webpack 的开发环境与生产环境

时至今日，Webpack 已经成为前端工程必备的基础工具之一，不仅被广泛用于前端工程发布前的打包，还在开发中担当本地前端资源服务器（assets server）、模块热更新（hot module replacement）、API Proxy 等角色，结合 ESLint 等代码检查工具，还可以实现在对源代码的严格校验检查。

正如上文中提到的，前端从开发到部署前都离不开 Webpack 的参与，而 Webpack 的默认配置文件只有一个，即 webpack.config.js，那么问题来了，开发期和部署前应该使用同一份 Webpack 配置吗？答案肯定是否定的，既然 webpack.config.js 是一个 JS 文件，我们当然可以在文件里写 JavaScript 业务逻辑，通过读取环境变量 NODE_ENV 来判断当前是在开发（dev）时还是最终的生产环境（production），然而很多同学习惯把这两者的配置都混写在根目录下的 webpack.config.js，通过很多零散的 if...else 来“临时”决定某一个 plugin 或者某一个 loader 的配置项，随着 loaders 和 plugins 的不断增加，久而久之 webpack.config.js 变得原来越隆长，代码的可读性和可维护性也大大下降。

先说说环境之间的区别。

开发环境：开发环境是程序猿们专门用于开发的服务器，配置可以比较随意， 为了开发调试方便，一般打开全部错误报告。

测试环境：一般是克隆一份生产环境的配置，一个程序在测试环境工作不正常，那么肯定不能把它发布到生产机上。

生产环境：是指正式提供对外服务的，一般会关掉错误报告，打开错误日志。

三个环境也可以说是系统开发的三个阶段：开发->测试->上线，其中生产环境也就是通常说的真实环境。

通俗一点就是：

1：开发环境：项目尚且在编码阶段，我们的代码一般在开发环境中 不会在生产环境中，生产环境组成：操作系统 ，web 服务器 ，语言环境。 php 。 数据库 。 等等

2：测试环境：项目完成测试，修改 bug 阶段

3：生产环境：项目数据前端后台已经跑通，部署在阿里云上之后，有客户使用，访问，就是网站正式运行了

开发到正式上线的流程：

应该是先在开发环境 中开发完成，测试环境测试，保证程序没有问题后，再上传到生产环境中。

对于 webpack 中的生产环境，与开发环境都有相应的模式，也是在 webpack 4 中提出的模式 mode。

我们来看一下 MODE 这个参数，他有三个参数 production，development，none，前两个是有预设的插件，而最后一个则是什么都没有，也就是说设置为 none 的话，webpack 就是最初的样子，无任何预设，需要从无到有开始配置。

在 webpack 的配置中，其他配置都可以没有！但是 mode 是必备的，如果不加 mode，官方虽然会打包，但同时也会给你一个警告：

```
WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/
```

意思很简单，就是 mode 没有被设置的情况下，系统就会给你一个默认的 production 模式。

mode 配置很简单，就只有 3 个值，任君挑选。none 这个参数，相信大家都能理解，那么我们就研究下其他两个 production 和 development，这为什么要有这两个状态，以及两者在 webpack 打包中都干了些啥事。

在 mode 为 production 或 development 的状态下，为了兼顾两个状态下的程序运行，webpack 创建了一个全局变量 process.env.NODE_ENV，等同于在插件 plugins 中加入了 new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("development|production") })，用来区分不同的状态，同时可以在程序中区分程序状态。

那么我们该如何在 coding 的时候进行区分呢？因为 process.env.NODE_ENV 是全局变脸给，所以可以这样来引用值，假设 mode:production：

```
if ("development" === process.env.NODE_ENV){
    ....
}else{
    ....
```

```
}
//编译之后：

if ("development" === "production"){
    ....
}else{
    ....
}
```

也就是最后 process.env.NODE_ENV 会被替换为一个常量。这个小功能可以帮助我们在写业务 JS 的时候，区分线上版本与开发版本。

none 模式下的模块打包

在没有任何优化处理的情况下，按照 webpack 默认的情况下打包出来的模块是怎么样的呢？下方是一个简易的例子，我们可以看出，他将模块打包至数组之中，调用模块的时候，就是直接调用模块在此数组中的一个序号。然后没有压缩混淆之类的优化，连注释都帮我们标的好好的，比如导入 /* harmony import ／，／harmony default export */。


```
[
    /* 0 */
    (function(module, __webpack_exports__, __webpack_require__) {
    "use strict";
    __webpack_require__.r(__webpack_exports__);
    /* harmony import */ var _page2_js__WEBPACK_IMPORTED_MODULE_0__ =
__webpack_require__(1);
    console.log(_page2_js__WEBPACK_IMPORTED_MODULE_0__["default"])
    }),
    /* 1 */
    (function(module, __webpack_exports__, __webpack_require__) {
    "use strict";
    __webpack_require__.r(__webpack_exports__);
    let str="page1"
    /* harmony default export */ __webpack_exports__["default"] = (str);
    })
]
```

但是无论是在开发环境 development 下，还是在正式坏境 production 下，这个代码都是不过关的，对于开发环境，此代码可读性太差，对于正式环境，此代码不够简洁，因此，为了减少一些重复操作，webpack4 提供的 development|production 可以很大程度上帮我们做掉一大部分事，我们要做的就是在这些事的基础上加功能。

development 模式下，webpack 做了那些打包工作

development 是告诉程序，我现在是开发状态，也就是打包出来的内容要对开发友好。在此 mode 下，就做了以下插件的事，其他都没做，所以这些插件可以省略。

```
// webpack.development.config.js
module.exports = {
+ mode: 'development'
- devtool: 'eval',
- plugins: [
-   new webpack.NamedModulesPlugin(),
-   new webpack.NamedChunksPlugin(),
-   new webpack.DefinePlugin({ "process.env.NODE_ENV":
JSON.stringify("development") }),
- ]
}
```


我们看看 NamedModulesPlugin 和 NamedChunksPlugin 这两个插件都做了啥，原本我们的 webpack 并不会给打包的模块加上姓名，一般都是按照序号来，从 0 开始，然后加载第几个模块。这个对机器来说无所谓，查找载入很快，但是对于人脑来说就是灾难了，所以这个时候给各个模块加上姓名，便于开发的时候查找。

没有 NamedModulesPlugin，模块就是一个数组，引用也是按照在数组中的顺序引用，新增减模块都会导致序号的变化，就是 webpack 默认打包下的情况。

有了 NamedModulesPlugin，模块都拥有了姓名，而且都是独一无二的 key，不管新增减多少模块，模块的 key 都是固定的。

```
{

"./src/index.js":   (function(module, __webpack_exports__,
__webpack_require__) {
                        "use strict";
                        __webpack_require__.r(__webpack_exports__);
```

```
                        var _page2_js__WEBPACK_IMPORTED_MODULE_0__ =
__webpack_require__("./src/page2.js");

console.log(_page2_js__WEBPACK_IMPORTED_MODULE_0__["default"])
                    }),
"./src/page2.js":    (function(module, __webpack_exports__,
__webpack_require__) {
                        "use strict";
                        __webpack_require__.r(__webpack_exports__);
                        let str="page1"
                         __webpack_exports__["default"] = (str);
                    })
}
```

除了 NamedModulesPlugin，还有一个 NamedChunksPlugin，这个是给配置的每个 chunks 命名，原本的 chunks 也是数组，没有姓名。

```
   Asset      Size  Chunks             Chunk Names
index.js  4.04 KiB       0  [emitted]  index
page2.js  3.75 KiB       1  [emitted]  page2

   Asset      Size          Chunks             Chunk Names
index.js   4.1 KiB           index  [emitted]  index
page1.js  4.15 KiB           page1  [emitted]  page1
```

NamedChunksPlugin 其实就提供了一个功能就是你可以自定义 chunks 的名字，假如我再不同的包中有相同 chunk 名，怎么办？这个时候就要在进行区分了，我么可以用所有的依赖模块名加本上的模块名。因为 Chunk.modules 已经废弃了，现在用其他的方法来代替 chunk.mapModules，然后重命名 chunk 的名字：

```
new webpack.NamedChunksPlugin((chunk) => {
    return chunk.mapModules(m => {
        return path.relative(m.context, m.request)
    }).join("_")
}),
```

看一眼做这一行代码的效果，我们可以看到 Chunks 这边已经重命名了，这样可以很大程度上解决 chunks 重名的问题：

```
    Asset      Size             Chunks             Chunk Names
index.js   4.1 KiB  index.js_page2.js  [emitted]  index
page2.js  3.78 KiB           page2.js  [emitted]  page2
```

总结：development 也就给我们省略了命名的过程，其他的还是要自己加的。

production

在正式版本中，所省略的插件们，如下所示，我们会一个个分析。

```
module.exports = {
+  mode: 'production',
-  plugins: [
-    new UglifyJsPlugin(/* ... */),
-    new webpack.DefinePlugin({ "process.env.NODE_ENV":
JSON.stringify("production") }),
-    new webpack.optimize.ModuleConcatenationPlugin(),
-    new webpack.NoEmitOnErrorsPlugin()
-  ]
}
```

UglifyJsPlugin

我们第一个需要处理的就要混淆&压缩 JS 了吧，这个时候就要请出 UglifyJs 了，在 webpack 中他的名字是 const UglifyJsPlugin = require('uglifyjs-webpack-plugin');，这样就可以使用他了。

不过 new UglifyJsPlugin()，这个插件我们可以在 optimize 中配置，效果是一样的，那么我们是不是就不用再导入一个新的插件了，这样反而会拖慢 webpack 的就打包速度。

```
optimization:{
    minimize: true,
},
```

将插件去除，混淆压缩放入 optimization，这样 webpack 速度快的飞起了。只有第一次打包会慢，之后再打包就快了。

ModuleConcatenationPlugin

webpack.optimize.ModuleConcatenationPlugin()这个插件的作用是什么呢？官方文档上是这么描述的：

> 记住，此插件仅适用于由 webpack 直接处理的 ES6 模块。在使用转译器(transpiler)时，你需要禁用对模块的处理（例如 Babel 中的 modules 选项）。

NoEmitOnErrorsPlugin

最后一个插件就是 webpack.NoEmitOnErrorsPlugin()，这个就是用于防止程序报错，就算有错误也给我继续编译，很暴力的做法呢。

others

还有一些默认的插件配置，也就是可以不在 plugins 中引用的配置：

flagIncludedChunks

flagIncludedChunks 这个配置的作用是，看结果：

未启用

```
   Asset       Size  Chunks             Chunk Names
index.js   1.02 KiB       0  [emitted]  index
page1.js  970 bytes       1  [emitted]  page1
```

启用后，如果只有二个文件似乎表现不明显，于是我增加了三个文件，page1 调用 page2，index 调用 page1，那么一目了然，在这里的 chunks 就是所有引用模块的 id。

```
  Asset       Size   Chunks             Chunk Names
index.js   1.08 KiB  0, 1, 2  [emitted]  index
page1.js   1.01 KiB     1, 2  [emitted]  page1
page2.js  971 bytes        2  [emitted]  page2
```

OccurrenceOrderPlugin

webpack.optimize.OccurrenceOrderPlugin 这个插件的作用是按照 chunk 引用次数来安排出现顺序，因为这让经常引用的模块和 chunk 拥有更小的 id。将上面的例子加上这个配置运行下就是这样的。

```
   Asset       Size   Chunks             Chunk Names
```

```
page2.js  969 bytes        0  [emitted]  page2
page1.js   1.01 KiB     1, 0  [emitted]  page1
index.js   1.08 KiB  2, 0, 1  [emitted]  index
```

SideEffectsFlagPlugin

webpack.optimize.SideEffectsFlagPlugin()这个插件如果需要生效的话，需要两个条件，一个是导入的模块已经标记了 sideEffect，即 package.json 中的 sideEffects 这个属性为 false，第二个就是当前模块引用了次无副作用的模块，而且没有使用。那么在打包的时候，就不会将这个模块打包到文件中。

总结

实际上 production mode 下，与官方文档相比，他的配置更加等同于如下配置：

```
module.exports = {
    mode:"none",
    optimization:{
        flagIncludedChunks:true,
        minimize: true,
    },
    plugins: [
        new webpack.DefinePlugin({ "process.env.NODE_ENV":
JSON.stringify("production") }),
        new webpack.optimize.ModuleConcatenationPlugin(),
        new webpack.NoEmitOnErrorsPlugin(),
        new webpack.optimize.SideEffectsFlagPlugin()
    ]
}
```