C.作用域插槽: 子组件内数据可以被父页面拿到(解决了数据只能从父页面传递给子组件)

```
// 父组件
<todo-list>
 <template v-slot:todo="slotProps" >
   {{slotProps.user.firstName}}
 </template>
</todo-list>
//slotProps 可以随意命名
//slotProps 接取的是子组件标签slot上属性数据的集合所有v-bind:user="user"

// 子组件
<slot name="todo" :user="user" :test="test">
    {{ user.lastName }}
 </slot>
data() {
    return {
      user:{
        lastName:"Zhang",
        firstName:"yue"
      },
      test:[1,2,3,4]
    }
  },
// {{ user.lastName }}是默认数据  v-slot:todo 当父页面没有(="slotProps")
```

3.11 EventBus

1.就是声明一个全局Vue实例变量 EventBus , 把所有的通信数据，事件监听都存储到这个变量上; 2.类似于 Vuex。但这种方式只适用于极小的项目 3.原理就是利用和emit 并实例化一个全局 vue 实现数据共享

```
// 在 main.js
Vue.prototype.$eventBus=new Vue()

// 传值组件
this.$eventBus.$emit('eventTarget','这是eventTarget传过来的值')

// 接收组件
this.$eventBus.$on("eventTarget",v=>{
  console.log('eventTarget',v);//这是eventTarget传过来的值
})
```

4.可以实现平级,嵌套组件传值,但是对应的事件名eventTarget必须是全局唯一的

3.12 broadcast和dispatch

vue 1.x 有这两个方法,事件广播和派发,但是 vue 2.x 删除了 下面是对两个方法进行的封装

```
function broadcast(componentName, eventName, params) {
  this.$children.forEach(child => {
    var name = child.$options.componentName;

    if (name === componentName) {
      child.$emit.apply(child, [eventName].concat(params));
    } else {
      broadcast.apply(child, [componentName, eventName].concat(params));
    }
  });
}
export default {
  methods: {
    dispatch(componentName, eventName, params) {
      var parent = this.$parent;
      var name = parent.$options.componentName;
      while (parent && (!name || name !== componentName)) {
        parent = parent.$parent;

        if (parent) {
          name = parent.$options.componentName;
        }
      }
      if (parent) {
        parent.$emit.apply(parent, [eventName].concat(params));
      }
    },
    broadcast(componentName, eventName, params) {
      broadcast.call(this, componentName, eventName, params);
    }
  }
}
```

3.13 路由传参

1.方案一

```
// 路由定义
{
  path: '/describe/:id',
  name: 'Describe',
  component: Describe
}
// 页面传参
this.$router.push({
  path: `/describe/${id}`,
})
// 页面获取
this.$route.params.id
```

2.方案二

```
// 路由定义
{
  path: '/describe',
  name: 'Describe',
  omponent: Describe
}
// 页面传参
this.$router.push({
  name: 'Describe',
  params: {
    id: id
  }
})
// 页面获取
this.$route.params.id
```

3.方案三

```
// 路由定义
{
  path: '/describe',
  name: 'Describe',
  component: Describe
}
// 页面传参
this.$router.push({
  path: '/describe',
    query: {
      id: id
  `}
)
// 页面获取
this.$route.query.id
```

4.三种方案对比 方案二参数不会拼接在路由后面,页面刷新参数会丢失 方案一和三参数拼接在后面,丑,而且暴露了信息

3.14 Vue.observable

2.6.0 新增 用法:让一个对象可响应。Vue 内部会用它来处理 data 函数返回的对象;

> 返回的对象可以直接用于渲染函数和计算属性内，并且会在发生改变时触发相应的更新；
> 也可以作为最小化的跨组件状态存储器，用于简单的场景。

通讯原理实质上是利用Vue.observable实现一个简易的 vuex

```
// 文件路径 - /store/store.js
import Vue from 'vue'

export const store = Vue.observable({ count: 0 })
export const mutations = {
  setCount (count) {
    store.count = count
  }
}

//使用
<template>
    <div>
        <label for="bookNum">数 量</label>
            <button @click="setCount(count+1)">+</button>
            <span>{{count}}</span>
            <button @click="setCount(count-1)">-</button>
    </div>
</template>

<script>
import { store, mutations } from '../store/store' // Vue2.6新增API Observable

export default {
  name: 'Add',
  computed: {
    count () {
      return store.count
    }
  },
  methods: {
    setCount: mutations.setCount
  }
}
</script>
```

4.render 函数

1.场景:有些代码在 template 里面写会重复很多,所以这个时候 render 函数就有作用啦

```
// 根据 props 生成标签
// 初级
<template>
  <div>
    <div v-if="level === 1"> <slot></slot> </div>
```

```
    <p v-else-if="level === 2"> <slot></slot> </p>
    <h1 v-else-if="level === 3"> <slot></slot> </h1>
    <h2 v-else-if="level === 4"> <slot></slot> </h2>
    <strong v-else-if="level === 5"> <slot></slot> </stong>
    <textarea v-else-if="level === 6"> <slot></slot> </textarea>
  </div>
</template>

// 优化版,利用 render 函数减小了代码重复率
<template>
  <div>
    <child :level="level">Hello world!</child>
  </div>
</template>

<script type='text/javascript'>
  import Vue from 'vue'
  Vue.component('child', {
    render(h) {
      const tag = ['div', 'p', 'strong', 'h1', 'h2', 'textarea'][this.level-1]
      return h(tag, this.$slots.default)
    },
    props: {
      level: {  type: Number,  required: true  }
    }
  })
  export default {
    name: 'hehe',
    data() { return { level: 3 } }
  }
</script>
```

2.render 和 template 的对比 前者适合复杂逻辑,后者适合逻辑简单; 后者属于声明是渲染，前者属于自定Render函数; 前者的性能较高，后者性能较低。

5.异步组件

场景:项目过大就会导致加载缓慢,所以异步组件实现按需加载就是必须要做的事啦 1.异步注册组件 3种方法

```
// 工厂函数执行 resolve 回调
Vue.component('async-webpack-example', function (resolve) {
  // 这个特殊的 `require` 语法将会告诉 webpack
  // 自动将你的构建代码切割成多个包，这些包
  // 会通过 Ajax 请求加载
  require(['./my-async-component'], resolve)
})

// 工厂函数返回 Promise
Vue.component(
  'async-webpack-example',
  // 这个 `import` 函数会返回一个 `Promise` 对象。
  () => import('./my-async-component')
)
```

```
// 工厂函数返回一个配置化组件对象
const AsyncComponent = () => ({
  // 需要加载的组件 (应该是一个 `Promise` 对象)
  component: import('./MyComponent.vue'),
  // 异步组件加载时使用的组件
  loading: LoadingComponent,
  // 加载失败时使用的组件
  error: ErrorComponent,
  // 展示加载时组件的延时时间。默认值是 200 (毫秒)
  delay: 200,
  // 如果提供了超时时间且组件加载也超时了，
  // 则使用加载失败时使用的组件。默认值是：`Infinity`
  timeout: 3000
})
```

异步组件的渲染本质上其实就是执行2次或者2次以上的渲染, 先把当前组件渲染为注释节点, 当组件加载成功后, 通过 forceRender 执行重新渲染。或者是渲染为注释节点, 然后再渲染为loading节点, 在渲染为请求完成的组件

2.路由的按需加载

```
webpack< 2.4 时
{
  path:'/',
  name:'home',
  components:resolve=>require(['@/components/home'],resolve)
}

webpack> 2.4 时
{
  path:'/',
  name:'home',
  components:()=>import('@/components/home')
}

import()方法由es6提出，import()方法是动态加载，返回一个Promise对象，then方法的参数是加载到的模块。类似于Node.js的require方法，主要import()方法是异步加载的。
```

6.动态组件

场景:做一个 tab 切换时就会涉及到组件动态加载

```
<component v-bind:is="currentTabComponent"></component>
```

但是这样每次组件都会重新加载,会消耗大量性能,所以 就起到了作用

```
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
```

这样切换效果没有动画效果,这个也不用着急,可以利用内置的

```
<transition>
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
</transition>
```

7.递归组件

场景:如果开发一个 tree 组件,里面层级是根据后台数据决定的,这个时候就需要用到动态组件

```
// 递归组件: 组件在它的模板内可以递归的调用自己，只要给组件设置name组件就可以了。
// 设置那么House在组件模板内就可以递归使用了,不过需要注意的是，
// 必须给一个条件来限制数量，否则会抛出错误: max stack size exceeded
// 组件递归用来开发一些具体有未知层级关系的独立组件。比如：
// 联级选择器和树形控件

<template>
  <div v-for="(item,index) in treeArr">
      子组件，当前层级值： {{index}} <br/>
      <!-- 递归调用自身，后台判断是否不存在改值 -->
      <tree :item="item.arr" v-if="item.flag"></tree>
  </div>
</template>
<script>
export default {
  // 必须定义name，组件内部才能递归调用
  name: 'tree',
  data(){
    return {}
  },
  // 接收外部传入的值
  props: {
     item: {
      type:Array,
      default: ()=>[]
    }
  }
}
</script>
```

递归组件必须设置name 和结束的阀值

8.函数式组件

定义:无状态,无法实例化，内部没有任何生命周期处理方法 规则:在 2.3.0 之前的版本中，如果一个函数式组件想要接收 prop，则 props 选项是必须的。

```
在 2.3.0 或以上的版本中，你可以省略 props 选项，所有组件上的特性都会被自动隐式解析为 prop
在 2.5.0 及以上版本中，如果你使用了单文件组件(就是普通的.vue 文件),可以直接在 template 上声明functional
组件需要的一切都是通过 context 参数传递
```

context 属性有: 1.props：提供所有 prop 的对象 2.children: VNode 子节点的数组 3.slots: 一个函数，返回了包含所有插槽的对象 4.scopedSlots: (2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。 5.data：传递给组件的整个数据对象，作为 createElement 的第二个参数传入组件 6.parent：对父组件的引用 7.listeners: (2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器的对象。这是 data.on 的一个别名。 8.injections: (2.3.0+) 如果使用了 inject 选项，则该对象包含了应当被注入的属性

```
<template functional>
  <div v-for="(item,index) in props.arr">{{item}}</div>
</template>
```

9.components和 Vue.component

components:局部注册组件

```
export default{
  components:{home}
}
```

Vue.component:全局注册组件

```
Vue.component('home',home)
```

10.Vue.extend

场景:vue 组件中有些需要将一些元素挂载到元素上,这个时候 extend 就起到作用了 是构造一个组件的语法器 写法:

```
// 创建构造器
var Profile = Vue.extend({
  template: '<p>{{extendData}}</br>实例传入的数据为:{{propsExtend}}</p>',//template对应的标签最外层必须只有一个标签
  data: function () {
    return {
      extendData: '这是extend扩展的数据',
    }
  },
  props:['propsExtend']
})

// 创建的构造器可以挂载到元素上,也可以通过 components 或 Vue.component()注册使用
// 挂载到一个元素上。可以通过propsData传参.
new Profile({propsData:{propsExtend:'我是实例传入的数据'}}).$mount('#app-extend')

// 通过 components 或 Vue.component()注册
Vue.component('Profile',Profile)
```

11.mixins

场景:有些组件有些重复的 js 逻辑,如校验手机验证码,解析时间等,mixins 就可以实现这种混入 mixins 值是一个数组

```
const mixin={
    created(){
      this.dealTime()
    },
    methods:{
      dealTime(){
        console.log('这是mixin的dealTime里面的方法');
      }
  }
}

export default{
  mixins:[mixin]
}
```

12.extends

extends用法和mixins很相似,只不过接收的参数是简单的选项对象或构造函数,所以extends只能单次扩展一个组件

```
const extend={
    created(){
      this.dealTime()
    },
    methods:{
      dealTime(){
        console.log('这是mixin的dealTime里面的方法');
      }
  }
}

export default{
  extends:extend
}
```

13.Vue.use()

场景:我们使用 element时会先 import,再 Vue.use()一下,实际上就是注册组件,触发 install 方法; 这个在组件调用会经常使用到; 会自动组织多次注册相同的插件.

14.install

场景:在 Vue.use()说到,执行该方法会触发 install 是开发vue的插件,这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象(可选)

```
var MyPlugin = {};
  MyPlugin.install = function (Vue, options) {
    // 2. 添加全局资源,第二个参数传一个值默认是update对应的值
    Vue.directive('click', {
      bind(el, binding, vnode, oldVnode) {
        //做绑定的准备工作,添加时间监听
        console.log('指令my-directive的bind执行啦');
      },
```

```
      inserted: function(el){
      //获取绑定的元素
      console.log('指令my-directive的inserted执行啦');
      },
      update: function(){
      //根据获得的新值执行对应的更新
      //对于初始值也会调用一次
      console.log('指令my-directive的update执行啦');
      },
      componentUpdated: function(){
      console.log('指令my-directive的componentUpdated执行啦');
      },
      unbind: function(){
      //做清理操作
      //比如移除bind时绑定的事件监听器
      console.log('指令my-directive的unbind执行啦');
      }
    })

    // 3. 注入组件
    Vue.mixin({
      created: function () {
        console.log('注入组件的created被调用啦');
        console.log('options的值为',options)
      }
    })

    // 4. 添加实例方法
    Vue.prototype.$myMethod = function (methodOptions) {
      console.log('实例方法myMethod被调用啦');
    }
  }

  //调用MyPlugin
  Vue.use(MyPlugin,{someOption: true })

  //3.挂载
  new Vue({
    el: '#app'
  });
```

更多请戳 vue中extend，mixins，extends，components,install的几个操作

15. Vue.nextTick

2.1.0 新增 场景:页面加载时需要让文本框获取焦点 用法:在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM

```
mounted(){ //因为 mounted 阶段 dom 并未渲染完毕,所以需要$nextTick
  this.$nextTick(() => {
    this.$refs.inputs.focus() //通过 $refs 获取dom 并绑定 focus 方法
  })
}
```

16.Vue.directive

16.1 使用

场景:官方给我们提供了很多指令,但是我们如果想将文字变成指定的颜色定义成指令使用,这个时候就需要用到Vue.directive

```
// 全局定义
Vue.directive("change-color",function(el,binding,vnode){
  el.style["color"]= binding.value;
})

// 使用
<template>
<div v-change-color="color">{{message}}</div>
</template>
<script>
  export default{
    data(){
      return{
        color:'green'
      }
    }
  }
</script>
```

16.2 生命周期

1.bind 只调用一次，指令第一次绑定到元素时候调用，用这个钩子可以定义一个绑定时执行一次的初始化动作。2.inserted:被绑定的元素插入父节点的时候调用(父节点存在即可调用，不必存在document中) 3.update: 被绑定与元素所在模板更新时调用，而且无论绑定值是否有变化，通过比较更新前后的绑定值，忽略不必要的模板更新 4.componentUpdate :被绑定的元素所在模板完成一次更新更新周期的时候调用 5.unbind: 只调用一次，指令月元素解绑的时候调用

17. Vue.filter

场景:时间戳转化成年月日这是一个公共方法,所以可以抽离成过滤器使用

```
// 使用
// 在双花括号中
{{ message | capitalize }}

// 在 `v-bind` 中
<div v-bind:id="rawId | formatId"></div>

// 全局注册
Vue.filter('stampToYYMMDD', (value) =>{
  // 处理逻辑
})

// 局部注册
filters: {
  stampToYYMMDD: (value)=> {
```

```
    // 处理逻辑
  }
}

// 多个过滤器全局注册
// /src/common/filters.js
let dateServer = value => value.replace(/(\d{4})(\d{2})(\d{2})/g, '$1-$2-$3')
export { dateServer }
// /src/main.js
import * as custom from './common/filters/custom'
Object.keys(custom).forEach(key => Vue.filter(key, custom[key]))
```

18.Vue.compile

场景:在 render 函数中编译模板字符串。只在独立构建时有效

```
var res = Vue.compile('<div><span>{{ msg }}</span></div>')

new Vue({
  data: {
    msg: 'hello'
  },
  render: res.render,
  staticRenderFns: res.staticRenderFns
})
```

19.Vue.version

场景:有些开发插件需要针对不同 vue 版本做兼容,所以就会用到 Vue.version 用法:Vue.version()可以获取 vue 版本

```
var version = Number(Vue.version.split('.')[0])

if (version === 2) {
  // Vue v2.x.x
} else if (version === 1) {
  // Vue v1.x.x
} else {
  // Unsupported versions of Vue
}
```

20.Vue.set()

场景:当你利用索引直接设置一个数组项时或你修改数组的长度时,由于 Object.defineprototype()方法限制,数据不响应式更新

```
不过vue.3.x 将利用 proxy 这个问题将得到解决
```

解决方案:

```
// 利用 set
this.$set(arr,index,item)

// 利用数组 push(),splice()
```

21.Vue.config.keyCodes

场景:自定义按键修饰符别名

```
// 将键码为 113 定义为 f2
Vue.config.keyCodes.f2 = 113;
<input type="text" @keyup.f2="add"/>
```

22.Vue.config.performance

场景:监听性能

```
Vue.config.performance = true
```

只适用于开发模式和支持 performance.mark API 的浏览器上

23.Vue.config.errorHandler

1.场景:指定组件的渲染和观察期间未捕获错误的处理函数 2.规则:

```
  从 2.2.0 起，这个钩子也会捕获组件生命周期钩子里的错误。同样的，当这个钩子是 undefined 时，被捕获的错误会通过 console.error 输出而避免应用崩溃
  从 2.4.0 起，这个钩子也会捕获 Vue 自定义事件处理函数内部的错误了
  从 2.6.0 起，这个钩子也会捕获 v-on DOM 监听器内部抛出的错误。另外，如果任何被覆盖的钩子或处理函数返回一个 Promise 链（例如 async 函数），则来自其 Promise 链的错误也会被处理
```

3.使用

```
Vue.config.errorHandler = function (err, vm, info) {
  // handle error
  // `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子
  // 只在 2.2.0+ 可用
}
```

24.Vue.config.warnHandler

2.4.0 新增 1.场景:为 Vue 的运行时警告赋予一个自定义处理函数,只会在开发者环境下生效 2.用法:

```
Vue.config.warnHandler = function (msg, vm, trace) {
  // `trace` 是组件的继承关系追踪
}
```

25.v-pre

场景:vue 是响应式系统,但是有些静态的标签不需要多次编译,这样可以节省性能

```
<span v-pre>{{ this will not be compiled }}</span>   显示的是{{ this will not be compiled }}
<span v-pre>{{msg}}</span>     即使data里面定义了msg这里仍然是显示的{{msg}}
```

26.v-cloak

场景:在网速慢的情况下,在使用vue绑定数据的时候，渲染页面时会出现变量闪烁 用法:这个指令保持在元素上直到关联实例结束编译。和 CSS 规则如 [v-cloak] { display: none } 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕

```
// template 中
<div class="#app" v-cloak>
    <p>{{value.name}}</p>
</div>

// css 中
[v-cloak] {
    display: none;
}
```

这样就可以解决闪烁,但是会出现白屏,这样可以结合骨架屏使用

27.v-once

场景:有些 template 中的静态 dom 没有改变,这时就只需要渲染一次,可以降低性能开销

```
<span v-once> 这时只需要加载一次的标签</span>
```

v-once 和 v-pre 的区别: v-once只渲染一次；v-pre不编译,原样输出

28.事件修饰符

```
.stop:阻止冒泡
.prevent:阻止默认行为
.self:仅绑定元素自身触发
.once: 2.1.4 新增,只触发一次
.passive: 2.3.0 新增,滚动事件的默认行为 (即滚动行为) 将会立即触发,不能和.prevent 一起使用
```

29.按键修饰符和按键码

场景:有的时候需要监听键盘的行为,如按下 enter 去查询接口等

```
// 对应键盘上的关键字
.enter
.tab
.delete (捕获“删除”和“退格”键)
.esc
.space
.up
.down
.left
.right
```

30.Vue-router

场景:Vue-router 是官方提供的路由插件

30.1 缓存和动画

1.路由是使用官方组件 vue-router,使用方法相信大家非常熟悉; 2.这里我就叙述下路由的缓存和动画; 3.可以用 exclude(除了)或者include(包括),2.1.0 新增来坐判断

```
<transition>
  <keep-alive :include="['a', 'b']">
  //或include="a,b" :include="/a|b/",a 和 b 表示组件的 name
  //因为有些页面,如试试数据统计,要实时刷新,所以就不需要缓存
    <router-view/> //路由标签
  </keep-alive>
  <router-view exclude="c"/>
  // c 表示组件的 name值
</transition>
```

注:匹配首先检查组件自身的 name 选项，如果 name 选项不可用，则匹配它的局部注册名称 (父组件 components 选项的键值)。匿名组件不能被匹配 4.用 v-if 做判断,组件会重新渲染,但是不用一一列举组件 name

30.2 全局路由钩子

1.router.beforeEach

```
router.beforeEach((to, from, next) => {
  console.log('全局前置守卫：beforeEach -- next需要调用') //一般登录拦截用这个,也叫导航钩子守卫
  if (path === '/login') {
    next()
    return
  }
  if (token) {
    next();
  }
})
```

2.router.beforeResolve (v 2.5.0+) 和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 即在 beforeEach之后调用

3.router.afterEach 全局后置钩子 在所有路由跳转结束的时候调用 这些钩子不会接受 next 函数也不会改变导航本身

30.3 组件路由钩子

1.beforeRouteEnter 在渲染该组件的对应路由被确认前调用，用法和参数与router.beforeEach类似，next需要被主动调用 此时组件实例还未被创建，不能访问this 可以通过传一个回调给 next来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数

```
beforeRouteEnter (to, from, next) {
  // 这里还无法访问到组件实例，this === undefined
  next( vm => {
    // 通过 `vm` 访问组件实例
  })
}
```

2.beforeRouteUpdate (v 2.2+) 在当前路由改变，并且该组件被复用时调用，可以通过this访问实例， next需要被主动调用，不能传回调

3.beforeRouteLeave 导航离开该组件的对应路由时调用，可以访问组件实例 this，next需要被主动调用，不能传回调

30.4 路由模式

设置 mode 属性:hash或 history

30.5 Vue.$router

```
this.$router.push():跳转到不同的url，但这个方法回向history栈添加一个记录，点击后退会返回到上一个页面
this.$router.replace():不会有记录
this.$router.go(n):n可为正数可为负数。正数返回上一个页面,类似 window.history.go(n)
```

30.6 Vue.$route

表示当前跳转的路由对象,属性有: name:路由名称 path:路径 query:传参接收值 params:传参接收值 fullPath:完成解析后的 URL，包含查询参数和 hash 的完整路径 matched:路由记录副本 redirectedFrom:如果存在重定向，即为重定向来源的路由的名字

```
this.$route.params.id:获取通过 params 或/:id传参的参数
this.$route.query.id:获取通过 query 传参的参数
```

30.7 router-view 的 key

场景:由于 Vue 会复用相同组件, 即 /page/1 => /page/2 或者 /page?id=1 => /page?id=2 这类链接跳转时, 将不在执行created, mounted之类的钩子

```
<router-view :key="$route.fullpath"></router-view>
```

这样组件的 created 和 mounted 就都会执行

31.Object.freeze

场景:一个长列表数据,一般不会更改,但是vue会做getter和setter的转换 用法:是ES5新增的特性，可以冻结一个对象，防止对象被修改 支持:vue 1.0.18+对其提供了支持，对于data或vuex里使用freeze冻结了的对象，vue不会做getter和setter的转换 注意:冻结只是冻结里面的单个属性,引用地址还是可以更改

```
new Vue({
    data: {
        // vue不会对list里的object做getter、setter绑定
        list: Object.freeze([
            { value: 1 },
            { value: 2 }
        ])
    },
    mounted () {
        // 界面不会有响应,因为单个属性被冻结
        this.list[0].value = 100;

        // 下面两种做法，界面都会响应
        this.list = [
            { value: 100 },
            { value: 200 }
        ];
        this.list = Object.freeze([
            { value: 100 },
            { value: 200 }
        ]);
    }
})
```

32.调试 template

场景:在Vue开发过程中, 经常会遇到template模板渲染时JavaScript变量出错的问题, 此时也许你会通过console.log来进行调试 这时可以在开发环境挂载一个 log 函数

```
// main.js
Vue.prototype.$log = window.console.log;

// 组件内部
<div>{{$log(info)}}</div>
```

33.vue-loader 小技巧

33.1 preserveWhitespace

场景:开发 vue 代码一般会有空格,这个时候打包压缩如果不去掉空格会加大包的体积 配置preserveWhitespace可以减小包的体积

```
{
  vue: {
    preserveWhitespace: false
  }
}
```

33.2 transformToRequire

场景:以前在写 Vue 的时候经常会写到这样的代码：把图片提前 require 传给一个变量再传给组件

```
// page 代码
<template>
  <div>
    <avatar :img-src="imgSrc"></avatar>
  </div>
</template>
<script>
  export default {
    created () {
      this.imgSrc = require('./assets/default-avatar.png')
    }
  }
</script>
```

现在:通过配置 transformToRequire 后，就可以直接配置，这样vue-loader会把对应的属性自动 require 之后传给组件

```
// vue-cli 2.x在vue-loader.conf.js 默认配置是
transformToRequire: {
    video: ['src', 'poster'],
    source: 'src',
    img: 'src',
    image: 'xlink:href'
}

// 配置文件,如果是vue-cli2.x 在vue-loader.conf.js里面修改
  avatar: ['default-src']

// vue-cli 3.x 在vue.config.js
// vue-cli 3.x 将transformToRequire属性换为了transformAssetUrls
module.exports = {
  pages,
  chainWebpack: config => {
    config
      .module
        .rule('vue')
        .use('vue-loader')
        .loader('vue-loader')
        .tap(options => {
      options.transformAssetUrls = {
        avatar: 'img-src',
      }
      return options;
      });
  }
}

// page 代码可以简化为
<template>
  <div>
    <avatar img-src="./assets/default-avatar.png"></avatar>
```

```
  </div>
</template>
```

34.为路径设置别名

1.场景:在开发过程中，我们经常需要引入各种文件，如图片、CSS、JS等，为了避免写很长的相对路径（../），我们可以为不同的目录配置一个别名

2.vue-cli 2.x 配置

```
// 在 webpack.base.config.js中的 resolve 配置项，在其 alias 中增加别名
resolve: {
    extensions: ['.js', '.vue', '.json'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js',
      '@': resolve('src'),
    }
  },
```

3.vue-cli 3.x 配置

```
// 在根目录下创建vue.config.js
var path = require('path')
function resolve (dir) {
  console.log(__dirname)
  return path.join(__dirname, dir)
}
module.exports = {
  chainWebpack: config => {
    config.resolve.alias
      .set(key, value) // key,value自行定义，比如.set('@@', resolve('src/components'))
  }
}
```

35.img 加载失败

场景:有些时候后台返回图片地址不一定能打开,所以这个时候应该加一张默认图片

```
// page 代码
<img :src="imgUrl" @error="handleError" alt="">
<script>
export default{
  data(){
    return{
      imgUrl:''
    }
  },
  methods:{
    handleError(e){
      e.target.src=reqiure('图片路径') //当然如果项目配置了transformToRequire,参考上面 27.2
    }
  }
```

```
}
</script>
```

36.css

36.1 局部样式

1.Vue中style标签的scoped属性表示它的样式只作用于当前模块，是样式私有化.

2.渲染的规则/原理： 给HTML的DOM节点添加一个 不重复的data属性 来表示 唯一性 在对应的 CSS选择器 末尾添加一个当前组件的 data属性选择器来私有化样式，如：.demo[data-v-2311c06a]{} 如果引入 less 或 sass 只会在最后一个元素上设置

```
// 原始代码
<template>
  <div class="demo">
    <span class="content">
      Vue.js scoped
    </span>
  </div>
</template>

<style lang="less" scoped>
  .demo{
    font-size: 16px;
    .content{
      color: red;
    }
  }
</style>

// 浏览器渲染效果
<div data-v-fed36922>
  Vue.js scoped
</div>
<style type="text/css">
.demo[data-v-039c5b43] {
  font-size: 14px;
}
.demo .content[data-v-039c5b43] { //.demo 上并没有加 data 属性
  color: red;
}
</style>
```

36.2 deep 属性

```
// 上面样式加一个 /deep/
<style lang="less" scoped>
  .demo{
    font-size: 14px;
  }
  .demo /deep/ .content{
    color: blue;
```

```
  }
</style>

// 浏览器编译后
<style type="text/css">
.demo[data-v-039c5b43] {
  font-size: 14px;
}
.demo[data-v-039c5b43] .content {
  color: blue;
}
</style>
```

●输入m获取到文章目录