# 重构项目总结

# **FastClick:**用于移动设备上的浏览器默认会在用户点击屏幕大约延迟300毫秒后才会触发点击事件(延迟事件)

**原理**：在检测到touchend事件的时候，会通过DOM自定义事件立即触发模拟一个click事件，并把浏览器在300ms之后真正的click事件阻止掉

**FastClick的使用：**

1. 要使用首先npm install fastClick来安装
2. 然后在页面引入fastclick.js （import FastClick from 'fastclick'）
3. 使用： FastClick.attach(document.body);
4. 初始化FastClick实例建议在页面的DOM文档加载完成后。

if ('addEventListener' in document) {

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

FastClick.attach(document.body);

}, false);

}

**不需要使用fastclick的情况：**

1. FastClick是不会对PC浏览器添加监听事件
2. Android版Chrome 32位的浏览器，如果设置viewport meta的值为width=device-width，这种情况下浏览器会马上触发点击事件，不会延迟300毫秒。

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

String.fromCharCode(numX,numX,...,numX)

一个或多个 Unicode 值，即要创建的字符串中的字符的 Unicode 编码。

* **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

}

* **this.$router.go()和this.$router.push()**

this.$router.go(val) => 在history记录中前进或者后退val步，当val为0时刷新当前页面。

this.$router.push(path) => 在history栈中添加一条新的记录。

* **插槽、具名插槽、作用域插槽**

1. 插槽：如果父组件想在调用的子组件设置内容，那么，就在子组件定义一个solt,就可以在父组件对子组件进行设置内容了
2. 具名插槽:具名插槽说白了就是给插槽起个名字让他们对应的插槽指向对应的内容位置。在父组件调用子组件的时候，给子组件中的solt起个name名，然后在父组件设置solt属性，属性值对应子组件的name属性值。 没定义name就是默认插槽
3. 作用域插槽:如果需要一个从子组件获取一个可用的插槽的时候我们就可以用作用域插槽。先在子组件找到需要插入内容的地方，然后用solt包裹起来，然后将要传的内容通过solt标签v-bind绑定，将跟他有关联的数据传递给slot里，在父组件调用的时候通过slot-scope获取到内容。在2.5版本以上的后不再限制只能在template标签上使用了。再是在插槽内的所有标签上都可以使用。

* **Async**

async. 默认是true，即为异步方式，$.Ajax执行后，会继续执行ajax后面的脚本，直到服务器端返回数据后，触发$.Ajax里的success方法，这时候执行的是两个线程。若要将其设置为false，则所有的请求均为同步请求，在没有返回值之前，同步请求将锁住浏览器，用户其它操作必须等待请求完成才可以执行。

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。

* **@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

# **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

* **Polyline**

用当前画笔描绘一系列线段，使用PolylineTo函数时，当前位置会设为最后一条线段的终点。它不会由Polyline函数改动。Polyline既不使用也不改变目前位置。

* **Svg**

可缩放矢量图形是基于可扩展标记语言（标准通用标记语言的子集），用于描述二维矢量图形的一种图形格式。它由万维网联盟制定，是一个开放标准。

SVG-Edit 是一个基于浏览器的图像编辑器，可进行常用的一些图像处理功能，无需服务器端支持，支持各种浏览器。

**svg比较优势：**

1. 任意放缩。

用户可以任意缩放图像显示，而不会破坏图像的清晰度、细节等。

1. 文本独立。

SVG图像中的文字独立于图像，文字保留可编辑和可搜寻的状态。也不会再有字体的限制，用户系统即使没有安装某一字体，也会看到和他们制作时完全相同的画面。

1. 较小文件。

总体来讲，SVG文件比那些GIF和JPEG格式的文件要小很多，因而下载也很快。

1. 超强显示效果。

SVG图像在屏幕上总是边缘清晰，它的清晰度适合任何屏幕分辨率和打印分辨率。

1. 超级颜色控制。

SVG图像提供一个1 600万种颜色的调色板，支持ICC颜色描述文件标准、RGB、线X填充、渐变和蒙版。

1. 交互X和智能化。

SVG面临的主要问题一个是如何和已经占有重要市场份额的矢量图形格式Flash竞争的问题，另一个问题就是SVG的本地运行环境下的厂家支持程度。

* **Await**

 后面必须跟Promise对象，若非Promise，则不会拦截后面代码执行。当promise对象resolve过后并且执行完then里面的代码，就执行下一步代码，不resolve不会触发下一行代码执行。

需注意：如果then()中需要异步操作，不会等then中的异步执行完过后再执行下一个then（）的函数。原因就是，异步函数中，没有地方给你return给then（）回调函数。解决方案是async函数。

也就是说Promise对于异步的帮助 其实很有限，.then()只有第一个有用而已。

const getdirs = async function(ctx) {

let sendFiles = []

const files = await readdir(filePath) // 读文件夹

const promises = files.map(function(file){ // 利用map函数特性，返回值组成新的数组，这儿并没有用async函数，map内并不等待一个stat回来后再进行另一个stat，是同时进行的。

return stat(filePath + '/' + file)

.then(res => {

if(res === 'dir') sendFiles.push(file)

})

})

await Promise.all(promises) // 这儿是异步并发的关键，在这个位置等待所有promise对象resolve。

ctx.body = sendFiles

}

* **Promise**

很多现代浏览器都已经实现，但是为了兼容，建议自行对Promise进行封装或者使用第三方的解决方案（如webpack对es6语法进行编译）。 那么，我么将得到一个Promise构造函数，新建一个Promise的实例：

var \_promise = new Promise(function(resolve, reject){

setTimeout(function(){

var rand = Math.random();

if(rand<0.5){

resolve("resolve" + rand);

}else{

reject("reject" + rand);

}

},1000);

});

/\*运行结果:

\*有两种情况：

\*1)无事发生

\*2)报错形如：d.js:7 Uncaught (in promise) reject0.9541820247347901

\*/

Promise的构造函数接收一个函数作为参数，该函数接受两个额外的函数，resolve和reject，这两个函数分别代表将当前Promise置为fulfilled(解决)和rejected(拒绝)两个状态。Promise正是通过这两个状态来控制异步操作的结果。接下来我们将讨论Promise的用法，实际上Promise上的实例\_promise是一个对象，不是一个函数。在声明的时候，Promise传递的参数函数会立即执行，因此Promise使用的正确姿势是在其外层再包裹一层函数。

var run = function(){

var \_promise = new Promise(function(resolve, reject){

setTimeout(function(){

var rand = Math.random();

if(rand<0.5){

resolve("resolve" + rand);

}else{

reject("reject" + rand);

}

},1000);

});

return \_promise;

}

run();

这是Promise的正常用法，接下来，就是对异步操作结果的处理，接着上面创建的函数run()

run().then(function(data){

console.log(data);

});

每个Promise的实例对象，都有一个then的方法，这个方法就是用来处理之前各种异步逻辑的结果。

* **vue路由的懒加载**

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

require-ensure和require-amd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。

* **minxin使用及介绍**

vue中提供了一种混合机制--mixins，用来更高效的实现组件内容的复用。最开始我一度认为这个和组件好像没啥区别。。后来发现错了。下面我们来看看mixins和普通情况下引入组件有什么区别？

组件在引用之后相当于在父组件内开辟了一块单独的空间，来根据父组件props过来的值进行相应的操作，单本质上两者还是泾渭分明，相对独立。

而mixins则是在引入组件之后，则是将组件内部的内容如data等方法、method等属性与父组件相应内容进行合并。相当于在引入后，父组件的各种属性方法都被扩充了。

单纯组件引用：

父组件 + 子组件 >>> 父组件 + 子组件

mixins：

父组件 + 子组件 >>> new父组件

作用：多个组件可以共享数据和方法，在使用mixin的组件中引入后，mixin中的方法和属性也就并入到该组件中，可以直接使用。钩子函数会两个都被调用，mixin中的钩子首先执行。

下面给大家介绍vue mixin的用法，具体介绍如下所示：

1. 定义一个 js 文件(mixin.js)

export default {

data() {

return {

name: 'mixin'

}

},

created() {

console.log('mixin...', this.name);

},

mounted() {},

methods: {}

}

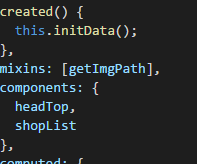
1. 在vue文件中使用mixin

import '@/mixin'; // 引入mixin文件

export default {

mixins: [mixin]

}



详见网址：<https://blog.csdn.net/qq_36838191/article/details/81004590>

* **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。

* **v-load-more实现上拉加载**

[vue](https://www.baidu.com/s?wd=vue&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/przlovecsdn/article/details/_blank)通过自定义指令实现loadMore上拉加载

详见网址：<https://blog.csdn.net/przlovecsdn/article/details/82900062>

* **Polygon**

<polygon> 标签用来创建含有不少于三个边的图形。

下面的例子创建一个四边形：

<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"

"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"

xmlns="http://www.w3.org/2000/svg">

<polygon points="220,100 300,210 170,250 123,234"

style="fill:#cccccc;

stroke:#000000;stroke-width:1"/>

</svg>

* **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

}