# 面试总结

**编写日期：**2020.5.11-

目录

[面试总结 1](#_Toc40183079)

[一、 Webpack 3](#_Toc40183080)

[二、 Vue 12](#_Toc40183081)

[三、 React 13](#_Toc40183082)

[四、 NodeJS 14](#_Toc40183083)

[五、 JavaScript 15](#_Toc40183084)

[六、 Uniapp 16](#_Toc40183085)

[七、 微信小程序 17](#_Toc40183086)

[八、 Html 18](#_Toc40183087)

[九、 Css 19](#_Toc40183088)

## Webpack

参考链接

Webpack中文文档：<https://www.webpackjs.com/concepts/>

webpack4 css打包、压缩、分离、去重等优化配置详解

<https://my.oschina.net/itlangz/blog/2986976>

#### 基础结构（entry、output、loader、plugins）

##### 入口(entry)

入口起点(entry point)指示 webpack 应该使用哪个模块，来作为构建其内部依赖图的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

方式一，单文件时

entry: path.resolve(\_\_dirname, '../src/index.js'),

方式二，多文件时

数组形式

    entry: [SRC\_PATH + '/index.js', SRC\_PATH + '/test.js'],

    对象形式

    entry: {

        index: SRC\_PATH + '/index.js',

        test: SRC\_PATH + '/test.js'

    },

##### 输出(output)

output 属性告诉 webpack 在哪里输出它所创建的 bundles，以及如何命名这些文件，默认值为 ./dist。基本上，整个应用程序结构，都会被编译到你指定的输出路径的文件夹中

  output: {

    path: DIST\_PATH, //打包后地址

    filename: `static/js/[name].[${isDev ? 'hash' : 'chunkhash:5'}].js`   //[chunkhash:number] hash生成位数

  },

##### 文件预处理器（loader）

loader 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只理解 JavaScript）。loader 可以将所有类型的文件转换为 webpack 能够处理的有效模块，然后你就可以利用 webpack 的打包能力，对它们进行处理

###### ES6+语法转换为ES5（babel）

参考链接：

1. Babel中文文档

<https://www.babeljs.cn/docs/>

1. @babel/preset-env 与@babel/plugin-transform-runtime 使用及场景区别<https://segmentfault.com/a/1190000021188054>
2. corejs3按需导入polyfill

<https://segmentfault.com/a/1190000020237817>

1. Babel-loader，babel-core和babel-preset之间是什么关系

<http://www.fly63.com/article/detial/5736>

`babel-loader` 是一个 npm 包，它使得 webpack 可以通过 babel 转译 JavaScript 代码。（在 babel 7 中 `babel-core` 和 `babel-preset` 被建议使用 `@babel` 开头声明作用域，因此应该分别下载 `@babel/core` 和· `@babel/presets`。就类似于 vue-cli 升级后 使用@vue/cli一样的道理 ）

babel 的功能在于「代码转译」，具体一点，即将目标代码转译为能够符合期望语法规范的代码。在转译的过程中，babel 内部经历了「解析 - 转换 - 生成」三个步骤。而 `@babel/core` 这个库则负责「解析」，具体的「转换」和「生成」步骤则交给各种插件（plugin）和预设（preset）来完成。

`@babel/preset-\*` 实际上就是各种插件的打包组合，也就是说各种转译规则的统一设定，目的是告诉loader要以什么规则来转化成对应的js版本

所需基础插件：babel-loader babel-core babel-preset-env

npm install babel-loader @babel/core @babel/preset-env --save-dev

module: {

    rules: [

      { test: /\.m?jsx?$/, loader: 'babel-loader', exclude: /node\_modules/ },

      {

        test: /\.vue$/,

        use: 'vue-loader'

      },

]

}

默认 @babel/preset-env 只会转换语法，也就是我们看到的箭头函数、const一类。

如果进一步需要转换内置对象、实例方法，那就得用polyfill

Polyfill 是一块代码（通常是 Web 上的 JavaScript），用来为旧浏览器提供它没有原生支持的较新的功能。

比如说 polyfill 可以让 IE7 使用 Silverlight 插件来模拟 HTML Canvas 元素的功能，或模拟 CSS 实现 rem 单位的支持，或 text-shadow，或其他任何你想要的功能。

但是Polyfill会把全部的兼容性代码导入最后编译的包，产生大量冗余代码、污染全局变量

此处只记录corejs@3版本方案,并实现按需加载

（需要 babel-loader 版本升级到 8.0.0 以上，@babel/core 版本升级到 7.4.0 及以上）

Babel 7的改动还是不少的，一个比较大的改动在于移除了之前的stage-x插件

[@babel/plugin-proposal-class-properties](https://babeljs.io/docs/en/next/babel-plugin-proposal-class-properties.html" \t "https://segmentfault.com/a/_blank)：用来解析类的属性的

箭头函数被当作成class的属性来看待，this也不会指向undefined

**class** **Bork** {

boundFunction = () => {

**return** **this**.state;

}

}

配置方法：.babelrc文件

{

  "plugins": [

    "@babel/proposal-class-properties"

  ]

}

1. 方式一：core-js + regenerator-runtime

// corejs@3升级的两种方案

// polyfill是在你的source code 前运行的，所以安装的时候是 --save

// 1.polyfill 垫片方案（根据 useBuiltIns 配置参数不同，既可以按需加载，也可以不按需）的升级策略：

// @babel/polyfill不必再安装，转而需要依靠core-js和regenerator-runtime（详细原因请看作者的阐述），替代方案用法如下：

// 安装依赖

// npm i -D babel-loader @babel/core @babel/preset-env

// npm i -S core-js regenerator-runtime

{

  "presets": [

    "@babel/preset-react",

    "@babel/preset-typescript",

    [

      "@babel/preset-env",

      {

        "targets": {

          "chrome": "50",

          "ie": "8"

        },

        "modules": false,

        "useBuiltIns": "usage",

        "corejs": {

          "version": 3, // 使用core-js@3

          "proposals": true,

        }

      }

    ]

  ]

}

1. 方式二：@babel/runtime-corejs3

// 2.transform-runtime 按需加载方案 的升级策略：

// 安装依赖

// npm i -D babel-loader @babel/core @babel/preset-env @babel/plugin-transform-runtime

// npm i -S @babel/runtime-corejs3

{

  "presets": [

    "@babel/preset-react",

    "@babel/preset-typescript",

    [

      "@babel/preset-env",

      {

        "modules": false

      }

    ]

  ],

  "plugins": [

    [

      "@babel/plugin-transform-runtime",

      {

        "corejs": {

          "version": 3,

          "proposals": true

        },

        "useESModules": true

      }

    ]

  ]

}

###### 处理css、less、scss

基础模块：css-loader style-loader

css-loader使你能够使用类似@import 和 url(…)的方法实现 require()的功能；

style-loader将所有的计算后的样式加入页面中

// less模块编译

npm install --save-dev less-loader less

// sass模块编译

npm install --save-dev node-sass sass-loader

module: {

  rules: [

    {

      test: /\.scss$/,

      use: ["style-loader", "css-loader"]

    },

    {

      test: /\.scss$/,

      // 加载loader的三种方式

      use: ["style-loader", "css-loader", "sass-loader"],

      // use: [

      //     {

      //         loader: "style-loader" // 将 JS 字符串生成为 style 节点

      //     },

      //     {

      //         loader: "css-loader" // 将 CSS 转化成 CommonJS 模块

      //     },

      //     {

      //         loader: "sass-loader" // 将 Sass 编译成 CSS

      //     }

      // ],

      // loader: 'style-loaer!css-loader!sass-loader'

    },

    {

      test: /\.less$/,

      use: ['style-loader', 'css-loader', 'less-loader']

    }

  ]

}

###### 处理图片、字体图标等文件

file-loader 返回的是图片的url

url-loader可以通过limit属性对图片分情况处理，当图片小于limit（单位：byte）大小时转base64，大于limit时调用file-loader对图片进行处理。

相同点：都是在webpack中处理图片、字体图标等文件。

关系：url-loader封装了file-loader，但url-loader并不依赖于file-loader。

npm install--save file-loader url-loader

**错误记录**

打包图片文件时路径错误输出[object-module] ：http://localhost:3000/[object%20Module] 404 (Not Found)

file-loader新版本默认使用了esModule语法，造成了引用图片文件时的方式和以前的版本不一样，通过查看其仓库的release会发现：在v4.3.0版本就引入这一新特性。

url-loader是基于file-loader的，所以使用url-loader打包图片也会出现相应问题，所以需要在webpack.config中将esModule默认的true，手动设置为false。

{

                test: /\.(png|jpe?g|gif|svg)(\?.\*)?$/,

                use: [

                    {

                        loader: 'url-loader',

                        options: {

esModule: false, //关闭esModule，启用CommonJS模块语法

                            limit: 10000,

                            name: 'static/images/[name].[hash:7].[ext]'

                        }

                    }

                ]

            },

            {

                test: /\.(mp4|webm|ogg|mp3|wav|flac|aac)(\?.\*)?$/,

                loader: 'url-loader',

                options: {

                    limit: 10000,

                    name: 'static/images/[name].[hash:7].[ext]'

                }

            },

            {

                test: /\.(woff2?|eot|ttf|otf)(\?.\*)?$/,

                loader: 'url-loader',

                options: {

                    limit: 10000,

                    name: 'static/images/[name].[hash:7].[ext]'

                }

            }

###### 处理Vue

npm install--save vue

npm install--save - dev vue-loader vue-style-loader  vue-template-compiler

module: {

    rules: [

      {

        test: /\.vue$/,

        use: 'vue-loader'

      },

]

}

##### 插件(plugins)

loader 被用于转换某些类型的模块，而插件则可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量。[插件接口](https://www.webpackjs.com/api/plugins)功能极其强大，可以用来处理各种各样的任务。

###### htmlWebpackPlugin

该插件将为你生成一个 HTML5 文件， 其中包括使用 script 标签的 body 中的所有 webpack 包

为html文件中引入的外部资源如script、link动态添加每次compile后的hash，防止引用缓存的外部文件问题

可以生成创建html入口文件，比如单页面可以生成一个html文件入口，配置N个html-webpack-plugin可以生成N个页面入口

plugins: [

new htmlWebpackPlugin({

     filename: DIST\_PATH + `/${name}.html`,

     title: name + '页面',

     // template: path.resolve(\_\_dirname, '../public/index.html'),

     template: file,

     inject: true,    //打包后script标签位置 body（body底部） head（head里） true(默认值，html底部)

     hash: true,      //js文件是否添加hash字符串

     minify: false,    //html是否压缩

     chunks: [name, 'vender'],

   })

]

###### mini-css-extract-plugin

参考链接：<https://www.cnblogs.com/ysk123/p/9990082.html>

将CSS提取为独立的文件的插件，对每个包含css的js文件都会创建一个CSS文件，支持按需加载css和sourceMap

extract-text-webpack-plugin废弃使用，webpack4.0后推荐使用mini-css-extract-plugin

plugins: [

        new MiniCssExtractPlugin({

            filename: 'static/css/[name].[contenthash:5].css',

            // chunkFilename: "assets/css/[name].[hash:5].css",

        }),

],

module: {

rules: [

{

test: /\.css$/,

use: [

{

loader: MiniCssExtractPlugin.loader,

options: {

// 这里可以指定一个 publicPath

// 默认使用 webpackOptions.output中的publicPath

publicPath: '../'

},

},

'css-loader',

],

}

]

}

###### clean-webpack-plugin

删除webpack打包后的文件夹以及文件

注意的是：在最新版的webpack中 new CleanWebpackPlugin()；中不需要写里面的目标路径，会自动清除生成的文件夹，比如是build文件夹。

###### 模块热替换(hot module replacement)

不刷新页面，只刷新更改的模块

plugins: [

//热更新是显示该模块路径、文件名

        new webpack.NamedModulesPlugin(),

//热更新模块

        new webpack.HotModuleReplacementPlugin()

]，

devserver:{  
 hot:true

}

#### 扩展属性

##### source map

为了更容易地追踪错误和警告，JavaScript 提供了 [source map](http://blog.teamtreehouse.com/introduction-source-maps) 功能，将编译后的代码映射回原始源代码。如果一个错误来自于 b.js，source map 就会明确的告诉你。

module.exports = {

//开发环境

devtool: 'cheap-module-eval-source-map'

//生产环境

devtool: '#source-map',

}

##### 模式

通过选择 development 或 production 之中的一个，来设置 mode 参数，你可以启用相应模式下的 webpack 内置的优化

//两种方式

//webpack.config.js

module.exports = {

mode: 'production'

};

//package.json

{

"dev": "webpack-dev-server --mode=development --config",

"build": "webpack --mode=production --config"

}

##### Resolve

Webpack 在启动后会从配置的入口模块出发找出所有依赖的模块，Resolve 配置 Webpack 如何寻找模块所对应的文件。 Webpack 内置 JavaScript 模块化语法解析功能，默认会采用模块化标准里约定好的规则去寻找，但你也可以根据自己的需要修改默认的规则

resolve: {

//配置项通过别名来把原导入路径映射成一个新的导入路径。例如使用以下配置

alias: {

Utilities: path.resolve(\_\_dirname, 'src/utilities/'),

},

//在导入语句没带文件后缀时，Webpack 会自动带上后缀后去尝试访问文件是否存在

extensions: ['.wasm', '.mjs', '.js', '.json'],

// Webpack 去哪些目录下寻找第三方模块，默认是只去node\_modules目录下寻找。 有时你的项目里会有一些模块会大量被其它模块依赖和导入，由于其它模块的位置分布不定，针对不同的文件都要去计算被导入模块文件的相对路径， 这个路径有时候会很长，就像这样  import '../../../components/button'  这时你可以利用  modules  配置项优化，假如那些被大量导入的模块都在  ./src/components  目录下，把  modules  配置成

modules: ['./src/components', 'node\_modules']

}

##### Externals（CDN引入模块）

**index.html**

<script

src="https://code.jquery.com/jquery-3.1.0.js"

integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="

crossorigin="anonymous">

</script>

**webpack.config.js**

module.exports = {

//...

externals: {

jquery: 'jQuery'

}

};

这将使所有相关模块保持不变，即，下面显示的代码将仍然有效：

import $ from 'jquery';

$('.my-element').animate(/\* ... \*/);

Optimization（优化）

optimization = {

    splitChunks: {

        chunks: 'all',     //默认作用于异步chunk，值为all/initial/async/function(chunk),值为function时第一个参数为遍历所有入口chunk时的chunk模块，chunk.\_modules为chunk所有依赖的模块，通过chunk的名字和所有依赖模块的resource可以自由配置,会抽取所有满足条件chunk的公有模块，以及模块的所有依赖模块，包括css

        minSize: 30000,  //表示在压缩前的最小模块大小,默认值是30kb

        minChunks: 1,  // 表示被引用次数，默认为1；

        maxAsyncRequests: 5,  //所有异步请求不得超过5个

        maxInitialRequests: 3,  //初始话并行请求不得超过3个

        automaticNameDelimiter: '~',//名称分隔符，默认是~

        name: true,  //打包后的名称，默认是chunk的名字通过分隔符（默认是～）分隔

        cacheGroups: { //设置缓存组用来抽取满足不同规则的chunk,下面以生成common为例

            common: {

                name: 'common',  //抽取的chunk的名字

                chunks(chunk) { //同外层的参数配置，覆盖外层的chunks，以chunk为维度进行抽取

                },

                test(module, chunks) {  //可以为字符串，正则表达式，函数，以module为维度进行抽取，只要是满足条件的module都会被抽取到该common的chunk中，为函数时第一个参数是遍历到的每一个模块，第二个参数是每一个引用到该模块的chunks数组。自己尝试过程中发现不能提取出css，待进一步验证。

                },

                priority: 10,  //优先级，一个chunk很可能满足多个缓存组，会被抽取到优先级高的缓存组中

                minChunks: 2,  //最少被几个chunk引用

                reuseExistingChunk: true,//  如果该chunk中引用了已经被抽取的chunk，直接引用该chunk，不会重复打包代码

                enforce: true  // 如果cacheGroup中没有设置minSize，则据此判断是否使用上层的minSize，true：则使用0，false：使用上层minSize

            },

            //第三方库抽离

            vendor: {

                priority: 1, //权重

                test: /node\_modules/,

                chunks: 'all'

            },

styles: {

                name: 'styles',

                test: /\.(scss|css|less)$/,

                chunks: 'all',

                minChunks: 1,

                reuseExistingChunk: true,

                enforce: true

            }

        }

    },

minimizer: [

//压缩优化

        new UglifyJsPlugin({

            cache: true,

            parallel: true,

            sourceMap: true

        }),

        new OptimizeCSSAssetsPlugin({

            assetNameRegExp: /\.css$/g,

            cssProcessor: require('cssnano'),

            // cssProcessorOptions: cssnanoOptions,

            cssProcessorPluginOptions: {

                preset: ['default', {

                    discardComments: {

                        removeAll: true,

                    },

                    normalizeUnicode: false

                }]

            },

            canPrint: true

        }),

    ],

}

##### Optimization（优化）

###### splitChunks 抽取公共代码

<https://www.cnblogs.com/ygunoil/p/12419069.html>

<https://blog.csdn.net/zy444263/article/details/85303964>

<https://juejin.im/post/5c31c3c6e51d45522f565535>

optimization: {

        splitChunks: {

            chunks: 'all',

            minSize: 30000,

            minChunks: 1,

            maxAsyncRequests: 5,

            maxInitialRequests: 3,

            name: true,

            cacheGroups: {

                common: {

                    // name: "common",  // 指定公共模块 bundle 的名称

                    chunks: 'all',

                    minSize: 0, //大于0个字节,默认值30000，过小没有必要打包，优化不大

                    minChunks: 2, //抽离公共代码时，这个代码块最小被引用的次数

                    priority: -20,

                },

                // 第三方库抽离

                vendors: {

                    chunks: 'all',

                    priority: -10, //权重

                    test: /node\_modules/

                },

                // 主要是针对多入口，会产生多分样式文件，合并成一个样式文件，减少加载次数 配置如下

                styles: {

                    name: 'styles',

                    test: /\.(scss|css|less)$/,

                    chunks: 'all',

                    minChunks: 1,

                    reuseExistingChunk: true,

                    enforce: true

                }

            }

        },

}

###### 压缩优化

<https://my.oschina.net/itlangz/blog/2986976>

minimizer: [

            new UglifyJsPlugin({

                cache: true,

                parallel: true,

                sourceMap: true

            }),

            new OptimizeCSSAssetsPlugin({

                assetNameRegExp: /\.css$/g,

                cssProcessor: require('cssnano'),

                // cssProcessorOptions: cssnanoOptions,

                cssProcessorPluginOptions: {

                    preset: ['default', {

                        discardComments: {

                            removeAll: true,

                        },

                        normalizeUnicode: false

                    }]

                },

                canPrint: true

            }),

        ],

## Vue

#### 核心概念

数据驱动 组件系统

<template>

  <div>

    <!-- 数据绑定 v-text指令等价于{{}} -->

    <h1>值：{{num}}</h1>

    <!-- <h1 v-text="num"></h1> -->

    <!-- 绑定html -->

    <span v-html="rawHtml"></span>

    <!-- 双向绑定 -->

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

    <!-- 事件绑定：v-on指令简写@ v-on:click === @click

    修饰符.stop .prevent ....-->

    <button @click.stop.prevent="addNum">绑定事件</button>

    <br>

    <!-- 动态绑定标签属性attribute、传递proprs v-bind指令 === : -->

    <img src="@/assets/images/logo.png" />

  </div>

</template>

<script>

import imgUrl from "@/assets/images/logo.png";

export default {

  components: {

    comName: () => import("path") //懒加载组件

  },

  props: {

    propsName: {

      type: Number,

      default: 1

    }

  },

  // 为什么vue中data必须是一个函数

  // 组件是可复用的vue实例，一个组件被创建好之后，就可能被用在各个地方，而组件不管被复用了多少次，组件中的data数据都应该是相互隔离，互不影响的，基于这一理念，组件每复用一次，data数据就应该被复制一次，之后，当某一处复用的地方组件内data数据被改变时，其他复用地方组件的data数据不受影响

  data: () => ({

    num: 111,

    rawHtml: "<div>1233</div>",

    imageSrc: imgUrl

  }),

  computed: {

    dobuleNum() {

      return this.num \* 2;

    }

  },

  methods: {

    addNum() {

      this.num = 2;

    }

  },

  watch: {

    num(newVal, oldVal) {

      console.log(newVal);

    }

  }

};

</script>

<style>

</style>

#### 主要属性

#### vue 等单页面应用的优缺点

优点：

* 良好的交互体验
* 良好的前后端工作分离模式
* 减轻服务器压力

缺点：

* SEO难度较高
* 前进、后退管理
* 初次加载耗时多

#### 双向绑定原理

参考链接：<https://segmentfault.com/a/1190000006599500?utm_source=tag-newest>

##### 简述

采用**数据劫持**结合**发布者-订阅者模式**的方式，通过 **Object.defineProperty()** 来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调

##### 详细

采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty() 来劫持各个属性的 setter，getter，页面每次绑定属性，都会触发getter方法，通过dep添加一个订阅者（watcher）。

当属性值发生变化时，会触发setter方法，通知dep的所有订阅者触发更新（update）方法

第一步：需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter

这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化

第二步：compile解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

第三步：Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是:

1、在自身实例化时往属性订阅器(dep)里面添加自己

2、自身必须有一个update()方法

3、待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

第四步：MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果。

#### Vue.use( plugin )

// 参数：

// {Object | Function} plugin

// 用法：

// 安装 Vue.js 插件。如果插件是一个对象，必须提供 install 方法。如果插件是一个函数，它会被作为 install 方法。install 方法调用时，会将 Vue 作为参数传入。

// 该方法需要在调用 new Vue() 之前被调用。

// 当 install 方法被同一个插件多次调用，插件将只会被安装一次。

// Vue.use 会自动阻止多次注册相同插件，届时即使多次调用也只会注册一次该插件。

#### 路由（vue-router）

##### 改history模式

// Vue 如何去除url中的 #

// vue-router 默认使用 hash 模式，所以在路由加载的时候，项目中的 url 会自带 #。如果不想使用 #， 可以使用 vue-router 的另一种模式 history

new Router({

  mode: 'history',

  routes: [ ]

})

// 需要注意的是，当我们启用 history 模式的时候，由于我们的项目是一个单页面应用，所以在路由跳转的时候，就会出现访问不到静态资源而出现 404 的情况，这时候就需要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

##### $route和$router的区别

// 使用了 v-if 的时候，如果值为 false ，那么页面将不会有这个 html 标签生成。

// v-show 则是不管值为 true 还是 false ，html 元素都会存在，只是 CSS 中的 display 显示或隐藏

##### [vue路由传参的三种基本方式](https://segmentfault.com/a/1190000012393587)

参考链接：<https://segmentfault.com/a/1190000012393587>

###### 动态路由

getDescribe(id) {

  //   直接调用$router.push 实现携带参数的跳转

  this.$router.push({

    path: `/describe/${id}`,

  })

<router-link to="/name/123">User</router-link>

  // 方案一，需要对应路由配置如下：

  {

    path: '/describe/:id',

      name: 'Describe',

        component: Describe

  }

  // 很显然，需要在path中添加/:id来对应 $router.push 中path携带的参数。在子组件中可以使用来获取传递的参数值。

  this.$route.params.id

###### params

{/\* 父组件中：通过路由属性中的name来确定匹配的路由，通过params来传递参数。 \*/}

this.$router.push({

   name: 'Describe',

   params: {

     id: id

   }

 })

<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>

{/\* 对应路由配置: 这里可以添加:/id 也可以不添加，不添加数据会在url后面显示，不添加数据就不会显示 \*/}

{

path: '/describe',

name: 'Describe',

component: Describe

}

{/\* 子组件中: 这样来获取参数 \*/}

this.$route.params.id

###### query

{/\* 父组件：使用path来匹配路由，然后通过query来传递参数 \*/ }

  {/\* 这种情况下 query传递的参数会显示在url后面?id=？ \*/ }

  this.$router.push({

    path: '/describe',

    query: {

      id: id

    }

  })

  this.$router.push('/describe?aaa=1&bbb=2')

  <router-link :to="{ path: 'register', query: { plan: 'private' }}"

  <router-link to="/describe?aaa=1&bbb=2"

  {/\* 对应路由配置： \*/ }

  {

    path: '/describe',

      name: 'Describe',

        component: Describe

  }

  {/\* 对应子组件: 这样来获取参数 \*/ }

  this.$route.query.id

##### **插槽**

**匿名插槽（默认插槽）**

// 父组件

<template>

  <div class="father">

    <h3>这里是父组件</h3>

    <child>

      <div class="tmpl">

        <span>菜单1</span>

      </div>

    </child>

  </div>

</template>

// 子组件

<template>

    <div class="child">

        <h3>这里是子组件</h3>

        <slot></slot>

    </div>

</template>

**具名插槽**

<template>

  <div class="father">

    <h3>这里是父组件</h3>

<child>

//缩写

//<div class="tmpl" #up>

      <div class="tmpl" v-slot:"up">

        <span>菜单1</span>

      </div>

      <div class="tmpl" v-slot:"down">

        <span>菜单-1</span>

      </div>

      <div class="tmpl">

        <span>菜单->1</span>

      </div>

    </child>

  </div>

</template>

// 子组件

<template>

  <div class="child">

    {/\* // 具名插槽 \*/}

    <slot name="up"></slot>

    <h3>这里是子组件</h3>

    {/\* // 具名插槽 \*/}

    <slot name="down"></slot>

    {/\* // 匿名插槽 \*/}

    <slot></slot>

  </div>

</template>

**作用域插槽**

<slot name="todo" :user="user"></slot>

 export default {

    data: function(){

      return {

        data: ['zhangsan','lisi','wanwu','zhaoliu','tianqi','xiaoba']

      }

    },

}

// <template #todo="slotProps">

<template v-slot:todo="slotProps">

  {{slotProps.user.age}}

  {{slotProps.user.lastname}}

</template>

##### 路由权限控制

###### 通过vue-router addRoutes 方法注入路由实现控制

参考链接：

**router.matcher:**https://blog.csdn.net/qq\_42991509/article/details/91971948

实现方式：<https://www.cnblogs.com/zhengrunlin/p/8981017.html>

<https://www.cnblogs.com/xianxiang886/p/12101665.html>

<https://blog.csdn.net/suolong914/article/details/89432563>

通过请求服务端获取当前用户路由配置，编码为 vue-router 所支持的基本格式（具体如何编码取决于前后端协商好的数据格式），通过调用 this.$router.addRoutes 方法将编码好的用户路由注入到现有的 vue-router 实例中去，以实现用户路由。

1. 路由表存在本地，通过角色权限筛选

import Vue from 'vue'

import Router from 'vue-router'

Vue.use(Router)

const createRouter = () => new Router({

  mode: 'history',

  routes: []

})

const router = createRouter()

export function resetRouter() {

  const newRouter = createRouter()

  router.matcher = newRouter.matcher  // 新路由实例matcer，赋值给旧路由实例的matcher，（相当于replaceRouter）

}

export default router

// permission.js

import router from './router'

import store from './store'

import { Message } from 'element-ui'

import { getToken } from '@/utils/auth' // 验权

const whiteList = ['/login', '/authredirect'] // 不重定向白名单

// 待筛选路由表

const routeList = [

  {

    path: '/permission',

    // name: 'permissionhome',

    meta: { //存放一些自定义内容

      title: 'permission',

      icon: 'el-icon-setting',

      roles: ['admin']  //能访问的角色列表

    },

    component: () => import(['../components/common/Home.vue']),

    children: [

      {

        name: 'permission',

        path: '/permission',

        meta: {

          title: 'permission', icon: 'el-icon-menu', roles: ['admin']

        },

        component: resolve => require(['../components/page/permission.vue'], resolve)

      }

    ]

  },

  { path: '\*', redirect: '/404', hidden: true }

]

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

  if (getToken()) { // 判断是否有token,有token表示已经登录过

    if (to.path === '/login') {   //已登录状态不允许访问登录页

      next({ path: '/' })

    } else {

      if (store.getters.roles.length === 0) { //判断vuex里是否有用户角色信息

        console.log('roles====0')

        store.dispatch('GetInfo').then(res => { // 拉取用户信息

          const roles = res.data.roles // note: roles must be a array! such as: ['editor','develop']

          console.log('roles?', roles)

          store.dispatch('GenerateRoutes', { roles }).then(() => { // 根据roles权限筛选出可访问的路由表，并添加

            console.log('addrouters', store.getters.addRouters)

            resetRouter() //重置路由表

            router.addRoutes(store.getters.addRouters) // 动态添加可访问路由表

            next({ ...to, replace: true }) // hack方法 确保addRoutes已完成 ,set the replace: true so the navigation will not leave a history record

          })

        }).catch(() => {

          store.dispatch('FedLogOut').then(() => {

            Message.error('验证失败,请重新登录')

            next({ path: '/login' })

          })

        })

      } else {  //有角色信息

        console.log('====1')

        next() // 当有用户权限的时候，说明所有可访问路由已生成 如访问没权限的全面会自动进入404页面

      }

    }

  } else {  //未登录逻辑

    if (whiteList.indexOf(to.path) !== -1) {

      next()

    } else {

      next('/login')

    }

  }

})

1. 通过服务器获取路由表，然后本地根据路径挂载组件对象并添加路由表

import Vue from 'vue'

import Router from 'vue-router'

Vue.use(Router)

const createRouter = () => new Router({

  mode: 'history',

  routes: []

})

const router = createRouter()

export function resetRouter() {

  const newRouter = createRouter()

  router.matcher = newRouter.matcher  // 新路由实例matcer，赋值给旧路由实例的matcher，（相当于replaceRouter）

}

export default router

// permission.js

import router from './router'

import store from './store'

const whiteList = ['/login', '/authredirect'] // 不重定向白名单

// 不需要权限的公用路由表

const routeList = [

  {

    path: '/login',

    component: () => import(['../components/common/login.vue']),

  },

  { path: '\*', redirect: '/404', hidden: true }

]

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

  if (getToken()) { // 判断是否有token,有token表示已经登录过

    if (to.path === '/login') {   //已登录状态不允许访问登录页

      next({ path: '/' })

    } else {

      if (store.getters.roles.length === 0) { //判断vuex里是否有用户角色信息

        console.log('roles====0')

        store.dispatch('GetList').then(res => { // 拉取路由列表

          const routeList = res.data.routeList // note: roles must be a array! such as: ['editor','develop']

          const resultList = routeList.map(e=>{

            // e.component为路径信息

            e.component = ()=>import(e.component)

          })

          resetRouter() //重置路由表

          router.addRoutes(resultList) // 动态添加可访问路由表

        }).catch(() => {

          store.dispatch('FedLogOut').then(() => {

            Message.error('验证失败,请重新登录')

            next({ path: '/login' })

          })

        })

      } else {  //有角色信息

        console.log('====1')

        next() // 当有用户权限的时候，说明所有可访问路由已生成 如访问没权限的全面会自动进入404页面

      }

    }

  } else {  //未登录逻辑

    if (whiteList.indexOf(to.path) !== -1) {

      next()

    } else {

      next('/login')

    }

  }

})

###### 通过vue-router beforeEach 钩子限制路由跳转

import Vue from 'vue'

import Router from 'vue-router'

Vue.use(Router)

export default new Router({

  routes: [

    {

      path: '/',

      name: 'HelloWorld',

      component: () => import('@/components/HelloWorld'),

      meta: {

        roles: ['admin']

      }

    }

  ]

})

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

  // 后台获取的用户信息的角色

  const USER\_ROLE = ['admin', 'aaa', 'bbb']

  // ---------------------------------------

  const isAllow = USER\_ROLE.some(e => to.meta.roles.includes(e))

  if (isAllow) {

    next()

  } else {

    if (to.path === '/error') {

      next()

    } else {

      next('/error')

    }

  }

})

#### Vuex

// Vuex是通过全局注入store对象，来实现组件间的状态共享

// 使用vue.use安装插件

// Vue.use(Vuex); // vue的插件机制,安装vuex插件

// 当ues(Vuex)时候，会调用vuex中的install方法，装在vuex!

// 下面时install的核心源码：

Vue.mixin({

    beforeCreate() {

        if (this.$options && this.$options.store) {

            //找到根组件 main 上面挂一个$store

            this.$store = this.$options.store

            // console.log(this.$store);

        } else {

            //非根组件指向其父组件的$store

            this.$store = this.$parent && this.$parent.$store

        }

    }

})

// store注入 vue的实例组件的方式，是通过vue的 mixin机制，借助vue组件的生命周期 钩子 beforeCreate 完成的。即 每个vue组件实例化过程中，会在 beforeCreate 钩子前调用 vuexInit 方法。

单模块

// 首先在 src 中新建一个 store.js 文件

import Vue from 'vue';

import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({

    state: {

        name: 'Lucy',

    },

    mutations: {

        setName(state, newName) {

            state.name = newName;

        }

    },

    actions: {

        modifyName({commit}, newName) {

            commit('setName', newName);

        }

    }

});

// 然后，在 main.js 中导入 store，并全局注入:

import store from './store';

// ...

new Vue({

    store,

    render: h => h(App),

}).$mount('#app')

// 最后在页面中使用

<script>

    import {mapState, mapActions} from 'vuex';

    export default {

        computed: {

            ...mapState(['name'])

        },

        methods: {

            ...mapActions(['modifyName']),

            modifyNameAction() {

                this.modifyName('bighone');

            }

        },

    }

</script>

分模块（module）

// 新建一个 moduleA.js、moduleB.js 文件，并编写如下代码：

// 模块内部的 action、mutation 和 getter 默认是注册在全局命名空间的。如果我们只想让他们在当前的模块中生效,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。

export default {

  namespaced: true,

  state: {

      text: 'moduleA'

  },

  getters: {},

  mutations: {},

  actions: {}

}

export default {

  namespaced: true,

  state: {

      text: 'moduleB'

  },

  getters: {},

  mutations: {},

  actions: {}

}

// 然后打开 store.js 文件，导入这两个 module ：

import moduleA from './module/moduleA';

import moduleB from './module/moduleB';

export default new Vuex.Store({

    modules: {

        moduleA, moduleB,

    },

    // ...

}

// 页面引用

<script>

    import {mapActions, mapGetters} from 'vuex';

    export default {

        computed: {

            // 两种方式

            // ...mapGetters({

            //     name: 'moduleA/detail'

            // }),

            ...mapGetter('moduleA',['detail'])

        },

        methods: {

            ...mapActions({

                call: 'moduleA/callAction'

            }),

            modifyNameAction() {

                this.call();

            }

        },

    }

</script>

#### 生命周期

**创建前/后：** 在beforeCreated阶段，vue实例的挂载元素$el和数据对象data都为undefined，还未初始化。在created阶段，vue实例的数据对象data有了，$el还没有。

**载入前/后：**在beforeMount阶段，vue实例的$el和data都初始化了，但还是挂载之前为虚拟的dom节点，data.message还未替换。在mounted阶段，vue实例挂载完成，data.message成功渲染。

**更新前/后：**当data变化时，会触发beforeUpdate和updated方法。

**销毁前/后：**在执行destroy方法后，对data的改变不会再触发周期函数，说明此时vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在

//vue实例有一个完整的生命周期，生命周期也就是指一个实例从开始创建到销毁的这个过程

beforeCreated() //在实例创建之间执行，数据未加载状态

created() //在实例创建、数据加载后，能初始化数据，dom渲染之前执行

beforeMount() //虚拟dom已创建完成，在数据渲染前最后一次更改数据

mounted() //页面、数据渲染完成，真实dom挂载完成

beforeUpadate() //重新渲染之前触发

两个特殊的生命周期，只有使用keep-alive时才有

updated() //数据已经更改完成，dom 也重新 render 完成,更改数据会陷入死循环

activated() //被 keep-alive 缓存的组件再次访问时调用。该钩子在服务器端渲染期间不被调用。

deactivated() //被 keep-alive 缓存的组件离开时调用。该钩子在服务器端渲染期间不被调用。

beforeDestory() //销毁前执行（实例仍然完全可用）

destoryed() //销毁后执行

#### 解释单向数据流和双向数据绑定

**单向数据流：** 顾名思义，数据流是单向的。数据流动方向可以跟踪，流动单一，追查问题的时候可以更快捷。缺点就是写起来不太方便。要使UI发生变更就必须创建各种 action 来维护对应的 state

**双向数据绑定：**数据之间是相通的，将数据变更的操作隐藏在框架内部。优点是在表单交互较多的场景下，会简化大量与业务无关的代码。缺点就是无法追踪局部状态的变化，增加了出错时 debug 的难度

#### 组件通信方式

参考链接：<https://segmentfault.com/a/1190000019208626>

##### 父传子 props

##### 子传父 $emit

this.$emit("doSomething ","子向父组件传值")

<my-component v-on:my-event="doSomething"></my-component>

//事件绑定简洁写法

<my-component @my-event="doSomething"></my-component>

##### 兄弟组件 $emit/$on

// 创立中间组件 event.js

var Event=new Vue();

//a组件

Event.$emit(事件名,数据);

//b组件

Event.$on(事件名,data => {});

##### Vuex

// 新建一个 moduleA.js、moduleB.js 文件，并编写如下代码：

// 模块内部的 action、mutation 和 getter 默认是注册在全局命名空间的。如果我们只想让他们在当前的模块中生效,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。

export default {

  namespaced: true,

  state: {

    text: 'moduleA'

  },

  // 对于模块内部的 mutation 和 getter，接收的第一个参数是模块的局部状态对象

  getters: {

    // 注意:rootState必须是第三个参数

    detail(state, getters, rootState) {

      console.log(getters)

      return state.text + '-' + rootState.name;

    }

  },

  mutations: {

    // 接受 state 作为第一个参数，第二参数开始为自定义传入参数

    setText(state,param){

      state.text = param

    }

  },

  // 对于模块内部的 action，局部状态通过 context.state 暴露出来，根节点状态则为 context.rootState：

  actions: {

    aaa({ state, rootState,commit }) {

      alert(state.text + '-' + rootState.name);

    },

    // 在带命名空间的模块内注册全局 action。

    callAction: {

      root: true,

      handler(namespacedContext, payload) {

        let { state, commit } = namespacedContext;

        commit('setText');

        alert(state.text);

      }

    }

  }

}

export default {

  namespaced: true,

  state: {

    text: 'moduleB'

  },

  getters: {},

  mutations: {},

  actions: {}

}

// 然后打开 store.js 文件，导入这两个 module ：

import moduleA from './module/moduleA';

import moduleB from './module/moduleB';

export default new Vuex.Store({

  modules: {

    moduleA, moduleB,

  },

  // ...

}

  // 页面引用

  < script >

    import { mapActions, mapGetters } from 'vuex';

export default {

  computed: {

    // 两种方式

    // ...mapGetters({

    //     name: 'moduleA/detail'

    // }),

    ...mapGetter('moduleA', ['detail'])

  },

  methods: {

    ...mapActions({

      call: 'moduleA/callAction'

    }),

    modifyNameAction() {

      this.call();

    }

  },

}

</script>

##### localStorage/sessionStoage

##### $attrs/$listeners

$attrs：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 interitAttrs 选项一起使用。

$listeners：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件

  // 假设有父组件Parent和子组件Child

  // Parent

  <template template >

    <child

      v-on: event-one="methodOne"

      v-on: event-two="methodTwo"

      :foo="foo"

      :boo="boo"

      :coo="coo"

      :doo="doo"

      title="前端工匠"

    />

  </template>

  // 那么你在使用Child时，传入的所有v-on事件都可以在$listeners对象中找到。

  // Child

  created() {

  console.log(this.$listeners) // { 'event-one': f(), 'event-two': f() }

  console.log(this.$attrs)

}

##### provide/inject

Vue2.2.0新增API,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。一言而蔽之：祖先组件中通过provider来提供变量，然后在子孙组件中通过inject来注入变量。

provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。

// A.vue

export default {

  provide: {

    name: '浪里行舟'

  }

}

// B.vue

export default {

  inject: ['name'],

  mounted () {

    console.log(this.name);  // 浪里行舟

  }

}

##### $parent / $children

this.$children.\*/this.$parent.\*

##### ref

export default {

  data () {

    return {

      title: 'Vue.js'

    }

  },

  methods: {

    sayHello () {

      window.alert('Hello');

    }

  }

}

// 父组件

<template>

  <component-a ref="comA"></component-a>

</template>

<script>

  export default {

    mounted () {

      const comA = this.$refs.comA;

      console.log(comA.title);  // Vue.js

      comA.sayHello();  // 弹窗

    }

  }

</script>

#### 过滤器

// 全局注册

  Vue.filter('reverse', function (value) {

    return value.split('').reverse().join('')

  })

  // 局部注册

  filters: {

    reverse: function (value) {

      return value.split('').reverse().join('')

    }

  }

  // 效果：'abc' => 'cba'

<template>

<div>

  <span v-text="message | reverse"></span>

  <span>{{message | reverse}}</span>

</div>

</template>

#### 自定义指令

// 注册一个全局自定义指令 `v-focus`

Vue.directive('focus', {

  // 当被绑定的元素插入到 DOM 中时……

  inserted: function (el) {

    // 聚焦元素

    el.focus()

  }

})

// 局部注册

directives: {

  focus: {

    // 指令的定义

    inserted: function (el) {

      el.focus()

    }

  }

}

#### Computed和methods

// 计算属性computed 和事件 methods 有什么区别

// 我们可以将同一函数定义为一个 method 或者一个计算属性。对于最终的结果，两种方式是相同的

// 不同点：

// computed: 计算属性是基于它们的依赖进行缓存的,只有在它的相关依赖发生改变时才会重新求值，不管绑定多少次都只会计算一次

// 对于 method ，只要发生重新渲染，method 调用总会执行该函数，而且页面绑定了多少次就会触发多少次

#### keep-alive

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染，离开当前路由不会被销毁，再次进入会触发

activated() //被 keep-alive 缓存的组件再次访问时调用。该钩子在服务器端渲染期间不被调用。

deactivated() //被 keep-alive 缓存的组件离开时调用。该钩子在服务器端渲染期间不被调用。

#### Key

key 的特殊属性主要用在 Vue 的虚拟 DOM 算法，在新旧 nodes 对比时辨识 VNodes。如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试修复/再利用相同类型元素的算法。使用 key，它会基于 key 的变化重新排列元素顺序，并且会移除 key 不存在的元素。

有相同父元素的子元素必须有独特的 key。重复的 key 会造成渲染错误

// 最常见的用例是结合 v-for：

<ul>

  <li v-for="item in items" :key="item.id">...</li>

</ul>

// 它也可以用于强制替换元素/组件而不是重复使用它。当你遇到如下场景时它可能会很有用：

// 完整地触发组件的生命周期钩子

// 触发过渡

// 例如：

<transition>

  <span :key="text">{{ text }}</span>

</transition>

// 当 text 发生改变时，<span> 总是会被替换而不是被修改，因此会触发过渡。

#### 常用Api

## React

#### 生命周期

componentWillMount：组件即将被装载、渲染到页面上

render:组件在这里生成虚拟的 DOM 节点

componentDidMount:组件真正在被装载之后

运行中状态：

componentWillReceiveProps: props发生变化时执行，初始化render时不执行

shouldComponentUpdate:组件接受到新属性或者新状态的时候（可以返回 false，接收数据后不更新，阻止 render 调用，后面的函数不会被继续执行了）

componentWillUpdate:组件即将更新不能修改属性和状态

render:组件重新描绘

componentDidUpdate:组件已经更新

销毁阶段：

componentWillUnmount:组件即将销毁

#### **React 中有三种构建组件的方式**

React.createClass()、ES6 class 和无状态函数

#### 为什么虚拟 dom 会提高性能?(必考)

虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能。

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

#### react diff 原理（常考，大厂必考）

把树形结构按照层级分解，只比较同级元素。

给列表结构的每个单元添加唯一的 key 属性，方便比较。

React 只会匹配相同 class 的 component（这里面的 class 指的是组件的名字）

合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.

选择性子树渲染。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能。

#### **类组件和函数式组件（无状态组件）之间有何不同**

* 类组件不仅允许你使用更多额外的功能，如组件自身的状态和生命周期钩子，也能使组件直接访问 store 并维持状态

// ES6 的 class 定义组件：

class Welcome extends React.Component {

  render() {

    return <h1>Hello, {this.props.name}</h1>;

  }

}

* 当组件仅是接收 props，并将组件自身渲染到页面时，该组件就是一个 '无状态组件(stateless component)'，可以使用一个纯函数来创建这样的组件。这种组件也被称为哑组件(dumb components)或展示组件

// 定义组件最简单的方式就是编写 JavaScript 函数：

// 该函数是一个有效的 React 组件，因为它接收唯一带有数据的 “props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。

function Welcome(props) {

  return <h1>Hello, {props.name}</h1>;

}

#### **(组件的)状态(state)和属性(props)之间有何不同**

· props是指组件间传递的一种方式，props自然也可以传递state。由于React的数据流是自上而下的，所以是从父组件向子组件进行传递；另外组件内部的this.props属性是只读的不可修改！

· state是组件内部的状态（数据），不能够直接修改，必须要通过setState来改变值的状态，从而达到更新组件内部数据的作用。

#### ref

Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄。

##### 字符串（废弃）

class MyComponent extends React.Component {

  componentDidMount() {

    this.refs.myRef.focus();

  }

  render() {

    return <input ref="myRef" />;

  }

}

##### 回调函数

我们可以为元素添加 ref 属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回：

Class组件

class CustomForm extends Component {

    handleSubmit = () => {

      console.log("Input Value: ", this.input.value)

    }

    render () {

      return (

        <form onSubmit={this.handleSubmit}>

          <input

            type='text'

            ref={(input) => this.input = input} />

          <button type='submit'>Submit</button>

        </form>

      )

    }

  }

##### React.createRef()

class MyComponent extends React.Component {

  constructor(props) {

    super(props);

    this.myRef = React.createRef();

  }

  componentDidMount() {

    // this.myRef是一个对象，current属性才是ref对象

    this.myRef.current.focus();

  }

  render() {

    return <input ref={this.myRef} />;

  }

}

##### React.forwardRef （ref转发）

会创建一个React组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中

函数组件（React.forwardRef）

const FancyButton = React.forwardRef((props, ref) => (

    <button ref={ref} className="FancyButton">

        {props.children}

    </button>

));

// 你可以直接获取 DOM button 的 ref：

const ref = React.createRef();

<FancyButton ref={ref}>Click me!</FancyButton>;

高阶组件

// 高阶组件

import React from 'react'

function withA(Component, newProps) {

    class MidComponent extends React.Component {

        componentDidMount() {

            // console.log(this.props, newProps)

        }

        render() {

            const { forwardedRef, ...rest } = this.props;

            return (

                <div>

                    <h1>12113</h1>

                    {/\* 指向被包装的组件 \*/}

                    <Component ref={forwardedRef} {...rest} />

                </div>

            )

        }

    }

    // ref转发

    return React.forwardRef((props, ref) => {

        return <MidComponent {...props} forwardedRef={ref} />

    });

}

export default withA

// 子组件

import React, { Component } from 'react'

import withA from './hightCom.jsx'

class header extends Component {

  render() {

    return (

      <div>

        12312331

      </div>

    )

  }

}

//  const Header = React.forwardRef((props, ref) => (

//   <header></header>

// ))

export default withA(header, () => ({ aaa: 123 }))

// 父组件

import React, { Component } from 'react';

import Header from './component/header'

import "antd/dist/antd.css";

import './index'

export default class Root extends Component {

  constructor(props) {

    super(props)

    this.forwardedRef = React.createRef()

    this.inputRef = React.createRef()

  }

  componentDidMount() {

    console.log(111, this.forwardedRef)

    console.log(222, this.inputRef)

  }

  render() {

    return (

      <div style={{ textAlign: 'center' }}>

        <h1>只更新这11里</h1>

        {/\* <Header ref={(ref)=>this.forwardedRef = ref}></Header> \*/}

        <Header ref={this.forwardedRef}></Header>

        <input type="text" ref={this.inputRef} />

      </div>);

  }

};

#### React.Fragment

// React.Fragment 组件能够在不额外创建 DOM 元素的情况下，让 render() 方法中返回多个元素。

render() {

  return (

    <React.Fragment>

      Some text.

      <h2>A heading</h2>

    </React.Fragment>

  );

}

#### React事件绑定

由于类的方法默认不会绑定this，因此在调用的时候如果忘记绑定，this的值将会是undefined。

通常如果不是直接调用，应该为方法绑定this。绑定方式有以下几种：

**只要是需要在调用的地方传参，就必须在事件绑定的地方使用bind或者箭头函数**

##### 在构造函数中使用bind绑定this

class Button extends React.Component {

  constructor(props) {

      super(props);

      // 使用bind绑定this为当前组件对象

      this.handleClick = this.handleClick.bind(this);

    }

    handleClick(){

      console.log('this is:', this);

    }

    render() {

      return (

        <button onClick={this.handleClick}>

          Click me

        </button>

      );

    }

  }

##### 在调用的时候使用bind绑定this

class Button extends React.Component {

  handleClick() {

    console.log("this is:", this);

  }

  render() {

    return <button onClick={this.handleClick.bind(this,参数)}>Click me</button>;

  }

}

##### 在调用的时候使用箭头函数绑定this

class Button extends React.Component {

  handleClick() {

    console.log("this is:", this);

  }

  render() {

    return <button onClick={() => this.handleClick(参数)}>Click me</button>;

  }

}

##### 使用属性初始化器语法绑定this(实验性)

class Button extends React.Component {

  handleClick = () => {

    console.log("this is:", this);

  };

  render() {

    return <button onClick={this.handleClick}>Click me</button>;

  }

}

#### **受控组间和非受控组件**

总结：表单组件的值是否由组件控制（是否绑定state，由setState设值）

##### 受控组件

对于受控组件来说，输入的值始终由 React 的 state 驱动

class NameForm extends React.Component {

  state = {

    value: 1231

  }

  setval = (e) => {

    this.setState({

      value: e.target.value

    })

  };

  render() {

    return (

      <div>

        <input type="text" value={this.state.value} onChange={this.setval} />

        <button onClick={this.getval}>获取值</button>

      </div>

    );

  }

}

##### 非受控组件

非受控组件将真实数据储存在 DOM 节点中，可以使用ref获取

class NameForm extends React.Component {

  getval = () => {

    console.log(this.input.value)

  };

  render() {

    return (

      <div>

        <input type="text" ref={input => this.input = input} />

        <button onClick={this.getval}>获取值</button>

      </div>

    );

  }

}

#### **Key**

key 帮助 React 识别哪些元素改变了，比如被添加或删除。因此你应当给数组中的每一个元素赋予一个确定的标识。

**给组件添加key**

key值相同：组件属性发生变化，React只更新对应组件的属性

key值不同：React先销毁原来的组件，然后在重新创建新的组件

尽量不要用数组的index去作为key

之所以这里说的是尽量，是因为如果你List中的子组件如果只是负责纯展示的话，（不涉及数组的动态变更），那么是可以采用index作为key的。

但如果组件设计到数组的动态变更，例如数组新增元素、删除元素或者重新排序等，这时index作为key会导致展示错误的数据。

#### **setState**

##### 异步同步问题

setState 只在合成事件和钩子函数中是“异步”的，在原生事件和 setTimeout 中都是同步的

合成事件：就是react 在组件中的onClick等都是属于它自定义的合成事件

原生事件：比如通过addeventListener添加的，dom中的原生事件

// setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

// setState 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次 setState ， setState 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 setState 多个不同的值，在更新时会对其进行合并批量更新。

state = { val: 0 }

batchUpdates = () => {

  this.setState({ val: this.state.val + 1 })

  this.setState({ val: this.state.val + 1 })

  this.setState({ val: this.state.val + 1 })

}

##### 为什么建议传递给 setState 的参数是一个 callback 而不是一个对象

因为 this.props 和 this.state 的更新可能是异步的，不能依赖它们的值去计算下一个 state。

state = { val: 0 }

// 最后的值为1

batchUpdates = () => {

  this.setState({ val: this.state.val + 1 })

  this.setState({ val: this.state.val + 1 })

  this.setState({ val: this.state.val + 1 })

}

// 给 setState 传递一个函数，而不是一个对象，就可以确保每次的调用都是使用最新版的 state

// 因为 setState 的调用是分批的，所以你可以链式地进行更新，并确保它们是一个建立在另一个之上的，这样才不会发生冲突：

batchUpdates = () => {

  this.setState(state => {

    return {

      val: state.val + 1

    }

  }, () => {

    // 因为 setState 的调用是分批的，所以你可以链式地进行更新，并确保它们是一个建立在另一个之上的，这样才不会发生冲突：

    // 在这里能获取到最新值:3

    console.log(111, this.state.value)

  })

  this.setState(state => {

    return {

      val: state.val + 1

    }

  })

  this.setState(state => {

    return {

      val: state.val + 1

    }

  })

  // 如果你现在在这里读取 `this.state.count`，它还是会为 0。

  // 但是，当 React 重新渲染该组件时，它会变为 3

  console.log(this.state.value) //1

}

##### 给 setState 传递一个对象与传递一个函数的区别是什么？

传递一个函数可以让你在函数内访问到当前的 state 的值

#### **(在构造函数中)调用 super(props) 的目的是什么**

在 super() 被调用之前，子类是不能使用 this 的，在 ES2015 中，子类必须在 constructor 中调用 super()。传递 props 给 super() 的原因则是便于(在子类中)能在 constructor 访问 this.props。

#### **何为纯函数(pure function)**

一个纯函数是一个不依赖于且不改变其作用域之外的变量状态的函数，这也意味着一个纯函数对于同样的参数总是返回同样的结果

#### **react-router与react-router-dom**

##### 注意要点

###### 404页面

<Route component={NoMatch}/>

// 路由指定的所有路径没有匹配时，就会加载这个NoMatch组件，也就是404页面

###### 重定向

// 3.路由重定向与404

// 重定向：

<Route exact path="/" render={() => (<Redirect to="/home"/>)}/>

###### <Switch>标签

**有<Switch>标签，则其中的<Route>在路径相同的情况下，只匹配第一个，这个可以避免重复匹配；**

###### 路由嵌套

* 1. 在component组件内部需要嵌套的位置直接嵌套Route标签

// 这个方法会使得路由标签比较分散, 子组件我们必须直接将Route标签写入到父组件之中，而且路由必须包含根路径.

// Dashboard.jsx

import Menu from '~/components/Menu/Menu';

import React from 'react';

import { Route, Switch } from "react-router-dom";

import Index from '~/container/Index/Index';

import TaskList from '~/container/TaskManage/TaskList/TaskList'

const routes = [

  {

    path: "/Dashboard/Index",

    component: Index

  },

  {

    path: "/Dashboard/TaskManage/TaskList",

    component: TaskList

  }

];

class Dashboard extends React.Component {

  constructor(props, context) {

    super(props, context);

  }

  render() {

    return (

      <div>

        <Menu />

        <div className='container'>

          <div>

            {routes.map((route, index) => (

              <Route

                exact

                key={index}

                path={route.path}

                component={route.component}

              />

            ))}

          </div>

        </div>

      </div>

    )

  }

}

export default Dashboard

* 1. 使用Route render渲染作内联嵌套

  // component使用this.props.children进行嵌套渲染,Dashboard为父组件,Index和TaskList为子组件

  // router.js

  < Router history = { history } >

    <Switch>

      <Route exact path="/" component={Login} />

      <Route path="/" render={({ history, location, match }) => (

        <Dashboard history={history} location={location} match={location}>

          <Route path="/Dashboard/Index" component={Index}></Route>

          <Route path="/Dashboard/TaskManage/TaskManageIng" component={TaskManageIng}></Route>

          <Route path="/Dashboard/TaskManage/TaskList" component={TaskList}></Route>

          <Route path="/Dashboard/TaskManage/TaskResource" component={TaskResource}></Route>

        </Dashboard>

      )} />

    </Switch>

            </Router >

// Dashboard.jsx

import Menu from '~/components/Menu/Menu';

import React from 'react';

class Dashboard extends React.Component {

  constructor(props, context) {

    super(props, context);

  }

  render() {

    return (

      <div>

        <Menu />

        <div className='container'>

          <div>

            {this.props.children}

          </div>

        </div>

      </div>

    )

  }

}

export default Dashboard

###### 动态路由

// \*\*设置路由\*\*

<Route path="/about/:id" component={About} />

// \*\*访问方式\*\*

<Link to={'/about/' + '2' }>About</Link>

// 或者

this.props.history.push('/about/'+'2')

// \*\*about页面获取参数\*\*

componentDidMount(){

    console.log(this.props.match.params.id);

}

###### 路有传参

1. **Params**

// 方法一：通过params

// 优点:简单快捷,并且，在刷新页面的时候，参数不会丢失。

// 缺点:只能传字符串，并且，如果传的值太多的话，url会变得长而丑陋。

// 如果，你想传对象的话，可以用JSON.stringify(),想将其转为字符串，然后另外的页面接收后，用JSON.parse()转回去。

// \*\*设置路由\*\*

<Route path="/about/:id" component={About} />

// \*\*访问方式\*\*

<Link to={'/about/' + '2' }>About</Link>

// 或者

this.props.history.push('/about/'+'2')

// \*\*about页面获取参数\*\*

componentDidMount(){

    console.log(this.props.match.params.id);

}

1. **Query**

// 方法二：通过query

// 优点：优雅，可传对象

// 缺点：刷新页面，参数丢失

// \*\*访问方式\*\*

<Link to={{pathname:'/contact',query:{id:'456'}}}>contact</Link>

// 或者

this.props.history.push({pathname :'/contact',query :{id:'456'}})

// \*\*contact页面获取参数\*\*

componentDidMount(){

   console.log(this.props.location.query.id);

}

1. **State**

// 方法三：通过state

// 优点：优雅，可传对象

// 缺点：刷新页面，参数丢失

// \*\*访问方式\*\*

<Link to={{pathname:'/contact',state:{id:'456'}}}>contact</Link>或者

this.props.history.push({pathname :'/contact',state:{id:'456'}})

// \*\*contact页面获取参数\*\*

componentDidMount(){

   console.log(this.props.location.state.id);

}

1. **url字符串拼接**

// 方法四：url字符串拼接

// 备注：如果想要http://localhost:3000/#/contact?sort=name&type=1这种形式传参

<Link to={`/ProductDetail?aid=${value.aid}`}>{value.title}</Link>

this.props.history.push({pathname :'/contact',search:'?sort=name&type=1',query :{id:'456'}})

// \*\*contact页面获取参数\*\*

 this.props.location.search

// this.props.location.search取参数，然后在进行解析

###### 权限控制

const isAuthenticated = false;  //判断登录

function PrivateRoute({ component: Component, ...rest }) {

  return (

    <Route {...rest} render={

      props => isAuthenticated ? (<Component {...props} />) : (<Redirect to={{ pathname: "/home" }} />)

    } />

  );

}

export default class router extends Component {

  render() {

    return (

      <HashRouter>

        <App>

          <Switch>

            <Route exact path="/" render={() => (<Redirect to="/home" />)} />

            <Route path="/home" component={Home} />

            <Route path="/about/:id" component={About} />

            <PrivateRoute path="/contact" component={Contact} />

            <Route component={NoMatch} />

          </Switch>

        </App>

      </HashRouter>

    )

  }

}

const isAuthenticated = false;  //判断登录

function PrivateRoute({ component: Component, ...rest }) {

  return (

    <Route {...rest} render={

      props => isAuthenticated ? (<Component {...props} />) : (<Redirect to={{ pathname: "/home" }} />)

    } />

  );

}

export default class router extends Component {

  render() {

    return (

      <HashRouter>

        <App>

          <Switch>

            {Routers.map((item, index) => {

              return <Route key={index} path={item.path} exact render={props =>

                ((token ? <item.component {...props} /> : <Redirect to={{

                  pathname: '/login',

                  state: { from: props.location }

                }} />)

                )} />

            })}

          </Switch>

        </App>

      </HashRouter>

    )

  }

}

##### 区别

react-router-dom是依赖react-router的，所以我们使用npm安装react-router-dom的时候，不需要npm安装react-router。

###### api对比

React-router：提供了router的核心api。如Router、Route、Switch等，但没有提供有关dom操作进行路由跳转的ap；

React-router-dom：提供了BrowserRouter、Route、Link等api，可以通过dom操作触发事件控制路由。

###### 功能对比

React-router：实现了路由的核心功能

React-router-dom：基于React-router，加入了一些在浏览器运行下的一些功能

例如：Link组件会渲染一个a标签

BrowserRouter使用 HTML5 提供的 history API可以保证你的 UI 界面和 URL 保持同步

HashRouter使用 URL 的 hash 部分保证你的 UI 界面和 URL 保持同步

BrowserRouter和HashRouter组件，前者使用pushState和popState事件构建路由，后者使用window.location.hash和hashchange事件构建路由。

###### 写法对比

// React-router不能通过操作dom控制路由，此时还需引入React-router-dom

import {Switch, Route, Router} from 'react-router';

import {HashHistory, Link} from 'react-router-dom';

// React-router-dom在React-router的基础上扩展了可操作dom的api

import {Swtich, Route, Router, HashHistory, Link} from 'react-router-dom';

###### 路由跳转对比

React-router：router4.0以上版本用this.props.history.push('/path')实现跳转；

　　　　　　 router3.0以上版本用this.props.router.push('/path')实现跳转

React-router-dom：直接用this.props.history.push('/path')就可以实现跳转

#### **Mobx**

##### 装饰器 说明

**@observable** 可以在 ES7 或者 TypeScript 类属性中属性使用，将其转换成可观察的。 @observable 可以在实例字段和属性 getter 上使用。

**@action** 用于装饰可观察变量的状态的更新。在严格模式下，对于可观察变量的状态的更新必须使用此装饰器

**@observer** 用来将 React 组件转变成响应式组件。它用 mobx.autorun 包装了组件的 render 函数以确保任何组件渲染中使用的数据变化时都可以强制刷新组件。

**@computed** 如果已经启用 decorators 的话，可以在任意类属性的 getter 上使用 @computed 装饰器来声明式的创建计算属性。

使用

const { observable, action, computed, autorun } = mobx;

class Store {

  // 通过 @observable 修饰器可以将其中的属性转变为可观察的状态值

  @observable count = 0;

  // 如果已经启用 decorators 的话，可以在任意类属性的 getter 上使用 @computed 装饰器来声明式的创建计算属性。

  @computed get result() {

    return this.count + 100;

  }

  // @action，其规定对于 store 对象中所有可观察状态属性的改变都应该在 @action 中完成，它使代码可以组织的更好，并且对于数据改变的时机也更加清晰。

  @action add () {

    this.count = this.count;

  }

};

const mstore = new Store();

// autorun 接受一个函数作为参数，在使用 autorun 的时候，该函数会被立即调用一次，之后当该函数中依赖的可观察状态属性（或者计算属性）发生变化的时候，该函数会被调用，注意，该函数的调用取决的函数中使用了哪些可观察状态属性（或者计算属性）。

autorun(() => {

  console.log(mstore.result);

});

## NodeJS

#### 终端基本命令

****npm uninstall 模块****：删除模块，但不删除模块留在package.json中的对应信息

****npm uninstall 模块 --save**** 删除模块，同时删除模块留在package.json中dependencies下的对应信息

****npm uninstall 模块 --save-dev**** 删除模块，同时删除模块留在package.json中devDependencies下的对应信息

#### MongoDB

##### MongoDB启动相关

C:\Program Files\MongoDB\Server\4.2\bin

到mongod安装目录下的bin文件夹使用cmd

使用mongod.cfg配置文件启动

mongod -f mongod.cfg

##### mongoDB开启用户权限校验

开启用户权限校验(两种方式)

1.命令行

mongod -f mongod.cfg --auth

2.config文件

security:

authorization: enabled

##### mongoDB命令行操作相关

**mongodb命令行操作**

先输入**mongo**命令启动

使用admin权限

use admin

显示数据库列表

show dbs

权限校验

ds.auth(用户名,密码)

##### 增删改查用户

创建一个数据库新用户用db.createUser()方法，如果用户存在则返回一个用户重复错误。

## 用法

db.createUser(user, writeConcern)

* user：这个文档创建关于用户的身份认证和访问信息；
* writeConcern：这个文档描述保证MongoDB提供写操作的成功报告。

### user文档

user文档，定义了用户的以下形式：

{ user: "<name>",

pwd: "<cleartext password>",

customData: { <any information> },

roles: [

{ role: "<role>", db: "<database>" } | "<role>",

...

]

}

user文档字段介绍：

* user字段，为新用户的名字；
* pwd字段，用户的密码；
* cusomData字段，为任意内容，例如可以为用户全名介绍；
* roles字段，指定用户的角色，可以用一个空数组给新用户设定空角色；

在roles字段,可以指定内置角色和用户定义的角色。

MongoDB内置角色有如下：

1. 数据库用户角色：read、readWrite;
2. 数据库管理角色：dbAdmin、dbOwner、userAdmin；
3. 集群管理角色：clusterAdmin、clusterManager、clusterMonitor、hostManager；
4. 备份恢复角色：backup、restore；
5. 所有数据库角色：readAnyDatabase、readWriteAnyDatabase、userAdminAnyDatabase、dbAdminAnyDatabase
6. 超级用户角色：root

// 这里还有几个角色间接或直接提供了系统超级用户的访问（dbOwner 、userAdmin、userAdminAnyDatabase）

1. 内部角色：\_\_system

你可以[点击这里](https://docs.mongodb.com/manual/reference/built-in-roles/" \l "built-in-roles)查看每个角色所拥有的操作权限。

### writeConcern文档

* w选项：允许的值分别是 1、0、大于1的值、"majority"、<tag set>；
* j选项：确保mongod实例写数据到磁盘上的journal（日志），这可以确保mongd以外关闭不会丢失数据。设置true启用。
* wtimeout：指定一个时间限制,以毫秒为单位。wtimeout只适用于w值大于1。

## 示例

例如：在products数据库创建用户accountAdmin01，并给该用户admin数据库上clusterAdmin和readAnyDatabase的角色，products数据库上readWrite角色。

use products

db.createUser( { "user" : "accountAdmin01",

"pwd": "cleartext password",

"customData" : { employeeId: 12345 },

"roles" : [ { role: "clusterAdmin", db: "admin" },

{ role: "readAnyDatabase", db: "admin" },

"readWrite"

] },

{ w: "majority" , wtimeout: 5000 } )

验证：

mongo -u accountAdmin01 -p yourpassward --authenticationDatabase products

## JavaScript

#### 数据类型

##### 基本数据类型

基本数据类型是按值访问的，因为可以直接操作保存在变量中的实际值

Number,String,Boolean,undefined,null,Symbol

##### 引用数据类型

Object 、Array 、Function 、Data、Regexp

javascript的引用数据类型是保存在堆内存中的对象。引用类型数据（变量）在栈内存中保存的是对象在堆内存中的引用地址。通过这个引用地址可以快速查找到保存中堆内存中的对象。

#### 装箱、拆箱

##### 装箱：把基本数据类型转换为对应的引用类型的操作称为装箱

每当读取一个基本类型的时候，后台就会创建一个对应的基本包装类型对象，从而让我们能够调用一些方法来操作这些数据。

###### 隐式装箱

num.toFixed(2); // '123.00'

//上方代码在后台的真正步骤为

var c = new Number(123);

c.toFixed(2);

c = null;

###### 显示装箱

var obj = new String('123');

##### 拆箱：把引用类型转换为基本的数据类型称为拆箱

将引用类型对象转换为对应的值类型对象，它是通过引用类型的valueOf()或者toString()方法来实现的。如果是自定义的对象，你也可以自定义它的valueOf()/tostring()方法，实现对这个对象的拆箱。

var objNum = new Number(123);

var objStr = new String("123");

console.log(typeof objNum); //object

console.log(typeof objStr); //object

console.log(typeof objNum.valueOf()); //number

console.log(typeof objStr.valueOf()); //string

console.log(typeof objNum.toString()); // string

console.log(typeof objStr.toString()); // string

#### CommonJs模块规范和es6模块规范的区别

// CommonJS模块规范

// 为了方便，Node为每个模块提供一个exports变量，指向module.exports。这等同在每个模块头部，有一行这样的命令。

var exports = module.exports;

// 造成的结果是，在对外输出模块接口时，可以向exports对象添加方法。

// 但是不能直接将exports变量指向一个值，因为这样等于切断了exports与module.exports的联系。exports只能通过打点属性赋值导出

// exports = function(x) {console.log(x)};  //错误写法

exports.area = function (r) {

  return Math.PI \* r \* r;

};

exports.circumference = function (r) {

  return 2 \* Math.PI \* r;

};

// ES6模块规范

// 不同于CommonJS，ES6使用 export 和 import 来导出、导入模块。

// 使用export

var firstName = 'Michael';

var lastName = 'Jackson';

var year = 1958;

export {firstName, lastName, year};

// 导入方式 import {firstName, lastName, year} from '文件路径'

// 使用export default命令，为模块指定默认输出。

// export-default.js

export default function () {

  console.log('foo');

}

// 导入方式 import name from '文件路径'

#### var与let、const

##### var声明的变量会挂载在window上，而let和const声明的变量不会：

var a = 100;

console.log(a,window.a);    // 100 100

let b = 10;

console.log(b,window.b);    // 10 undefined

const c = 1;

console.log(c,window.c);    // 1 undefined

##### var声明变量存在变量提升，let和const不存在变量提升

console.log(a); // undefined  ===>  a已声明还没赋值，默认得到undefined值

var a = 100;

console.log(b); // 报错：b is not defined  ===> 找不到b这个变量

let b = 10;

console.log(c); // 报错：c is not defined  ===> 找不到c这个变量

const c = 10;

##### let和const声明形成块作用域

if(1){

  var a = 100;

  let b = 10;

}

console.log(a); // 100

console.log(b)  // 报错：b is not defined  ===> 找不到b这个变量

if(1){

  var a = 100;

  const c = 1;

}

console.log(a); // 100

console.log(c)  // 报错：c is not defined  ===> 找不到c这个变量

##### 同一作用域下let和const不能声明同名变量，而var可以

var a = 100;

console.log(a); // 100

var a = 10;

console.log(a); // 10

let a = 100;

let a = 10;

//  控制台报错：Identifier 'a' has already been declared  ===> 标识符a已经被声明了。

##### 暂存死区

var a = 100;

if(1){

    a = 10;

    //在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，

    // 而这时，还未到声明时候，所以控制台Error:a is not defined

    let a = 1;

}

##### Const

/\*

\* 　　1、一旦声明必须赋值,不能使用null占位。

\* 　　2、声明后不能再修改

\* 　　3、如果声明的是复合类型数据，可以修改其属性

\*

\* \*/

const a = 100;

const list = [];

list[0] = 10;

console.log(list);　　// [10]

const obj = {a:100};

obj.name = 'apple';

obj.a = 10000;

console.log(obj);　　// {a:10000,name:'apple'}

## Uniapp

## 微信小程序

## Html

## Css