Webpack-Day3

webpack打包原理分析

webpack 在执行npx webpack进行打包后，都干了什么事情?

```
(function(modules) {
  var installedModules = {};

  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 =
"./index.js"));
})({
  "./index.js": function(module, exports) {
    eval(
      '// import a from "./a";\n\nconsole.log("hello
word");\n\n\n//# sourceURL=webpack:///./index.js?'
    ),
       "./a.js": function(module, exports) {
    eval(
      '// import a from "./a";\n\nconsole.log("hello
word");\n\n\n//# sourceURL=webpack:///./index.js?'
    ),
       "./b.js": function(module, exports) {
    eval(
      '// import a from "./a";\n\nconsole.log("hello
word");\n\n\n//# sourceURL=webpack:///./index.js?'
    );
  }
});
```

大概的意思就是，我们实现了一个**webpack_require** 来实现自己的模块化，把代码都缓存在installedModules里，代码文件以对象传递进来，key是路径，value是包裹的代码字符串，并且代码内部的require，都被替换成了**webpack_require**

自己实现一个bundle.js

- 模块分析：读取入口文件，分析代码

```
const fs = require("fs");

const fenximokuai = filename => {
  const content = fs.readFileSync(filename, "utf-8");
  console.log(content);
};

fenximokuai("./index.js");
```

- 拿到文件中依赖，这里我们不推荐使用字符串截取，引入的模块名越多，就越麻烦，不灵活，这里我们推荐使用@babel/parser，这是babel7的工具，来帮助我们分析内部的语法，包括es6，返回一个ast抽象语法树

@babel/parser:https://babeljs.io/docs/en/babel-parser

```
//安装@babel/parser
npm install @babel/parser --save


//bundle.js
const fs = require("fs");
const parser = require("@babel/parser");

const fenximokuai = filename => {
  const content = fs.readFileSync(filename, "utf-8");

  const Ast = parser.parse(content, {
    sourceType: "module"
  });
  console.log(Ast.program.body);
};
```

```
fenximokuai("./index.js");
```

- 接下来我们就可以根据body里面的分析结果，遍历出所有的引入模块，但是比较麻烦，这里还是推荐babel推荐的一个模块@babel/traverse，来帮我们处理。

  npm install @babel/traverse --save

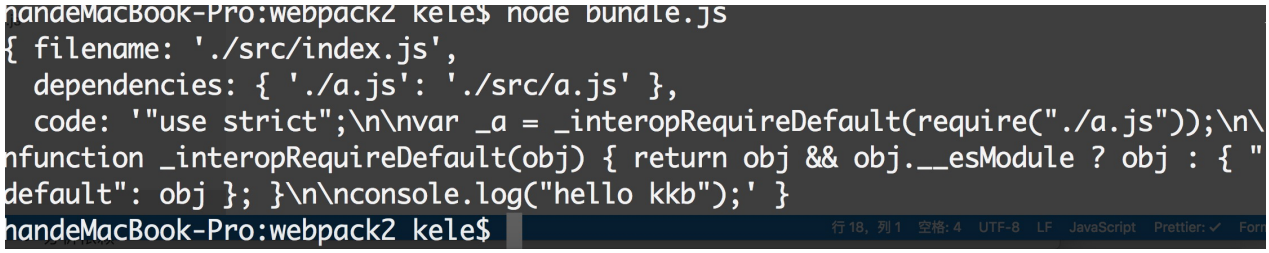
```
const fs = require("fs");
const path = require("path");
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;

const fenximokuai = filename => {
  const content = fs.readFileSync(filename, "utf-8");

  const Ast = parser.parse(content, {
    sourceType: "module"
  });

  const dependencies = [];
  //分析ast抽象语法树，根据需要返回对应数据，
  //根据结果返回对应的模块，定义一个数组，接受一下node.source.value的值
  traverse(Ast, {
    ImportDeclaration({ node }) {
      console.log(node);
      dependencies.push(node.source.value);
    }
  });
  console.log(dependencies);
};

fenximokuai("./index.js");
```



分析上图，我们要分析出信息：

- 入口文件
- 入口文件引入的模块
  - 引入路径
  - 在项目中里的路径
- 可以在浏览器里执行的代码

处理现在的路径问题：

```
//需要用到path模块
const parser = require("@babel/parser");

//修改 dependencies 为对象，保存更多的信息
const dependencies = {};

//分析出引入模块，在项目中的路径
const newfilename =
        "./" + path.join(path.dirname(filename),
node.source.value);

//保存在dependencies里
dependencies[node.source.value] = newfilename;
```

把代码处理成浏览器可运行的代码，需要借助@babel/core，和@babel/preset-env，把ast语法树转换成合适的代码

```
const babel = require("@babel/core");

const { code } = babel.transformFromAst(Ast, null, {
    presets: ["@babel/preset-env"]
  });
```

导出所有分析出的信息：

```
return {
    filename,
    dependencies,
    code
  };
```

完成代码参考：

```
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');

const moduleAnalyser = (filename) => {
  const content = fs.readFileSync(filename, 'utf-8');
  const ast = parser.parse(content, {
    sourceType: 'module'
  });
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
```

```
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname,
node.source.value);
      dependencies[node.source.value] = newFile;
    }
  });
  const { code } = babel.transformFromAst(ast, null, {
    presets: ["@babel/preset-env"]
  });
  return {
    filename,
    dependencies,
    code
  }
}

const moduleInfo = moduleAnalyser('./src/index.js');
console.log(moduleInfo);
```

- 分析依赖

上一步我们已经完成了一个模块的分析，接下来我们要完成项目里所有模块的分析：

```
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');

const moduleAnalyser = (filename) => {
  const content = fs.readFileSync(filename, 'utf-8');
```

```
  const ast = parser.parse(content, {
    sourceType: 'module'
  });
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname,
node.source.value);
      dependencies[node.source.value] = newFile;
    }
  });
  const { code } = babel.transformFromAst(ast, null, {
    presets: ["@babel/preset-env"]
  });
  return {
    filename,
    dependencies,
    code
  }
}

const makeDependenciesGraph = (entry) => {
  const entryModule = moduleAnalyser(entry);
  const graphArray = [ entryModule ];
  for(let i = 0; i < graphArray.length; i++) {
    const item = graphArray[i];
    const { dependencies } = item;
    if(dependencies) {
      for(let j in dependencies) {
        graphArray.push(
          moduleAnalyser(dependencies[j])
        );
      }
    }
  }
  const graph = {};
```

```
  graphArray.forEach(item => {
    graph[item.filename] = {
      dependencies: item.dependencies,
      code: item.code
    }
  });
  return graph;
}

const graghInfo = makeDependenciesGraph('./src/index.js');
console.log(graghInfo);
```

- 生成代码

```
const fs = require('fs');
const path = require('path');
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const babel = require('@babel/core');

const moduleAnalyser = (filename) => {
  const content = fs.readFileSync(filename, 'utf-8');
  const ast = parser.parse(content, {
    sourceType: 'module'
  });
  const dependencies = {};
  traverse(ast, {
    ImportDeclaration({ node }) {
      const dirname = path.dirname(filename);
      const newFile = './' + path.join(dirname,
node.source.value);
      dependencies[node.source.value] = newFile;
    }
  });
  const { code } = babel.transformFromAst(ast, null, {
    presets: ["@babel/preset-env"]
```

```
  });
  return {
    filename,
    dependencies,
    code
  }
}

const makeDependenciesGraph = (entry) => {
  const entryModule = moduleAnalyser(entry);
  const graphArray = [ entryModule ];
  for(let i = 0; i < graphArray.length; i++) {
    const item = graphArray[i];
    const { dependencies } = item;
    if(dependencies) {
      for(let j in dependencies) {
        graphArray.push(
          moduleAnalyser(dependencies[j])
        );
      }
    }
  }
  const graph = {};
  graphArray.forEach(item => {
    graph[item.filename] = {
      dependencies: item.dependencies,
      code: item.code
    }
  });
  return graph;
}

const generateCode = (entry) => {
  const graph = JSON.stringify(makeDependenciesGraph(entry));
  return `
    (function(graph){
      function require(module) {
```

```
        function localRequire(relativePath) {
          return
require(graph[module].dependencies[relativePath]);
        }
        var exports = {};
        (function(require, exports, code){
          eval(code)
        })(localRequire, exports, graph[module].code);
        return exports;
      };
      require('${entry}')
    })(${graph});
  `;
}

const code = generateCode('./src/index.js');
console.log(code);
```

node调试工具使用

- 修改scripts

```
    "debug": "node --inspect --inspect-brk
node_modules/webpack/bin/webpack.js"
```

如何自己编写一个Loader

自己编写一个Loader的过程是比较简单的，

Loader就是一个函数，**声明式函数**，不能用箭头函数

拿到源代码，作进一步的修饰处理，再返回处理后的源码就可以了

官方文档：https://webpack.js.org/contribute/writing-a-loader/

接口文档：https://webpack.js.org/api/loaders/

简单案例

- 创建一个替换源码中字符串的loader

```
//index.js
console.log("hello kkb");


//replaceLoader.js
module.exports = function(source) {
  console.log(source, this, this.query);
  return source.replace('kkb','开课吧')
};


//需要用声明式函数，因为要上到上下文的this,用到this的数据，该函数接受一个参数，是源码
```

- 在配置文件中使用loader

```
//需要使用node核心模块path来处理路径
const path = require('path')
module: {
    rules: [
      {
        test: /\.js$/,
        use: path.resolve(__dirname,
"./loader/replaceLoader.js")
      }
    ]
  },
```

- 如何给loader配置参数，loader如何接受参数?
  - this.query
  - loader-utils

```
//webpack.config.js
module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            loader: path.resolve(__dirname,
"./loader/replaceLoader.js"),
            options: {
              name: "开课吧"
            }
          }
        ]
      }
    ]
  },

//replaceLoader.js
```

```
//const loaderUtils = require("loader-utils");//官方推荐处理
loader,query的工具

module.exports = function(source) {
  //this.query 通过this.query来接受配置文件传递进来的参数

  //return source.replace("kkb", this.query.name);
  const options = loaderUtils.getOptions(this);
  const result = source.replace("kkb", options.name);
  return source.replace("kkb", options.name);
}
```

- **this.callback** :如何返回多个信息，不止是处理好的源码呢，可以使用this.callback来处理

```
//replaceLoader.js
const loaderUtils = require("loader-utils");//官方推荐处理
loader,query的工具

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  const result = source.replace("kkb", options.name);
  this.callback(null, result);
};


//this.callback(
  err: Error | null,
  content: string | Buffer,
  sourceMap?: SourceMap,
  meta?: any
);
```

- **this.async**：如果loader里面有异步的事情要怎么处理呢

```
const loaderUtils = require("loader-utils");

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  setTimeout(() => {
    const result = source.replace("kkb", options.name);

    return result;
  }, 1000);
};
//先用setTimeout处理下试试，发现会报错
```

我们使用this.asycn来处理，他会返回this.callback

```
const loaderUtils = require("loader-utils");

module.exports = function(source) {
  const options = loaderUtils.getOptions(this);

  //定义一个异步处理，告诉webpack,这个loader里有异步事件,在里面调用下这个异步
  //callback 就是 this.callback 注意参数的使用
  const callback = this.async();
  setTimeout(() => {
    const result = source.replace("kkb", options.name);
    callback(null, result);
  }, 3000);
};
```

- 多个loader的使用

```
//replaceLoader.js
module.exports = function(source) {
```

```
  return source.replace("开课吧", "word");
};
//replaceLoaderAsync.js
const loaderUtils = require("loader-utils");
module.exports = function(source) {
  const options = loaderUtils.getOptions(this);
  //定义一个异步处理，告诉webpack,这个loader里有异步事件,在里面调用下这个异步
  const callback = this.async();
  setTimeout(() => {
    const result = source.replace("kkb", options.name);
    callback(null, result);
  }, 3000);
};

//webpack.config.js
module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          path.resolve(__dirname, "./loader/replaceLoader.js"),
          {
            loader: path.resolve(__dirname,
"./loader/replaceLoaderAsync.js"),
            options: {
              name: "开课吧"
            }
          }
        ]
        // use: [path.resolve(__dirname,
"./loader/replaceLoader.js")]
      }
    ]
  },
```

顺序，自下而上，自右到左

- 处理loader的路径问题

```
resolveLoader: {
    modules: ["node_modules", "./loader"]
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          "replaceLoader",
          {
            loader: "replaceLoaderAsync",
            options: {
              name: "开课吧"
            }
          }
        ]
        // use: [path.resolve(__dirname,
"./loader/replaceLoader.js")]
      }
    ]
  },
```

参考： loader API

https://webpack.js.org/api/loaders

Webpack.run(config) complier.hooks.before complier.hooks.after complier.hooks.end complier.hooks.emit

如何自己编写一个Plugin

Plugin: 开始打包，在某个时刻，帮助我们处理一些什么事情的机制

plugin要比loader稍微复杂一些，在webpack的源码中，用plugin的机制还是占有非常大的场景，可以说plugin是webpack的灵魂

设计模式

事件驱动

发布订阅

plugin是一个类，里面包含一个apply函数，接受一个参数，compiler

官方文档：https://webpack.js.org/contribute/writing-a-plugin/

案例：

- 创建copyright-webpack-plugin.js

```
class CopyrightWebpackPlugin {
  constructor() {
  }

  //compiler：webpack实例
  apply(compiler) {

  }
}
module.exports = CopyrightWebpackPlugin;
```

- 配置文件里使用

```
const CopyrightWebpackPlugin = require("./plugin/copyright-webpack-plugin");

plugins: [new CopyrightWebpackPlugin()]
```

- 如何传递参数

```
//webpack配置文件
plugins: [
    new CopyrightWebpackPlugin({
      name: "开课吧"
    })
  ]

//copyright-webpack-plugin.js
class CopyrightWebpackPlugin {
  constructor(options) {
    //接受参数
    console.log(options);
  }

  apply(compiler) {}
}
module.exports = CopyrightWebpackPlugin;
```

- 配置plugin在什么时刻进行

```
class CopyrightWebpackPlugin {
  constructor(options) {
    // console.log(options);
  }
```

```
  apply(compiler) {
    //hooks.emit 定义在某个时刻
    compiler.hooks.emit.tapAsync(
      "CopyrightWebpackPlugin",
      (compilation, cb) => {
        compilation.assets["copyright.txt"] = {
          source: function() {
            return "hello copy";
          },
          size: function() {
            return 20;
          }
        };
        cb();
      }
    );

    //同步的写法
    //compiler.hooks.compile.tap("CopyrightWebpackPlugin",
compilation => {
    //  console.log("开始了");
    //});
  }
}
module.exports = CopyrightWebpackPlugin;
```

参考：compiler-hooks

https://webpack.js.org/api/compiler-hooks

end