复习

JavaScript模块

1. 实现一个new

```
function my_new () {
    let obj = new Object()  // 新创建一个空对象
    let Fun = [].shift.call(arguments) // 获取构造函数
    obj.__proto__ = Fun.prototype  // 将obj连接到Fun的原型上
    let result = Fun.apply(obj, arguments) // 绑定this
    return typeof obj === 'Object' ? result : obj // 确保返回的是一个对象
}
```

2. 实现一个instanceof

```
function my_instanceof(left, right) {
    let prototype = right.prototype // 获取类型的原型
    left = left.__proto__ // 获取对象的原型
    while (true) {   // 判断对象的类型是否等于类型的原型
        if (left === null)
            return false
        if (prototype === left)
            return true
        left = left.__proto__
    }
}
```

3. 实现一个call

call是多个参数

```
Function.prototype.my_call = function(context){
    let context = context || window // 不传入第一个参数，默认window
    context.fn = this // 添加一个属性（这里的this是call前面的函数，比如
obj.sayhi.my_call(m) 这里m就是context， this就是sayhi）
    var args = [...arguments].slice(1) // 将context后面的参数取出来
    var result = context.fn(...args)
    delete context.fn // 删除fn
    return result
}
```

4. 实现一个apply

apply 同理， B.apply(A, arguments); apply的参数是一个数组

```
Function.prototype.my_apply = function (context) {
    var context = context || window
    context.fn = this
    var result
    // 需要判断是否存储第二个参数，如果存在就将第二个参数展开
    if(arguments[1]){
        result = context.fn(...arguments[1])
    } else {
        result = context.fn()
    }
    delete context.fn
    return result
}
```

5. 实现bind

bind和apply，call的方法作用也是一致的，区别是bind会返回一个函数。并且我们可以通过bind实现柯里化

```
Function.prototype.my_bind = function (context){
    console.log(this)
    if(typeof this !== 'function ') {
        throw new TypeError('error')
    }
    var that = this
    console.log(that)
    var args = [...arguments].slice(1)
    console.log(args)
    // 返回一个函数
    return function F() {
        console.log(this)
        // 可以new F()，所以需要判断
        if (this instanceof F) {
            return new _this(...args, ...arguments)
        }
        return _this.apply(context, args.concat(...arguments))
    }
}
```

Vue

1. v-show 和v-if 的区别

切换比较频繁选v-show，否则v-if

2. Vue生命周期

beforeCreate

created

beforeMount

mounted

beforeUpdate

updated

beforeDestroy

destroyed

3. Vue响应式

vue2.x核心API——Object.defineProperty

能监听对象，不能原生监听数组，需要特殊处理

深度监听需要一次性递归到底，一次性计算量大

无法监听新增属性/删除属性（Vue.set Vue.delete）

```
function updateView () {
  console.log('更新啦!')
}

function defineObserve (target, key, value) {
  /**深度监听
   * 否则value如果是对象将无法被监听
   */
  observer(value)
  Object.defineProperty(target, key, {
    get: function () {
      return value
    },
    set: function (newval) {
      if (newval !== value) {

        /**深度监听（如果不监听设置新值，将会监听不到的情况）
         * 比如 刚开始data.name = 'hahabboom'
         * data.name = {number: 12}
         * data.name.number = 1
         * 最后一次number改变将无法被监听
         */
        observer(newval)

        value = newval
        updateView()
      }
    }
  })
}

function observer (target) {
  if (typeof target === null || typeof target !== 'object') {
    return target
  }
  for(let key in target) {
    defineObserve(target, key, target[key])
  }
}

let data = {
  name: 'hahabboom',
  age: 11,
```

```
    info: {
      detail: 'hi you!!'
    },
    part: ['one', 'two', 'three']
  }
  observer(data)

  // data.name = '231'
  // data.age = 1
  // data.info.detail = 'change'
  data.name = {number: 12}
  data.name.number = 1
```

vue3.x——Proxy

Proxy可深度监听，性能更好（因为他是什么时候get什么时候递归，不像上面那种一次性递归到底）

可监听新增/删除属性

可监听数组变化

Proxy兼容性不好，且无法polyfill

```
function reactive(target = {}) {
    if (typeof target !== 'object' || target == null) {
        // 不是对象或数组，则返回
        return target
    }

    // 代理配置
    const proxyConf = {
        get(target, key, receiver) {
            // 只处理本身（非原型的）属性
            const ownKeys = Reflect.ownKeys(target)
            if (ownKeys.includes(key)) {
                console.log('get', key) // 监听
            }

            const result = Reflect.get(target, key, receiver)

            // 深度监听
            // 性能如何提升的？
            return reactive(result)
        },
        set(target, key, val, receiver) {
            // 重复的数据，不处理
            if (val === target[key]) {
                return true
            }

            const ownKeys = Reflect.ownKeys(target)
            if (ownKeys.includes(key)) {
                console.log('已有的 key', key)
            } else {
                console.log('新增的 key', key)
            }

            const result = Reflect.set(target, key, val, receiver)
```

```
            console.log('set', key, val)
            // console.log('result', result) // true
            return result // 是否设置成功
        },
        deleteProperty(target, key) {
            const result = Reflect.deleteProperty(target, key)
            console.log('delete property', key)
            // console.log('result', result) // true
            return result // 是否删除成功
        }
    }

    // 生成代理对象
    const observed = new Proxy(target, proxyConf)
    return observed
}
```

4. Vue如何监听数组变化

```
  function updateView () {
    console.log('更新啦!')
  }

  // 避免污染全局原型
  const oldArray = Array.prototype

  const newArray = Object.create(oldArray);

  ['pop','push'].forEach(name => {
    newArray[name] = function () {
      updateView()
      oldArray[name].call(this, ...arguments)
    }
  })

  function defineObserve (target, key, value) {
    /**深度监听
     * 否则value如果是对象将无法被监听
     */
    observer(value)
    Object.defineProperty(target, key, {
      get: function () {
        return value
      },
      set: function (newval) {
        if (newval !== value) {

          /**深度监听（如果不监听设置新值，将会监听不到的情况）
           * 比如 刚开始data.name = 'hahabboom'
           * data.name = {number: 12}
           * data.name.number = 1
           * 最后一次number改变将无法被监听
           */
          observer(newval)
```

```
          value = newval
          updateView()
        }
      }
    })
  }

  function observer (target) {
    if (typeof target === null || typeof target !== 'object') {
      return target
    }
    if (Array.isArray(target)) {
      target.__proto__ = newArray
    }
    for(let key in target) {
      defineObserve(target, key, target[key])
    }
  }
```

5. Vue双向绑定

方案一：

子组件$emit派发input事件，父组件监听input事件中的value值，并存起来；父组件通过prop的形式传递给子组件value值，子组件将其绑定在value上

父组件

```
<message-demo :text="name" @change="name = $event"></message-demo>
```

子组件

```
 <input type="text" :value="text" @input="$emit('change', $event.target.value)">
```

方案二：

使用model指令，指定prop和event

父组件

```
<message-demo v-model="name"></message-demo>
```

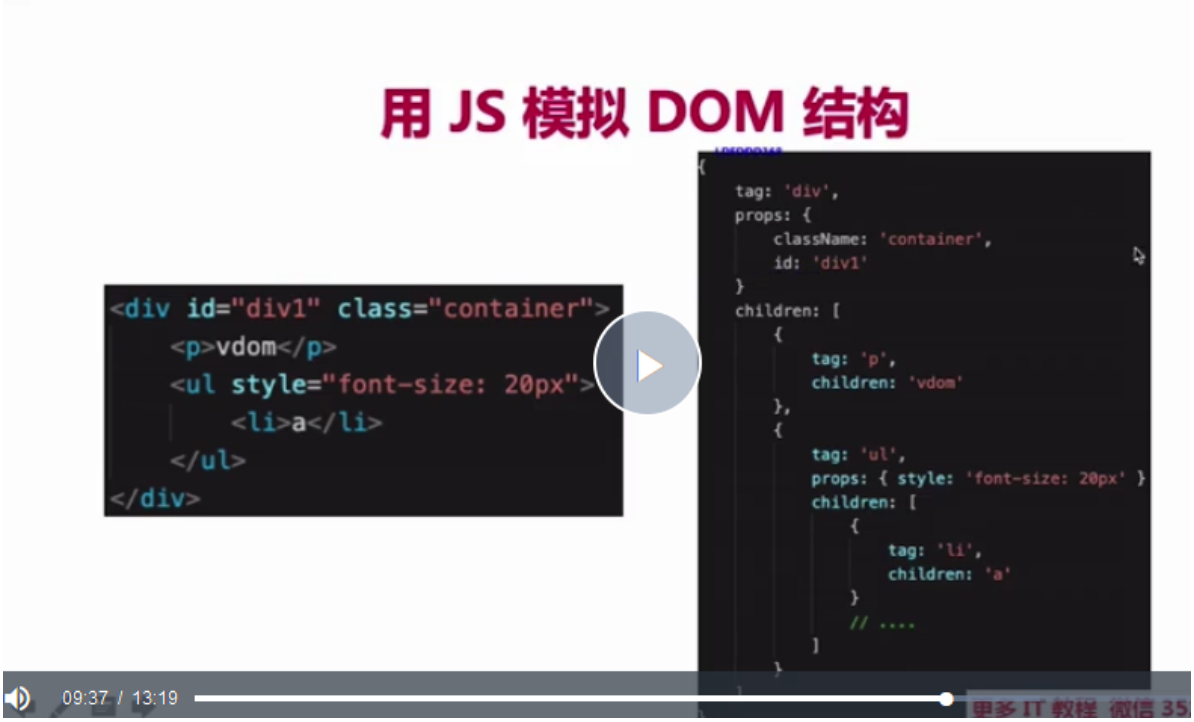
子组件

```
<input type="text" :value="text">
<script>
export default {
  name: 'messageDemo',
  model: {
    prop: 'text',
    event: 'change'
  },
  data () {
    return {
```

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

6. 虚拟DOM

JS如何模拟DOM结构




```
{
    tag: 'div',
    props: {
        className: 'container',
        id: 'div1'
    },
    children: [
        {
            tag: 'p',
            children: 'vdom'
        },
        ...
    ]
}
```

7. diff算法

优化时间复杂度到On

只比较同一层级，不跨级比较

tag不同，直接删除重建，不在深度比较

tag和key，两者都想吐，则认为是相同节点，不再深度比较

webpack模块

1. 前端模块化

什么是模块?

- 将一个复杂的程序依据一定的规则（规范）封装成几个块（文件）并进行组合在一起
- 块的内部数据与实现是私有的，只是向外部暴露一些接口（方法）与外部其他模块通信

模块化的好处

- 避免命名冲突
- 更好的分离，按需加载
- 更好复用性
- 高可维护性

引入多个script后出现的问题

- 请求过多
- 依赖模糊--不知道具体的依赖关系是什么，很容易因为不了解他们之间的依赖关系导致加载先后顺序出错
- 难以维护--以上两个问题就导致难以维护，很有可能牵一发动全身

模块化规范

- CommonJS（node，每个文件就是一个模块，有自己的作用域，在一个文件里面定义的变量函数类都是私有的，对其他文件不可见；在服务端，模块加载是运行时同步加载的；在浏览器端，模块是需要提前编译打包处理）

  特点：

  1. 所有代码都运行在模块作用域，不会污染全局作用域
  2. 所有模块都可多参加在，但是只会在第一次加载是运行一次，然后运行结果就被缓存了，以后直接读取缓存
  3. 模块加载顺序是按照在代码中出现的顺序

  基本语法：

  ```
  1. 暴露模块
  var x = 5;
  var addX = function (value) {
    return value + x;
  };
  module.exports.x = x;
  module.exports.addX = addX;
  2. 引入模块
  var example = require('./example.js');//如果参数字符串以“./”开头，则表示加载的是一个位于相对路径
  ```

  模块加载机制：

  输入的是被输出的值的拷贝，因为比如输出一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动后的值

- AMD（非同步加载模块，允许指定回调函数，AMD模式可以用于浏览器环境，并且允许非同步加载模块，也可以根据需要动态加载模块。）

  语法：

```
//定义有依赖的模块
define(['module1', 'module2'], function(m1, m2){
    return 模块
})
引入使用模块:

require(['module1', 'module2'], function(m1, m2){
    使用m1/m2
})
```

AMD不会污染全局环境，能够清楚地显示依赖关系，AMD可以用于浏览器环境，并且允许非同步加载模块，也可以根据需要动态加载模块

- CMD（专门用于浏览器端，模块加载异步，模块使用时才会加载执行。CMD整合了CommonJS和AMD规范的特点）

  语法：

```
// 定义没有依赖模块
define(function(require, exports, module){
  exports.xxx = value
  module.exports = value
})
// 引用使用模块
define(function (require) {
  var m1 = require('./module1')
  var m4 = require('./module4')
  m1.show()
  m4.show()
})
```

```
- ES6模块化（设计思想还是尽量静态化，使得编译的时候就能确定模块依赖关系）

  语法：

  ```javascript
  // export-default.js
  export default function () {
    console.log('foo');
  }
  // import-default.js
  import customName from './export-default';
  customName(); // 'foo
```

2. tree-shaking

webpack 2增加了tree-shaking的功能

Tree-shaking的本质是消除无用的js代码
```

3. 基本配置

设置代理

```
proxy: {
    '/api': 'http://local:3000',
    '/api2': {
        target: 'http://local:3000',
        pathRewrite: {
            '/api2': ''
        }
    }
}
```

loader执行顺序从后往前

```
{
    loader: ['style-loader', 'css-loader', 'postcss-loader' // 负责添加浏览器前缀之类，兼容性更好]（生产环境写法）
}
```

4. 多入口配置

- 在webpack中配置

    1. common.js中配置entry和plugins

```
    entry: {
        index: path.join(srcPath, 'index.js'),
        other: path.join(srcPath, 'other.js')
    },

    plugins: [
        // 多入口 - 生成 index.html
        new HtmlWebpackPlugin({
            template: path.join(srcPath, 'index.html'),
            filename: 'index.html',
            // chunks 表示该页面要引用哪些 chunk （即上面的 index 和 other），默认全部引用
            chunks: ['index']  // 只引用 index.js
        }),
        // 多入口 - 生成 other.html
        new HtmlWebpackPlugin({
            template: path.join(srcPath, 'other.html'),
            filename: 'other.html',
            chunks: ['other']  // 只引用 other.js
        })
    ]
```

    2. 在prod里面配置output

```
    output: {
        // filename: 'bundle.[contentHash:8].js',  // 打包代码时，加上 hash 戳
        filename: '[name].[contentHash:8].js', // name 即多入口时 entry 的 key
        path: distPath,
    },
```

- vue-cli 3中配置

  配置pages

```
pages: {
  business: {
    // page 的入口
    entry: 'src/views/business/main.js',
    // 模板来源
    template: 'public/index.html',
    // 在 dist/index.html 的输出
    filename: 'index.html',
    // 当使用 title 选项时，
    // template 中的 title 标签需要是 <title><%=
htmlWebpackPlugin.options.title %></title>
    title: '综合气象观测业务运行信息化平台',
    chunks: ['chunk-vendors', 'chunk-common', 'business']
  },
  ksh: {
    // page 的入口
    entry: 'src/views/ksh/main.js',
    // 模板来源
    template: 'public/ksh.html',
    // 在 dist/index.html 的输出
    filename: 'ksh.html',
    // 当使用 title 选项时，
    // template 中的 title 标签需要是 <title><%=
htmlWebpackPlugin.options.title %></title>
    title: '全国天气雷达在线分析系统',
    chunks: ['chunk-vendors', 'chunk-common', 'ksh']
  }
}
```

5. webpack如何抽离压缩css文件

1. 在prod.js里面安装引入mini-css-extract-plugin插件
2. 在rules里写上

```
            {
                test: /\.css$/,
                loader: [
                    MiniCssExtractPlugin.loader,  // 注意，这里不再用 style-
loader
                    'css-loader',
                    'postcss-loader'
                ]
            },
            // 抽离 less --> css
```

```
            {
                test: /\.less$/,
                loader: [
                    MiniCssExtractPlugin.loader,  // 注意，这里不再用 style-
loader
                    'css-loader',
                    'less-loader',
                    'postcss-loader'
                ]
            }
```

3. 在plugins里增加配置（生产环境使用hash，增加被命中缓存的概率，速度会快一些）

```
        new MiniCssExtractPlugin({
            filename: 'css/main.[contentHash:8].css'
        })
```

接着就是压缩，安装插件 terser-webpack-plugin 和 optimize-css-assets-webpack-plugin ：

```
    optimization: {
        // 压缩 css
        minimizer: [new TerserJSPlugin({}), new
OptimizeCSSAssetsPlugin({})],
    }
```

6. webpack如何抽离公共代码和第三方代码

在optimization里面进行配置：

```
        // 分割代码块
        splitChunks: {
            chunks: 'all',
            /**
             * initial 入口 chunk，对于异步导入的文件不处理
                async 异步 chunk，只对异步导入的文件处理
                all 全部 chunk
             */

            // 缓存分组
            cacheGroups: {
                // 第三方模块
                vendor: {
                    name: 'vendor', // chunk 名称
                    priority: 1, // 权限更高，优先抽离，重要！！！,比如我有个代码又是第三
方又是公共模块，优先命中第三方里面
                    test: /node_modules/, // 命中范围
                    minSize: 0,  // 大小限制
                    minChunks: 1  // 最少复用过几次
                },

                // 公共的模块
                common: {
                    name: 'common', // chunk 名称
```

```
                priority: 0, // 优先级
                minSize: 0,  // 公共模块的大小限制
                minChunks: 2  // 公共模块最少复用过几次
            }
        }
    }
```

所以在进行多入口配置的时候会有chunks的配置，就是因为考虑代码分割的问题了

7. webpack如何实现异步加载

```
setTimeOut(() => {
// 类似定义一个chunk
    import('./xxx.js').then(res => {
        console.log(res)
    })
}, 1500)
```

webpack实现懒加载就是import + vue/react异步组件 + vue-router/react-router异步加载路由

8. module chunk bundle区别

module——各个源码文件，webpack中一切皆模块

chunk——多模块合并成的（webpack分析过程中），如entry import() splitChunks

bundle——最终输出文件（一个chunk对应一个bundle）

9. webpack性能优化

- 优化打包构建速度——开发体验和效率
    1. 优化babel-loader
        - 开启缓存
        - 明确范围（include和exclude二者选一个就可以）

        ```
        {
            test: /\.js$/,
            use: ['babel-loader?cacheDirecyory'], //开启缓存
            include:  path.resolve(_dirname, 'src') // 明确范围
        }
        ```

    2. IgnorePlugin（webpack内置插件，忽略第三方指定目录，让这些指定目录不要被打包进去）

        ```
        plugins:[
            new Webpack.IgnorePlugin(/\.\/locale/,/moment/),//moment这个库中，如果引用了./locale/目录的内容，就忽略掉，不会打包进去
        ]
        // 因为上面忽略了那个目录，需要手动引入中文语言目录
        ```

    3. noParse（避免去打包那些东西）
    4. happyPack（多进程打包，生产和开发环境都可以）

        因为js是单线程，开启多进程打包

        项目较大，打包较慢，开启多进程能提高速度

项目较小，打包较快，开启多进程会降低速度（进程开销）

```
引入happypack
然后在rules改写js的
    {
        test: /\.js$/,
        // 把对 .js 文件的处理转交给 id 为 babel 的 HappyPack 实例
        use: ['happypack/loader?id=babel'],
        include: srcPath,
        // exclude: /node_modules/
    }
然后在plugins里面 new HappyPack
        new HappyPack({
            // 用唯一的标识符 id 来代表当前的 HappyPack 是用来处理一类特定的文件
            id: 'babel',
            // 如何处理 .js 文件，用法和 Loader 配置中一样
            loaders: ['babel-loader?cacheDirectory']
        }),
```

5. ParallelUglifyPlugin（多进程压缩JS，只能生产环境）

   webpack内置Uglify工具压缩JS

   JS单线程，开启多进程压缩更快

   和happyPack同理

```
        // 使用 ParallelUglifyPlugin 并行压缩输出的 JS 代码
        new ParallelUglifyPlugin({
            // 传递给 UglifyJS 的参数
            // （还是使用 UglifyJS 压缩，只不过帮助开启了多进程）
            uglifyJS: {
                output: {
                    beautify: false, // 最紧凑的输出
                    comments: false, // 删除所有的注释
                },
                compress: {
                    // 删除所有的 `console` 语句，可以兼容ie浏览器
                    drop_console: true,
                    // 内嵌定义了但是只用到一次的变量
                    collapse_vars: true,
                    // 提取出出现多次但是没有定义成变量去引用的静态值
                    reduce_vars: true,
                }
            }
        })
```

6. 自动刷新（不能用于生产环境）

   watch:true 开启监听

   // 但是在开发环境下开了devServer就会自动开启刷新浏览器，其实就会覆盖watch

```
    watch: true, // 开启监听，默认为 false
    watchOptions: {
        ignored: /node_modules/, // 忽略哪些
        // 监听到变化发生后会等300ms再去执行动作，防止文件更新太快导致重新编译频率太高
        // 默认为 300ms
        aggregateTimeout: 300,
        // 判断文件是否发生变化是通过不停的去询问系统指定文件有没有变化实现的
        // 默认每隔1000毫秒询问一次
        poll: 1000
    }
```

7. 热更新（不能用于生产环境）

   因为自动刷新的话，整个网页全部刷新，速度较慢，状态会丢失

   热更新：新代码生效，网页不刷新，状态不丢失（比如全局变量还在，输入框的内容还在，路由不丢失）

   优化会付出代价

   首先引入插件 HotModuleReplacementPlugin

   然后将entry的index改写成数组：

```
    entry: {
        // index: path.join(srcPath, 'index.js'),
        index: [
            'webpack-dev-server/client?http://localhost:8080/',
            'webpack/hot/dev-server',
            path.join(srcPath, 'index.js')
        ],
        other: path.join(srcPath, 'other.js')
    },
```

   然后在plugins里面new HotModuleReplacementPlugin

```
    plugins: [
        new HotModuleReplacementPlugin()
    ],
```

   最后在devServer里面 开启hot：true

   如果在监听范围之外需要手动允许那些模块进行热更新（所以需要成本的）

```
if (module.hot) {
    module.hot.accept(['./math'], () => {
        const sumRes = sum(10, 30)
        console.log('sumRes in hot', sumRes)
    })
}
```

8. DllPlugin（动态链接库插件，只开发环境）

   前端框架Vue体积大，构建慢，但是较稳定，不会经常升级版本；同一个版本只会构建一次，不用每次都重新构建

   webpack已内置DllPlugin支持

DllPlugin——将vue打包出dll文件

DllReferencePlugin——使用dll文件

步骤1：新建一个webpack.dll.js文件，在里面引入DllPlugin，在module.exports里面配置entry，output，plugin

```
const path = require('path')
const DllPlugin = require('webpack/lib/DllPlugin')
const { srcPath, distPath } = require('./paths')

module.exports = {
  mode: 'development',
  // JS 执行入口文件
  entry: {
    // 把 React 相关模块的放到一个单独的动态链接库
    react: ['react', 'react-dom']
  },
  output: {
    // 输出的动态链接库的文件名称，[name] 代表当前动态链接库的名称，
    // 也就是 entry 中配置的 react 和 polyfill
    filename: '[name].dll.js',
    // 输出的文件都放到 dist 目录下
    path: distPath,
    // 存放动态链接库的全局变量名称，例如对应 react 来说就是 _dll_react
    // 之所以在前面加上 _dll_ 是为了防止全局变量冲突
    library: '_dll_[name]',
  },
  plugins: [
    // 接入 DllPlugin
    new DllPlugin({
      // 动态链接库的全局变量名称，需要和 output.library 中保持一致
      // 该字段的值也就是输出的 manifest.json 文件 中 name 字段的值
      // 例如 react.manifest.json 中就有 "name": "_dll_react"
      name: '_dll_[name]',
      // 描述动态链接库的 manifest.json 文件输出时的文件名称
      path: path.join(distPath, '[name].manifest.json'),
    }),
  ],
}
```

步骤二：在package.json里面的scripts里面再加一条dll指令，就可以执行了

```
 "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "webpack-dev-server --config build/webpack.dev.js",
    "dll": "webpack --config build/webpack.dll.js"
  },
```

步骤三：然后就可以在index.html里面引用了，但是需要DllReferencePlugin插件配置：1.在dev.js里面引入该插件，2. 在rules忽略node_modules ,3. 在 plugins 里面new一下，高数webpack使用了哪些动态链接库

```
// 第一，引入 DllReferencePlugin
const DllReferencePlugin = require('webpack/lib/DllReferencePlugin');

// 第二，不要再转换 node_modules 的代码
```

```
    module: {
        rules: [
            {
                test: /\.js$/,
                loader: ['babel-loader'],
                include: srcPath,
                exclude: /node_modules/ // 第二，不要再转换 node_modules 的
代码
            },
        ]
    },


      plugins: [
        new webpack.DefinePlugin({
            // window.ENV = 'production'
            ENV: JSON.stringify('development')
        }),
        // 第三，告诉 Webpack 使用了哪些动态链接库
        new DllReferencePlugin({
            // 描述 react 动态链接库的文件内容
            manifest: require(path.join(distPath,
'react.manifest.json')),
        }),
    ],
```

- 优化产出代码——产品性能

  体积更小；合理分包，不重复加载；速度更快，内存使用更少

  1. 小图片base64编码
  2. bundle加hash（根据文件内容算hash值，如果文件没有改变hash值不会改变，就更容易命中缓存）
  3. 懒加载
  4. 提取公共代码
  5. IngorePlugin（忽略多语言，只引用自己需要的语言）
  6. 使用CDN加速

     步骤： 1. 在prod.js的output设置一个公共publicPath，修改所有静态文件的url前缀

     ```
         output: {
             filename: 'bundle.[contentHash:8].js',  // 打包代码时，加上 hash 戳
             path: distPath,
             // publicPath: 'http://cdn.abc.com'  // 修改所有静态文件 url 的前缀
     （如 cdn 域名），这里暂时用不到
         },
     ```

     ```
             2. 把打包出来的js，css，图片上传到cdn上面
     ```

  7. 使用production

     自动开启压缩代码——webpack4.x后才有

     vue react会自动删掉调试代码（如开发环境的warning）

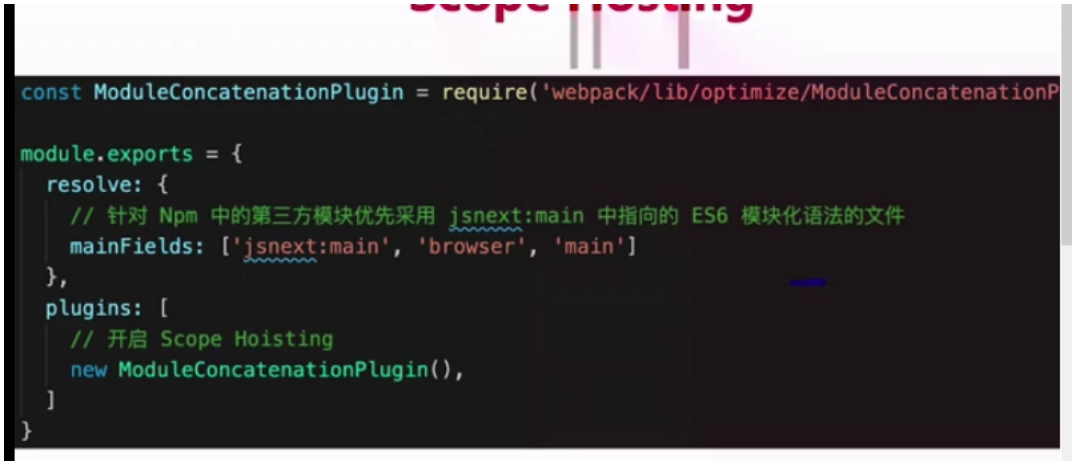
启动Tree-Shaking（必须使用ES6 Module才能使tree-shaking生效）

为什么?

ES6Module静态引入，编译时引入；Commonjs动态引入，执行时引入；只有静态分析才能实现Tree-Shaking

1. Scope Hosting（代码体积越大效果越好）

   代码体积更小；创建作用域更少；代码可读性更好



10. babel（将各种语言转换为浏览器可以认识的语言， ECMAScript 2015+ 代码转换为当前和旧版浏览器或环境中的向后兼容版本的 JavaScript）

webpack 通过 babel-loader 使用 Babel

11. babel-polyfill

所有的函数的polyfill的集合——core.js

支持generator语法——regenerator

babel-polyfill就是以上的集合

babel-polyfill如何按需引入？——

在babellrc文件里面的presets增加配置"useBuiltIns": "usage"

```
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage",
                "corejs": 3
            }
        ]
    ]
}
```

babel-pollyfill存在的问题?

会污染全局（因为重新定义了函数，如果做一个独立的web系统就没有问题；如果做第三方库，就存在问题），使用babel-runtime，首先安装插件

```
  "devDependencies": {
    "@babel/cli": "^7.7.5",
    "@babel/core": "^7.7.5",
    "@babel/plugin-transform-runtime": "^7.7.5",
    "@babel/preset-env": "^7.7.5"
  },
  "dependencies": {
    "@babel/polyfill": "^7.7.0",
    "@babel/runtime": "^7.7.5"
  }
```

然后修改babelrc的配置

```
    "plugins": [
        [
            "@babel/plugin-transform-runtime",
            {
                "absoluteRuntime": false,
                "corejs": 3,
                "helpers": true,
                "regenerator": true,
                "useESModules": false
            }
        ]
    ]
```

12. 前端为什么要进行打包和构建

1. 体积更小（tree-shaking，压缩，合并），加载更快
2. 编译更加高级语言或语法（TS，ES6+，模块化，scss）
3. 兼容性和错误检查（polyfill，postcss，eslint）
4. 统一，高效的开发环境
5. 统一的构建流程和产出标准
6. 集成公司构建规范（提测，上线等）

13. loader和plugin的区别

loader——模块转换器

plugin——扩展插件，如HtmlWebpackPlugin（吧js，css塞进一个html文件里面）

常见的loader和plugin

14. babel和webpack的区别

babel——js新语法编译工具，不关心模块化

webpack——打包构建工具，不关心语法，是多个loader，plugin的集合

15. 为什么proxy不能被polyfill

因为proxy不能用Object.defineProperty模拟

websoket

介绍

1. 服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种
2. 建立在TCP协议之上，服务端的实现比较容易
3. 与HTTP协议有着良好的兼容性，默认端口也是80和443，并且握手阶段采用HTTP协议，因此握手不容易被屏蔽，能通过各种HTTP代理服务器
4. 数据格式比较轻量，性能开销小，通信高效
5. 可以发送文本，也可以发送二进制数据
6. 没有同源限制，客户端可以与任意服务器通信
7. 协议标识符是ws（如果加密是wss），服务器网址就是URL

简单用法

```
var ws = new WebSocket('')

// 用于指定连接成功后的回调函数
ws.onopen = function () {
    console.log('连接成功')
}
// 用于指定收到服务器数据后的回调函数
ws.onmessage = function(evt) {
  console.log( "Received Message: " + evt.data);
  ws.close();
  ws.send('your message');  // 向服务器发送数据
};
// 用于指定连接关闭后的回调函数
ws.onclose = function(evt) {
```

```
  console.log("Connection closed.");
};

// 用于指定报错时的回调函数
ws.onerror = function(event) {
  // handle error event
};
```