极客时间

目 录

CONTENTS

极客时间

扫码试看/订阅
《玩转webpack》

当前构建时的问题

每次构建的时候不会清理目录，造成构建的输出目录 output 文件越来越多

通过 npm scripts 清理构建目录

rm –rf ./dist && webpack

rimraf ./dist && webpack

自动清理构建目录

避免构建前每次都需要手动删除 dist

使用 clean-webpack-plugin

- 默认会删除 output 指定的输出目录

```
module.exports = {
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name][chunkhash:8].js',
    path: __dirname + '/dist'
  },
  plugins: [
+   new CleanWebpackPlugin()
};
```

CSS3 的属性为什么需要前缀?

Trident(-ms)

Geko(-moz)

Webkit(-webkit)

Presto(-o)

举个例子

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

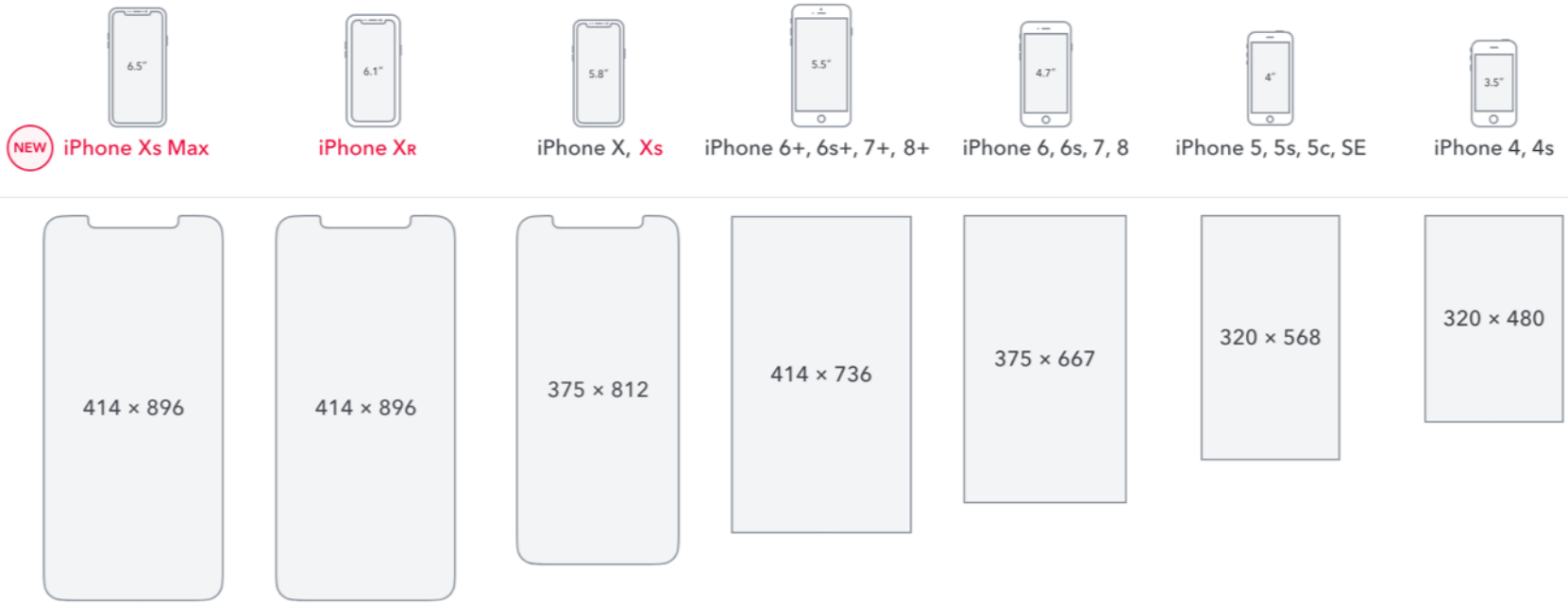
PostCSS 插件 autoprefixer 自动补齐 CSS3 前缀

使用 autoprefixer 插件

根据 Can I Use 规则（ https://caniuse.com/ ）

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          'style-loader',
          'css-loader',
          'less-loader',
+         {
+           loader: 'postcss-loader',
+           options: {
+             plugins: () => [
+               require('autoprefixer')({
+                 browsers: ["last 2 version", "> 1%",  "iOS 7"]
+               })
+             ]
+           }
+         }
        ]
      }
    ]
  }
};
```

浏览器的分辨率


6.5"
NEW iPhone Xs Max
414 × 896
6.1"
iPhone XR
414 × 896
5.8"
iPhone X, Xs
375 × 812
5.5"
iPhone 6+, 6s+, 7+, 8+
414 × 736
4.7"
iPhone 6, 6s, 7, 8
375 × 667
4"
iPhone 5, 5s, 5c, SE
320 × 568
3.5"
iPhone 4, 4s
320 × 480

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

```
@media screen and (max-width: 980px) {
  .header {
    width: 900px;
  }
}
@media screen and (max-width: 480px) {
  .header {
    height: 400px;
  }
}
@media screen and (max-width: 350px) {
  .header {
    height: 300px;
  }
}
```

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

移动端 CSS px 自动转换成 rem

使用 px2rem-loader

页面渲染时计算根元素的 font-size 值

- 可以使用手淘的lib-flexible库
- https://github.com/amfe/lib-flexible

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          'style-loader',
          'css-loader',
          'less-loader',
+         {
+           loader: "px2rem-loader",
+           options: {
+             remUnit: 75,
+             remPrecision: 8
+           }
+         }
        ]
      }
    ]
  }
};
```

资源内联的意义

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

请求层面：减少 HTTP 网络请求数

- 小图片或者字体内联 (url-loader)

HTML 和 JS 内联

极客时间

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

方案二：html-inline-css-webpack-plugin

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: 'style-loader',
            options: {
              insertAt: 'top', // 样式插入到 <head>
              singleton: true, //将所有的style标签合并成一个
            }
          },
          "css-loader",
          "sass-loader"
        ],
      },
    ]
  },
};
```

多页面应用(MPA)概念

每一次页面跳转的时候，后台服务器都会给返回一个新的 html 文档，这种类型的网站也就是多页网站，也叫做多页应用。

多页面打包基本思路

每个页面对应一个 entry，一个 html-webpack-plugin

缺点：每次新增或删除页面需要改 webpack 配置

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

多页面打包通用方案

动态获取 entry 和设置 html-webpack-plugin 数量

利用 glob.sync

- entry: glob.sync(path.join(__dirname, './src/*/index.js')),

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

使用 source map

极客时间

作用：通过 source map 定位到源代码

- source map科普文：http://www.ruanyifeng.com/blog/2013/01/javascript_source_map.html

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

source map 关键字

eval: 使用eval包裹模块代码

source map: 产生.map文件

cheap: 不包含列信息

inline: 将.map作为DataURI嵌入，不单独生成.map文件

module:包含loader的sourcemap

source map 类型

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

基础库分离

·思路：将 react、react-dom 基础包通过 cdn 引入，不打入 bundle 中

·方法：使用 html-webpack-externals-plugin

```
const HtmlWebpackExternalsPlugin = require('html-webpack-externals-plugin');

plugins: [
    new HtmlWebpackExternalsPlugin({
        externals: [
            {
                module: 'react',
                entry: '//11.url.cn/now/lib/15.1.0/react-with-addons.min.js?_bid=3123',
                global: 'React'
            }, {
                module: 'react-dom',
                entry: '//11.url.cn/now/lib/15.1.0/react-dom.min.js?_bid=3123',
                global: 'ReactDOM'
            }
        ]
    });
];
```

```
<!doctype html>
<html lang="zh_CN" style="font-size: 146.5px;">
 ▶<head>…</head>
 ▼<body style="font-size: 18px;">
   ▶<script>…</script>
   ▶<div id="container">…</div>
    <script type="text/javascript" src="//11.url.cn/now/lib/16.2.0/react.min.js?_bid=3123"></script>
    <script type="text/javascript" src="//11.url.cn/now/lib/16.2.0/react-dom.min.js?_bid=3123"></script>
    <script type="text/javascript" src="//s.url.cn/qqun/qun/qunpay/qq/withdraw/income_455d05c8.js?_bid=152"
    TWIaa8ZD/rQZptX8Urp5O2Ef3IT48JbtHS07nW2U= sha384-6E2BbRVmJ2ZLQCQyWHyOYRftEVktwLsaWnC+8h1oyip/0F+6Xa3Lo+
    "anonymous"></script>
  </body>
</html>
```

利用 SplitChunksPlugin 进行公共脚本分离

极客时间

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)
- initial 同步引入的库进行分离
- all 所有引入的库进行分离(推荐)

```
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'async',
      minSize: 30000,
      maxSize: 0,
      minChunks: 1,
      maxAsyncRequests: 5,
      maxInitialRequests: 3,
      automaticNameDelimiter: '~',
      name: true,
      cacheGroups: {
        vendors: {
          test: /[\\/]node_modules[\\/]/,
          priority: -10
        }
      }
    }
  }
};
```

利用 SplitChunksPlugin 分离基础包

test: 匹配出需要分离的包

```
module.exports = {
  optimization: {
    splitChunks: {
      cacheGroups: {
        commons: {
          test: /(react|react-dom)/,
          name: 'vendors',
          chunks: 'all'
        }
      }
    }
  }
};
```

利用 SplitChunksPlugin 分离页面公共文件

minChunks: 设置最小引用次数为2次

minuSize: 分离的包体积的大小

```
module.exports = {
  optimization: {
    splitChunks: {
      minSize: 0,
      cacheGroups: {
        commons: {
          name: 'commons',
          chunks: 'all',
          minChunks: 2
        }
      }
    }
  }
};
```

tree shaking(摇树优化)

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 里设置 modules: false 即可

- production mode的情况下默认开启

要求：必须是 ES6 的语法，CJS 的方式不支持

DCE (Dead code elimination)

代码不会被执行，不可到达

代码执行的结果不会被用到

代码只会影响死变量（只写不读）

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

Tree-shaking 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
- import binding 是 immutable的

代码擦除： uglify 阶段删除无用代码

现象：构建后的代码存在大量闭包代码

```
// a.js
export default 'xxxx';

// b.js
import index from './a';
console.log(index);
```

编译前 (source code)

```
/***/ "./app/index/app.js":
/*!**************************!*\
  !*** ./app/index/app.js ***!
  \**************************/
/*! no exports provided */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
__webpack_require__.r(__webpack_exports__);
/* harmony import */ var _js_index__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*! ./js/index */ "./a

console.log(_js_index__WEBPACK_IMPORTED_MODULE_0__["default"]);

/***/ }),

/***/ "./app/index/js/index.js":
/*!*******************************!*\
  !*** ./app/index/js/index.js ***!
  \*******************************/
/*! exports provided: default */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
__webpack_require__.r(__webpack_exports__);

/* harmony default export */ __webpack_exports__["default"] = ('xxxx');

/***/ })
```

编译后（bundle.js）

会导致什么问题?

大量作用域包裹代码，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

模块转换分析

```
import { helloworld } from './helloworld';
import '../../common';

document.write(helloworld());
```

模块

```
/* 0 */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
__webpack_require__.r(__webpack_exports__);
/* harmony import */ var _common__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(1);
/* harmony import */ var _helloworld__WEBPACK_IMPORTED_MODULE_1__ = __webpack_require__(2);


document.write(Object(_helloworld__WEBPACK_IMPORTED_MODULE_1__["helloworld"])());

/***/ })
```

模块初始化函数

结论：

- 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

```
(function(modules) {
  var installedModules = {};

  function __webpack_require__(moduleId) {
    if (installedModules[moduleId])
      return installedModules[moduleId].exports;
    var module = installedModules[moduleId] = {
      i: moduleId,
      l: false,
      exports: {}
    };
    modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);
    module.l = true;
    return module.exports;
  }
  __webpack_require__(0);
})([
  /* 0 module */
  (function (module, __webpack_exports__, __webpack_require__) {
    ...
  }),
  /* 1 module */
  (function (module, __webpack_exports__, __webpack_require__) {
    ...
  }),
  /* n module */
  (function (module, __webpack_exports__, __webpack_require__) {
    ...
  })
]);
```

分析：

- 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以防止变量名冲突

对比: 通过 scope hoisting 可以减少函数声明代码和内存开销

```
/***/ "./app/index/app.js":
/*!**************************!*\
  !*** ./app/index/app.js ***!
  \**************************/
/*! no exports provided */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
__webpack_require__.r(__webpack_exports__);
/* harmony import */ var _js_index__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(/*! ./js/index */ "./

console.log(_js_index__WEBPACK_IMPORTED_MODULE_0__["default"]);

/***/ }),

/***/ "./app/index/js/index.js":
/*!*******************************!*\
  !*** ./app/index/js/index.js ***!
  \*******************************/
/*! exports provided: default */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
__webpack_require__.r(__webpack_exports__);

/* harmony default export */ __webpack_exports__["default"] = ('xxxx');

/***/ })
```

```
/***/ "./app/index/app.js":
/*!**************************************!*\
  !*** ./app/index/app.js + 1 modules ***!
  \**************************************/
/*! no exports provided */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";

// CONCATENATED MODULE: ./app/index/js/index.js

/* harmony default export */ var js = ('xxxx');
// CONCATENATED MODULE: ./app/index/app.js


console.log(js);

/***/ })
```

scope hoisting 使用

webpack mode 为 production 默认开启

必须是 ES6 语法，CJS 不支持

```
module.exports = {
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name][chunkhash:8].js',
    path: __dirname + '/dist'
  },
  plugins: [
+   new webpack.optimize.ModuleConcatenationPlugin()
};
```

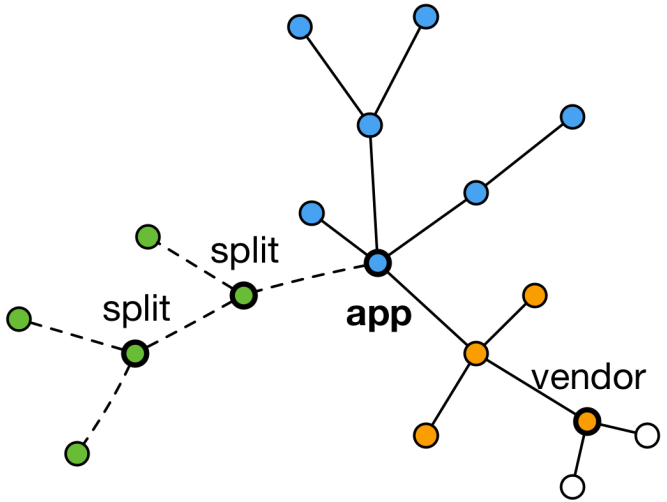
代码分割的意义

对于大的 Web 应用来讲，将所有的代码都放在一个文件中显然是不够有效的，特别是当你的某些代码块是在某些特殊的时候才会被使用到。webpack 有一个功能就是将你的代码库分割成 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，使得初始下载的代码更小



懒加载 JS 脚本的方式

CommonJS：require.ensure

ES6：动态 import（目前还没有原生支持，需要 babel 转换）

如何使用动态 import?

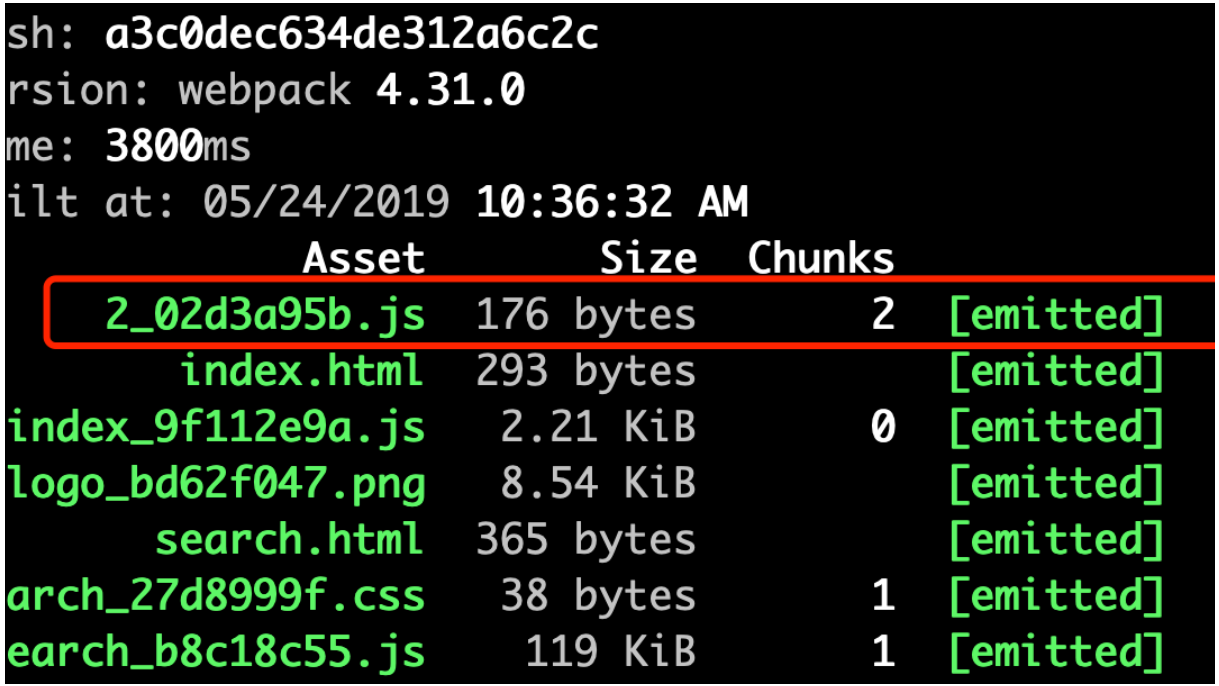
安装 babel 插件

```
npm install @babel/plugin-syntax-dynamic-import --save-dev
```

ES6：动态 import（目前还没有原生支持，需要 babel 转换）

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

代码分割的效果


sh: a3c0dec634de312a6c2c
rsion: webpack 4.31.0
me: 3800ms
ilt at: 05/24/2019 10:36:32 AM
Asset Size Chunks
2_02d3a95b.js 176 bytes 2 [emitted]
index.html 293 bytes [emitted]
index_9f112e9a.js 2.21 KiB 0 [emitted]
logo_bd62f047.png 8.54 KiB [emitted]
search.html 365 bytes [emitted]
arch_27d8999f.css 38 bytes 1 [emitted]
earch_b8c18c55.js 119 KiB 1 [emitted]

ESLint 的必要性

2017年4月13日，腾讯高级工程师小明在做充值业务时，修改了苹果 iap 支付配置，将 JSON 配置增加了重复的 key 。代码发布后，有小部分使用了 vivo 手机的用户反馈充值页面白屏，无法在 Now app 内进行充值。最后问题定位是：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。

行业里面优秀的 ESLint 规范实践

Airbnb: eslint-config-airbnb、 eslint-config-airbnb-base

腾讯：

- alloyteam团队 eslint-config-alloy(https://github.com/AlloyTeam/eslint-config-alloy)
- ivweb 团队：eslint-config-ivweb(https://github.com/feflow/eslint-config-ivweb)

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

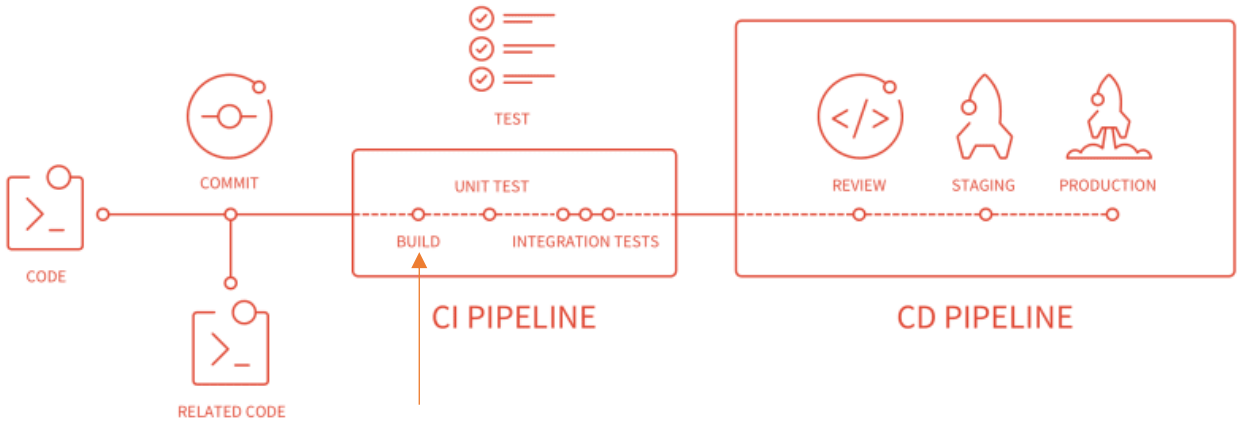
规则名称	错误级别	说明
for-direction	error	for 循环的方向要求必须正确
getter-return	error	getter必须有返回值，并且禁止返回值为undefined, 比如 return;
no-await-in-loop	off	允许在循环里面使用await
no-console	off	允许在代码里面使用console
no-prototype-builtins	warn	直接调用对象原型链上的方法
valid-jsdoc	off	函数注释一定要遵守jsdoc规则
no-template-curly-in-string	warn	在字符串里面出现{和}进行警告
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

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 install husky --save-dev
```

增加 npm script，通过 lint-staged 增量检查修改的文件

```
"scripts": {
  "precommit": "lint-staged"
},
"lint-staged": {
  "linters": {
    "*.{js,scss}": ["eslint --fix", "git add"]
  }
},
```

方案二：webpack 与 ESLint 集成

使用 eslint-loader，构建时检查 JS 规范

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

webpack 打包库和组件

webpack 除了可以用来打包应用，也可以用来打包 js 库

实现一个大整数加法库的打包

- 需要打包压缩版和非压缩版本
- 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 large-number.min.js

```
+ |- /dist
+   |- large-number.js
+   |- large-number.min.js
+ |- webpack.config.js
+ |- package.json
+ |- index.js
+ |- /src
+   |- index.js
```

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

支持 AMD

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

支持的使用方式

可以直接通过 script 引入

```
<!doctype html>
<html>
 ...
 <script src="https://unpkg.com/large-number"></script>
 <script>
  // ...
  // Global variable
  largeNumber.add('999', '1');
  // Property in the window object
  window. largeNumber.add('999', '1');
  // ...
 </script>
</html>
```

如何将库暴露出去?

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

通过 include 设置只压缩 min.js 结尾的文件

```
module.exports = {
  mode: "none",
  entry: {
    "large-number": "./src/index.js",
    "large-number.min": "./src/index.js"
  },
  output: {
    filename: "[name].js",
    library: "largeNumber",
    libraryTarget: "umd"
  },
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        include: /\.min\.js$/,
      }),
    ],
  }
};
```

设置入口文件

package.json 的 main 字段为 index.js

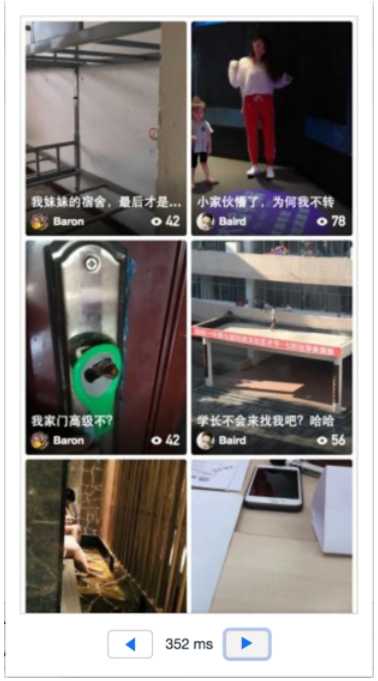
```
if (process.env.NODE_ENV === "production") {
    module.exports = require("./dist/large-number.min.js");
} else {
    module.exports = require("./dist/large-number.js");
}
```

页面打开过程

开始加载

HTML加载成功
开始加载数据

数据加载成功
渲染成功开始
加载图片资源



图片加载成功
页面可交互

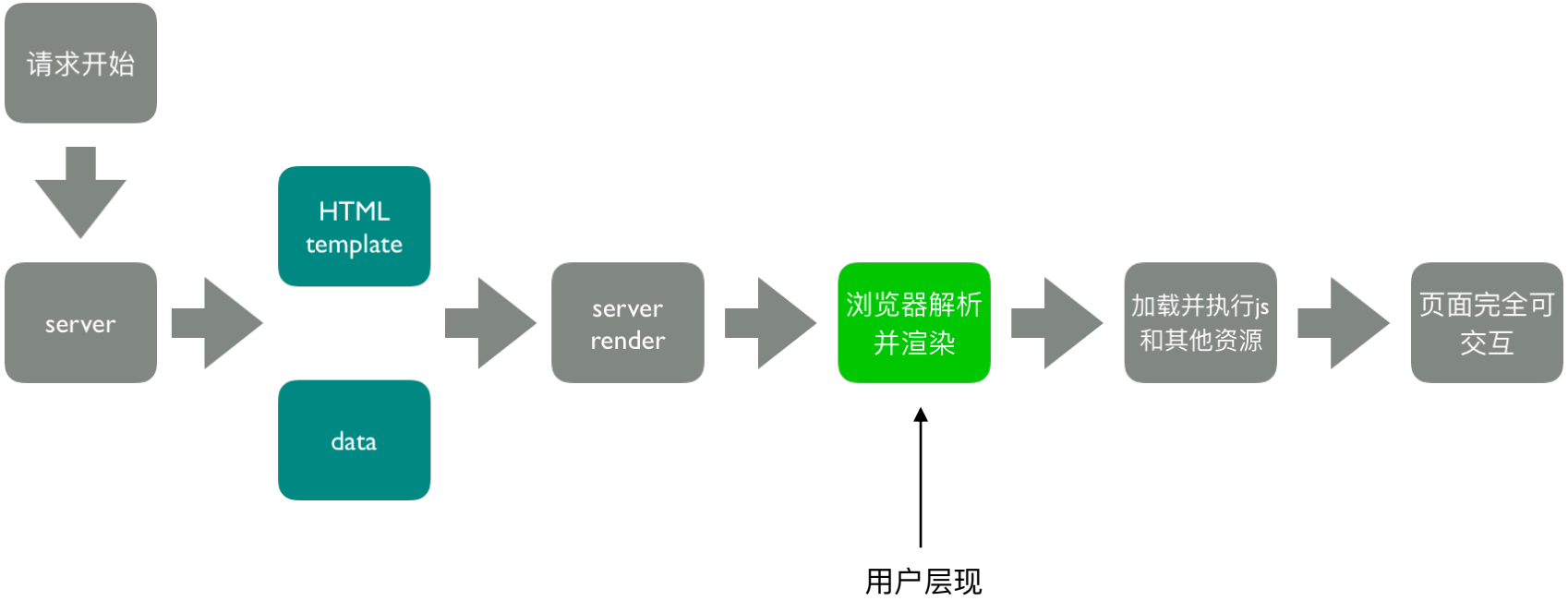
服务端渲染 (SSR) 是什么?

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

	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	图片等静态资源加载完成，JS逻辑执行完成可交互	

总结：服务端渲染 (SSR) 的核心是减少请求

SSR 的优势

减少白屏时间

对于 SEO 友好

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>
    <title>服务端渲染</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div id="app">${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 或者 ajax 发送请求的写法改成 isomorphic-fetch 或者 axios

样式问题 (Node.js 无法解析 css)

- 方案一：服务端打包通过 ignore-loader 忽略掉 CSS 的解析
- 方案二：将 style-loader 替换成 isomorphic-style-loader

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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root"><!--HTML_PLACEHOLDER--></div>
</body>
</html>
```

首屏数据如何处理?

服务端获取数据

替换占位符

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <div id="root"><!--HTML_PLACEHOLDER--></div>

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

当前构建时的日志显示

展示一大堆日志，很多并不需要开发者关注

统计信息 stats

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输出
`"none"`	`false`	没有输出
`"normal"`	`true`	标准输出
`"verbose"`	*none*	全部输出

如何优化命令行的构建日志

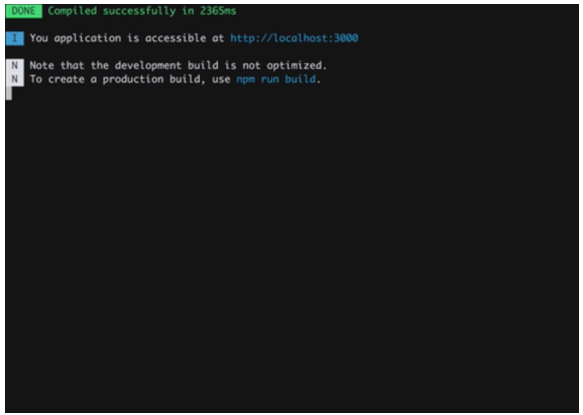
使用 friendly-errors-webpack-plugin

- 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

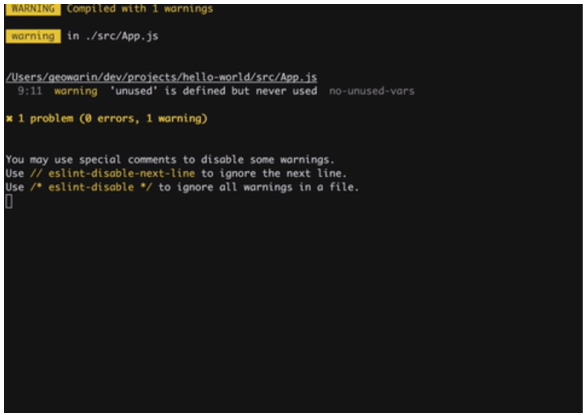
stats 设置成 errors-only

```
module.exports = {
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name][chunkhash:8].js',
    path: __dirname + '/dist'
  },
  plugins: [
+   new FriendlyErrorsWebpackPlugin()
  ],
+ stats: 'errors-only'
};
```

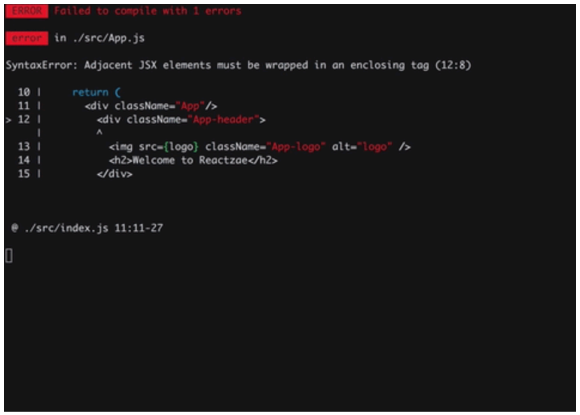
使用效果



构建成功



构建警告



构建失败

如何判断构建是否成功?

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

构建异常和中断处理

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 主动处理构建报错

```
plugins: [
  function() {
    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);
      }
    })
  }
]
```

极客时间

扫码试看/订阅
《玩转webpack》