05.23 公开课 - 2021大厂前端核心面试题详解

#2021#

一、Webpack

1、Webpack中的module是指什么?

其实webpack中提到的module概念, 和咱们平时前端开发的module概念是一样的.

webpack 支持ESModule, CommonJS, AMD, Assets等.

简单说几个模块的引入和导出方式:

1. ESM

关键字 export 允许将 ESM 中的内容暴露给其他模块,
关键字 import 允许从其他模块获取引用到 ESM 中.

```
import { aa } from './a.js';

export { bb };
```

可以设置 package.json 中的属性来显式设置文件模块类型。
在 package.json 中

- 设置 “type”: “module” 会强制 package.json 下的所有文件使用 ECMAScript 模块。
- 设置 “type”: “commonjs” 将会强制使用 CommonJS 模块。

2. CommonJS

module.exports 允许将 CommonJS 中的内容暴露给其他模块,
require 允许从其他模块获取引用到 CommonJS 中.

```
const path = require('path');
```

```
module.exports = {
    aa,
    bb
}
```

所以webpack modules 如何表达自己的各种依赖关系?

- ESM import 语句
- CommonJS require() 语句
- AMD define 和 require 语句
- css/sass/less 文件中的 @import 语句。
- stylesheet url(...) 或者 HTML 文件中的图片链接。

那么问题又来了, 我们常说的 chunk 和 bundle 的区别是什么?

1. Chunk

   Chunk是Webpack打包过程中Modules的集合，是打包过程中的概念。
   Webpack的打包是从一个入口模块开始，入口模块引用其他模块，模块再引用模块。
   Webpack通过引用关系逐个打包模块，这些module就形成了一个Chunk。
   当然如果有多个入口模块，可能会产出多条打包路径，每条路径都会形成一个Chunk。

2. Bundle

   Bundle是我们最终输出的一个或多个打包好的文件.

3. Chunk 和 Bundle 的关系?

   大多数情况下，一个Chunk会生产一个Bundle。比如咱们来简单写个项目试一下

代码：project/webpack-run-demo

```
module.exports = {
    mode: "production",
    entry: {
        index: "./src/index.js"
    },
```

```
    output: {
        filename: "[name].js"
    }
};
```

但是当我们开启source-map后, chunk和bundle就不是一对一的关系了.

```
module.exports = {
    mode: "production",
    entry: {
        index: "./src/index.js"
    },
    output: {
        filename: "[name].js"
    },
    devtool: "source-map"
};
```

可以看一下webpack的输出, ChunkNames只有一个Index, 而输出了两个bundle. index.js, index.js.map.

所以可以有这样的一个总结:

Chunk是过程中的代码块，Bundle是打包结果输出的代码块, Chunk在构建完成就呈现为Bundle。

4. 生成Chunk的几种方式

- entry配置一个key, value为数组

```
module.exports = {
    mode: "production",
    entry: {
        index: ["./src/index.js", "./src/add.js"]
    },
```

```
    output: {
        filename: "[name].js"
    }
};
```

可以看到这种情况, 也只会产生一个chunk.

- entry配置多个key

```
module.exports = {
    mode: "production",
    entry: {
        index: "./src/index.js",
        common: "./src/common.js"
    },
    output: {
        filename: "[name].js"
    },
};
```

可以看到这种情况, 产生了common和index两个chunk, 配置的key也就会被用为chunkName.

而output中filename字段, 将被用为bundle的名称。

- Split Chunk

咱们来改一下文件引用结构, 修改一下配置。

1. add.js 和 multiply.js 都引用common.js

```
// add.js
import CommonFn from './common.js';

export default function add(a, b) {
    return CommonFn(a + b);
}
```

```
// multiply.js
import CommonFn from './common.js';

export default function multiply(a, b) {
    return CommonFn(a * b);
}
```

2. 安装lodash

```
npm i lodash
```

3. 修改index.js

```
import {
    once
} from 'lodash';
import Add from './add.js';
import Multiply from './multiply.js';

const onceAdd = once(Add);
const addRes = onceAdd(1, 3);
const mulRes = Multiply(2, 4);

console.log(addRes);
console.log(mulRes);
```

4. 修改webpack.config.js

```
module.exports = {
    entry: {
        main: "./src/index.js",
        other: "./src/multiply.js",
    },
    output: {
```

```
        filename: "[name].js",
    },

    optimization: {
        runtimeChunk: "single",
        splitChunks: {
            cacheGroups: {
                commons: {
                    chunks: "initial",
                    minChunks: 2,
                    minSize: 0 // 默认是20000, 这里为了演示生成commonChunks, 对最小
体积不做限制
                },
                vendor: {
                    test: /node_modules/,
                    chunks: "initial",
                    name: "vendor",
                    enforce: true
                }
            },
        }
    }
}
```

5. 看一下这种配置会产生几个chunk?

   配置介绍：https://webpack.docschina.org/configuration/optimization/#optimizationruntimechunk
   官方runtime解释：https://www.webpackjs.com/concepts/manifest/#runtime

- entry main
- entry other
- runtimeChunk: single
- splitChunks commons
- splitChunks vendor

2、比较重要的一些概念

1. Compiler 对象包含了 Webpack 环境所有的的配置信息，包含 options，loaders，plugins 这些信息，这个对象在 Webpack 启动时候被实例化，它是全局唯一的，可以简单地把它理解为 Webpack 实例；
2. Compilation 对象包含了当前的模块资源、编译生成资源、变化的文件等。当 Webpack 以开发模式运行时，每当检测到一个文件变化，一次新的 Compilation 将被创建。Compilation 对象也提供了很多事件回调供插件做扩展。通过 Compilation 也能读取到 Compiler 对象。

Plugin 和 Loader 分别是做什么的? 怎么工作的?

1. Loader

   一句话描述：模块转换器，将非js模块转化为webpack能识别的js模块.

   loader 让 webpack 能够去处理那些非 JavaScript 文件.
   Loader 可以将所有类型的文件转换为 webpack 能够处理的有效模块,然后你就可以利用 webpack 的打包能力,对它们进行处理。
   本质上,webpack loader 将所有类型的文件,转换为应用程序的依赖图（和最终的 bundle）可以直接引用的模块。

2. Plugin

   一句话描述：扩展插件，在webpack运行的各个阶段，都会广播出去相对应的事件，插件可以监听到这些事件的发生，在特定的时机做相对应的事情

   Loader 被用于转换某些类型的模块,而插件则可以用于执行范围更广的任务。
   插件的范围包括,从打包优化和压缩,一直到重新定义环境中的变量。插件接口功能极其强大,可以用来处理各种各样的任务。

   在 webpack 运行的生命周期中会广播出各种事件，Plugin 就可以监听这些事件，在触发时通过 webpack 提供的 API 改变输出结果。
   在插件中，可以拿到 Compile 和 Compilation 的引用对象，使用它们广播事件，这些事件可以被其他插件监听到，或者对他们做出一定修改，其他插件拿到的也是变化的对象。

3、能简单描述一下webpack的打包流程吗?

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数,得出最终的参数。
2. 开始编译：用上一步得到的参数初始化 Compiler 对象,加载所有配置的插件,执行对象的 run

方法开始执行编译。

3. 确定入口：根据配置中的 entry 找出所有的入口文件。
4. 编译模块：从入口文件出发,调用所有配置的 Loader 对模块进行翻译,再找出该模块依赖的模块,再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。
5. 完成模块编译：在经过第 4 步使用 Loader 翻译完所有模块后,得到了每个模块被翻译后的最终内容以及它们之间的依赖关系。
6. 输出资源：根据入口和模块之间的依赖关系,组装成一个个包含多个模块的 Chunk,再把每个 Chunk 转换成一个单独的文件加入到输出列表,这步是可以修改输出内容的最后机会。
7. 输出完成：在确定好输出内容后,根据配置确定输出的路径和文件名,把文件内容写入到文件系统

这么说可能还是有点迷糊, 咱们可以来一步一步的写个打包工具, 感受一下webpack的打包流程

代码： **project/mywebpack**

二、Promise 常见面试代码题（有时间的话可以讲一下）

1. 实现Promise.allSelttled

2. Promise.limit的多种实现