Webpack

是一個開源的模塊打包器，.js、.css、.sass、.jpg等各種檔案視為一個個模塊，將各模塊間的相依關係繪製成相依圖dependency graph，依照相依圖解析並處理每一個模塊，最後建置成一個或多個 bundle(將模塊化編譯成瀏覽器看得懂的模塊)化，bundle可以直接被瀏覽器載入

webpack會將下面語意都視為相依於其他模塊

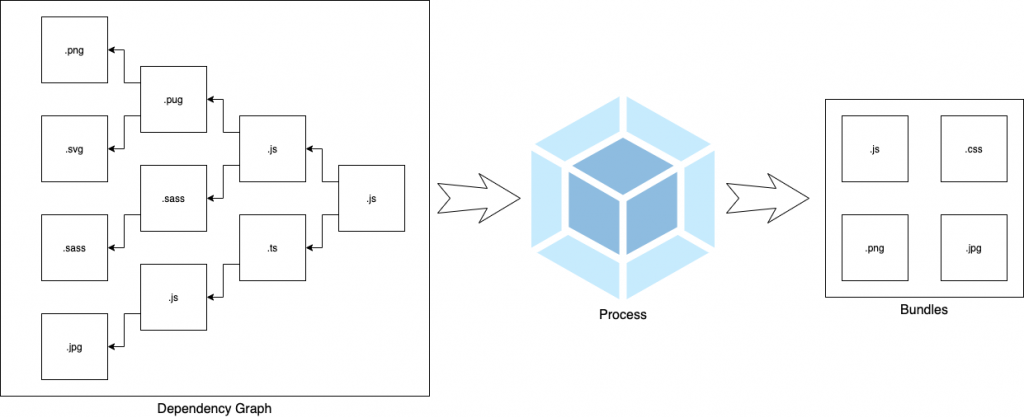
ES2015的import

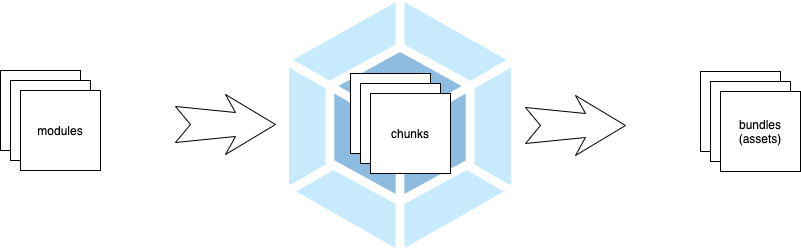
CommonJS的require()

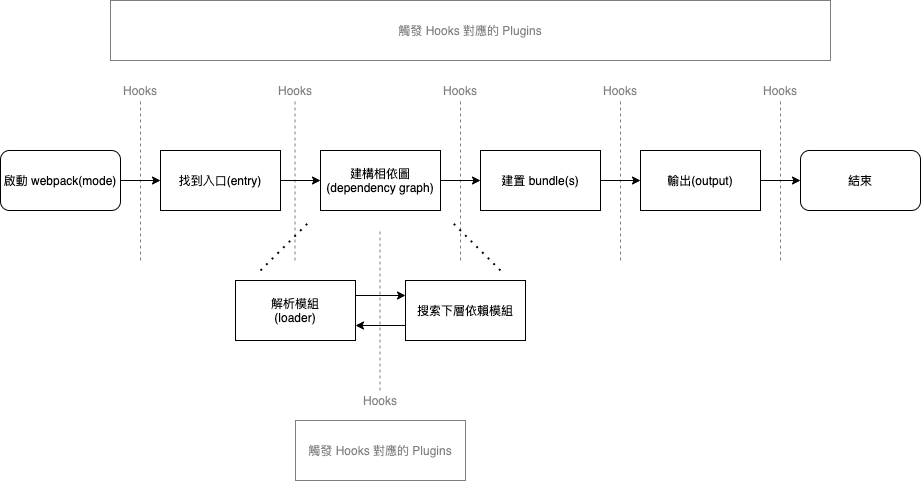
AMD的define、require

CSS的@import與url(...) (需要loader)

HTML的<img src=...> (需要loader)







webpack核心概念

entry入口

entry指示webpack作為構建相依圖的起點模塊，進入entry後，webpack會找出entry直接、間接依賴的模塊和庫，默認值是./src/index.js

multi-main entry

使用陣列的entry將複數個相依圖設定為chunk入口，使不相關的模塊打包在同個Chunk(main.js輸出檔)中

entry設定為物件時，物件中的key為Chunk名稱，因此設定多個key會產生多個Chunk，並產生出各別的bundle

output輸出

output指示webpack所創建的bundle的路徑、名稱，默認路徑是./dist目錄中，而主要bundle名稱默認是./dist/main.js

loader

webpack只能理解JavaScript和JSON文件，loader能讓webpack將其他類型的文件轉譯成webpack讀得懂的模塊，讓webpack可以將此模塊加入相依圖

loader主要用兩個屬性:

test屬性

定義轉換文件

use屬性

定義轉換文件使用的loader

use.options

定義特定loader的配置

enforce屬性

定義loader執行順序(默認由下至上)

pre > normal > inline > post

loader配置步驟:

1.安裝loader

2.設定配置檔

plugin插件

webpack在建置過程中，會依序觸發不同的事件鉤子，藉以完成各個時期的工作， plugin可以藉著這些事件鉤子執行其所設定的工作，包括打包優化、資源管理、注入環境

plugin配置步驟:

1.安裝plugin(如果是內建插件則不用安裝)

2.引入plugin

3.設定配置檔(使用new來初始化plugin)

mode模式

有development、production、none三種，默認值為production

development

將DefinePlugin中process.env.NODE\_ENV的值設置為development

為模塊和chunk啟用有效名稱，而非單純編號

production

比development多一步壓縮js代碼

將DefinePlugin中process.env.NODE\_ENV的值設置為production

為模塊和chunk啟用確定性的混淆名稱，FlagDependencyUsagePlugin、FlagIncludedChunksPlugin、ModuleConcatenationPlugin、NoEmitOnErrorsPlugin、TerserPlugin

none

不使用任何默認優化選項

CommonJS require/exports模塊化語法

導出

exports.fs = fs; //單個特性導出，可導出多個

module.exports = fs; //整個模塊導出，每个模塊只包含一個

引入

const fs = require('fs'); //引入整個模塊

Source Map

在建置後代碼發生Bug時，利用映射(建置後代碼與源代碼相互對應)追蹤源代碼Bug

devtool屬性可以設定Source Map種類:

false

默認值，代表不使用Source Map(無法對應)

eval

可以對應檔名(將各個模塊以eval()包起來，並在最後加上sourceURL)

source-map

可以對應檔名、loader處理前的行數、列數(產生外部map檔，bundle中加上map檔路徑資訊)

inline

可以對應檔名、loader處理前的行數、列數，(用內聯DataUrl代替外部map檔載入map資訊，bundle中加上map檔路徑資訊)

hidden

只有建構後錯誤代碼位置(bundle中沒有map檔路徑資訊(sourceMappingURL註解))

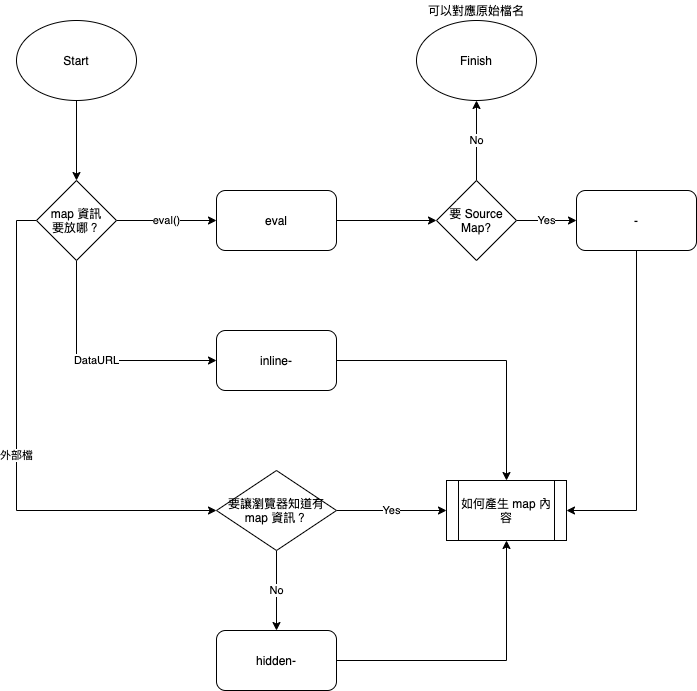
cheap

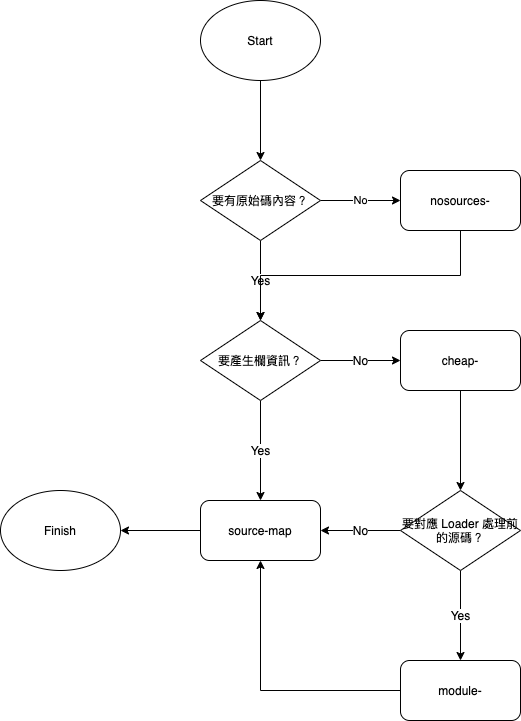
可以對應檔名、loader處理後的行數(sourceContent為Loader載入後的內容)，不會對應列數(mappings中不產生列資訊)

cheap-module

可以對應檔名、loader處理前的行數(sourceContent為Loader載入前的內容)，不會對應列數(mappings中不產生列資訊)

nosource

瀏覽器不會產生錯誤代碼內容(sourceContent內容)，但可以看到代碼錯誤訊息



optimization屬性

如果對mode屬性的預設配置模式不滿，可針對bundle輸出環境做詳細配置

HMR模塊熱替換 Hot Module Replacement

在應用程序、網頁運行時，交換、添加、刪除模塊(在一個模塊修改後，重新打包此模塊)，不需重新加載整個應用程序、網頁，提升建構速度

.css文件:可用HMR，因為style-loader中的module.hot.accept在幕後修補<style>

.js文件:默認不可用HMR，需添加js代碼以支持HMR(只能用於非entry的js文件)

**index.js**

// 如果開啟HMR，則module.hot為true

if (module.hot) {

// 監聽./demoName，只重新打包此文件

module.hot.accept("./demoName", function () {

// 函數，只要demoName更新，就在console輸出新內容

console.log("Accepting the updated demoName module!");

console.log(demoName);

});

.html文件:默認不可用HMR(產生問題: html文件不能HMR)，修改entry，引入.html文件即可

entry: ['./src/index.js', './src/index.html',]

side effect副作用

指當調用函數時，除了回傳函數值之外，還對主調用函數產生附加的影響，例如修改全域變數(函數外的變數)、參數或改變外部存儲

Tree Shaking

將檔案內容中不會執行到的代碼刪去，使用ES2015模塊語法(import、export)

**webpack.config.js**

{

module.exports = {

optimization: {

// 從模塊package.json中的sideEffects判斷，將未使用物件從bundle中去除

sideEffects: true,

},

};

// 生產環境下使用Tree Shaking

mode: 'production'

};

**package.json**

{

// 上面提到的所有代碼都不包含副作用，webpack皆可進行Tree Shaking

// 可標記有副作用的代碼文件(css、less)，避免無意中刪除

"sideEffects": false|["\*.css"]

}

Template String

可將Chunk的資料帶入字串值，以產生不同名稱的輸出(由Webpack內建的TemplatePathPlugin驅動)

[name] : Chunk名稱

[id] : Chunk ID

[hash] : Compilation建構的hash值，(只要建構有改變，就會變化)

[contenthash] : 每個bundle的hash值(只有bundle改變，才會變化)

[chunkhash] : 每個chunk的hash值(只有chunk改變，才會變化)

code split切割代碼

webpack會將從entry模塊開始的所有相依模塊都輸出到同一個bundle中，雖然減少了瀏覽器對伺服器的請求次數，但第一次載入時間會因為bundle體積過大而十分漫長，可以切割代碼，依照使用者點擊適時載入，以加速運作

切割代碼方式有三種:

1.entry:依照entry物件語法(多入口)切割代碼

用在完全不相關的兩個頁面的建置(多頁應用程式)，讓每一頁只讀取所需資源

// 產生index.js、test.js兩個bundle

entry: {

index: "./src/index.js",

test: "./src/test.js",

},

output: {

filename: '[name].[contenthash:10].js',

path: path.resolve(\_\_dirname, 'build'),

},

2.動態引入:藉由js代碼import語法，將文件單獨打包

function getComponent() {

const element = document.createElement('div');

return import('lodash')

.then(({ default: \_ }) => {

const element = document.createElement('div');

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

})

.catch((error) => 'An error occurred while loading the component');

}

getComponent().then((component) => {

document.body.appendChild(component);

});

import()返回promise，可與async函數一起使用來簡化代碼

async function getComponent() {

const element = document.createElement('div');

const { default: \_ } = await import('lodash');

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

getComponent().then((component) => {

document.body.appendChild(component);

});

3.optimization.splitChunks:依照splitChunks配置，用內建SplitChunksPlugin切割代碼

可以將依賴項模塊(node\_modules)，單獨分割(多入口多輸出的多個bundle，可以共用同一模塊bundle，而非在多個bundle中各自加入模塊代碼)

// 兩個bundle各自加入模塊代碼，index.js(30 kb) + test.js(40kb) = 70kb

// 模塊單獨分割，index.js(10 kb) + test.js(20kb) + 模塊(20kb) = 50kb

optimization: {

splitChunks: {

// chunks預設是async，如果模塊非同步引入，webpack會將其拆為另一個bundle

// all模塊同步、非同步引入，皆會將其拆為另一個bundle

chunks: "all",

},

},

漸進式網絡應用程序progressive web application PWA

一種可以提供類似於native app原生應用程序的web app網絡應用程序，在離線offline時應用程序能夠繼續運行功能，通過Service Workers服務工作線程(瀏覽器在後臺獨立於網頁運行的腳本，實現不需要網頁或用戶交互的功能)來實現

Workbox

一組庫，可以為PWA自動化管理service worker

外部擴展externals

提供從輸出的bundle中排除依賴項模塊的方法

**index.html**

<script

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

integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="

crossorigin="anonymous">

</script>

**webpack.config.js**

externals: {

// 藉由externals告知webpack，在模組中使用jQuery，且自行從外部載入，jQuery不需打包

jquery: 'jQuery'

}

動態連結函式庫Dynamic-link library DLL

微軟公司在微軟視窗作業系統中實現共享函式函式庫概念的一種實作方式，把一些經常會共享的程式碼製作成DLL檔，當執行檔呼叫到DLL檔內的函式時，Windows作業系統才會把DLL檔載入記憶體內，副檔名是.DLL、.OCX(包含ActiveX控制的函式庫)、.DRV(舊式的系統驅動程式)

webpack.config.js配置物件Configuration Object

在項目根目錄下創建一個webpack.config.js文件，webpack會自動作為配置文件使用，在使用webpack命令時，會使用其中設置

const path = require('path');

// 導入node.js原生的path模塊

const HtmlWebpackPlugin = require('html-webpack-plugin');

// 導入HtmlWebpackPlugin插件

const devtools = [

false|"source-map"|…

];

// 各種source map

const commonCssLoader = [

"style-loader",

// 將模塊導出的內容作為樣式並添加到DOM中(在<head>中新增<style>，並插入js中的樣式資源)

'css-loader',

// 加載CSS文件並解析import的CSS文件，返回CSS代碼(將css檔以字符串刑式變成CommonJS模塊導入js中)

]

//宣告重複使用loader

module.exports = {

// CommonJS模塊化語法，外部程式可讀到導出的module.exports物件所帶值

// webpack設置

entry: './src/index.js',

// 入口，代表編譯前的檔名

output: {

// 輸出，分為filename、path，編譯結果會在path/filename

// 瀏覽器快取cache更新仰賴檔名的變換，webpack可用hash值，使產出的bundle在內容更新時改變檔名，使瀏覽器更新

// 為了讓快取保留較長，需要將bundle切割，將不同類型且沒關聯的模塊分開快取(瀏覽器可以只更新縮需要的檔案)

// filename用contenthash分隔.js、.css產出的快取(更改其一不需載入所有快取)

filename: 'build.[contenthash:10].js',

// 編譯輸出的檔名

path: path.resolve(\_\_dirname, 'build'),

// 編譯輸出的路徑，必須是絕對路徑

// 使用path模塊中的resolve()把相對路徑轉換成絕對路徑

// \_\_dirname，node.js的變量，代表當前文檔(webpack.config.js)的目錄絕對路徑

clean: true,

// 在每次構建之前清理/dist文件夾，這樣只會生成使用過的文件

},

mode: "production|development|none",

// 模式

watch: true,

// 監聽模式，默認為false，true代表每當文件變化都自動去編譯

// loader設置

module: {

// 模塊編譯設置

rules: [

// 陣列，代表使用loader編譯的規則，不同類型檔案要配置不同的loader

// css rule

{

oneOf: [

// 規則數組，當規則匹配時，只使用第一個匹配規則

// 使用oneOf，不需讓文件把所有rules都詢問一遍，提高loader效率

{

test: /\.css$/,

// regex(正則表達式)，如果檔名符合，就會使用use指定的loader

use: [...commonCssLoader]

// use可以是陣列，順序由下至上

//重複使用loader

},

// less rule

{

test: /\.less$/,

use: [...commonCssLoader, 'less-loader']

},

]

}

]

}

// plugins設置

plugins: [

// 創建默認index.html，在<script>中引入bundle

new HtmlWebpackPlugin(),

// 創建test.html，在<script>中引入bundle

new HtmlWebpackPlugin({

filename: 'test.html',

template: './src /test.html',

minify: true

})

]

};

命令

npm:

安裝

npm install --save-dev|-D webpack webpack-cli

使用npm安裝webpack、webpack cli

npm install –D webpack-dev-server

使用npm安裝webpack-dev-server

webpack-cli:

webpack build|bundle|b(默認命令，可省略)

運行webpack(使用webpack.config.js中的設置)，進行打包後輸出(webpack能處理js/json，不能處理css/img等)

命令行接口cli參數的優先級高於配置文件參數

--entry(默認參數，可省略) <entry>

以此為entry

--output-path|-o <output-path>

以此為output

--mode=<mode>

以此(development|production|none)為整體打包環境

--watch|-w

每當文件變化都自動去編譯

--config|-c <配置文件名稱>

使用指定配置文件

webpack-dev-server:

webpack-dev-server

用於開啟DevServer開發服務器，並啟動監聽模式(自動編譯bundle、刷新瀏覽器)

module種類

webpack-cli

提供webpack的cli，用於在命令行中運行webpack

webpack-dev-server

提供實時重新加載的開發服務器，自動編譯不會在./dist資料夾中產生bundle，而是儲存在Memory內存中

module.exports = {

devServer: {

// contentBase設定伺服器的根目錄

contentBase: path.resolve(\_\_dirname, './dist')

// compress設定是否使用gzip壓縮(常見的HTTP compression壓縮方法，HTTP數據在從服務器發送之前被壓縮)

compress: true,

// port設定用於偵聽請求的端口號

port: 8080,

// open是否在服務器啟動後打開默認瀏覽器

open: true,

// hot是否啟用webpack的HMR功能

//，當配置devServer.hot:true時，webpack-dev-sever內部對HotModuleReplacementPlugin插件做了自動添加此插件的判斷

hot: true,

}

};

loader種類

樣式:

style-loader

將模塊導出的內容以<style>形式添加到HTML中

css-loader

加載CSS文件並解析import的CSS文件，最終返回CSS代碼(CSS -> CSS代碼)

less-loader

加載並編譯LESS文件(LESS -> CSS)

sass-loader

加載並編譯SASS/SCSS文件(SASS/SCSS -> CSS)

載入圖片資源:

file-loader

將靜態資源載入到Webpack內，並且解析資源的相互依賴關係，最後 output到指定的位置

參數

name

設置output時的文件名稱，默認為[contenthash].[ext]

outputPath

指定目標文件的公共路徑，默認為underfined

url-loader

將指定大小上限內的圖片資源(icon圖標等小檔案)轉換為base64格式內聯到bundle中，如遇到超過上限的資源，將fallback給file-loader做處理

優點:減少請求數，減輕伺服器壓力(往後網頁在渲染圖片時，不需要以request加載圖片，直接向JavaScript檔案拿取即可)

缺點:圖片體積較大，請求速度慢(base64存在bundle.js內)

參數(file-loader的參數url-loader皆可用)

limit

限制可轉為base64的檔案大小上限，單位為byte，默認為underfined

fallback

指定當文件大小超過 limit 限制時，需轉向的加載程序，默認為file-loader

esModule

使用ES6模塊語法，默認為true

html-loader

將HTML靜態資源(<img>)導出為字符串(解決url-loader、file-loader默認無法處理HTML靜態資源的問題)

test: /\.(png|jpg|gif)$/,

use: [

{

// url-loader會fallback file-loader，只要寫url-loader就好

loader: 'url-loader',

options: {

// [name]目標文件名稱[ext]目標文件的文件擴展名[hash:10]前十位hash值

name: '[name].[ext]| [hash:10].[ext]',

outputPath : 'images',

limit: 8192|8\*1024,

// 關閉ES6，採用CommonJS，因為html-loader使用CommonJS引入

esModule: false

},

},

{

test: /\.html$/,

use:['html-loader']

},

],

postcss-loader(記得用npm安裝PostCSS)

使用PostCSS(後處理器，一個使用JS插件轉換樣式的工具)處理CSS

PostCSS插件:

css兼容性:

postcss-preset-env

將現代CSS轉換為瀏覽器理解的內容，根據目標瀏覽器或運行環境(browserslist)決定需要的polyfill(一塊代碼，通常是Web上的JS，用於提供舊瀏覽器沒有支持的新功能)

CSS優化:

cssnano

縮小(採用一些代碼並使用各種方法來減小其在磁盤上的大小的過程，與gzip保留CSS原始語義的無損技術不同，是一個固有的有損過程，，最終生成代碼將與原始文件的行為相同，但會更改部分以盡可能減小大小)CSS體積

PostCSS插件配置寫法有以下三種:

1.在**webpack.config.js**中的loader.options.postcssOptions

2.在**postcss.config.js**(一般放置在項目根目錄)

3.在**package.json**中的postcss屬性(不推薦)

**webpack.config.js**

// 設置nodejs環境變量

process.env.NODE\_ENV = "development";

test: /\.css$/i,

use: [

'style-loader',

'css-loader',

{

loader: 'postcss-loader',

options: {

postcssOptions: {

plugins: [

[

// 使用PostCSS插件postcss-preset-env

'postcss-preset-env',

// 使用PostCSS插件cssnano

'cssnano',

],

],

},

},

},

],

或者使用PostCSS的配置文件postcss.config.js，postcss-loader會自動使用

**webpack.config.js**

// 設置nodejs環境變量

process.env.NODE\_ENV = "development";

test: /\.css$/i,

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

**postcss.config.js**

module.exports = {

plugins: [

[

"postcss-preset-env",

"cssnano",

],

],

};

package.json中的browserslist用於在不同前端工具之間共享目標瀏覽器、Node.js版本配置

**package.json:**

"browserslist": {

// Browserslist依賴BROWSERSLIST\_ENV或NODE\_ENV環境變量查詢瀏覽器版本範圍，若皆無則先查production，若無production則使用默認值(> 0.5%, last 2 versions, Firefox ESR, not dead)

"development" : [

// 每個瀏覽器的最後1個版本

last 1 versions

],

"production" : [

// 全球使用統計選擇的瀏覽器版本

"> 1%",

// 24個月內有官方支持或更新的瀏覽器

"not dead"

]

}

js兼容性:

babel-loader(記得用npm安裝@babel/core、@babel/preset-env)

使用Babel轉譯 JavaScript文件

Babel 默認只轉換語法Syntax:

1.Arrow function () => {}

2.Spread [ …state, isOpen ]

3.Destructing const { data } = res

4.Default Parameters ( id = NaN ) => {}

而自帶的Api、原生內置Methods，需要加上polyfill後才能使用:

1.Promise

2.Symbol

3.Array.from

4.Object.assign

5.fetch

module: {

rules: [

{

test: /\.m?js$/,

// 排除node\_modules，以加快babel-loader效率

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

// js兼容規則

// @babel/preset-env智能預設，允許使用最新的JavaScript，而無需對目標環境需要的語法轉換和瀏覽器polyfill進行微觀管理

presets: [

'@babel/preset-env',

{

// useBuiltIns有三種設定entry、usage(在code中，加需要的polyfill)、disable(不加polyfill)

useBuiltIns: "entry",

// 指定core-js (JavaScript標準庫的polyfill)版本(記得用npm安裝core-js)

corejs: {

version: 3

},

// 定義兼容至指定版本瀏覽器

"targets": {

"ie": "11"

"chrome": "60"

},

}

],

// 默認值為false，在webpack構建過程中讀取緩存(避免執行時，產生的Babel重新編譯過程recompilation process)，以加快babel-loader效率

// 可以設置指定目錄來緩存loader的執行結果，如果cacheDirectory設置空值、true，loader將使用默認緩存目錄node\_modules/.cache/babel-loader(如果在任何根目錄下都沒有找到node\_modules目錄，會降級回退到操作系統默認的臨時文件目錄)

cacheDirectory: true,

}

}

}

]

}

thread-loader

進行多進程打包，每個worker是一個單獨的node.js進程，其啟動開銷約為600ms毫秒，此外還有進程間通信開銷(需要編譯文件很多時，能提高打包速度；需要編譯文件少，則反之)，一般給babel使用

無選項

rules: [

{

test: /\.js$/,

use: [

'thread-loader',

// your expensive loader (e.g babel-loader)

],

},

],

有選項

use: [

{

loader: 'thread-loader',

options: {

// 2個進程

workers: 2,

},

},

// your expensive loader (e.g babel-loader)

];

plugin種類

HtmlWebpackPlugin

創建包含bundle的HTML文件

參數

template

webpack模板的相對或絕對路徑，默認為src/index.html

filename

要將HTML寫入的文件名稱，默認為index.html

minify

控制是否、以何種方式壓縮(用html-minifier-terser壓縮)，如果是true，則production下用默認壓縮；如果是對象，則production下用自訂義壓縮

// 默認值

minify: {

collapseWhitespace：true, // 移除空格

keepClosingSlash：true, // 保持單例元素的尾部斜杠

removeComments：true, // 移除注釋

removeRedundantAttributes：true,// 當值與默認值匹配時刪除屬性

removeScriptTypeAttributes：true,// 當type="text/javascript"時，從<script>刪除

removeStyleLinkTypeAttributes：true,// 當type="text/css"時，從<style>、<link>刪除

useShortDoctype：true// doctype用簡短的doctype(HTML5)替換

}

MiniCssExtractPlugin

將CSS提取到單獨的文件中，為每個包含CSS的JS文件創建一個CSS文件

參數

filename

輸出的CSS文件名稱，默認為[name].css

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {

module: {

rules: [

{

test: /\.css$/i,

use: [

// 取代style-loader，將JS文件的CSS提取成單獨文件

MiniCssExtractPlugin.loader,

"css-loader"

],

},

],

},

plugins: [

new MiniCssExtractPlugin({

// 更新檔名，推動瀏覽器的快取更新

filename: "build.[contenthash:10].css"

})

],

};

CssMinimizerWebpackPlugin

使用cssnano優化和壓縮CSS

**webpack.config.js**

const CssMinimizerPlugin = require('css-minimizer-webpack-plugin');

module.exports = {

module: {

rules: [

{

test: /\.css$/i,

use: [

"style-loader",

"css-loader"

],

},

],

},

optimization: {

minimizer: [

new CssMinimizerPlugin(),

],

// 默認為生產環境使用，開發環境為true

minimize: true,

},

};

ESLintWebpackPlugin(記得用npm安裝ESLint)

ESLint (JavaScript代碼檢查、格式化工具)的Webpack插件

const ESLintPlugin = require('eslint-webpack-plugin');

module.exports = {

plugins: [

new ESLintPlugin(options)

],

};

WorkboxPlugin

為PWA自動化管理service worker

生成Service Worker文件

**webpack.config.js**

const WorkboxPlugin = require('workbox-webpack-plugin');

plugins: [

new WorkboxPlugin.GenerateSW({

// 這些選項幫助ServiceWorkers快速啟動、刪除舊的ServiceWorkers

clientsClaim: true,

skipWaiting: true,

}),

]

註冊Service Worker

**index.js**

// 添加註冊代碼

if ('serviceWorker' in navigator) {

window.addEventListener('load', () => {

navigator.serviceWorker

.register('/service-worker.js')

.then(registration => {

// 註冊成功

console.log('SW registered: ', registration);

})

.catch(registrationError => {

// 註冊失敗

console.log('SW registration failed: ', registrationError);

});

});

}

sw代碼必須在服務器上運行

npm install serve --save-dev

運行build目錄下所有資源作為靜態資源

serve –s build

DllPlugin

打包出一個個獨立的dll文件(透過一份manifest.json來映射專案中第三方模塊，將其獨立出來打包)

新增第三方模塊配置文件

**webpack.dll.js**

const path = require('path')

// 引入webpack內置插件webpack.DllPlugin

const webpack = require('webpack')

module.exports = {

entry: {

jquery: [

// 獨立打包的第三方模塊名稱jquery

'jquery',

],

},

output: {

filename: '[name].js',

path: resolve(\_\_dirname, 'dll'),

// 第三方模塊向外暴露名稱

library: '[name]\_[hash]',

},

plugins: [

new webpack.DllPlugin({

// 映射第三方模塊向外暴露名稱，要跟output.library一致

name: '[name]\_dll',

// 輸出文件路徑

path: resolve(\_\_dirname, 'dll/manifest.json'),

})

]

}

使用webpack --config webpack.dll.js採用第三方模塊配置文件參數(除非第三方模塊改變，不然只需打包一次)

DllReferencePlugin

在webpack主配置文件中設置，引入DllPlugin插件打包好的dll文件(通過manifest.json文件把依賴名稱映射到第三方模塊(只有dll的bundle)的id上，之后在需要時通過内置的\_\_webpack\_require\_\_函數来require第三方模塊)

**webpack.config.js**

const path = require('path')

// 引入webpack內置插件webpack.DllReferencePlugin

const webpack = require('webpack')

module.exports = {

plugins: [

// 告訴Webpack使用的dll文件(不參與打包的第三方模塊)

new webpack.DllReferencePlugin({

manifest: resolve(\_\_dirname, 'dll/manifest.json'),

})

]

}

因為第三方模塊不參與打包，所以要在出口文件夾的index.html中單獨引入dll文件(不參與打包的第三方模塊)

**index.html**

<body>

<script src="../dll/ jquery.js"></script>

</body>

AddAssetHtmlPlugin

將dll文件(不參與打包的第三方模塊)複製到出口文件夾下， HTMLWebpackPlugin 會自動將其注入到生成的html文件中(不需手動在index.html中引入dll文件)

**webpack.config.js**

const AddAssetHtmlPlugin = require('add-asset-html-webpack-plugin');

module.exports = {

plugins: [

new AddAssetHtmlPlugin([

{

// 要複製的dll文件(不參與打包的第三方模塊)絕對路徑

filepath: resolve(\_\_dirname, 'dll/jquery.js'),

}

])

]

}