# 重构的收获

1. String.fromCharCode（numX） 可接受一个指定的 Unicode 值，然后返回一个字符串。 1

参数：numX ，必填的，一个或多个 Unicode 值。

2. @click="$router.go(-1)" 返回上一个页面

### 3. 移动端点击延迟事件

1. 移动端浏览器在派发点击事件的时候，通常会出现300ms左右的延迟

2. 原因: 移动端的双击会缩放导致click判断延迟

# 解决方法：Fastclick

// 安装 cnpm install fastclick -S

// 引入 import FastClick from 'fastclick'

    // 使用 FastClick.attach(document.body);

使用js 引用：<script type='application/javascript' src='/path/to/fastclick.js'></script>

使用jquery ： $(function() {

        FastClick.attach(document.body);

    });

// 没使用jquery的时候

    if ('addEventListener' in document) {

        document.addEventListener('DOMContentLoaded', function() {

            FastClick.attach(document.body);

        }, false);

    }

# 4.go back与return有什么区别

go back回去,回到某一个地方  
 return是从A地到B地,然后强调了又从B地回到A地  
 而且return还有一个用法就是指借了东西,归还回去

# 5. [Fetch](https://www.cnblogs.com/libin-1/p/6853677.html)

**Fetch获取数据**

假设我们想通过GitHub获取一个仓库，我们可以像下面这样使用：

fetch('https://api.github.com/users/chriscoyier/repos');

Fetch会返回Promise，所以在获取资源后，可以使用.then方法做你想做的。

fetch('https://api.github.com/users/chriscoyier/repos')

.then(response => {/\* do something \*/})

Github返回的响应是JSON格式的，所以调用response.json方法来转换数据。还有其他方法来处理不同类型的响应。如果请求一个XML格式文件，则调用response.text。如果请求图片，使用response.blob方法

# 列如： fetch('https://api.github.com/users/chriscoyier/repos')

# .then(response => response.json())

# .then(data => {

# // data就是我们请求的repos

# console.log(data)

# });

# Fetch发送数据

使用Fetch发送也很简单，只需要配置三个参数。

第一个参数是设置请求方法（如post、put或del），Fetch会自动设置方法为get。

第二个参数是设置头部。因为一般使用JSON数据格式，所以设置ContentType为application/json。

第三个参数是设置包含JSON内容的主体。因为JSON内容是必须的，所以当设置主体时会调用JSON.stringify。

列如： let content = {some: 'content'};

// The actual fetch request

fetch('some-url', {

method: 'post',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(content)

})

// .then()...

**6. async**

它作为一个关键字放到函数前面，用于表示函数是一个异步函数

async 函数返回的是一个promise 对象，如果要获取到promise 返回值，我们应该用then 方法。Promise 有一个resolved，这是async 函数内部的实现原理，如果async 函数中有返回一个值 ,当调用该函数时，内部会调用Promise.solve() 方法把它转化成一个promise 对象作为返回，但如果timeout 函数内部抛出错误呢？ 那么就会调用Promise.reject() 返回一个promise 对象， 这时修改一下timeout 函数

async function timeout(flag) {

if (flag) {

return 'hello world'

} else {

throw 'my god, failure'

}

}

console.log(timeout(true)) // 调用Promise.resolve() 返回promise 对象。

console.log(timeout(false)); // 调用Promise.reject() 返回promise 对象**。**

**7.** XLink，即 XML 链接语言，用于在 XML 文件上创建内部和外部链接，以及与这些链接相关联的元数据。

属性（xlink:href）值（*URL*） 描述（要连接的url）

**8.** scrollBehavior滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。接收to和from两个路由对象

scrollBehavior (to, from, savedPosition) {

// return 期望滚动到哪个的位置

}

**9. @include指令**

使用include指令可以方便的把多个页面中重复显的内容抽取出来，大大的减少代码的重复量，方便我们对重复内容的维护

Vue路由懒加载

const tvProgram = resolve => require(['路由'], resolve);

const game = resolve => require(['路由'], resolve);

const gameitem = resolve => require(['路由'], resolve)

## [require-ensure](http://webpack.github.io/docs/api-in-modules.html#require-ensure)和[require-amd](http://webpack.github.io/docs/api-in-modules.html" \l "require-amd" \t "_blank)的区别：

require-amd

说明: 同AMD规范的require函数，使用时传递一个模块数组和回调函数，模块都被下载下来且都被执行后才执行回调函数

语法: require(dependencies: String[], [callback: function(...)])

参数

dependencies: 模块依赖数组

callback: 回调函数

require-ensure

说明: require.ensure在需要的时候才下载依赖的模块，当参数指定的模块都下载下来了（下载下来的模块还没执行），便执行参数指定的回调函数。require.ensure会创建一个chunk，且可以指定该chunk的名称，如果这个chunk名已经存在了，则将本次依赖的模块合并到已经存在的chunk中，最后这个chunk在webpack构建的时候会单独生成一个文件。

语法: require.ensure(dependencies: String[], callback: function([require]), [chunkName: String])

dependencies: 依赖的模块数组

callback: 回调函数，该函数调用时会传一个require参数

chunkName: 模块名，用于构建时生成文件时命名使用

**注意点**：requi.ensure的模块只会被下载下来，不会被执行，只有在回调函数使用require(模块名)后，这个模块才会被执行

**Vue项目中使用的Mixins**

混入 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项

**使用：**mixins就是定义一部分公共的方法或者计算属性,然后混入到各个组件中使用,方便管理与统一修改

**1、在assets文件夹下创建一个js文件**

// 创建一个需要混入的对象

export const mixinTest1 = {

created() {

this.hello();

},

methods: {

hello() {

console.log('mixinTest1');

}

}

};

**2、在组件中使用刚刚创建的混入**

import {mixinTest1} from './../assets/js/mixin';

export default {

mixins:[mixinTest1],

name: 'hello',

data () {

return {

msg: 'Welcome to Your Vue.js App'

}

}

}

这样就可以直接调用到混入对象中的hello方法

3、如果组件中定义的方法与混入对象中的方法/属性一样,组件中的优先级大于混入对象中的(方法会调用多次)

4、混入对象中可以定义抽象方法,使用混入的组件必须重写该方法

methods: {

handlePlaylist() {

throw new Error('component must implement handlePlaylist method')

}

}

**Vue.nextTick**

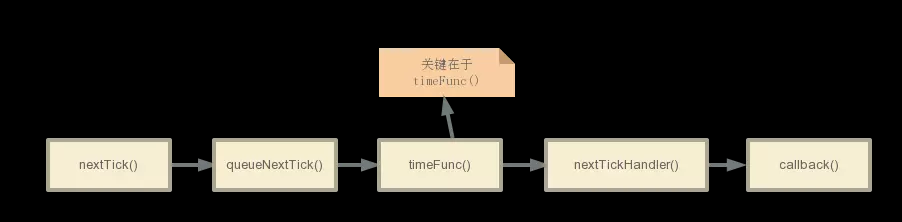
作用：Vue.nextTick用于延迟执行一段代码，它接受2个参数（回调函数和执行回调函数的上下文环境），如果没有提供回调函数，那么将返回promise对象。

参数： callbacks 用来存储所有需要执行的回调函数

Pending 用来标志是否正在执行回调函数

timerFunc 用来触发执行回调函数

nextTickHandler()函数用来执行callbacks里存储的所有回调函数，接下来是将触发方式赋值给timerFunc；最后是queueNextTick函数。因为nextTick是一个即时函数，所以queueNextTick函数是返回的函数，接受用户传入的参数，用来往callbacks里存入回调函数。



* 而timeFunc()一共有三种实现方式：Promise MutationObserver setTimeout

其中Promise和setTimeout是一个异步任务，会在同步任务以及更新DOM的异步任务之后回调具体函数。

MutationObserver是HTML5中的新API，是个用来监视DOM变动的接口。他能监听一个DOM对象上发生的子节点删除、属性修改、文本内容修改等等。调用时需要先绑回调

var mo = new MutationObserver(callback)

通过给MutationObserver的构造函数传入一个回调，能得到一个MutationObserver实例，这个回调就会在MutationObserver实例监听到变动时触发。

这个时候你只是给MutationObserver实例绑定好了回调，他具体监听哪个DOM、监听节点删除还是监听属性修改，还没有设置。而调用他的observer方法就可以完成这一步:

var domTarget = 你想要监听的dom节点

mo.observe(domTarget, {

characterData: true //说明监听文本内容的修改。

})

使用 MutationObserver的原因就是 nextTick想要一个异步API，用来在当前的同步代码执行完毕后，执行我想执行的异步回调，包括Promise和 setTimeout都是基于这个原因。

**Vue中watch监听**

在vue中，使用watch来响应数据的变化。watch的用法大致有三种。下面代码是watch的一种简单的用法：

<input type="text" v-model="cityName"/>

new Vue({

   el: '#root',

   data: {

     cityName: 'shanghai'

   },

   watch: {

     cityName(newName, oldName) {      // ...    }

   }

})

直接写一个监听处理函数，当每次监听到 cityName 值发生改变时，执行函数。也可以在所监听的数据后面直接加字符串形式的方法名：

watch: {

     cityName: 'nameChange'

     }

 }

这样使用watch时有一个特点，就是当值第一次绑定的时候，不会执行监听函数，只有值发生改变才会执行。如果我们需要在最初绑定值的时候也执行函数，则就需要用到immediate属性。

监听的数据后面写成对象形式，包含handler方法和immediate，之前我们写的函数其实就是在写这个handler方法；

immediate表示在watch中首次绑定的时候，是否执行handler，值为true则表示在watch中声明的时候，就立即执行handler方法，值为false，则和一般使用watch一样，在数据发生变化的时候才执行handler。

### deep

当需要监听一个对象的改变时，普通的watch方法无法监听到对象内部属性的改变，只有data中的数据才能够监听到变化，此时就需要deep属性对对象进行深度监听。

<input type="text" v-model="cityName.name"/>

new Vue({

  el: '#root',

  data: {

    cityName: {id: 1, name: 'shanghai'}

  },

  watch: {

    cityName: {

      handler(newName, oldName) {      // ...    },

    deep: true,

    immediate: true

    }

  }

})

设置deep: true 则可以监听到cityName.name的变化，此时会给cityName的所有属性都加上这个监听器，当对象属性较多时，每个属性值的变化都会执行handler。如果只需要监听对象中的一个属性值，则可以做以下优化：使用字符串的形式监听对象属性：

watch: {    'cityName.name': {

      handler(newName, oldName) {      // ...      },

      deep: true,

      immediate: true

    }

  }  
这样只会给对象的某个特定的属性加监听器。数组（一维、多维）的变化不需要通过深度监听，对象数组中对象的属性变化则需要deep深度监听。

**Svg**

SVG 指可伸缩矢量图形 (Scalable Vector Graphics)

SVG 用来定义用于网络的基于矢量的图形

SVG 使用 XML 格式定义图形

SVG 图像在放大或改变尺寸的情况下其图形质量不会有所损失

SVG 是万维网联盟的标准

SVG 与诸如 DOM 和 XSL 之类的 W3C 标准是一个整体 、

**优势：**

1.可被非常多的工具读取和修改（比如记事本）

2.与 JPEG 和 GIF 图像比起来，尺寸更小，且可压缩性更强。

3.图像可在任何的分辨率下被高质量地打印

4.可以与 Java 技术一起运行

**标签：**

### 1)<svg>标签

<svg width="100%" height="100%">

<circle id="mycircle" cx="50" cy="50" r="50" />

</svg>

<svg>的width属性和height属性，指定了 SVG 图像在 HTML 元素中所占据的宽度和高度。SVG 图像默认大小是300像素（宽） x 150像素（高）。

如果只想展示 SVG 图像的一部分，就要指定viewBox属性：

<svg width="100" height="100" viewBox="50 50 50 50">

<circle id="mycircle" cx="50" cy="50" r="50" />

</svg>

<viewBox>属性的值有四个数字，分别是左上角的横坐标和纵坐标、视口的宽度和高度。

如果不指定width属性和height属性，只指定viewBox属性，则相当于只给定 SVG 图像的长宽比。这时，SVG 图像的默认大小将等于所在的 HTML 元素的大小。

**2)**<circle>标签 代表圆形。

<svg width="300" height="180">

<circle cx="30" cy="50" r="25" />

<circle cx="90" cy="50" r="25" class="red" />

<circle cx="150" cy="50" r="25" class="fancy" />

</svg>

<circle>标签的cx、cy、r属性分别为横坐标、纵坐标和半径，单位为像素

**3)**<line>标签用来绘制直线。

<svg width="300" height="180">

<line x1="0" y1="0" x2="200" y2="0" style="stroke:rgb(0,0,0);stroke-width:5" />

</svg>

<line>标签的x1属性和y1属性，表示线段起点的横坐标和纵坐标；x2属性和y2属性，表示线段终点的横坐标和纵坐标；style属性表示线段的样式。

**4)** <polyline>标签用于绘制一根折线。

<svg width="300" height="180">

<polyline points="3,3 30,28 3,53" fill="none" stroke="black" />

</svg>

<polyline>的points属性指定了每个端点的坐标，横坐标与纵坐标之间与逗号分隔，点与点之间用空格分隔。

**5)** <rect>标签用于绘制矩形。

<svg width="300" height="180">

<rect x="0" y="0" height="100" width="200" style="stroke: #70d5dd; fill: #dd524b" />

</svg>

<rect>的x属性和y属性，指定了矩形左上角端点的横坐标和纵坐标；width属性和height属性指定了矩形的宽度和高度（单位像素）。

**6)** <ellipse>标签用于绘制椭圆。

<svg width="300" height="180">

<ellipse cx="60" cy="60" ry="40" rx="20" stroke="black" stroke-width="5" fill="silver"/>

</svg>

<ellipse>的cx属性和cy属性，指定了椭圆中心的横坐标和纵坐标（单位像素）；rx属性和ry属性，指定了椭圆横向轴和纵向轴的半径（单位像素）。

**7)** <polygon>标签用于绘制多边形。

<svg width="300" height="180">

<polygon fill="green" stroke="orange" stroke-width="1" points="0,0 100,0 100,100 0,100 0,0"/>

</svg>

<polygon>的points属性指定了每个端点的坐标，横坐标与纵坐标之间与逗号分隔，点与点之间用空格分隔。**better-scroll**

better-scroll 是一款重点解决移动端（已支持 PC）各种滚动场景需求的插件，基于原生 JS 实现的，不依赖任何框架。html 结构

<div class="wrapper">

<ul class="content">

<li>...</li>

<li>...</li>

...

</ul>

<!-- 这里可以放一些其它的 DOM，但不会影响滚动 -->

</div>

上面的代码中 better-scroll 是作用在外层 wrapper 容器上的，滚动的部分是 content 元素。这里要注意的是，better-scroll 只处理容器（wrapper）的第一个子元素（content）的滚动，其它的元素都会被忽略。

初始化代码如下：

import BScroll from 'better-scroll'

let wrapper = document.querySelector('.wrapper')

let scroll = new BScroll(wrapper)

better-scroll 提供了一个类，实例化的第一个参数是一个原生的 DOM 对象。当然，如果传递的是一个字符串，better-scroll 内部会尝试调用 querySelector 去获取这个 DOM 对象，所以初始化代码也可以是这样：

import BScroll from 'better-scroll'

let scroll = new BScroll('.wrapper')

# keep-alive

在做电商有关的项目中，当我们第一次进入列表页需要请求一下数据，当我从列表页进入详情页，详情页不缓存也需要请求下数据，然后返回列表页，这时候我们使用keep-alive来缓存组件，防止二次渲染，这样会大大的节省性能。

**.keep-alive的基本用法**

在app.vue中

[?](https://www.jb51.net/article/140188.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5 | <!-- 缓存所有的页面 -->  <keep-alive>   <router-view v-if="$route.meta.keep\_alive"></router-view>  </keep-alive>  <router-view v-if="!$route.meta.keep\_alive"></router-view> |

需要缓存的组件内容直接在router中添加：

[?](https://www.jb51.net/article/140188.htm)

|  |  |
| --- | --- |
| 1  2  3 | meta: {   keepAlive: true // true 表示需要使用缓存 false表示不需要被缓存   } |

**keep-alive的生命周期**

当引入keep-alive的时候，页面第一次进入，钩子的触发顺序created-> mounted-> activated，退出时触发deactivated。当再次进入（前进或者后退）时，只触发activated。

**问题描述**

在业务开发中，会有路由跳转但是返回需要保留数据的场景；vue 中提供了 keep-alive 来处理

**解决方案**

返回dom不让其重新刷新，在vue-view外面包一层, 当引入keep-alive的时候，页面第一次进入，钩子的触发顺序created-> mounted-> activated，退出时触发deactivated。当再次进入（前进或者后退）时，只触发activated。

 事件挂载的方法等，只执行一次的放在 mounted 中；组件每次进去执行的方法放在 activated 中；   
 可以将 是否包裹 keep-alive 通过参数配置；

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <keep-alive>    <router-view v-if="$route.meta.keepAlive" style="min-height:100%"></router-view>  </keep-alive>  <router-view v-if="!$route.meta.keepAlive" style="min-height:100%"></router-view>  //不需要刷新的路由配置里面配置 meta: {keepAlive: true}, 这个路由则显示在上面标签；  //需要刷新的路由配置里面配置 meta: {keepAlive: false}, 这个路由则显示在下面标签； |