从 Webpack 的产出代码来看 Webpack 是怎么执行的

更新时间：2019-07-01 12:09:35

> 老骥伏枥，志在千里；烈士暮年，壮心不已。
>
> ——曹操

通过之前的章节内容，我们已经了解了 Webpack 的整个打包流程，并且针对源码做了分析。这篇文章，来分析下 Webpack 打包产出物是怎样执行的。本文基于 Webpack@4.29.6 版本来分析产出物，Webpack 版本不同产出物可能有差异，但是基本原理是一致的。

首先我使用下面两个 js 文件：`app.js` 和 `name.js`，其中 `name.js` 内容如下：

```
// name.js
module.exports = 'Alex';
```

而 `app.js` 中直接使用 `require` 引入了 `name.js`：

```
// app.js
const name = require('./name.js');
console.log(name);
```

对应的 `webpack.config.js` 也是相当简单，使用 `mode="development"` 让 js 不压缩，`devtool=false` 保证不输出 `sourcemap`，然后指定下 `entry`：

```
const config = {
    mode: 'development', // 采用dev模式，不会压缩代码
    devtool: false, // 不用sourcemap
    // 没有output则默认输出是到dist的main
    entry: './src/app.js'
};
```

执行`npx webpack --config webpack.config.js`后，在`dist`文件夹下找到了打包后的代码（去掉部分不必要的注释）：

```
(function(modules) {
    // webpackBootstrap
    // The module cache
    var installedModules = {};

    // The require function
    function __webpack_require__(moduleId) {
        // Check if module is in cache
        if (installedModules[moduleId]) {
            return installedModules[moduleId].exports;
        }
        // Create a new module (and put it into the cache)
        var module = (installedModules[moduleId] = {
            i: moduleId,
            l: false,
            exports: {}
        });

        // Execute the module function
        modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);

        // Flag the module as loaded
        module.l = true;

        // Return the exports of the module
        return module.exports;
    }

    // expose the modules object (__webpack_modules__)
    __webpack_require__.m = modules;

    // expose the module cache
    __webpack_require__.c = installedModules;

    // define getter function for harmony exports
    __webpack_require__.d = function(exports, name, getter) {
        if (!__webpack_require__.o(exports, name)) {
            Object.defineProperty(exports, name, {enumerable: true, get: getter});
        }
    };

    // define __esModule on exports
    __webpack_require__.r = function(exports) {
        if (typeof Symbol !== 'undefined' && Symbol.toStringTag) {
            Object.defineProperty(exports, Symbol.toStringTag, {value: 'Module'});
        }
        Object.defineProperty(exports, '__esModule', {value: true});
    };

    // create a fake namespace object
    // mode & 1: value is a module id, require it
    // mode & 2: merge all properties of value into the ns
    // mode & 4: return value when already ns object
    // mode & 8|1: behave like require
    __webpack_require__.t = function(value, mode) {
        if (mode & 1) value = __webpack_require__(value);
        if (mode & 8) return value;
        if (mode & 4 && typeof value === 'object' && value && value.__esModule) return value;
        var ns = Object.create(null);
        __webpack_require__.r(ns);
        Object.defineProperty(ns, 'default', {enumerable: true, value: value});
        if (mode & 2 && typeof value != 'string')
            for (var key in value)
                __webpack_require__.d(
                    ns,
                    key,
                    function(key) {
                        return value[key];
```

```
                }.bind(null, key)
            );
        return ns;
    };

    // getDefaultExport function for compatibility with non-harmony modules
    __webpack_require__.n = function(module) {
        var getter =
            module && module.__esModule
                ? function getDefault() {
                      return module['default'];
                  }
                : function getModuleExports() {
                      return module;
                  };
        __webpack_require__.d(getter, 'a', getter);
        return getter;
    };

    // Object.prototype.hasOwnProperty.call
    __webpack_require__.o = function(object, property) {
        return Object.prototype.hasOwnProperty.call(object, property);
    };

    // __webpack_public_path__
    __webpack_require__.p = '';

    // Load entry module and return exports
    return __webpack_require__((__webpack_require__.s = './src/app.js'));
})({
    './src/app.js': function(module, exports, __webpack_require__) {
        const name = __webpack_require__(/*! ./name.js */ './src/name.js');
        console.log(name);
    },
    './src/name.js': function(module, exports) {
        // name.js
        module.exports = 'Alex';
    }
});
```

分析打包后的代码结构

上面代码可以看出，整个 Webpack 的打包产物是一个**立即执行函数表达式**（IIFE），函数外部结构如下：

```
(function(modules) {
    // 内部内容忽略
    // 加载 entry 模块，并且 return 他的 exports
    return __webpack_require__((__webpack_require__.s = './src/app.js'));
})({
    './src/app.js': function(module, exports, __webpack_require__) {
        const name = __webpack_require__(/*! ./name.js */ './src/name.js');
        console.log(name);
    },
    './src/name.js': function(module, exports) {
        // name.js
        module.exports = 'Alex';
    }
});
```

即将下面对象当成一个参数传入立即执行函数，该函数唯一参数是`modules`，所以称这个对象为`modules`吧，`modules`格式如下：

```
{
    './src/app.js': function(module, exports, __webpack_require__) {
        const name = __webpack_require__(/*! ./name.js */ './src/name.js');
        console.log(name);
    },
    './src/name.js': function(module, exports) {
        // name.js
        module.exports = 'Alex';
    }
}
```

在这里我们看到了，`modules`对象是的`key`是文件的路径，`value`则是函数的类似 AMD factory 格式的函数，整个 IIFE 函数的核心是四步：

1. 定义一个对象 `installedModules` ，用来保存已经注册成功的模块；
2. 定义`__webpack_require__` 函数来实现模块的加载，这是整个模块管理的核心；
3. 定义`__webpack_require__` 的一些属性；
4. 传入 `entry` 模块，执行`__webpack_require__` ，并且返回执行结果，即 `entry` 的`exports`。

`__webpack_require__` 函数

`__webpack_require__` 函数是 Webpack 的核心，它主要作用是调用并且注册模块，整个代码如下：

```
// 接受一个模块id
function __webpack_require__(moduleId) {
    // 1. 判断是否已经注册过，注册过的模块都在installedModules可以找到
    if (installedModules[moduleId]) {
        // 如果注册过，则直接返回
        return installedModules[moduleId].exports;
    }
    // 2. 没有注册的模块，就注册一个，并且放入`installedModules` 缓存起来
    var module = (installedModules[moduleId] = {
        i: moduleId,
        l: false,
        exports: {}
    });

    // 3. 执行模块的 factory 函数
    modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);

    // 4. 打个 flag 表示下模块已经加载了
    module.l = true;

    // 5. 返回模块的输出 exports
    return module.exports;
}
```

`__webpack_require__`函数接收`moduleId`（模块 ID）作为参数，然后开始执行，整个过程如下：

1. 根据`moduleId`在`installedModules`中判断是否已经注册过，注册过则直接返回`exports`；
2. 没有注册的模块，就注册一个，并且放入`installedModules`缓存起来，这里有三个属性：
    1. `i`：是模块 id，即 moduleId
    2. `l`：默认是 false，即有没有被注册执行过（应该是`loaded`简写）
    3. `exports`：模块的输出对象
3. 使用`call`方法执行模块的 `factory` 函数，其中`call`上下文是`module.exports`，同时传入 3 个参数：
    1. `module`：模块本身
    2. `module.exports`：模块 `exports` 对象

3. `__webpack_require__` 函数

4. 修改模块的 `l` 属性，标识已经注册完成；
5. 返回模块的 `exports` 对象。

这里我们继续看下 `app.js` 处理后的 `factory` 函数内容：

```
function(module, exports, __webpack_require__) {
    const name = __webpack_require__('./src/name.js');
    console.log(name);
}
```

我们可以看到，`factory` 的 3 个参数，分别对应的是 `__webpack_require__` 函数内执行 `call` 传入的 3 个参数，即：

1. `module`：模块本身
2. `module.exports`：模块 `exports` 对象
3. `__webpack_require__` 函数

而 `factory` 的上下文，即 `this`，则应该是 `call` 传入的 `module.exports`，即 `exports` 对象。

`__webpack_require__` 的属性

研究完 `__webpack_require__` 函数，在继续看下 IIFE 中添加 `__webpack_require__` 属性：

1. `__webpack_require__.s`：记录入口文件的 moduleId；
2. `__webpack_require__.m`：为 `modules` 对象，即传入的所有模块对象；
3. `__webpack_require__.c`：即 `cache` 对象，所有已注册的对象缓存，即 `installedModules` 对象；
4. `__webpack_require__.d`：用于 ES modules，输出的是值的引用；
5. `__webpack_require__.r`：给 `exports` 定义一个 `__esModule` 属性；
6. `__webpack_require__.t`：根据传入的 `moduleId` 的模块做 ES module `default` 和 CommonJS module 兼容；
7. `__webpack_require__.n`：解决 `ES module` 和 `CommonJS module` 导出不一致的问题，即 `ES` 模块，则返回 `module['default']`；
8. `__webpack_require__.o`：判断是否一个 `Object` 有没有 `property` 属性；
9. `__webpack_require__.p`：这个属性的取值来自于我们配置中的 `output.publicPath`。

`__webpack_require__.p` 跟 `output.publicPath` 有关系，如果修改下 `webpack.config.js` 的内容，增加 `output.publicPath`：

```
module.exports = {
    mode: 'development',
    devtool: false,
    entry: './src/app.js',
    output: {
        publicPath: 'http://baidu.com/'
    }
};
```

则 `__webpack_require__.p` 的内容就发生了变化：

```
__webpack_require__.p = 'http://baidu.com/';
```

上面是在普通的方式打包产出的分析。还有其他几种情况，得到的 `__webpack_require__` 属性会不同，这里可以直接在 `webpack/lib/MainTemplate.js` 中找到的定义说明。

Chunks Split 产出分析

接下来我们再来看下配置了`splitChunks`的产出差异，为了更好的理解入口文件不同对打包结果的影响，我们将入口文件增加到两个：`app.js`和`name.js`，内容分别如下：

```
// app.js
const $ = require('zepto');
console.log($);
// name.js
module.exports = 'Alex';
```

`zepto`是通过 NPM 安装的`npm i zepto`，在修改下`webpack.config.js`：

1. 增加`splitChunks`的内容；
2. entry 由只有一个字符串形式的单一 entry，换成数组`[string]`。

```
module.exports = {
    mode: 'development',
    devtool: false,
    // 注意这里，entry为数组形式
    entry: ['./src/app.js', './src/name.js'],
    // 增加 splitChunks
    optimization: {
        splitChunks: {
            chunks: 'all',
            minSize: 10,
            minChunks: 1,
            name: 'vendors.main',
            cacheGroups: {
                vendors: {
                    test: /[\\/]node_modules[\\/]/,
                    priority: -10
                }
            }
        }
    }
};
```

根据之前的常见配置项小结的内容，我们知道 entry 无论是`string`还是`[string]`的形式，都是单一文件入口，即打包产出上会生成一个 bundle，我们来看下打包后的结果：

```
//main.js
(function(modules) {
    // webpackBootstrap
    // install a JSONP callback for chunk loading
    // 新增内容
    function webpackJsonpCallback(data) {
        var chunkIds = data[0];
        var moreModules = data[1];
        var executeModules = data[2];

        var moduleId,
            chunkId,
            i = 0,
            resolves = [];
        for (; i < chunkIds.length; i++) {
            chunkId = chunkIds[i];
            // 这里代码在`import()`/`require.ensure`会有用，后面会详细介绍
            if (installedChunks[chunkId]) {
                resolves.push(installedChunks[chunkId][0]);
            }
            installedChunks[chunkId] = 0;
        }
        for (moduleId in moreModules) {
            if (Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
                modules[moduleId] = moreModules[moduleId];
```

```
            }
        }
        if (parentJsonpFunction) parentJsonpFunction(data);

        while (resolves.length) {
            resolves.shift()();
        }

        // add entry modules from loaded chunk to deferred list
        deferredModules.push.apply(deferredModules, executeModules || []);

        // run deferred modules when all chunks ready
        return checkDeferredModules();
    }
    // 遍历deferredModules数组，依次调用__webpack_require__完成模块注册
    function checkDeferredModules() {
        var result;
        for (var i = 0; i < deferredModules.length; i++) {
            var deferredModule = deferredModules[i];
            var fulfilled = true;
            for (var j = 1; j < deferredModule.length; j++) {
                var depId = deferredModule[j];
                if (installedChunks[depId] !== 0) fulfilled = false;
            }
            if (fulfilled) {
                deferredModules.splice(i--, 1);
                result = __webpack_require__((__webpack_require__.s = deferredModule[0]));
            }
        }
        return result;
    }

    // 一样，注册过的 module 缓存
    var installedModules = {};

    // installedChunks是一个存储对象，用于存储 Chunks的状态
    // 在splitChunk，没有使用 import()/require.ensure 的情况下，状态只有两种：
    // 1. undefined：没有加载
    // 2. 0 加载成功
    // undefined = chunk not loaded, null = chunk preloaded/prefetched
    // Promise = chunk loading, 0 = chunk loaded
    // 只在 SplitChunk 方式，状态只有0
    var installedChunks = {
        main: 0
    };
    // 存储定义后的 module
    var deferredModules = [];

    // __webpack_require__方法和属性部分代码都没变化
    function __webpack_require__(moduleId) {}
    __webpack_require__.m = modules;
    __webpack_require__.c = installedModules;
    __webpack_require__.d = function(exports, name, getter) {};
    __webpack_require__.r = function(exports) {};
    __webpack_require__.t = function(value, mode) {};
    __webpack_require__.n = function(module) {};
    __webpack_require__.o = function(object, property) {};
    __webpack_require__.p = '';

    // 新增内容
    var jsonpArray = (window['webpackJsonp'] = window['webpackJsonp'] || []);
    var oldJsonpFunction = jsonpArray.push.bind(jsonpArray);
    jsonpArray.push = webpackJsonpCallback;
    jsonpArray = jsonpArray.slice();
    for (var i = 0; i < jsonpArray.length; i++) webpackJsonpCallback(jsonpArray[i]);
    var parentJsonpFunction = oldJsonpFunction;

    // 将entry 等模块添加到deferredModules
    deferredModules.push([0, 'vendors.main']);
    // 检测已经注册到deferredModules的模块，并且触发 factory
    return checkDeferredModules();
})({
    './src/name.js': function(module, exports) {
```

```
        module.exports = 'Alex';
    },

    './src/app.js': function(module, exports, __webpack_require__) {
        __webpack_require__('./node_modules/zepto/dist/zepto.js');
    },

    0: function(module, exports, __webpack_require__) {
        __webpack_require__('./src/app.js');
        module.exports = __webpack_require__('./src/name.js');
    }
});
```

下面是拆出来的`vendors.main.js`文件：

```
(window['webpackJsonp'] = window['webpackJsonp'] || []).push([
    ['vendors.main'],
    {
        './node_modules/zepto/dist/zepto.js': function(module, exports, __webpack_require__) {
            var __WEBPACK_AMD_DEFINE_RESULT__;
            (function(global, factory) {
                if (true)
                    !((__WEBPACK_AMD_DEFINE_RESULT__ = function() {
                        return factory(global);
                    }.call(exports, __webpack_require__, exports, module)),
                    __WEBPACK_AMD_DEFINE_RESULT__ !== undefined && (module.exports = __WEBPACK_AMD_DEFINE_RESULT__));
                else {
                }
            })(this, function(window) {
                // zepto 源码
            });
        }
    }
]);
```

基本概念

在解析源码之前，为了方便理解，根据代码中涉及到的一些变量名，做概念上的统一：

1. `app.js`和`name.js`打包到了`main.js`；
2. `app.js`依赖的`zepto`库太大，被拆分到了`vendors.main.js`；
3. chunk 是打包后产出的文件，内部包含多个模块的代码，比如`main.js`就是一个 chunk，内部包含打包之前的`app.js`和`name.js`；
4. module 是 js 模块，例如：`app.js`和`name.js`。

> Tips：可以简单理解 chunk 是打包后的文件产物，一个 chunk 可能包含多个模块；模块是打包前的文件。

所以，通过上面的概念，`installedChunks`这个变量，从命名上我们应该知道跟 chunk 有关系，而`deferredModules`则跟模块有关系。

跟普通打包产出的差异

比较下之前的打包产出物，发现变化如下：

1. IIFE 函数 return 出来的是`checkDeferredModules`；
2. 在 entry 配置了入口文件是又两个文件组成的数组，但是打包之后 IIFE 的参数`modules`变成了三个，即`name.js`、`app.js`和一个自动生成的依赖`app.js`和`name.js`的`0`模块；
3. IIFE 内部新增`webpackJsonpCallback`用于加载 splitChunk 内容；

4. 在文件引用关系上，会先加载`vendors.main.js`，然后加载`main.js`（app.js），先将内容绑定到`webpackJsonp`数组。

`vendors.main.js`代码解析

接下来，我们看下代码执行的过程。因为`vendors.main.js`会在`main.js`（app.js）之前被引入到页面，所以首先发生在`vendors.main.js`中，会将`main.js`执行需要的依赖模块统一放入`webpackJsonp`数组，`webpackJsonp`数组的格式如下：

```
// webpackJsonp的每一项都是一个数组组成
[
    ['vendors.main'], // 打包后的chunkId数组，
    {moduleId: factory} // 文件对应的包含的模块对象，key 是模块 id，value 是 factory 函数
];
```

值得注意的是，其实源码中 `zepto` 的源码代码如下：

```
// npm zepto源码
(function(global, factory) {
  if (typeof define === 'function' && define.amd)
    define(function() { return factory(global) })
  else
    factory(global)
}(this, function(window) {
    // zepto source
});
```

而经过 Webpack 处理之后的这部分定义代码发生了下面的变化：

```
function(module, exports, __webpack_require__) {
    var __WEBPACK_AMD_DEFINE_RESULT__;
    // 下面是 zepto 的代码
    (function(global, factory) {
        if (true)
            !((__WEBPACK_AMD_DEFINE_RESULT__ = function() {
                return factory(global);
            }.call(exports, __webpack_require__, exports, module)),
            __WEBPACK_AMD_DEFINE_RESULT__ !== undefined && (module.exports = __WEBPACK_AMD_DEFINE_RESULT__));
        else {
        }
    })(this, function(window) {
        // zepto 源码
    });
}
```

1. `typeof define === 'function' && define.amd`直接被替换成`true`
2. `define` 被直接作为执行了，执行结果非`undefined`（return）作为`module.exports`了

简单来说上面的操作，是把 AMD 的 define 方法，换成了 Webpack 的 `module.export = factory(global)`。这部分是直接将 AMD 的模块定义替换 Webpack 自己的模块管理方式，相关的转换可以在`webpack/lib/dependencies/AMDDefineDependency.js`找到。

`vendors.main.js`内容相对简单，一句话概括就是给`webpackJsonp`数组增加子项，等待`main.js`处理。

> Tips：这里有个坑，实际 zepto 代码执行的时候会报错。这里本文最后会说原因和解决方式，下面继续产出源码分析。

`main.js`内容介绍

当进入`main.js`文件后，执行 IIFE 函数，遍历`webpackJsonp`数组，将数组项交给`webpackJsonpCallback`处理：

```
var jsonpArray = (window['webpackJsonp'] = window['webpackJsonp'] || []);
var oldJsonpFunction = jsonpArray.push.bind(jsonpArray);
// **这里重写了webpackJsonp数组的 push 函数
// 实际当后续还有模块通过 push 方法添加到webpackJsonp，则直接调用webpackJsonp
jsonpArray.push = webpackJsonpCallback;
jsonpArray = jsonpArray.slice();
for (var i = 0; i < jsonpArray.length; i++) webpackJsonpCallback(jsonpArray[i]);
```

下面来重点解读下`webpackJsonpCallback`的源码。

> Tips：注意上面的代码重写了 webpackJsonp 数组的 push 函数，用处有：
>
> 1. 实际当后续还有模块通过 push 方法添加到 webpackJsonp，则直接调用 webpackJsonp，比如再加载个 webpack 打包后的 verdons 文件；
> 2. `import()` / `require.ensure`这种异步加载的方式，加载之后实际`webpackJsonp.push`就是`webpackJsonpCallback`函数了，后面详细解释，记住即可。

`webpackJsonpCallback` 函数

`webpackJsonpCallback` 函数主要是用于加载 chunk 及其内部包含模块的函数，用到了`installedChunks`对象来存储 chunk 的状态。

> Tips: `installedChunks`是一个存储 chunk 状态的对象，在 splitChunk，没有使用 `import()` / `require.ensure` 的情况下，状态只有两种：
>
> 1. undefined：没有加载
> 2. 0 加载成功

```
function webpackJsonpCallback(data) {
    var chunkIds = data[0]; // 数组，保存了该 chunk 文件的 chunkId 值
    var moreModules = data[1]; // 对象，保存的chunk 文件包含的模块
    var executeModules = data[2]; // 数组，可执行的模块，被传进来的 moduleId 会被优先执行

    var moduleId,
        chunkId,
        i = 0,
        resolves = [];
    // 1. 循环判断chunks 的加载状态
    for (; i < chunkIds.length; i++) {
        chunkId = chunkIds[i];
        // 使用installChunks中存储chunk的加载状态，据此判断chunk是否加载完毕
        if (installedChunks[chunkId]) {
            // 在 splitChunk 没有 import() /require.ensure下，状态始终是
            resolves.push(installedChunks[chunkId][0]);
        }
        // 给 chunk 添加加载完成的标识，状态0
        installedChunks[chunkId] = 0;
    }
    // 2. 将 chunk 文件中的 module 添加到 modules 对象上，modules 是 IIFE 函数的参数
    // 即 app.js name.js 0 的对象
    for (moduleId in moreModules) {
        if (Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
            modules[moduleId] = moreModules[moduleId];
        }
    }
    if (parentJsonpFunction) parentJsonpFunction(data);

    while (resolves.length) {
        resolves.shift()();
    }

    // 3. 如果有需要执行的模块，则加入 deferredModules 数组
    deferredModules.push.apply(deferredModules, executeModules || []);

    // 4. 每次执行完最后，都要执行下checkDeferredModules 函数，检测下状态，达到entry 可执行状态没（即依赖模块都加载没）
    return checkDeferredModules();
}
```

`checkDeferredModules` 函数

介绍完`webpackJsonpCallback`，再看下 IIFE 函数最后的执行代码，实际是将`0`和`vendors.main`俩模块放入`deferredModules`，然后检测`deferredModules`内模块的状态。

```
// 将entry 等模块添加到deferredModules
deferredModules.push([0, 'vendors.main']);
// 检测已经注册到deferredModules的模块，并且触发 factory
return checkDeferredModules();
```

在`checkDeferredModules`函数主要是遍历`deferredModules`数组中模块状态，保证 Chunk 文件中模块和依赖的模块都已经加载成功，然后执行`__webpack_require__`，触发 entry 的 factory，详细过程可以查看下面的代码注释。

```
// `checkDeferredModules` 函数
function checkDeferredModules() {
    var result;
    for (var i = 0; i < deferredModules.length; i++) {
        var deferredModule = deferredModules[i];
        var fulfilled = true;
        // 1. 循环开始检测依赖模块的状态
        for (var j = 1; j < deferredModule.length; j++) {
            var depId = deferredModule[j];
            // 2. 使用installedChunks 对象值来检测状态
            // installedChunks中存储的状态，在webpackJsonpCallback得到修改
            if (installedChunks[depId] !== 0) fulfilled = false;
        }
        // 3. 只有模块所需的chunk都加载完毕，该模块才会被执行
        if (fulfilled) {
            deferredModules.splice(i--, 1);
            result = __webpack_require__((__webpack_require__.s = deferredModule[0]));
        }
    }
    return result;
}
```

在`checkDeferredModules`函数的最后：

```
result = __webpack_require__((__webpack_require__.s = deferredModule[0]));
return result;
```

将`deferredModule`第一个子项，赋值给`__webpack_require__.s`，即作为入口文件，然后执行`__webpack_require__`进行注册，并且返回第一个模块的`exports`结果，至此整个打包产物的执行流程就结束了。

多文件打包的产出分析

多文件的打包产出物，其实核心的内容跟单文件打包没有差别，如果多文件+`splitChunks`方式的打包，也跟单文件`splitShunks`没有差别，只不过在拆分 chunk 上会略有不同，所以不再继续讨论。下面来讨论下在模块中使用`import()`或者`require.ensure`异步加载模块的打包产物执行过程。

`import()` 方式打包的产出物解析

首先修改`app.js`的内容，修改成下面的代码：

```
// app.js
const name = require('./name.js');
import('zepto');
```

`webpack.config.js`的代码：

```
module.exports = {
    mode: 'development',
    devtool: false,
    entry: './src/app.js',
    optimization: {
        // runtimeChunk:{name: 'runtime'},
        splitChunks: {
            chunks: 'all',
            minSize: 10,
            minChunks: 1,
            name: 'vendors.main',
            cacheGroups: {
                vendors: {
                    test: /[\\/]node_modules[\\/]/,
                    priority: -10
                }
            }
        }
    }
};
```

那么执行`npx webpack --config webpack.config.js`之后，在 dist 文件夹会找到`main.js`和`verdors.main.js`，其中`verdors.main.js`跟上一部分`splitChunks`没有区别，`main.js`却发生了变化：

```
// main.js，忽略掉相同的部分代码
(function(modules) {
    // webpackBootstrap
    // webpackJsonpCallback 发生变化
    function webpackJsonpCallback(data) {
        var chunkIds = data[0];
        var moreModules = data[1];

        // add "moreModules" to the modules object,
        // then flag all "chunkIds" as loaded and fire callback
        var moduleId,
            chunkId,
            i = 0,
            resolves = [];
        for (; i < chunkIds.length; i++) {
            chunkId = chunkIds[i];
            if (installedChunks[chunkId]) {
                resolves.push(installedChunks[chunkId][0]);
            }
            installedChunks[chunkId] = 0;
        }
        for (moduleId in moreModules) {
            if (Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
                modules[moduleId] = moreModules[moduleId];
            }
        }
        if (parentJsonpFunction) parentJsonpFunction(data);

        while (resolves.length) {
            resolves.shift()();
        }
    }
    // 忽略一部分相同代码

    // script path function
    function jsonpScriptSrc(chunkId) {
        return __webpack_require__.p + '' + ({'vendors.main': 'vendors.main'}[chunkId] || chunkId) + '.js';
    }

    // 忽略一部分相同代码

    // 新增__webpack_require__.e方法，异步加载 js
    __webpack_require__.e = function requireEnsure(chunkId) {
        var promises = [];

        // JSONP chunk loading for javascript
```

```
        var installedChunkData = installedChunks[chunkId];
        if (installedChunkData !== 0) {
            // 0 means "already installed".

            // a Promise means "currently loading".
            if (installedChunkData) {
                promises.push(installedChunkData[2]);
            } else {
                // setup Promise in chunk cache
                var promise = new Promise(function(resolve, reject) {
                    installedChunkData = installedChunks[chunkId] = [resolve, reject];
                });
                promises.push((installedChunkData[2] = promise));

                // start chunk loading
                var script = document.createElement('script');
                var onScriptComplete;

                script.charset = 'utf-8';
                script.timeout = 120;
                if (__webpack_require__.nc) {
                    script.setAttribute('nonce', __webpack_require__.nc);
                }
                script.src = jsonpScriptSrc(chunkId);

                onScriptComplete = function(event) {
                    // avoid mem leaks in IE.
                    script.onerror = script.onload = null;
                    clearTimeout(timeout);
                    var chunk = installedChunks[chunkId];
                    if (chunk !== 0) {
                        if (chunk) {
                            var errorType = event && (event.type === 'load' ? 'missing' : event.type);
                            var realSrc = event && event.target && event.target.src;
                            var error = new Error(
                                'Loading chunk ' + chunkId + ' failed.\n(' + errorType + ': ' + realSrc + ')'
                            );
                            error.type = errorType;
                            error.request = realSrc;
                            chunk[1](error);
                        }
                        installedChunks[chunkId] = undefined;
                    }
                };
                var timeout = setTimeout(function() {
                    onScriptComplete({type: 'timeout', target: script});
                }, 120000);
                script.onerror = script.onload = onScriptComplete;
                document.head.appendChild(script);
            }
        }
        return Promise.all(promises);
    };

    // 忽略一部分相同代码
    // 新增一个 oe 方法，用于报错处理
    __webpack_require__.oe = function(err) {
        console.error(err);
        throw err;
    };

    // 下面部分代码没有变化
    var jsonpArray = (window['webpackJsonp'] = window['webpackJsonp'] || []);
    var oldJsonpFunction = jsonpArray.push.bind(jsonpArray);
    jsonpArray.push = webpackJsonpCallback;
    jsonpArray = jsonpArray.slice();
    for (var i = 0; i < jsonpArray.length; i++) webpackJsonpCallback(jsonpArray[i]);
    var parentJsonpFunction = oldJsonpFunction;

    // return 返回的是 __webpack_require__(entry)的代码
    return __webpack_require__((__webpack_require__.s = './src/app.js'));
})({
    './src/name.js': function(module, exports) {
```

```
    './src/name.js': function(module, exports) {
        module.exports = 'Alex';
    },

    './src/app.js': function(module, exports, __webpack_require__) {
        const name = __webpack_require__('./src/name.js');
        __webpack_require__
            .e('vendors.main')
            .then(
                (require => {
                    __webpack_require__('./node_modules/zepto/dist/zepto.js');
                }).bind(null, __webpack_require__)
            )
            .catch(__webpack_require__.oe);
    }
});
```

通过观察代码，发现增加了一个关键函数`__webpack_require__.e`，用 Promise 的方式来异步加载 js 模块。代码的起点 IIFE 函数，只不过`app.js`内的`import()`部分代码被处理成了下面代码：

```
__webpack_require__
    .e('vendors.main')
    .then(
        (require => {
            __webpack_require__('./node_modules/zepto/dist/zepto.js');
        }).bind(null, __webpack_require__)
    )
    .catch(__webpack_require__.oe);
```

所以核心是`__webpack_require__.e`函数

`__webpack_require__.e`函数

简单来说，`__webpack_require__.e`是通过创建`script`标签的方式来异步加载 js 文件，然后返回一个 Promise 对象：

```
__webpack_require__.e = function requireEnsure(chunkId) {
    // promise 数组
    var promises = [];
    // 1. 判断状态，状态为不为0，即没有加载，没加载则进入加载逻辑
    var installedChunkData = installedChunks[chunkId];
    if (installedChunkData !== 0) {
        // 还记得之前installedChunkData的状态吗？
        // 在这里chunk的状态有以下几种：
        // 1.1. undefined 没有加载
        // 1.2. 0 加载完成
        // 1.3. Promise 正在加载中，单一个模块被多次依赖引用的时候，这时候通过检测这个状态可以判断模块正在被加载，不需要重复创建script 加载 js
        if (installedChunkData) {
            promises.push(installedChunkData[2]);
        } else {
            // 2. 首先创建一个 promise 对象
            var promise = new Promise(function(resolve, reject) {
                installedChunkData = installedChunks[chunkId] = [resolve, reject];
            });
            // installedChunks[chunkId] 内容是：[resole, reject, promise实例]
            promises.push((installedChunkData[2] = promise));

            // 3. 创建一个 script 标签
            var script = document.createElement('script');
            var onScriptComplete;

            script.charset = 'utf-8';
            script.timeout = 120;
            if (__webpack_require__.nc) {
                // 添加 script 的 nonce 属性
                script.setAttribute('nonce', __webpack_require__.nc);
            }
            // 4. 添加 script 标签的 src
            script.src = jsonpScriptSrc(chunkId);
            // 5. onload 回到函数
            onScriptComplete = function(event) {
                // avoid mem leaks in IE.
                script.onerror = script.onload = null;
                clearTimeout(timeout);
                var chunk = installedChunks[chunkId];
                // 判断状态，保证只执行一次，0表示加载完成
                // 划重点：下面逻辑中需要注意，并没有加载成功的逻辑！！而是只有加载失败的逻辑
                if (chunk !== 0) {
                    if (chunk) {
                        var errorType = event && (event.type === 'load' ? 'missing' : event.type);
                        var realSrc = event && event.target && event.target.src;
                        var error = new Error(
                            'Loading chunk ' + chunkId + ' failed.\n(' + errorType + ': ' + realSrc + ')'
                        );
                        error.type = errorType;
                        error.request = realSrc;
                        // 出现错误了，执行 promise 的 reject
                        chunk[1](error);
                    }
                    installedChunks[chunkId] = undefined;
                }
            };
            // 设置加载超时时间为120s
            var timeout = setTimeout(function() {
                onScriptComplete({type: 'timeout', target: script});
            }, 120000);
            // 6. 添加回调函数绑定
            script.onerror = script.onload = onScriptComplete;
            // 7. 添加到 document head 中，开始执行
            document.head.appendChild(script);
        }
    }
    // 返回一个 promise.all
    return Promise.all(promises);
};
```

在上面的代码中，需要重点说下：

1. `installedChunks`的状态，在这里 chunk 的状态有以下几种： 1. undefined：没有加载 2. 0：加载完成 3. Promise：正在加载中，当一个模块被多次依赖引用的时候，这时候通过检测这个状态可以判断模块正在被加载，不需要重复创建 script 加载 js
2. **上面代码解决了加载的问题，但是没有找到加载成功的代码，只有 error 的逻辑，详见上面的代码注释**

那么加载成功的代码去哪里了呢？

我们还记得在`splitChunks`部分提到的重写`webpackJsonp.push`的方法吗？这里加载的`vendors.main.js`实际在往数组添加项目，执行`webpackJsonp.push`的时候已经执行`webpackJsonpCallback`了：

```
(window['webpackJsonp'] = window['webpackJsonp'] || []).push([
    ['vendors.main'],
    {
        './node_modules/zepto/dist/zepto.js': function(module, exports, __webpack_require__) {}
    }
]);
```

首先在`__webpack_require__.e`中定义了`installChunks`的内容代码：

```
// 2. 首先创建一个 promise 对象
var promise = new Promise(function(resolve, reject) {
    installedChunkData = installedChunks[chunkId] = [resolve, reject];
});
// installedChunks[chunkId] 内容是: [resole, reject, promise实例]
promises.push((installedChunkData[2] = promise));
```

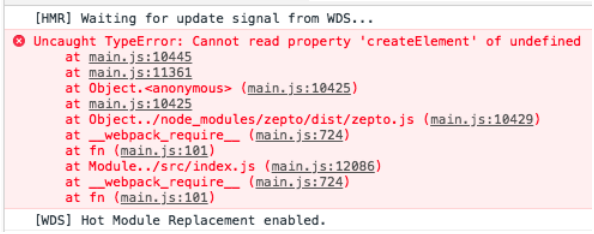
`installedChunks[chunkId]` 内容是：`[resole, reject, promise实例]`，在`webpackJsonpCallback`函数中，被读取出来`resolve`然后执行了！详见下面的注释：

```
// main.js webpackJsonpCallback
function webpackJsonpCallback(data) {
    var chunkIds = data[0];
    var moreModules = data[1];

    var moduleId,
        chunkId,
        i = 0,
        resolves = []; // 这个resolves数组收集 加载js文件成功的 promise resolve 函数
    for (; i < chunkIds.length; i++) {
        chunkId = chunkIds[i];
        // 1. 这里取出 installedChunks[chunkId], 判断成功
        if (installedChunks[chunkId]) {
            // 2. 收集 resolve, installedChunks[chunkId][0] 就是 resolve!
            resolves.push(installedChunks[chunkId][0]);
        }
        // 3. 设置成功状态, 修改为0
        installedChunks[chunkId] = 0;
    }
    for (moduleId in moreModules) {
        if (Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
            modules[moduleId] = moreModules[moduleId];
        }
    }
    if (parentJsonpFunction) parentJsonpFunction(data);

    // 4. 最后遍历数组执行 resolve
    while (resolves.length) {
        resolves.shift()();
    }
}
```

整个过程执行完毕！

「遗留问题」zepto 引用报错的问题

本文第二部分，解析`splitChunk`产出的代码时，故意预留一个坑：zepto 执行的时候会报错，具体如下：



通过分析报错原因，我们来更好地理解下 Webpack 的模块执行机制。

首先看下 Zepto 的源码结构：

```
/* Zepto v1.2.0 - zepto event ajax form ie - zeptojs.com/license */
(function(global, factory) {
    if (typeof define === 'function' && define.amd)
        define(function() {
            return factory(global);
        });
    else factory(global);
})(this, function(window) {
    var Zepto = (function() {
        // ...

        return $;
    })();

    window.Zepto = Zepto;
    window.$ === undefined && (window.$ = Zepto);

    return Zepto;
});
```

可以看出，它只使用了 AMD 规范的模块导出方法 define，没有用 CommonJs 规范的方法 `module.exports` 来导出模块，不过这不是造成报错的原因。

被 Webpack 打包后，zepto 的内容变成：

```
(function(module, exports, __webpack_require__) {
    var __WEBPACK_AMD_DEFINE_RESULT__; /* Zepto v1.2.0 - zepto event ajax form ie - zeptojs.com/license */
    (function(global, factory) {
        if (true)
            !((__WEBPACK_AMD_DEFINE_RESULT__ = function() {
                return factory(global);
            }.call(exports, __webpack_require__, exports, module)),
            __WEBPACK_AMD_DEFINE_RESULT__ !== undefined && (module.exports = __WEBPACK_AMD_DEFINE_RESULT__));
        else {
        }
    })(this, function(window) {
        var Zepto = (function() {
            // ...

            return $;
        })();

        window.Zepto = Zepto;
        window.$ === undefined && (window.$ = Zepto);
        return Zepto;
    });
});
```

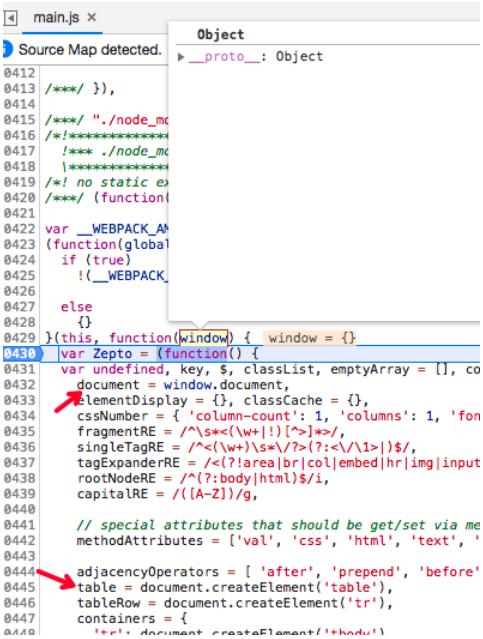
以上代码是模块执行的闭包，化简一下其实就是 Webpack 把 AMD 规范的 `define` 方法转换成了 `module.export = factory(global)`，以此来获取 factory 方法返回的对象。

在模块加载（import/require）时，Webpack 会通过下面这种方法来执行模块闭包并导入模块：

```
function __webpack_require__(moduleId) {
    // Check if module is in cache
    if (installedModules[moduleId]) {
        return installedModules[moduleId].exports;
    }
    // Create a new module (and put it into the cache)
    var module = (installedModules[moduleId] = {
        i: moduleId,
        l: false,
        exports: {},
        hot: hotCreateModule(moduleId),
        parents: ((hotCurrentParentsTemp = hotCurrentParents), (hotCurrentParents = []), hotCurrentParentsTemp),
        children: []
    });

    // Execute the module function
    modules[moduleId].call(module.exports, module, module.exports, hotCreateRequire(moduleId));

    // Flag the module as loaded
    module.l = true;

    // Return the exports of the module
    return module.exports;
}
```

其核心在于 `modules[moduleId].call`，它会传入新初始化的 `module.exports` 来作为模块闭包的上下文（`context`），并运行模块闭包来将模块暴露的对象加入到已加载的模块对象（`installedModules`）中。

所以对于 Zepto 来说，它初始化时使用的 `this`（见下图）其实就是 `module.exports`，但这个 `module.exports` 没有赋值过任何变量，即 Zepto 初始化使用的 `this` 为空对象。



而在 Zepto 中，实际想用的 `this` 是`window`对象，所以 `factory(global)` 中 `global` 为空对象，Zepto 运行函数中的 `window` 也就变成了空对象，而 `document = window.document`，这个 `document` 为 `undefined`，因此会造成 `document.createElement` 会报 `TypeError`。

解决 Webpack 引入 Zepto 报错问题

要解决这个问题，需要使用两个 loader：script-loader和exports-loader。

script-loader 会把我们指定的模块 JavaScript 文件转成纯字符串，并用 `eval.call(null, string)` 执行，这样执行的作用域就为全局作用域了，即这里的`this`就是`window`了。

但是如果只用 script-loader，我们要使用 Zepto 对象的时候就不能使用`import/require`引入了，而是直接作为一个全局对象来使用：

```
/*
 * 不能使用 `import $ from 'zepto'`
 * 因为 zepto.js 执行后返回值为 undefined
 * 因为 module.exports 默认初始为空对象
 * 所以 $ 也为空对象
 */

$(function() {});
```

这样的写法就是：当 Webpack 初始化时，zepto.js 会在`eval`下执行一遍，将 Zepto 对象赋值给 `window.$` 并挂载到 Window 上。因此后续的 `$`、`Zepto` 变量就都可用了，但是没有这种方式只执行一次，不是每次调用（`import/require`）的时候都会返回一个引入的对象。这种使用全局对象的实现方法不够友好，还是将对象以 ES6 Module/CommonJs/AMD 方式暴露出来更好。

为了让我们的模块导入更加地「模块化」，可以 `import/require`，而不是像上面那么「与众不同」，我们还需要 exports-loader 的帮助。

exports-loader 可以导出我们指定的对象:

```
require('exports?window.Zepto!./zepto.js');
```

他的作用就是在模块闭包最后加一句 `module.exports = window.Zepto` 来导出我们需要的对象，这样我们就可以愉快地 `import $ from 'zepto'` 了。

所以最后的配置是：

```
// webpack.config
{
    module: {
        rules: [
            // ....
            {
                test: require.resolve('zepto'),
                use: ['exports-loader?window.Zepto', 'script-loader']
            }
        ];
    }
}
```

总结

到此为止，我们已经理解了单文件打包、`splitChunks` 和 `import()` 打包产出物的代码执行全流程。通过分析 Webpack 的构建产出代码执行过程，能够让更加深入理解 Webpack 的内核实现。Webpack 的产出物是公共各自的模板进行拼接代码而成的，针对不同的打包配置，Webpack 打包出来的代码执行过程有所差异。

> Tips：为了便于学习与代码阅读，还可以在 Webpack 配置文件中添加 `optimization:{runtimeChunk: {name: 'runtime'}}` 配置项，这样会让 Webpack 将 runtime 与模块注册代码分开打包，runtime 部分汇单独打包到 `runtime.js` 中。

> 本小节 Webpack 相关面试题：
>
> 1. 在 Webpack 中怎么引入类似 jQuery 和 Zepto 这种没有模块化的代码？
> 2. Webpack 打包出来的代码是怎么执行的？异步加载的模块是怎么执行的？

Webpack 工作流程

Webpack 的模块热替换做了什么？

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论