01｜自动清理构建目录产物

- 问题：每次构建时不清理目录，造成构建的输出目录 output 文件越来越多
- 解决方案
  - 方案一通过 npm scripts 清理构建目录
    - rm -rf ./dist && webpack
    - rimraf ./dist && webpack
  - 方案二使用 clean-webpack-plugin
    - 作用默认会删除 output 指定的输出目录
    - 使用

```
plugins: [
  new CleanWebpackPlugin()
]
```

02｜CSS增强：PostCSS插件autoprefixer自动补齐CSS3前缀

- CSS3的属性为什么需要前缀？ 因为浏览器的标准并未统一 有四种内核
  - Trident：-ms - Dege
  - Geko：-moz - 火狐
  - Webkit：-webkit - Chrome、Safari
  - Presto：-o - 欧朋
- 例子

```
.box {
  -ms-border-radius: 10px;
  -moz-border-radius: 10px;
  -webkit-border-radius: 10px;
  -o-border-radius: 10px;
  border-radius: 10px;
}
```

- 使用autoprefixer 插件 CSS 后缀处理器 解决浏览器内核标准不统一的前缀问题
  - 根据 Can｜Use [规则][https://caniuse.com/]
  - 问题与 less和sass不同，less与sass是css的预处理器，是在打包前处理，auto 是在代码生成好之后再进行的一次后置处理

```
module.rules = [
  {
    test: /\.less$/,
    use: [
      'style-loader',
      'css-loader',
      'less-loader',
      {
        loader: 'postcss-loader', // 很强大 还可以做 css module 等
        options: {
          plugins: () => {
            require('autoprefixer')({
              broswers: ["last 2 version", "1%", "IOS 7"]
            })
          }
        }
      }
    ]
  }
]
```

03｜移动端CSS px自动转换成rem

- 起因浏览器的分辨率 不同设备的尺寸不同
- 解决方案演进
  - 方案一：CSS 媒体查询实现响应式布局
    - 缺陷需要写多套适配样式代码
  - 方案二：CSS3 之后出现的rem单位
    - rem 是什么？ W3C对 rem的定义：font-size of the root element
      - rem 和 px 的对比
        - rem 是相对单位
        - px 是绝对单位
    - 使用px2rem-loader 首页渲染时计算根元素的 font-size 值

- 可以使用 手套的 [lib-flexible 库][https://github.com/amfe/lib-flexible] 动态计算rem单位

```
{
  test: /\.less$/,
  use: [
    'style-loader',
    'css-loader',
    'less-loader',
    {
      loader: 'px2rem-loader',
      options: {
        remUnit: 75,
        remPrecision: 8
      }
    }
  ]
}
```

04 | 静态资源内联

- 意义两个层面：代码 & 请求
  - 代码层面
    - 页面框架的初始化脚本
    - 上报相关打点
    - css 内联避免页面闪动（首屏内容更快）
  - 请求层面：减少http网络请求数
    - 小图片或字体内联（url-loader）
- HTML 和 JS 内联 使用 raw-loader 必须是0.5.1版本 cnpm i -D raw-loader@0.5.1

```
<!--  raw-loader 内联 html -->
<script>${require('raw-loader!babel-loader!./meta.html')}</script>
<!-- raw-loader 内联 js -->
<script>${require('raw-loader!babel-loader!./node_modules/lib-flexible')}</script>
```

- CSS 内联
  - 方案一：借助 style-loader 更新了 按照最新的 npm 包来
  - 方案二：html-inline-css-webpack-plugin

```
  {
    loader: 'style-loader',
    options: { injectType: 'singletonStyleTag' },
  }
```

05 | 多页面打包（MPA）通用方案

- 多页面应用（MPA）概念
  - 每一次页面跳转的时候，后台服务器都会返回一个新的html文档，这种类型的网站也就是多页网站，也叫多页应用
- 优势是什么？ 1、天然接耦 2、对SEO更加友好
- 基本思路每一个页面对应一个 entry 一个 html-webpack-plugin
- 缺点每次新增或删除页面需要更改 webpack 配置
- 多页面打包 通用方案动态获取 entry 和 设置 html-webpack-plugin 数量
- 利用原理利用 glob.sync 约定好所有页面都放在 src 目录下 入口文件都是 index.js

```
{
  entry: glob.sync(path.join(__dirname, './src/-/index.js'))
}
{
  entry: {
    index: './src/index/index.js',
    search: './src/search/index.js'
  }
}
```

```
module.exports = {
  entry: {
    index: './src/index.js',
    search: './src/search.js'
  }
}
```

06 | 使用sourcemap 🌟

- 意义通过 source map 定位到源代码 传送门
- 使用开发环境开启，线上环境关闭
  - 线上排查问题的时候可以将 sourcemap 上传到错误监控系统

```
{
  // ...
  devtool: 'source-map'
}
```

- source map 关键字一共五个
  - eval：使用 eval 包裹模块代码
  - source map：产生.map文件

    - cheap：不包含列信息（发生错误 只能定位到列 不能定位到行）
    - inline：将 .map 作为 DataURL嵌入，不单独生成.map文件
    - module：包含 loader 的 source map
- source map 类型很多 十几种

devtool	首次构建	二次构建	是否适合生产环境	可以定位的代码
(none)	+++	+++	yes	最终输出的代码
eval	+++	+++	no	webpack生成的代码(一个个的模块)
cheap-eval-source-map	+	++	no	经过loader转换后的代码(只能看到行)
cheap-module-eval-source-map	o	++	no	源代码(只能看到行)
eval-source-map	--	+	no	源代码
cheap-source-map	+	o	yes	经过loader转换后的代码(只能看到行)
cheap-module-source-map	o	-	yes	源代码(只能看到行)
inline-cheap-source-map	+	o	no	经过loader转换后的代码(只能看到行)
inline-cheap-module-source-map	o	-	no	源代码(只能看到行)
source-map	--	--	yes	源代码
inline-source-map	--	--	no	源代码
hidden-source-map	--	--	yes	源代码

07 | 提取页面公共资源 🌟

- 基础库分离
- 方案一：使用 htm-webpack-externals-plugin 插件
    - 思路将 react、react-dom 基础包 通过 cdn 引入，不打入 bundle中
    - 方法使用 htm-webpack-externals-plugin
    - 配置

```
plugins: [
  new HtmlWebpackExternalsPlugin({
    {
      module: 'react',
      entry: '',
      global: 'React'
    },
    {
      module: 'react-dom',
      entry: '',
      global: 'ReactDOM'
```

```
    }
  })
]
```

- 方案二：利用 SplitChunksPlugin 进行公共脚本分离
  - webpack4 内置的 替代 CommonsCHunkPlugin 插件
  - chunks 参数说明
    - async 异步引入的库进行分离（默认）
    - inital 同步引入的库进行分离
    - all 所有引入的库进行分离（推荐）
  - 配置

```
module.exports = {
  // ...
  optimization: {
    splitChunks: {
      chunks: 'async',
      minSize: 30000, // 分离的包体积的大小 单位 字节 30k 的大小
      maxSize: 0,
      minChunks: 1, // 引用次数大于1 就提取
      maxAsyncRequests: 5,
      maxInitialRequests: 3,
      automaticNameDelimiter: '~',
      name: true,
      cacheGroups: {
        commons: {
          test: /(react|react-dom)/, // 匹配出需要分离的包
          name: 'vendors', // 分离出来的名称 vendors
          chunks: 'all',
          minChunks: 2, // 设置最小引用次数为2次
        },
        vendors : {
          test: /[\\/]node_modules[\\/]/,
          priority: -10
        },
      }
    }
  }
}
```

- 方案三：利用 SplitChunksPlugin 分离页面公共文件
  - minChunks：设置最小引用次数为2次
  - minSize：分离的包体积的大小

```
optimization: {
  splitChunks: {
    minSize: 0,
    cacheGroups: {
```

```
        commons: {
          name: 'commons',
          chunks: 'all',
          minChunks: 2
        }
      }
    }
  }
```

08 | Tree Shaking(摇树优化) 的使用和原理分析

- 概念1个模块可能有多个方法，只要其中的某个方法使用到了，则整个文件都会被打包到bundle里面去，tree shaking 就是只把用到的方法打入 bundle ，没用到的方法会在 uglify 阶段被擦除掉
- 使用webpack 默认支持，在 .babelrc 里面 设置 modules:false 即可（webpack2 开始就支持了）
  - production 生产环境下 默认开启
- 要求必须是ES6的语法 CJS的方式不支持 且不能有副作用
- 条件DCE（Elimination 消除）
  - 代码不会被执行、不可到达
  - 代码执行的结果不会被用到
  - 代码智慧影响死变量（只写不读）

```
if (false) {
  console.log('这段代码永远不会执行')
}
```

- 原理利用ES6 模块的特点
  - 只能作为模块顶层的语句出现
  - import 的模块名只能是字符串常量（不能动态去设置需要 import 的类型）
  - import binding 是 immutable（一成不变 / 不能修改） 的
  - 代码擦除：uglify 阶段删除无用代码
- 本质最本质的还是 做静态分析

09 | Scope Hoisting 使用和原理分析

- 现象构建后的代码存在大量闭包代码
- 问题体积增大 & 内存开销大
  - 大量函数闭包 包裹代码，导致体积增大（模块越多越明显）
  - 运行代码时创建的函数作用域变多，内存开销变大
- 原因分析
  - 模块转换分析

```
/- 模块 源代码 -/
import { helloworld } from './helloworld'
import '../../common'
```

```
document.writ(helloworld())
/- 模块初始化函数 编译后 -/
/---/ (function(module, __webpack_exports__, __webpack_require__)
{
'use strict';
__webpack_require_.r(__webpack_exports__);
/- harmony import 指的是ES6 -/  var
_common__WEBPACK_IMPORTED_MODULE_0 = __webpack_require__(1); // 1
指的是 moduleID
/- harmony import -/  var _common__WEBPACK_IMPORTED_MODULE_1 =
__webpack_require__(2); // 2 指的是 moduleID
document.writ(
Object(_helloworld__WEBPACK_IMPORTED_MODULE_1__["helloworld"])
());
})
```

    - 被 webpack 转换后的模块会带上一层包裹
    - import 会被转换成 __webpack_require
  - 进一步分析 webpack 的模块机制

    - 打包出来的是一个IIFE（匿名闭包）
    - modules 是一个数组，每一项是一个模块初始化函数
    - __webpack_require 用来加载模块 返回 module.exports
    - 通过 WEBPACK_REQUIRE_METHOD(0) 启动程序

- scope hoisting 原理
  - 原理将所有模块的代码 按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量防止变量名冲突
  - 作用通过 scope hosting 可以减少函数声明代码和内存开销（解决了构建后代码存在大量闭包代码）
  - 使用webpack4 开始 mode 为 production 默认开始 要求：必须是 ES6语法， CJS 不支持

```
plugins: [
  // ...
  new webpack.optimize.ModuleConcatenationPlugin()
]
```

10 | 代码分割和动态import 🌟

- 代码分割
  - 意义对于大的web应用来将，将所有的代码都放在一个文件中显示是不够有效的，特别是当你的某些代码是在某些特殊时候才会被使用到。webpack 有一个功能那个就是将你的代码分割成chunks（语块），当代码运行到需要它们的时候再进行加载
  - 适用的场景
    - 抽离相同代码到一个共享块
    - 脚本懒加载，使得厨师下载的代码更小（首屏优化）
  - 懒加载JS脚本的方式
    - CommonJS：require.ensure
    - ES6：动态import（目前还没有原声支持，需要babael转换）
      - 动态import是声明含义？ 比方我们写一个es6 语法的时候呢 我们 import 一个模块 然后 from 一个什么内容 这个时候呢 它是一个静态的 动态的就是我们实际使用到 在if...else 里面再去执行import 功能跟require 优点像 可以让我们通过一些逻辑去按需架子啊
- 如何使用 动态 import？
  - 安装 babel 插件： npm i -D @babel/plugin-syntax-dynamic-import
  - ES6：动态 import

```
{
  // ...
  "plugins": ["@babel/plugin-syntax-dynamic-import"]
}
```

11 | 在webpack中使用ESLint

- ESLint 的必要性？ 为什么要使用ESLint？ -- 及时将一些错误暴露出来 避免发布上线后出现问题 影响用户
  - 例子修改 瓶盖iap 支付配置，将jSON 配置增加了重复的key，导致小部分vivo手机的用户反馈充值页面白屏 问题定位：vivo 手机使用了系统自带的 webview 而没有使用X5内核，解析 JSON 时遇到重复key报错，导致页面白屏，如何避免类似代码问题？
  - 推荐使用 ESlint 对 JS 代码做规范检查
- 行业里面优秀的ESLint 规范实践
  - Airbnb（爱彼迎）：eslint-config-airbnb、eslint-config-airbnb-base 有十几个文件 配置很全面
  - 腾讯
    - alloyteam团队： eslint-config-alloy 大概三四百条规则
    - ivwev团队： eslint-config-ivweb 大概150条规则
- 制定团队的 ESLint 规范 -- 团队规则小于10人完全可以使用团队规范 一旦人数上升 可以根据制定不同规范
  - 不重复造轮子，基于 eslint：recommend 配置并改进
  - 能够帮助发现代码错误的规则，全部开启

- 帮助保持团队的代码风格统一，而不是限制开发体验

制定团队的 ESLint 规范

极客时间

不重复造轮子，基于 eslint:recommend 配置并改进

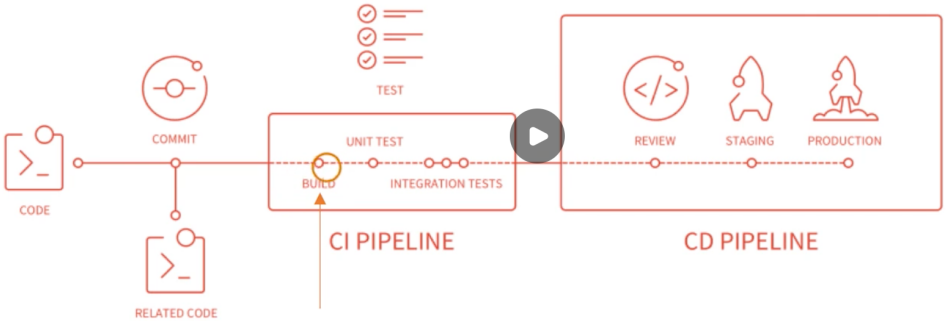
能够帮助发现代码错误的规则，全部开启

帮助保持团队的代码风格统一，而不是限制开发体验

规则名称	错误级别	说明
for-direction	error	for 循环的方向要求必须正确
getter-return	error	getter必须有返回值，并且禁止返回值为undefined, 比如 return;
no-await-in-loop	off	允许在循环里面使用await
no-console	off	允许在代码里面使用console
no-prototype-builtins	warn	直接调用对象原型链上的方法
valid-jsdoc	off	函数注释一定要遵守jsdoc规则
no-template-curly-in-string	warn	在字符串里面出现{和}进行警告
accessor-pairs	warn	getter和setter没有成对出现时给出警告
array-callback-return	error	对于数据相关操作函数比如reduce, map, filter等，callback必须有return
block-scoped-var	error	把var关键字看成块级作用域，防止变量提升导致的bug
class-methods-use-this	error	要求在Class里面合理使用this，如果某个方法没有使用this,则应该申明为静态方法
complexity	off	关闭代码复杂度限制
default-case	error	switch case语句里面一定需要default分支

- ESLint 如何执行落地？
  - 和 CI/CD 系统集成
  - 和 webpack集成 -- 在构建的时候遇到 eslint 错误就终止构建
    - 方案一：webpack 与 CI/CD 集成 -- 增加 lint pipline 在 build之前加入检查

方案一：webpack 与 CI/CD 集成



增加 lint pipline

- 本地开发阶段增加 precommit 钩子
  - 安装 husky：npm i -D husky
  - 增加 npm scrit ，通过 lint-staged 增量检查修改的文件

```
{
  "script": {
    "precommit": "lint-staged"
  },
  "lint-staged": {
    "linters": {
      "-.{js,sass}": ["eslint --fix", "git add"]
    }
  }
}
```

- 方案二：webpack 与 ESLint 集成
  - 使用 eslint-loader 构建时检查 JS 规范
  - 适用的场景 比较推荐在一些新项目中使用 在项目一开始就设置ESLint 不适合老项目接入 在构建的时候会检查所有的 require的模块
  - ESLint 配置

```
module.rules = [
  {
    test: /\.js$/,
    exclude: /node_modules/,
    use: [
      'babel-loader',
      'eslint-loader'
    ]
  }
]
```

```
{
    "parser": "babel-eslint", // ESLint 默认使用Espree作为其解析器，你可以在配置文件中指定一个不同的解析器
    "extends": "airbnb", // 集成 airbnb 的配置
    "env": { // 要在配置文件里指定环境，使用 env 关键字指定你想启用的环境，并设置它们为 true。例如，以下示例启用了 browser 和 Node.js 的环境
        "browser": true,
        "node": true
    },
    "rules": {
        "semi": "error"
    }
}
```

  - 如何修改一些配置 当遇到不符合团队风格时
    - 找到 eslint 对应的 规则配置 进行修改

```
 {
   "rules": {

   }
 }
```

12 | webpack打包组件和基础库

- webpack 除了可以用来打包应用还可以用来打包 js库
- 实现一个大整数加法库的打包（腾讯面试 机考题目 30min内）

    - 需要打包压缩版（适用与开发阶段）和非压缩版本（业务项目 线上打包）
    - 支持 AMD/CJS/ESM 模块引入
- 库的目录结构和打包要求
    - 打包输出的库名称
        - 未压缩版 large-number.js
        - 压缩版本 large-number.min.js
        - 目录结构
            - /dist
                - large-number.js
                - large-number.min.js
            - /src
                - index.js
            - index.js
            - webpack.config.js
            - package.json
- 支持的使用方式
    - 支持 ES Module

```
import - as largeNumber from 'large-number'
// ...
largeNumber.add('999', '1')
```

    - 支持 CJS

```
const largeNumber = require('large-number')
 // ...
largeNumber.add('999', '1')
```

    - 支持 AMD

```
require(['large-number'], function(large-number) {
  // ..
  largeNumber.add('999', '1')
} )
```

    - 直接 script 引入

```
<script src="https://unpkg.com/large-number"></script>
<script>
  // ...
  // global variable
  largeNumber.add('999', '1')
  // property in wendow object
```

```
    window.largeNumber.add('999', '1')
  </script>
```

- 如何将库暴露出去？ 配置好 output
  - library: 指定库的全局变量
  - libraryTarget：支持库引入的方式

```
module.exports = {
  mode: "production",
  entry: {
    "large-number": "./src/index.js",
    "large-number.min": "./src/index.js"
  },
  output: {
    filename: "[name].js",
    library: "largeNumber",
    libraryExport: "default",
    libraryTarget: "umd"
  }
}
```

- 如何指对 .min压缩
  - 安装 npm i -D terser-webpack-plugin
  - 通过 使用 terser-webpack-plugin 的 include 设置 只压缩 min.js 结尾的文件

```
module.exports = {
  mode: 'none',
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        include: /\.min\.js$/
      })
    ]
  }
}
```

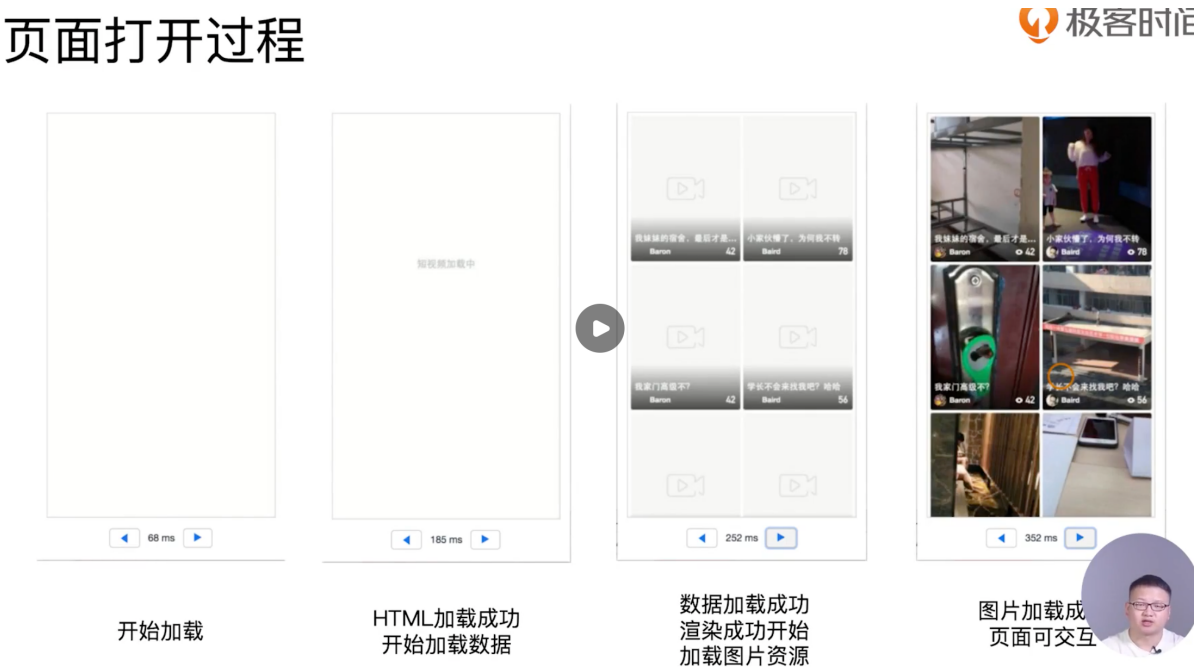
- 设置入口文件
  - package.json 的 main 字段为 index.js

```
if (process.env.NODE_ENV === 'production') {
  module.exports = require('./dist/large-number.min.js')
} else {
  module.exports = require('./dist/large-number.js')
}
```

- 发布 包
  - npm login 登录npm账号
  - npm publish
  - npm unpublish package-name --force

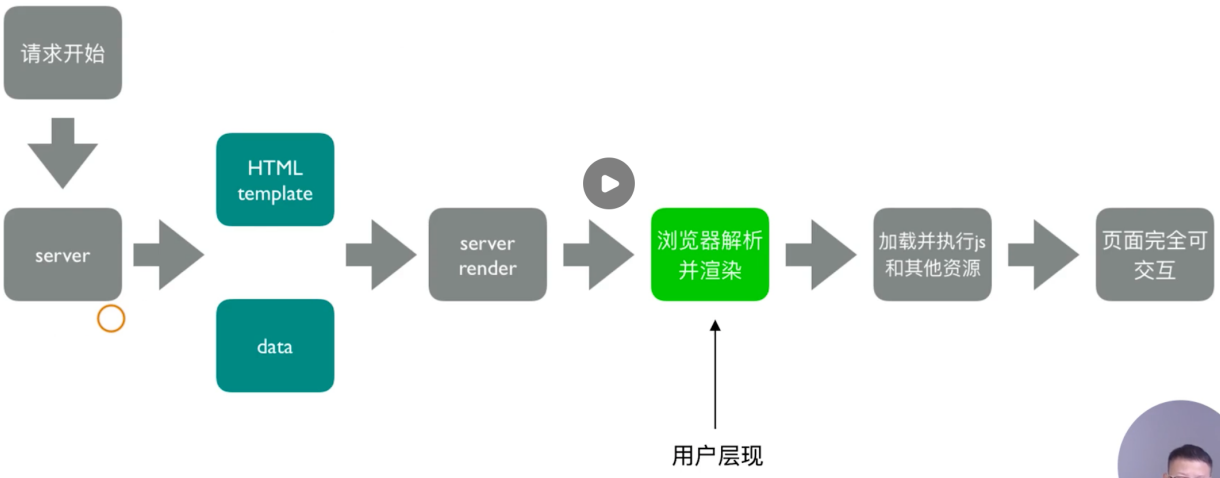
13 | webpack实现SSR打包（上）

- 页面打开过程 串行过程
  - 1.开始加载 -> 2.HTML加载成功 -> 3.数据加载成功、渲染成功、加载图片资源 -> 4.图片加载成功 & 页面可交互



- 服务端渲染（SSR）是什么？
  - 渲染HTML + CSS + JS + Data -> 渲染后的HTML
  - 服务端
    - 所有模板等资源都存储在服务端
    - 内网机器拉取数据更快
    - 一个 HTML 返回所有数据

浏览器和服务器交互流程



-
- 客户端渲染 VS 服务端渲染（小结：SSR 的核心是减少请求）

对比	客户端渲染	服务端渲染
请求	多个请求（HTML、数据等）	1个请求
加载过程	HTML & 数据串行加载	1个请求返回HTML & 数据
渲染	前端渲染	服务端渲染
可交互	图片静态资源加载完成，JS逻辑执行完成可交互 1个请求	

- SSR 的优势
  - 减少白屏时间
  - 对于 SEO 友好（空的 HTML文件 ）
- SSR 代码实现思路
  - 服务端
    - 使用 react-dom/server 的 renderToString 方法将 React 组件渲染成字符串
    - 服务端路由返回对应的模板
  - 客户端
    - 打包出针对服务端的组件

```
const express = require('express')
const { renderToString } = require('react-dom/server')
const SSR = require('../dist/search-server')

server(process.env.PORT || 3000)

function server(port) {
  const app = express()

  app.use(express.static('dist'))
  app.get('/search', (req, res) => {
```

```
    console.log('Server response template ', renderToString(SSR))
    res.status(200).send(renderMarkup(renderToString(SSR)))
  })

  app.listen(port, () => {
    console.log('server is running on port :', port)
  })
}

function renderMarkup(html) {
  return `<!DOCTYPE html>
  <html>
  <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-
scale=1.0">
      <title>服务端渲染</title>
  </head>
  <body>
      <div id="root">${html}</div>
  </body>
  </html>`
}
```

- webpack ssr 打包存在的问题
  - 浏览器的全局变量（Node.js 中没有 document，window）
    - 组件适配：将不兼容的组件根据打包环境进行适配
    - 请求适配：将 fetch 或 ajax 发送请求的写法 改成 isomorphic-fetch 或 axios
  - 样式问题（Node.js 无法解析 css）isomorphic 同构
    - 方案一：服务端打包通过 ignore-loader 忽略掉 CSS 的解析
    - 方案二： style-loader 替换成 isomorphic-style-loader

13 | webpack实现SSR打包（下）

- 如何解决样式不显示的问题？
  - 使用打包出来的 浏览器端html 为模板 设置占位符，动态插入组件

```
<body>
  <div id="root"><!--HTML_PLACEHOLDER--></div>
</body>
```

- 首屏数据如何处理？
  - 服务端获取数据
  - 替换占位符

```
<body>
  <div id="root"><!--HTML_PLACEHOLDER--></div>
```

```
    <!--INITIAL_DATA_PLACEHOLDER  -->
  </body>
```

- 找极客时间 拿到json数据 格式化一下

14 | 优化构建时命令行的显示日志

- 现状：当前构建时的日志显示 展示一大堆日志，很多不需要开发者关注
- 统计信息 stats： 期望 error

Preset	Alternative	Description
"errors-only"	none	只在发生错误时输出
"minimal"	none	只在发生错误或有新的编译时时输出
"none"	false	没有输出
"normal"	true	全部输出
"verbose"	none	标准输出

- 使用设置

```
module.exports = {
  // ...
  stats: 'error-only'
  // 或者在 开发调试阶段时
  devServer: {
      contentBase: './dist',
      hot: true,
      stats: 'errors-only'
  }
}
```

15 | 如何优化命令行的构建日志

- 使用 friendly-errors-webpack-plugin
  - success：构建成功的日志提示
  - warning：构建警告的日志提示
  - error：构建报错的日志提示
- stats 设置成 errors-only
- 构建异常和中断处理
- 如何判读构建是否成功？

    - 在 CI/CD 的 pipline 或者发布系统需要知道当前构建状态
    - 每次构建完成后输入 echo$? 获取错误码 不为0的话 这一次就是失败的
    - webpack4 之前的版本构建失败不会抛出错误码（error code）
    - Node.js 中的process.exit 规范
        - 0 表示成功完成，回调函数中，err 为null
        - 非0 表示执行失败，回调函数中，err不为null，err.code 就是传给 exit的数字

- 如何主动捕获并处理构建错误？

    - compiler 在每次构建结束后会触发 done 这个 hook
    - process.exit 主动处理构建错误

```
plugins: [
  function() {
    // this 指的是 compiler 对象
    this.hooks.done.tap('done', (stats) => {
      if (stats.compilation.errors &&
      stats.compilation.errors.length &&
      process.argv.indexOf('--watch') == -1)
      {
        console.log('build error')
        process.exit(1)
      }
    })
  }
]
```