webpack 的模块化机制

webpack 自己实现了一套模块机制，无论是 CommonJS 模块的 require 语法还是 ES6 模块的 import 语法，都能够被解析并转换成指定环境的可运行代码， webpack 作为当前主流的前端模块化工具，在 webpack 刚开始流行的时候，我们经常通过 webpack 将所有处理文件全部打包成一个 bundle 文件， 先通过一个简单的例子来看：

```
// src/single/index.js
var index2 = require('./index2');
var util = require('./util');
console.log(index2);
console.log(util);

// src/single/index2.js
var util = require('./util');
console.log(util);
module.exports = "index 2";

// src/single/util.js
module.exports = "Hello World";

// 通过 config/webpack.config.single.js 打包
const webpack = require('webpack');
const path = require('path')

module.exports = {
  entry: {
    index: [path.resolve(__dirname, '../src/single/index.js')],
  },
  output: {
    path: path.resolve(__dirname, '../dist'),
    filename: '[name].[chunkhash:8].js'
  },
}
```

我们手动构建后，可看到打包效果，打包内容大致如下(经过精简)：

```
// dist/index.xxxx.js
(function(modules) {
```

```
  // 已经加载过的模块
  var installedModules = {};

  // 模块加载函数
  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;
  }
  return __webpack_require__(__webpack_require__.s = 3);
})([
/* 0 */
(function(module, exports, __webpack_require__) {
  var util = __webpack_require__(1);
  console.log(util);
  module.exports = "index 2";
}),
/* 1 */
(function(module, exports) {
  module.exports = "Hello World";
}),
/* 2 */
(function(module, exports, __webpack_require__) {
  var index2 = __webpack_require__(0);
  index2 = __webpack_require__(0);
  var util = __webpack_require__(1);
  console.log(index2);
  console.log(util);
}),
/* 3 */
(function(module, exports, __webpack_require__) {
  module.exports = __webpack_require__(2);
```

```
})]);
```

1. 首先 webpack 将所有模块(可以简单理解成文件)包裹于一个函数中，并传入默认参数，这里有三个文件再加上一个入口模块一共四个模块，将它们放入一个数组中，取名为 modules，并通过数组的下标来作为 moduleId。
2. 将 modules 传入一个自执行函数中，自执行函数中包含一个 installedModules 已经加载过的模块和一个模块加载函数，最后加载入口模块并返回。
3. *__webpack_require__* 模块加载，先判断 installedModules 是否已加载，加载过了就直接返回 exports 数据，没有加载过该模块就通过 *modules[moduleId].call(module.exports, module, module.exports, __webpack_require__)* 执行模块并且将 module.exports 给返回。

有些点需要注意的是：

1. 每个模块 webpack 只会加载一次,所以重复加载的模块只会执行一次，加载过的模块会放到 installedModules，下次需要需要该模块的值就直接从里面拿了。
2. 模块的 id 直接通过数组下标去一一对应的，这样能保证简单且唯一，通过其它方式比如文件名或文件路径的方式就比较麻烦，因为文件名可能出现重名，不唯一，文件路径则会增大文件体积，并且将路径暴露给前端，不够安全。
3. *modules[moduleId].call(module.exports, module, module.exports, __webpack_require__)*保证了模块加载时 this 的指向 module.exports 并且传入默认参数，很简单，不过多解释。

webpack 单文件打包的方式应付一些简单场景就足够了，但是我们在开发一些复杂的应用，如果没有对代码进行切割，将第三方库(jQuery)或框架(React)和业务代码全部打包在一起，就会导致用户访问页面速度很慢。所以需要多个文件打包。

一个例子：

```
// src/multiple/pageA.js
const utilA = require('./js/utilA');
const utilB = require('./js/utilB');
console.log(utilA);
console.log(utilB);

// src/multiple/pageB.js
const utilB = require('./js/utilB');
console.log(utilB);
// 异步加载文件，类似于 import()
const utilC = () => require.ensure(['./js/utilC'], function(require) {
  console.log(require('./js/utilC'))
});
```

```
utilC();

// src/multiple/js/utilA.js 可类比于公共库，如 jQuery
module.exports = "util A";

// src/multiple/js/utilB.js
module.exports = 'util B';

// src/multiple/js/utilC.js
module.exports = "util C";
```

这里我们定义了两个入口 pageA 和 pageB 和三个库 util，我们希望代码切割做到：

1. 因为两入口都是用到了 utilB，我们希望把它抽离成单独文件，并且当用户访问 pageA 和 pageB 的时候都能去加载 utilB 这个公共模块，而不是存在于各自的入口文件中。
2. pageB 中 utilC 不是页面一开始加载时候就需要的内容，假如 utilC 很大，我们不希望页面加载时就直接加载 utilC，而是当用户达到某种条件(如：点击按钮)才去异步加载 utilC，这时候我们需要将 utilC 抽离成单独文件，当用户需要的时候再去加载该文件。

那么 webpack 需要怎么配置呢？

```
// 通过 config/webpack.config.multiple.js 打包
const webpack = require('webpack');
const path = require('path')

module.exports = {
  entry: {
    pageA: [path.resolve(__dirname, '../src/multiple/pageA.js')],
    pageB: path.resolve(__dirname, '../src/multiple/pageB.js'),
  },
  output: {
    path: path.resolve(__dirname, '../dist'),
    filename: '[name].[chunkhash:8].js',
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: 'vendor',
      minChunks: 2,
    }),
    new webpack.optimize.CommonsChunkPlugin({
```

```
      name: 'manifest',
      chunks: ['vendor']
    })
  ]
}
```

单单配置多 entry 是不够的，这样只会生成两个 bundle 文件，将 pageA 和 pageB 所需要的内容全部放入，跟单入口文件并没有区别，要做到代码切割，我们需要借助 webpack 内置的插件 CommonsChunkPlugin。

webpack 执行存在一部分运行时代码，即一部分初始化的工作，就像之前单文件中的 *__webpack_require__*，这部分代码需要加载于所有文件之前，相当于初始化工作，少了这部分初始化代码，后面加载过来的代码就无法识别并工作了。

另外 webpack 默认会抽离异步加载的代码，这个不需要你做额外的配置，pageB 中异步加载的 utilC 文件会直接抽离为 chunk.xxxx.js 文件。

所以这时候我们页面加载文件的顺序就会变成：

```
mainifest.xxxx.js // 初始化代码
vendor.xxxx.js    // pageA 和 pageB 共同用到的模块，抽离
pageX.xxxx.js     // 业务代码
当 pageB 需要 utilC 时候则异步加载
手动打包后，查看打包内容，首先来看下 manifest 如何做初始化工作(精简版)
// dist/mainifest.xxxx.js
(function(modules) {
  window["webpackJsonp"] = function webpackJsonpCallback(chunkIds,
moreModules) {
    var moduleId, chunkId, i = 0, callbacks = [];
    for(;i < chunkIds.length; i++) {
      chunkId = chunkIds[i];
      if(installedChunks[chunkId])
        callbacks.push.apply(callbacks, installedChunks[chunkId]);
      installedChunks[chunkId] = 0;
    }
    for(moduleId in moreModules) {
      if(Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
        modules[moduleId] = moreModules[moduleId];
      }
    }
    while(callbacks.length)
      callbacks.shift().call(null, __webpack_require__);
```

```
    if(moreModules[0]) {
      installedModules[0] = 0;
      return __webpack_require__(0);
    }
  };
  var installedModules = {};
  var installedChunks = {
    4:0
  };
  function __webpack_require__(moduleId) {
    // 和单文件一致
  }
  __webpack_require__.e = function requireEnsure(chunkId, callback) {
    if(installedChunks[chunkId] === 0)
      return callback.call(null, __webpack_require__);
    if(installedChunks[chunkId] !== undefined) {
      installedChunks[chunkId].push(callback);
    } else {
      installedChunks[chunkId] = [callback];
      var head = document.getElementsByTagName('head')[0];
      var script = document.createElement('script');
      script.type = 'text/javascript';
      script.charset = 'utf-8';
      script.async = true;
      script.src = __webpack_require__.p + "" + chunkId + "." +
({"0":"pageA","1":"pageB","3":"vendor"}[chunkId]||chunkId) + "." +
{"0":"e72ce7d4","1":"69f6bbe3","2":"9adbbaa0","3":"53fa02a7"}[chunkId] +
".js";
      head.appendChild(script);
    }
  };
})([]);
```

与单文件内容一致，定义了一个自执行函数，因为它不包含任何模块，所以传入一个空数组。除了定义了 *__webpack_require__*，还另外定义了两个函数用来进行加载模块。

首先讲解代码前需要理解两个概念，分别是 module 和 chunk

1. chunk 代表生成后 js 文件，一个 chunkId 对应一个打包好的 js 文件(一共五个)，从这段代码可以看出，manifest 的 chunkId 为 4，并且从代码中还可以看到：0-3 分别对应 pageA, pageB, 异步 utilC, vendor 公共模块文件，这也就是

我们为什么不能将这段代码放在 vendor 的原因，因为文件的 hash 值会变。内容变了，vendor 生成的 hash 值也就变了。

2. module 对应着模块，可以简单理解为打包前每个 js 文件对应一个模块，也就是之前 *__webpack_require__* 加载的模块，同样的使用数组下标作为 moduleId 且是唯一不重复的。

那么为什么要区分 chunk 和 module 呢？

首先使用 installedChunks 来保存每个 chunkId 是否被加载过，如果被加载过，则说明该 chunk 中所包含的模块已经被放到了 modules 中，注意是 modules 而不是 installedModules。我们先来简单看一下 vendor chunk 打包出来的内容。

```
// vendor.xxxx.js
webpackJsonp([3,4],{
  3: (function(module, exports) {
    module.exports = 'util B';
  })
});
```

在执行完 manifest 后就会先执行 vendor 文件，结合上面 webpackJsonp 的定义，我们可以知道 [3, 4] 代表 chunkId，当加载到 vendor 文件后，installedChunks[3] 和 installedChunks[4] 将会被置为 0，这表明 chunk3，chunk4 已经被加载过了。

webpackJsonpCallback 一共有两个参数，chuckIds 一般包含该 chunk 文件依赖的 chunkId 以及自身 chunkId，moreModules 代表该 chunk 文件带来新的模块。

```
var moduleId, chunkId, i = 0, callbacks = [];
for(;i < chunkIds.length; i++) {
  chunkId = chunkIds[i];
  if(installedChunks[chunkId])
    callbacks.push.apply(callbacks, installedChunks[chunkId]);
  installedChunks[chunkId] = 0;
}
for(moduleId in moreModules) {
  if(Object.prototype.hasOwnProperty.call(moreModules, moduleId)) {
    modules[moduleId] = moreModules[moduleId];
  }
}
while(callbacks.length)
  callbacks.shift().call(null, __webpack_require__);
if(moreModules[0]) {
  installedModules[0] = 0;
```

```
  return __webpack_require__(0);
}
```

简单说说 *webpackJsonpCallback* 做了哪些事，首先判断 chunkIds 在 installedChunks 里有没有回调函数函数未执行完，有的话则放到 callbacks 里，并且等下统一执行，并将 chunkIds 在 installedChunks 中全部置为 0，然后将 moreModules 合并到 modules。

这里面只有 modules[0] 是不固定的，其它 modules 下标都是唯一的，在打包的时候 webpack 已经为它们统一编号，而 0 则为入口文件即 pageA，pageB 各有一个 module[0]。

然后将 callbacks 执行并清空，保证了该模块加载开始前所以前置依赖内容已经加载完毕，最后判断 moreModules[0]，有值说明该文件为入口文件，则开始执行入口模块 0。

但是考虑到异步加载 js 文件的时候(比如 pageB 异步加载 utilC 文件)，就没那么简单，我们先来看下 webpack 是如何加载异步脚本的：

```
// 异步加载函数挂载在 __webpack_require__.e 上
__webpack_require__.e = function requireEnsure(chunkId, callback) {
  if(installedChunks[chunkId] === 0)
    return callback.call(null, __webpack_require__);

  if(installedChunks[chunkId] !== undefined) {
    installedChunks[chunkId].push(callback);
  } else {
    installedChunks[chunkId] = [callback];
    var head = document.getElementsByTagName('head')[0];
    var script = document.createElement('script');
    script.type = 'text/javascript';
    script.charset = 'utf-8';
    script.async = true;

    script.src = __webpack_require__.p + "" + chunkId + "." +
({"0":"pageA","1":"pageB","3":"vendor"}[chunkId]||chunkId) + "." +
{"0":"e72ce7d4","1":"69f6bbe3","2":"9adbbaa0","3":"53fa02a7"}[chunkId] +
".js";
    head.appendChild(script);
  }
};
```

大致分为三种情况，(已经加载过，正在加载中以及从未加载过)

1. 已经加载过该 chunk 文件，那就不用再重新加载该 chunk 了，直接执行回调函数即可，可以理解为假如页面有两种操作需要加载加载异步脚本，但是两个脚本都依赖于公共模块，那么第二次加载的时候发现之前第一次操作已经加载过了该 chunk，则不用再去获取异步脚本了，因为该公共模块已经被执行过了。
2. 从未加载过，则动态地去插入 script 脚本去请求 js 文件，这也就为什么取名 *webpackJsonpCallback*，因为跟 jsonp 的思想很类似，所以这种异步加载脚本在做脚本错误监控时经常出现 Script error。
3. 正在加载中代表该 chunk 文件已经在加载中了，比如说点击按钮触发异步脚本，用户点太快了，连点两次就可能出现这种情况，此时将回调函数放入 installedChunks。