Webpack实战：入门、进阶、调优分享

部门：hrg
分享人：皮玉含
日期：2020年11月06日

让 生 活 更 简 单


基础

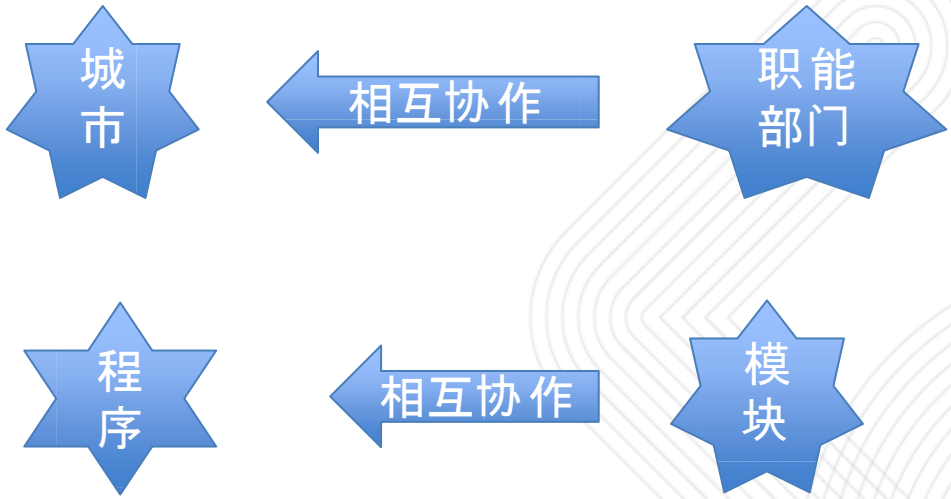
什么是webpack？

Webpack是一个JavaScript模块打包工具，它能够解决模块之间的依赖，把各个模块按照特定的规则和顺序组织在一起，最终合并为一个或多个能够直接在浏览器上运行的JS文件。

什么是模块？

- **模块**

把程序比作一个城市，程序中的模块就像城市内部的职能部门一样，每个模块都有特定的功能。协同工作，才能保证程序的正常运转。如工程中引入的npm包，或者提供工具方法的JS文件，都可以称为模块。



模块化解决的问题

无模块化问题

1需要手动维护JavaScript的加载顺序。页面中多个script之间依赖关系很不明确。

2每一个script标签都需要向服务器请求一次静态资源，过多的请求会严重拖慢网页的渲染速度。

3在每个script标签中，顶层作用域即全局作用域，很容易造成全局作用域的污染

模块化解决问题

1导入和导出语句，使模块间依赖关系清晰易懂

2可以借助工具来进行打包，在页面中只需要加载合并后的资源文件，减少了资源的消耗。

3多个模块之间的作用域是隔离的，彼此不会有命名冲突。

Common.js

每个文件是一个模块，模块会形成一个属于模块自身的作用域，所有的变量及函数只有自己能访问，对外是不可见的。

- 导出

module.export = {...}

模块内部会有一个module对象用于存放当前模块的信息, exports用来指定该模块要对外暴露哪些内容。

- 导入

const xxx = require('...');

require的模块首次加载，会首先执行该模块，再导出内容。再次加载，直接导出内容。Module对象中有loaded属性记录模块是否被加载过。

模块标准

ES6

每个文件是一个模块，自动采用严格模式.

- 导出

export default {...} 默认导出，只能有一个
export {xxx...} 命名导出

- 导入

import xxx(随意命名) from ‘xx’//默认导出
import {xxx} from ‘xxx’//命名导出导入时名称一致，可as重命名

区别

	Common.js	es6
依赖关系解决方案	动态的，在运行阶段确定	静态的，编译时即可确定
导入获得结果	值的拷贝，可修改	值的动态映射，不可修改

Es6优点

1 死代码检查和排除

2 代码类型检查，模块之间传递的值或接口类型是正确的

3编译器优化，eS6 支持直接导入变量，减少了引用层级，程序效率更高。

4解决循环依赖，只是需要由开发者来保证当导入的值被使用时已经设置好正确的导出值。

Webpack优点

1.支持多种模块标准，如common.js 、es6、AMD等等。

2.Webpack有完备的代码分割（code splitting）解决方案.它可以分割打包后的资源，首屏只加载必要的部分，不太重要的功能放到后面动态地加载，可以减小资源体积，提升首页渲染速度。

3.可以处理各种类型的资源,如vue、ts、scss等。

4拥有庞大的社区支持，有无数开发者来为它编写周边插件和工具

Webpack打包后内容

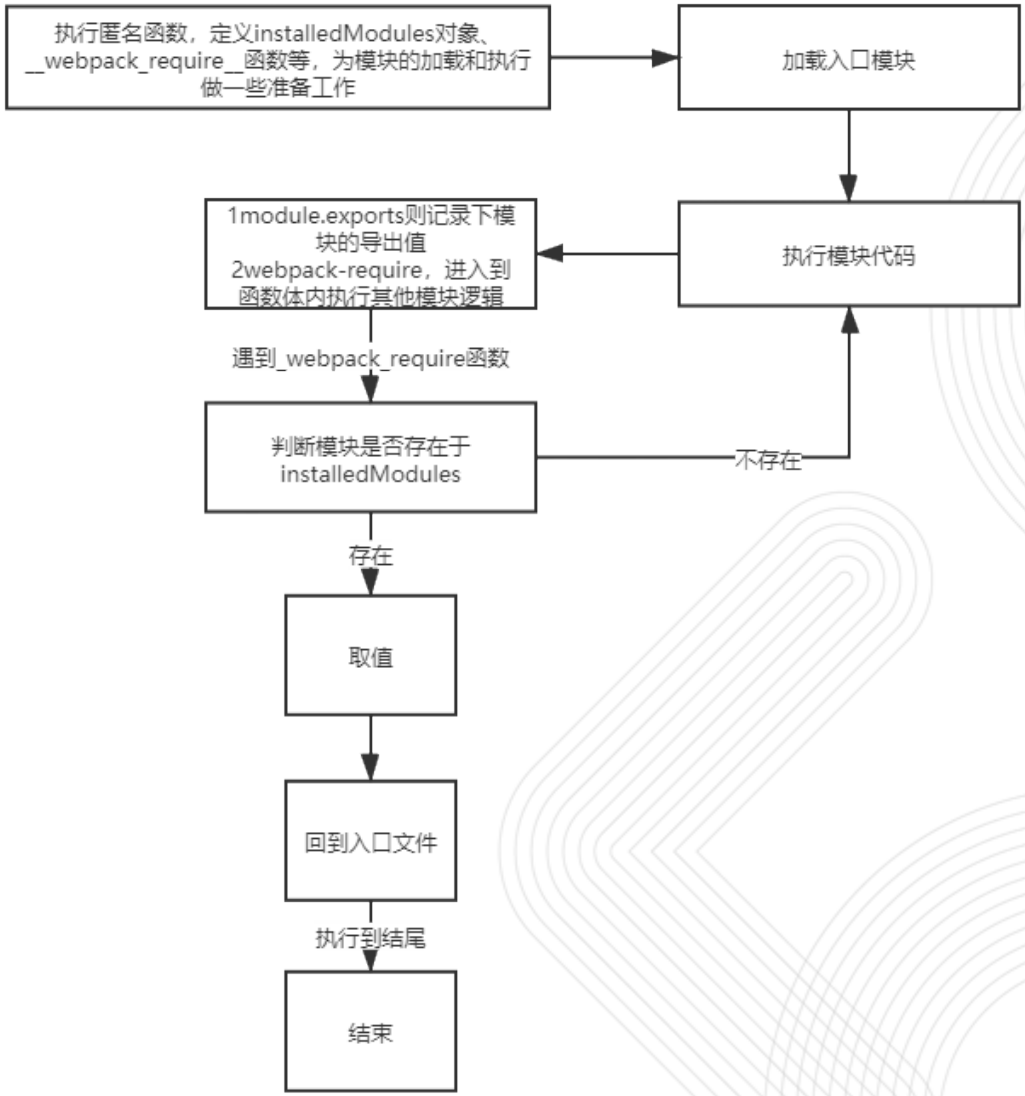
```
// 立即执行匿名函数
(function(modules) {
    //模块缓存
    var installedModules = {};
    // 实现require
    function __webpack_require__(moduleId) {
        ...
    }
    // 执行入口模块的加载
     return __webpack_require__(__webpack_require__.s
= 0);
})({
    // modules: 以key-value的形式储存所有被打包的模块
     0: function(module, exports, __webpack_require_
_) {
        // 打包入口
           module.exports = __webpack_require__("3qi
v");
    },
    "3qiv": function(module, exports, __webpack_requ
ire__) {
        // index.js内容
    },
    jkzz: function(module, exports) {
        // calculator.js 内容
```

最外层立即执行匿名函数。它用来包裹整个bundle，并构成自身的作用域。

installedModules对象。每个模块只在第一次被加载的时候执行，之后其导出值就被存储到这个对象里面，当再次被加载的时候直接从这里取值，而不会重新执行.

modules对象。工程中所有产生了依赖关系的模块都会以key-value的形式放在这里。key为一个模块的id，由数字或者一个很短的hash字符串构成；value则是由一个匿名函数包裹的模块实体

浏览器运行bundle文件流程



Webpack基础配置

- 基础概念：具有依赖关系的模块会在打包是封装为一个chunk，根据配置不同，一个程序会生成一个或多个chunk。
- 基础配置

```
module.export ={
    context: path.join(_dirname,'./src'),//入口路径前缀，可省略，默认为根目录
    entry: './',  //入口文件，可为字符串，数组，对象。可定义chunk名称，单文件默认为main
    //entry: ['react','react-dom','react-route']  预先合并，最后一个元素作为实际入口
    //entry:{
        // app: './src/index.js',  app和vendor即为chunk名称
        // vendor:  ['react','react-dom','react-route']
   // }
   output: {
       path: path.join(_dirname ,'./dist/js'),//资源生成路径，绝对路径，可省略，默认dist
       filename: 'bundle.js'//生成资源文件名称，字符串，相对路径，模板语言为[id][hash][name][chunkhash]
       publicPath: './js/' //间接资源的请求位置，页面中js或css请求的资源加字体或异步加载的js 分为html相关，host相关/，CDN相关绝对路径
   },
   mode: 'development', //development、production、none； 前两种模式下时，会自动添加适合于当前模式的一系列配置，减少了人为工作量。
   module:{ //主要用于配置预处理器规则，
       rules: [ {
                test: '/\.scss$/',//匹配上这条规则
            use:[//使用的loader，可链式的， 从后到前
                'style-loader',
                {
                loader: 'css-loader',
                options: {
                sourceMap: true,
                }
                },
                {
                loader: 'sass-loader',
                options:{
                    sourceMap: true // 对模块做一些额外的功能
                    }
                    }
                ],
                exclude: './node_modules/',//排除和包含指定目录下文件，exclude更高
                include: './src/'
            },]
   },
   plugin: [//接收一个插件数组，以在打包的各个环节中添加一些额外任务
    new Analyzer()
   ],
   devServer: {//服务器
       publicPath:'/dist'
    },

}
```

devServer

- 单纯使用**webpack**缺陷

开发时，修改代码后，就要打包，再刷新页面，导致开发效率低下

- 作用

接收浏览器的请求，然后将资源返回。并且源文件进行了更新操作就会自动刷新浏览器，显示更新后的内容，提高了开发效率。

Publicpath配置为资源请求路径。

启动devserver，会进行模块打包并将打包结果存放于内存中。当webpack-dev-server接收到浏览器的资源请求时，它会首先进行URL地址校验，如请求路径和publicpath一致，则返回内存中打包结果，否则读取硬盘中的源文件并将其返回。

loader

- 作用：

webpack只识别js,预处理器loader，将各种类型程序处理为webpack能够接收的形式。

- 常用**loader**介绍

1 babel-loader/@babel-core/@babel-preset:babel和webpack协同工作/babel核心功能/预置器，可根据用户设置的环境自动添加所需的插件和补丁来编译ES6+代码。babel-loader支持从.babelrc文件读取Babel配置.

2ts-loader，连Webpack与Typescript的模块。ts配置单独配置。

3 html-loader用于将HTML文件转化为字符串并进行格式化

4file-loader用于打包文件类型的资源，并返回其publicPath。

5url-loader与file-loader作用类似，可设置文件大小的阈值，当大于时返回publicPath，而小于该阈值时则返回文件base64形式编码。

6vue-loader将组件的模板、JS及样式进行拆分，vue-template-compiler来编译Vue模板

7css-loader: node-sass是真正用来编译SCSS,sass-loader只起粘合作用


优化

样式处理

分离样式文件

原因：
生产环境下，样式存在于CSS文件中更有利于客户端进行缓存。故我们需要提取样式到CSS文件中。

解决方案：使用mini-css-extract-plugin

```
module.export ={
    ...
    modules:{
        rules:[{
            test: '/\.css$/',
            use:[{
                loader: MiniCssExtractPlugin.loader,
                options:{
                    publicPath: '../',//异步css资源加载路劲
                }
            },'css-loader']
        }]
    },
    pulgin: [
        new MiniCssExtractPlugin({
            filename: '[name].css',//同步资源css名称
            chunkFilename: '[id].css'//异步资源css名称
        })
    ]
}
```

原因：
利用语言强大和便捷的特性，可以降低项目的开发和维护成本如sass和less。

postCss:接收样式源代码并交由编译插件处理，最后输出CSS。单独配置文件。

使用：使用post-loader即可将postcss和webpack连接起来，单独配置文件。

作用：
1. 结合Autoprefixer，添加厂商前缀
2. 结合stylelint，代码风格质量检查
3. 结合CSSNext，使用最新的特性和风格

代码分片

- 作用

把代码按照特定的形式进行拆分，不必一次全部加载，而是按需加载，可以有效降低首屏加载资源的大小，提高性能

- 方案

1入口划分

通常有一些库和工具是不常变动的，把它们放在单独的入口中。因不常更新，故可有效地利用客户端缓存，不必在每次请求页面时都重新加载。

2 SplitChunks

可将多个Chunk中公共的部分提取出来。

优点：减少资源体积；减少重复打包，提高打包速度；更合理的利用缓存

```
module.exports = {
    ...
    optimization: {
        splitChunks: {
            chunks: 'all' //对所有chuank生效，默认异步生效
        }
    }
    ...
}
```

代码分片：splitChunk

·提取后的chunk可被
共享或者来自node_modules目录

提取后的Javascript chunk体积大于30kB（压缩和gzip之前），CSS chunk体积大于50kB

在按需加载过程中，并行请求的资源最大值小于等于5

在首次加载时，并行请求的资源数最大值小于等于3

```
// splitChunks:{
//     chunks:'async',// async（默认）、initial、all  异步chunk 、入口chunk 、两种
//     minSize: {
//         javascript:  30000,
//         style: 50000
//     },
//     maxSize: 0,
//     minChunks: 1,
//     maxAsyncRequests: 5,
//     maxInitialRequest: 3,//提取规则
//     automaticNameDelimiter: '~',//命名分隔符
//     name: true,//根据cacheGroup和上规则生成提取文件名如vendors~a~b~c.js
//     cacheGroups:{//分离chunk的规则
//         vendors:{
//             test: /[\\/]node_modules[\\/]/,//提取node_modules
//             priority: -10,
//         },
//         default:{
//             minChunks:2,//提取多次引用的模块
//             priority: -20,//优先级
//             reuseExistingChunk: true
//         }
//     }
// }
```

代码分片：资源异步加载

- 解决的问题：

当模块数量过多、资源体积过大时，可以把一些暂时使用不到的模块延迟加载，可提高首屏加载速度。也叫做按需加载。

- 异步加载语法

import函数，import函数加载的模块及其依赖会被异步地进行加载，并返回一个Promise对象。不要求出现在顶层作用域，可在任何地方调用。在webpack打包后会生成异步chunk。

- 实现原理

通过JavaScript在页面的head标签里插入一个script标签

- 作用范围

可用在用户切换某些特定路由时，去渲染相应组件。

- 配置异步chunk

可通过配置output.chunkFilename，来设置异步chunk名称

生产环境：配置

生产环境的配置与开发环境有所不同，比如要设置mode、环境变量，为文件名添加chunk hash作为版本号等。

- 生产环境配置文件

一般配置文件有三个，开发配置文件、生产配置文件、公共配置文件，可通过webpack-merge插件，来合并公共配置文件和其他文件

- 生产环境变量

通过definePlugin来配置环境变量，通过环境变量来选择合适的配置。mode：production，已经设置好了，无需再设置

```
new webpack.DefinePlugin({
    process.env.NODE_ENV: 'production',
})
```

生产环境：sourcemap

source map指的是将编译、打包、压缩后的代码映射回源代码的过程。

原理：Webpack对源代码的每一步处理都会生成对应的source map。启用了devtool配置项，source map就会跟随源代码一步步传递，直到生成最后的map文件。

默认名称为打包文件名称.map，并且bundle文件中会标注。打开开发者工具时，map文件会被加载，浏览器使用它来对bundle文件进行解析，分析出源代码的目录结构和内容。在source选项卡中webpack可看到源文件。

```
//# sourceMappingURL=app.js.map
```

```
module.exports ={
    devtool: 'source-map',
    module:{
        rules:[{
            test: /\.scss$/,//scss、less等需要单独配置
            use:[
                'style-loader',
                {
                    loader: 'css-loader',
                    options:{
                        sourceMap:true
                    }
                },
                {
                    loader: 'sass-loader',
                    options:{
                        sourceMap:true
                    }
                }
            ]
        }]
    }
}
```

生产环境：sourceMap隐患

Sourcemap帮助开发者调试源码，当线上有问题产生时也有助于查看调用栈信息，是线上差错的重要信息。但也意味着任何人都可以通过dev tools看到工程源码，有安全隐患。

解决方案：
1hidden-source-map，产出完整的map文件，不会在bundle文件中添加map文件的引用。浏览器自然也无法对bundle进行解析，需借助第三方工具。如Sentry

2nosources-source-map，打包部署之后，可以在开发者工具的Sources选项卡中看到源码的目录结构，但是具体内容会被隐藏起来。能进行简单的错误回溯

3正常打包出source map，然后通过服务器的nginx设置（或其他类似工具）将.map文件只对固定的白名单（比如公司内网）开放

生产环境：压缩

在线上环境前，常都会进行代码压缩，移除多余的空格、换行及执行不到的代码，缩短变量名，在执行结果不变的前提下将代码替换为更短的形式。压缩之后的代码将基本上不可读，在一定程度上提升了代码的安全性。

- **Js**压缩

```
module.export ={
    optimization: {//可自定义配置或直接设置true，production mode无需配置
        minimizer: [
            new TerserPlugin({
                test: /\.js(\?.*)?$/i,
                exclude: /\/excludes/
            })
        ]
    }
}
```

- **Css**压缩

前提： mini-css-extract-plugin将样式提取出来。

使用：

```
module.export ={
    optimization: {
        minimizer: [
            new OptimizeCssAssetsPlugin({
                assetNameRegExp:/\.optimize\.css$/g,//生效范围
                cssProcessor: require('cssnano'),//压缩器
                cssProcessorOptions: {
                    discardComments: {removeAll: true}//压缩器配置
                },
                canPrint: true,//是否展示log

            })
        ]
    }
```

生产环境：缓存

缓存是指重复利用浏览器已经获取过的资源，合理利用缓存，能够提高性能。具体的缓存策略（如指定缓存时间等）由服务器来决定，浏览器会在资源过期前一直使用本地缓存进行响应。

问题：如何获取到最新的资源？
更改资源url，重新获取

1使用**chunkhash**来命名文件

问题：更改后都要手动地去维护html中url的引用路径很麻烦
解决方案：html-webpack-plugin
使用：

```
plugin: [
    new HtmlWebpackPlugin({
        index: "./src/index.html"//会在模板中自动引入打包后资
    })
]
```

生产环境：bundle体积监控

对打包输出的bundle体积进行持续的监控，以防止不必要的冗余模块被添加进来。

1VS Code中插件Import Cost对引入模块的大小进行实时监测，寻找一些更小的替代方案或者只引用其中的某些子模块

2webpack-bundle-analyzer，生成一张bundle的模块组成结构图

打包优化

1增加资源

happyPack

happyPack多线程来提升Webpack打包速度

原理：

使用loader将各种资源进行转译处理，一个模块依赖于几个模块，即使几个模块之间没有关系，webpack也只能串行处理。开启多个线程，并行地对不同模块进行转译。HappyPack适用于那些转译任务比较重的工程。

使用：

使用happypack中loader替换原有loader，并在plugin中使用happckPack插件来配置。多个loader需要标注id

```
loader: 'happypack/loader?id=ts',
```

```
new HappyPack({
  id: 'js',
  loaders: [{
    loader: 'babel-loader',
    options: {}, // babel options
  }],
}),
```

打包优化

2 缩小范围

- 使用exclude/include来缩小打包范围
- noParse忽略，webpack完全不会去解析，仍会打包到文件中

```
module: {
  noParse: /lodash/,
}
```

- IgnorePlugin完全排除，被排除的模块即便被引用了也不会被打包。如moment.js中的本地化的语言包。

```
new webpack.IgnorePlugin({
  resourceRegExp: /^\.\/locale$/, // 匹配资源文件
  contextRegExp: /moment$/, // 匹配检索目录
})
],
```

- Cache有些loader会有一个cache配置项，在编译代码后会缓存，下一次编译前会先检查源文件是否更新，进而是否重新编译或使用缓存

供 DLLReferencePlugin 映射到相关的依赖上去的

打包优化:DllPlugin

DllPlugin 第三方或者不常变化的模块，将它们预先编译和打包，在项目实际构建过程中直接取用即可。需单独配置文件.

```
"dll": 'webpack --config webpack.dll.config.js'
```

```
module.exports = {
    entry: ['react'], //要提取文件
    output:{
        path: path.join(_dirname,'dll')
        filename: 'vendor.js',
        library: dllLibraryName
    },
    plugins: [
        new webpack.DllPlugin({
            name: dllLibraryName //到处的dllLibray文字，和上libray对应
            path: path.join(_dirname,'./dll/mainfest.json')//资源清单路径，业务代码打包时进行模块索引
        })
    ]
}
```

运行指令后，生成vendor.js和manifest.json。库代码/提取的库索引

Vendor.js:

```
var dllExample = (function(params) {
    // ...
})(params);
```

立即执行函数声明，内容为使用数组保存的模块，索引值是id

```
{
  "name": "dllExample",
  "content": {
    "./node_modules/fbjs/lib/invariant.js": {
      "id": 0,
      "buildMeta": { "providedExports": true
    },
    ...
  }
}
```

name即为plugin中配置的，用来描

打包优化:DllPlugin 链接

链接到项目中：使用DllReferencePlugin，把依赖的名称映射到id上。在业务代码前引入dll链接库，再在引入主构建流程中即dist中资源 。

```
  plugins: [
    new webpack.DllReferencePlugin({
      manifest: require(path.join(__dirname, 'dll/ma
nifest.json')),
    })
```

```
// index.html
<body>
  <!-- ... -->
  <script src="dll/vendor.js"></script>
  <script src="dist/app.js"></script>
</body>
```

使用流程：执行vendor.js时会声明变量， manifest相当于我们注入app.js的资源地图，在执行app.js时会根据manifest的name， name字段找到名为dllExample的动态链接库library，进而获得内部文件。

打包优化:tree shaking

Es6静态依赖的关系，treeshaking在打包过程中检测工程中没有被引用过的模块，这部分代码将永远无法被执行到，因此也被称为“死代码”。Webpack会对这部分代码进行标记，并在资源压缩时将它们从最终的bundle中去掉。

tree shaking只能对ES6 Module生效。
使用了babel-loader，一定要通过配置来禁用它的模块依赖解析。否则webpack接收的转化过的CommonJS形式，无法tree shaking

```
presets: [
  // 这里一定要加上 modules: false
  [@babel/preset-env, { modules: false }]
],
```

开发环境调优

1 Webpack开发效率插件

- webpack-dashboard更好地展示打包相关信息，需用用webpack-dashboard模块命令替代原本的webpack或者webpack-dev-server的命令
- speed-measure-webpack-plugin，Webpack整个打包过程中在各个loader和plugin上耗费的时间
- Webpack-merge合并module.rules的过程中会以test属性作为标识符，有相同项出现时以后面的规则覆盖前面的规则
- size-plugin每次执行Webpack打包命令后，size-plugin都会输出本次构建的资源体积（gzip过后），以及与上次构建相比体积变化了多少

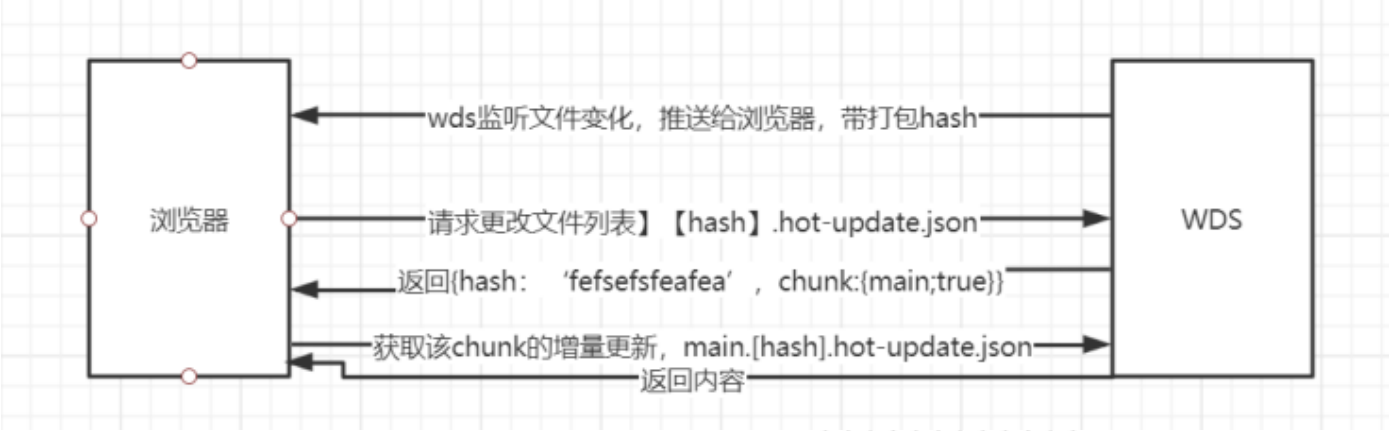
2模块热替换

可以让代码在网页不刷新的前提下得到最新的改动，我们甚至不需要重新发起请求就能看到更新后的效果。在webpack-dev-server模式下。

```
module.exports = {
  // ...
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  devServer: {
    hot: true,
  },
};
```

开发环境调优：热替换原理

HMR的核心就是客户端从服务端拉取更新后的资源 ，为chunk diff，即chunk需要更新的部分。




未来

其他打包工具

1 rollup
全局安装，更专注于js打包，通用性不如webpack

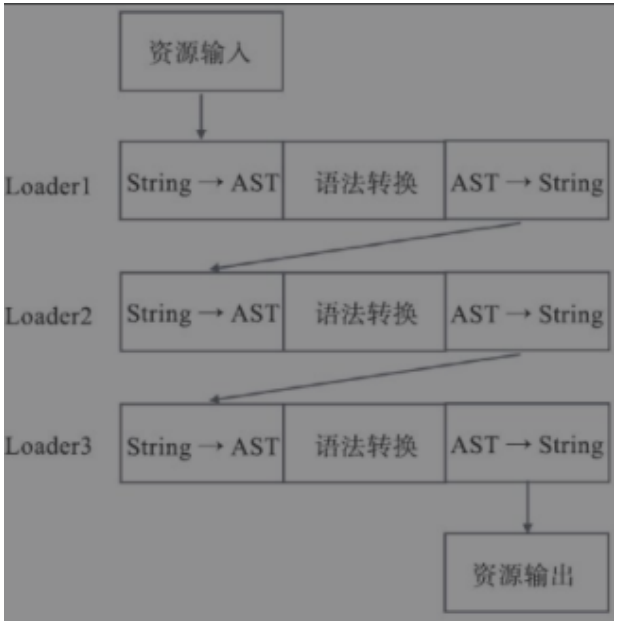
```
rollup -c rollup.config.js
```

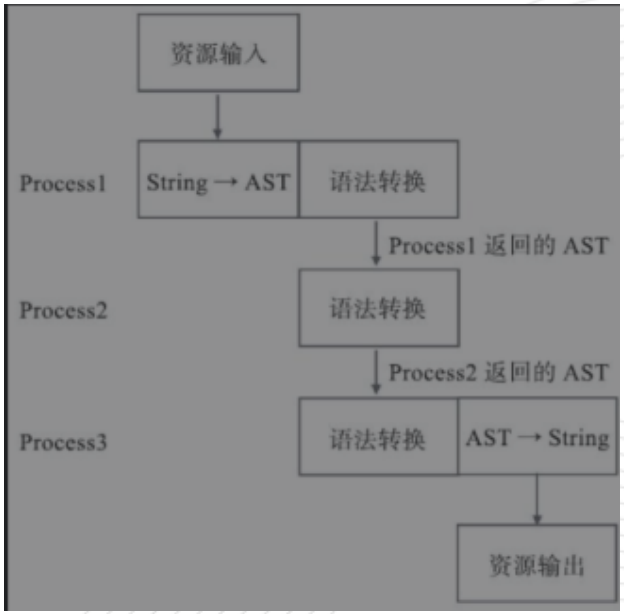
- 打包后文件很干净，就是代码本身内容。Webpack打包时将自身代码注入进去即使项目本身仅仅有一行代码
- tree shaking，基于es6，没有被引用过代码，不会出现在最终的bundle.js中。输出的内容非常清晰简洁，没有附加代码。
- output.format开发者可以选择输出资源的模块形式

2parcel
在Parcel官网的Benchmark测试中，有缓存时其打包速度比Webpack快将近8倍，且宣称自己是零配置的。
优化：用worker来并行执行任务；文件系统缓存；资源编译处理流程优化

前两种，webpack利用多核同时压缩多个资源，在某些loader中也有缓存选项。

Parcel：资源编译处理流程优化





Webpack：涉及大量的String和AST之间的转换，loader只能接受字符串； 不同loader只要完成好各自的工作，有助于保持loader的独立性和可维护性

Parcel：不同的编译处理流程之间可以用AST作为输入输出

Parcel：打包与运行

运行：

```
parcel index.html
```

打包：

```
parcel build index.html
```

打包后文件：自动为其生成了hash版本号及source map，内容压缩

打包工具发展趋势

性能与通用性有时是一对互相制衡的指标。性能上可能就不如那些更加专注于某一个小的领域的工具。新出现的工具的趋势是，专注在某一特定领域，比Webpack做得更好更精，性能更强。

置极小化甚至是零配置逐渐成为了一个重要的特性，背后是 JavaScript工程的标准化，如是源码目录和产出资源目录。比如用src作为源码目录，以dist作为资源输出目录。

性能更高，配置更少。但是还是要根据项目需求来选择合适工具。

Thank you!

皮玉含

PPT中配合内容的ICON