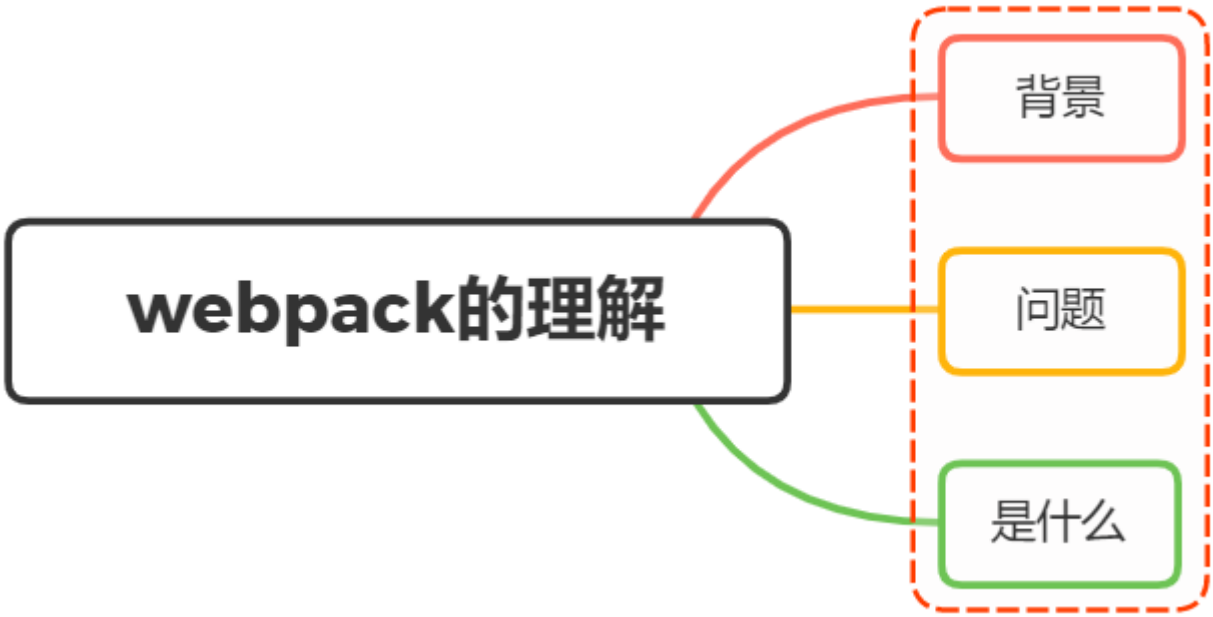
Webpack面试真题（10题）

1. 说说你对webpack的理解？解决了什么问题？



1.1. 背景

`Webpack` 最初的目标是实现前端项目的模块化，旨在更高效地管理和维护项目中的每一个资源

1.1.1. 模块化

最早的时候，我们会通过文件划分的形式实现模块化，也就是将每个功能及其相关状态数据各自单独放到不同的 `JS` 文件中

约定每个文件是一个独立的模块，然后再将这些 `js` 文件引入到页面，一个 `script` 标签对应一个模块，然后调用模块化的成员

```
1 <script src="module-a.js"></script>
2 <script src="module-b.js"></script>
```

但这种模块弊端十分的明显，模块都是在全局中工作，大量模块成员污染了环境，模块与模块之间并没有依赖关系、维护困难、没有私有空间等问题

项目一旦变大，上述问题会尤其明显

随后，就出现了命名空间方式，规定每个模块只暴露一个全局对象，然后模块的内容都挂载到这个对象中

```
window.moduleA = {
  method1: function () {
    console.log('moduleA#method1')
  }
}
```

这种方式也并没有解决第一种方式的依赖等问题

再后来，我们使用立即执行函数为模块提供私有空间，通过参数的形式作为依赖声明，如下

```
// module-a.js
(function ($) {
  var name = 'module-a'
  function method1 () {
    console.log(name + '#method1')
    $('body').animate({ margin: '200px' })
  }
  window.moduleA = {
    method1: method1
  }
})(jQuery)
```

上述的方式都是早期解决模块的方式，但是仍然存在一些没有解决的问题。例如，我们是用过 `script` 标签在页面引入这些模块的，这些模块的加载并不受代码的控制，时间一久维护起来也十分的麻烦

理想的解决方式是，在页面中引入一个 `JS` 入口文件，其余用到的模块可以通过代码控制，按需加载进来

除了模块加载的问题以外，还需要规定模块化的规范，如今流行的则是 `CommonJS` 、 `ES Modules`

1.2. 问题

从后端渲染的 `JSP` 、 `PHP` ，到前端原生 `JavaScript` ，再到 `jQuery` 开发，再到目前的三大框架 `Vue` 、 `React` 、 `Angular`

开发方式，也从 `javascript` 到后面的 `es5` 、 `es6、7、8、9、10` ，再到 `typescript` ，包括编写 `CSS` 的预处理器 `less` 、 `scss` 等

现代前端开发已经变得十分的复杂，所以我们开发过程中会遇到如下的问题：

- 需要通过模块化的方式来开发
- 使用一些高级的特性来加快我们的开发效率或者安全性，比如通过ES6+、TypeScript开发脚本逻辑，通过sass、less等方式来编写css样式代码

- 监听文件的变化来并且反映到浏览器上，提高开发的效率
- JavaScript 代码需要模块化，HTML 和 CSS 这些资源文件也会面临需要被模块化的问题
- 开发完成后我们还需要将代码进行压缩、合并以及其他相关的优化

而 `webpack` 恰巧可以解决以上问题

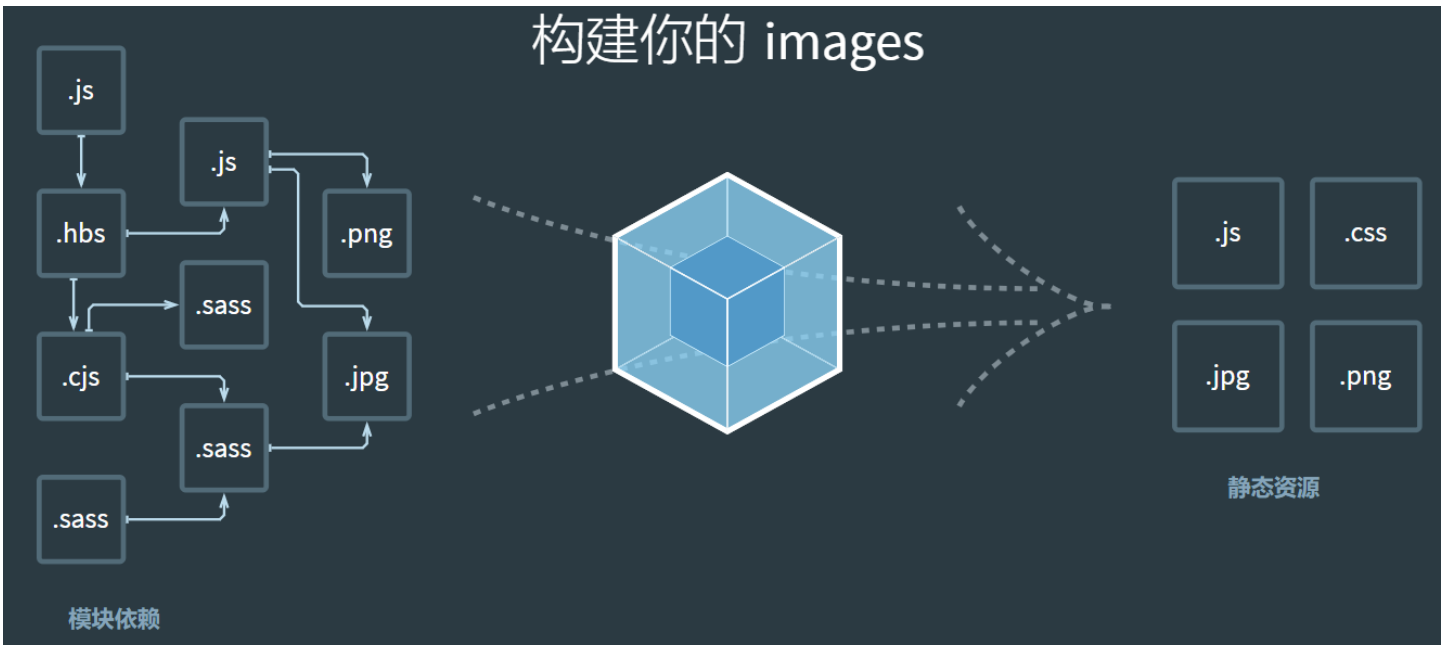
1.3. 是什么

`webpack` 是一个用于现代 `JavaScript` 应用程序的静态模块打包工具

- 静态模块

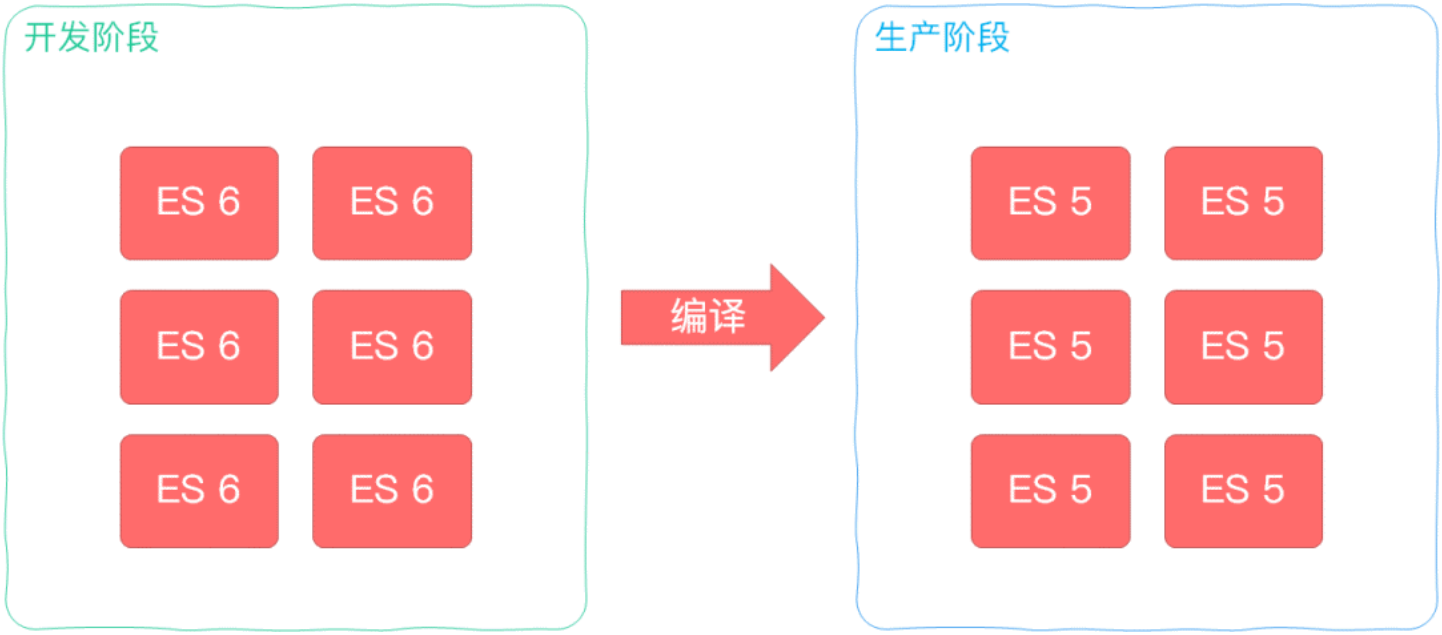
这里的静态模块指的是开发阶段，可以被 `webpack` 直接引用的资源（可以直接被获取打包进 `bundle.js` 的资源）

当 `webpack` 处理应用程序时，它会在内部构建一个依赖图，此依赖图对应映射到项目所需的每个模块（不再局限 `js` 文件），并生成一个或多个 `bundle`

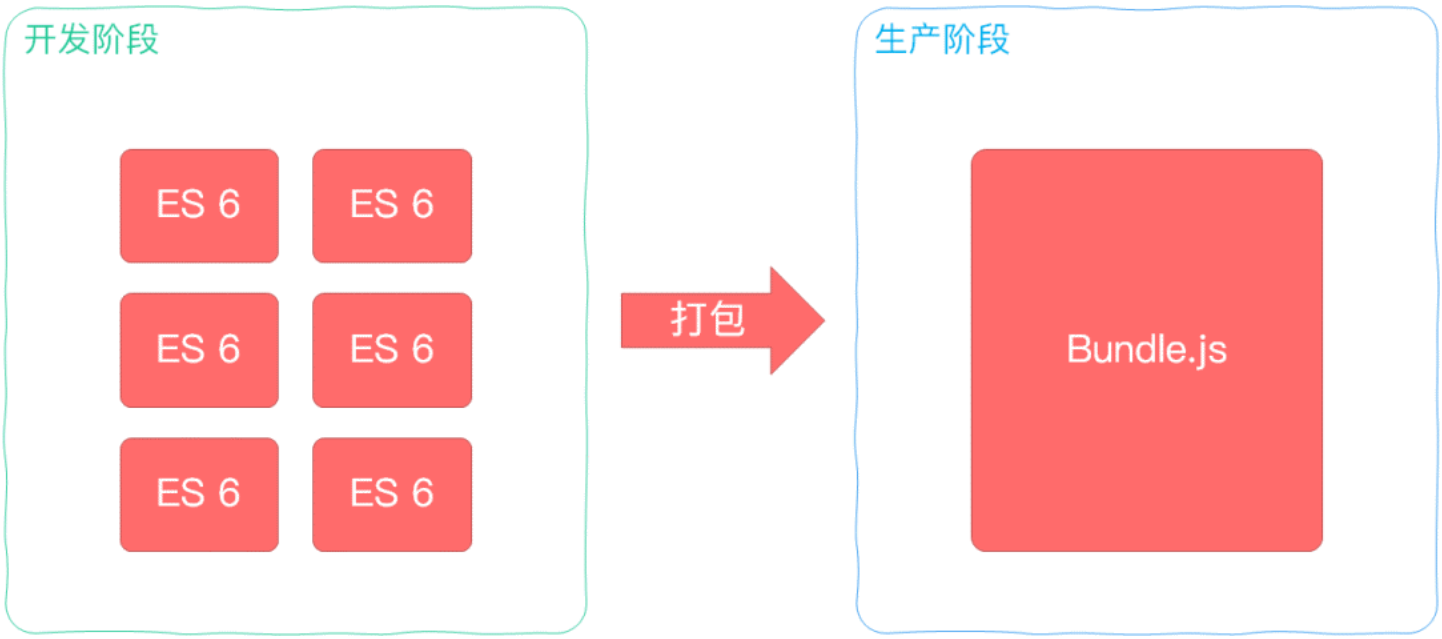


1.3.1. `webpack` 的能力：

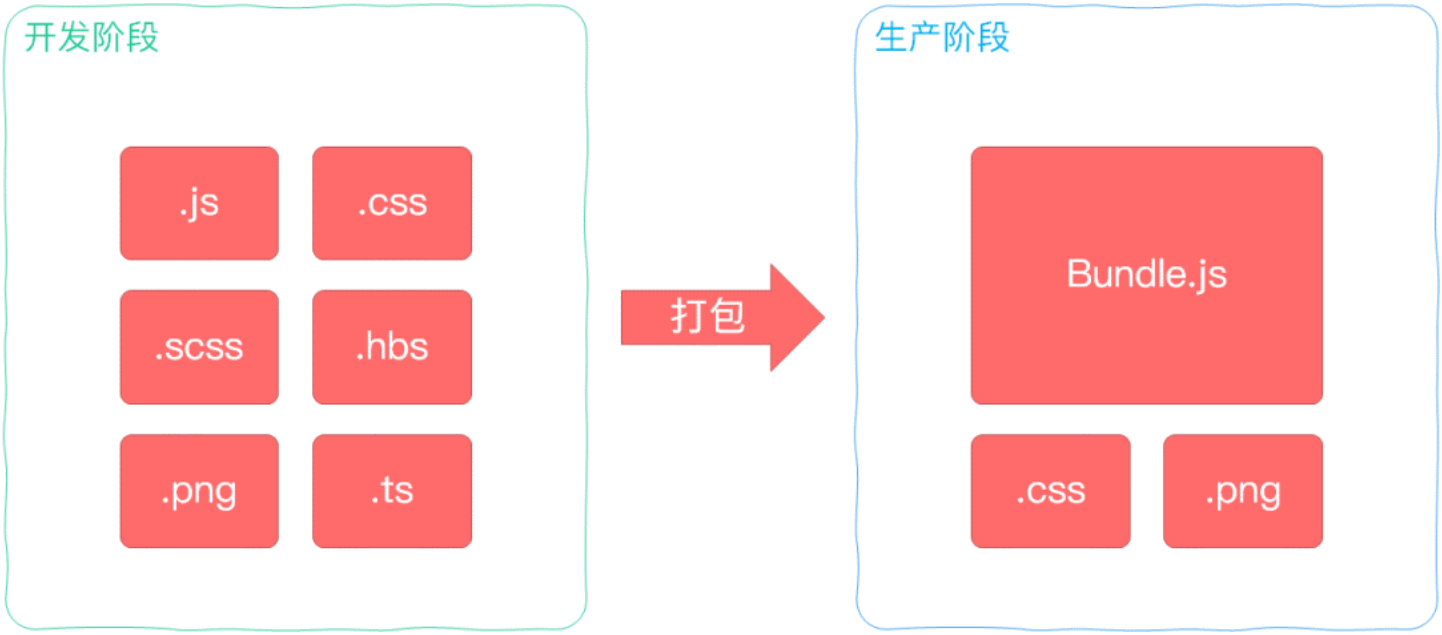
编译代码能力，提高效率，解决浏览器兼容问题



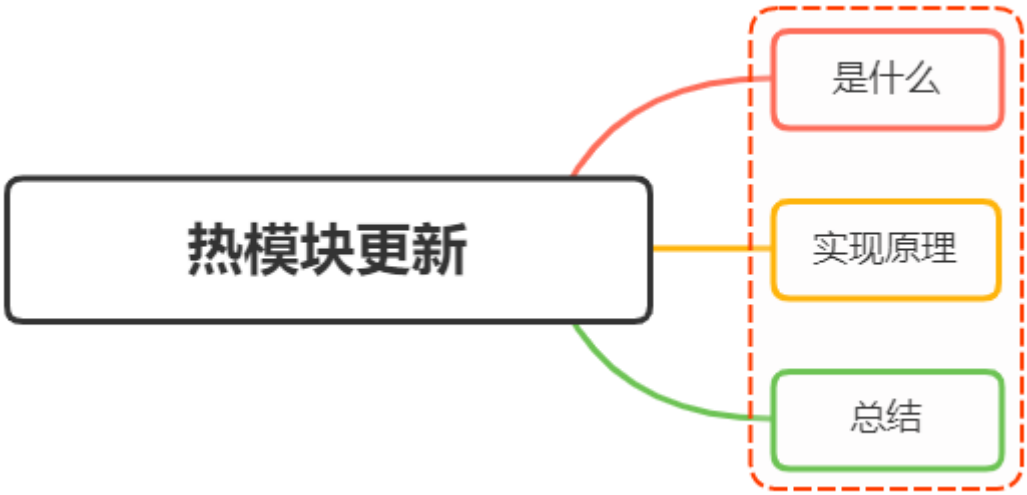
模块整合能力，提高性能，可维护性，解决浏览器频繁请求文件的问题



万物皆可模块能力，项目维护性增强，支持不同种类的前端模块类型，统一的模块化方案，所有资源文件的加载都可以通过代码控制



2. 说说webpack的热更新是如何做到的？原理是什么？



2.1. 是什么

`HMR` 全称 `Hot Module Replacement`，可以理解为模块热替换，指在应用程序运行过程中，替换、添加、删除模块，而无需重新刷新整个应用

例如，我们在应用运行过程中修改了某个模块，通过自动刷新会导致整个应用的整体刷新，那页面中的状态信息都会丢失

如果使用的是 `HMR`，就可以实现只将修改的模块实时替换至应用中，不必完全刷新整个应用

在 `webpack` 中配置开启热模块也非常的简单，如下代码：

```
1 const webpack = require('webpack')
2 module.exports = {
```

```
  // ...
  devServer: {
    // 开启 HMR 特性
    hot: true
    // hotOnly: true
  }
}
```

通过上述这种配置，如果我们修改并保存 `css` 文件，确实能够以不刷新的形式更新到页面中

但是，当我们修改并保存 `js` 文件之后，页面依旧自动刷新了，这里并没有触发热模块

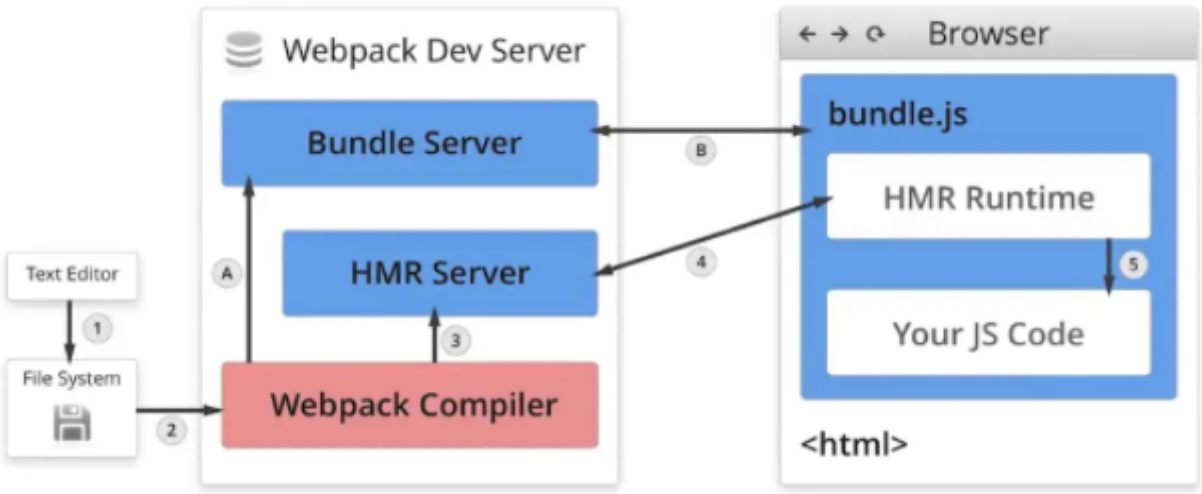
所以， `HMR` 并不像 `Webpack` 的其他特性一样可以开箱即用，需要有一些额外的操作

我们需要去指定哪些模块发生更新时进行 `HRM` ，如下代码：

```
if(module.hot){
    module.hot.accept('./util.js',()=>{
        console.log("util.js更新了")
    })
}
```

2.2. 实现原理

首先来看看一张图，如下：



- Webpack Compile：将 JS 源代码编译成 bundle.js
- HMR Server：用来将热更新的文件输出给 HMR Runtime
- Bundle Server：静态资源文件服务器，提供文件访问路径
- HMR Runtime：socket服务器，会被注入到浏览器，更新文件的变化

- bundle.js：构建输出的文件
- 在HMR Runtime 和 HMR Server之间建立 websocket，即图上4号线，用于实时更新文件变化

上面图中，可以分成两个阶段：

- 启动阶段为上图 1 - 2 - A - B

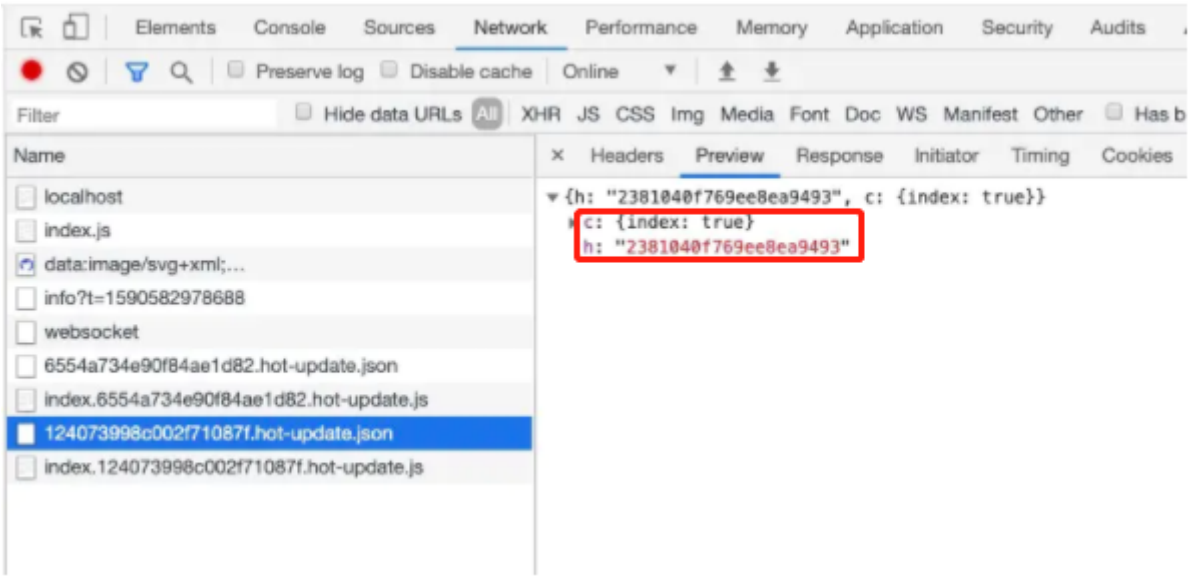
在编写未经过 `webpack` 打包的源代码后， `Webpack Compile` 将源代码和 `HMR Runtime` 一起编译成 `bundle` 文件，传输给 `Bundle Server` 静态资源服务器

- 更新阶段为上图 1 - 2 - 3 - 4

当某一个文件或者模块发生变化时， `webpack` 监听到文件变化对文件重新编译打包，编译生成唯一的 `hash` 值，这个 `hash` 值用来作为下一次热更新的标识

根据变化的内容生成两个补丁文件： `manifest` （包含了 `hash` 和 `chundId` ，用来说明变化的内容）和 `chunk.js` 模块

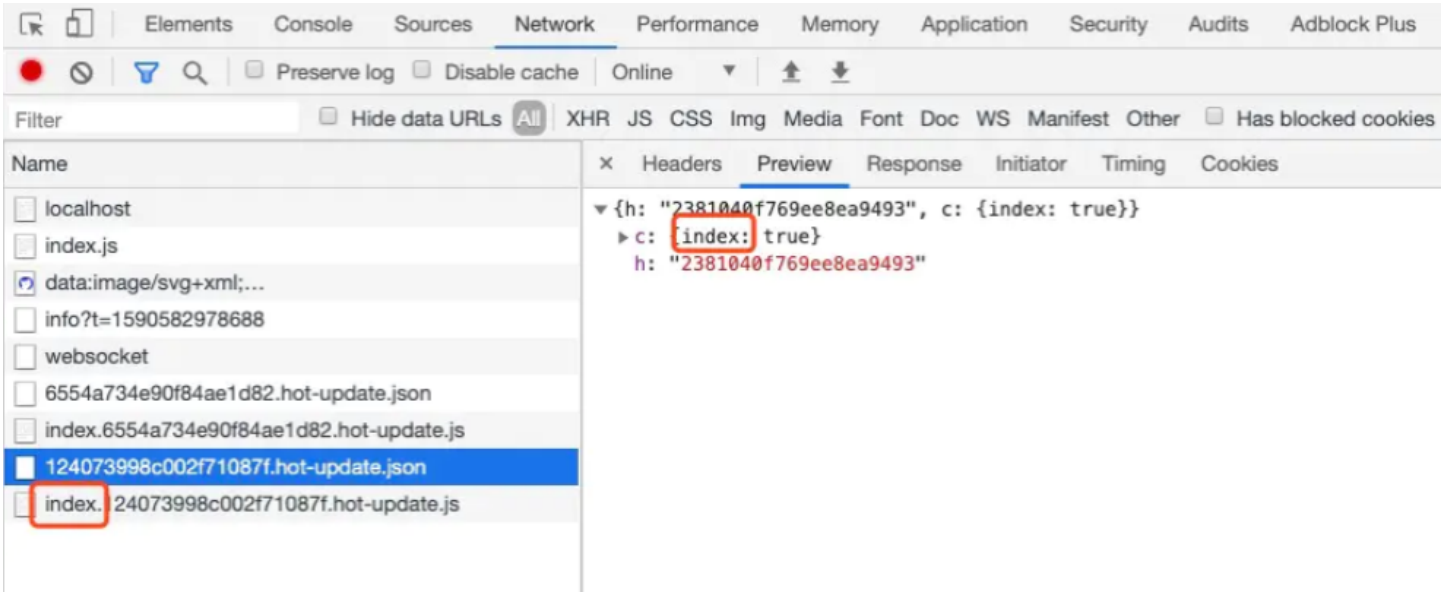
由于 `socket` 服务器在 `HMR Runtime` 和 `HMR Server` 之间建立 `websocket` 链接，当文件发生改动的时候，服务端会向浏览器推送一条消息，消息包含文件改动后生成的 `hash` 值，如下图的 `h` 属性，作为下一次热更细的标识



在浏览器接受到这条消息之前，浏览器已经在上一次 `socket` 消息中已经记住了此时的 `hash` 标识，这时候我们会创建一个 `ajax` 去服务端请求获取到变化内容的 `manifest` 文件

`mainfest` 文件包含重新 `build` 生成的 `hash` 值，以及变化的模块，对应上图的 `c` 属性

浏览器根据 `manifest` 文件获取模块变化的内容，从而触发 `render` 流程，实现局部模块更新

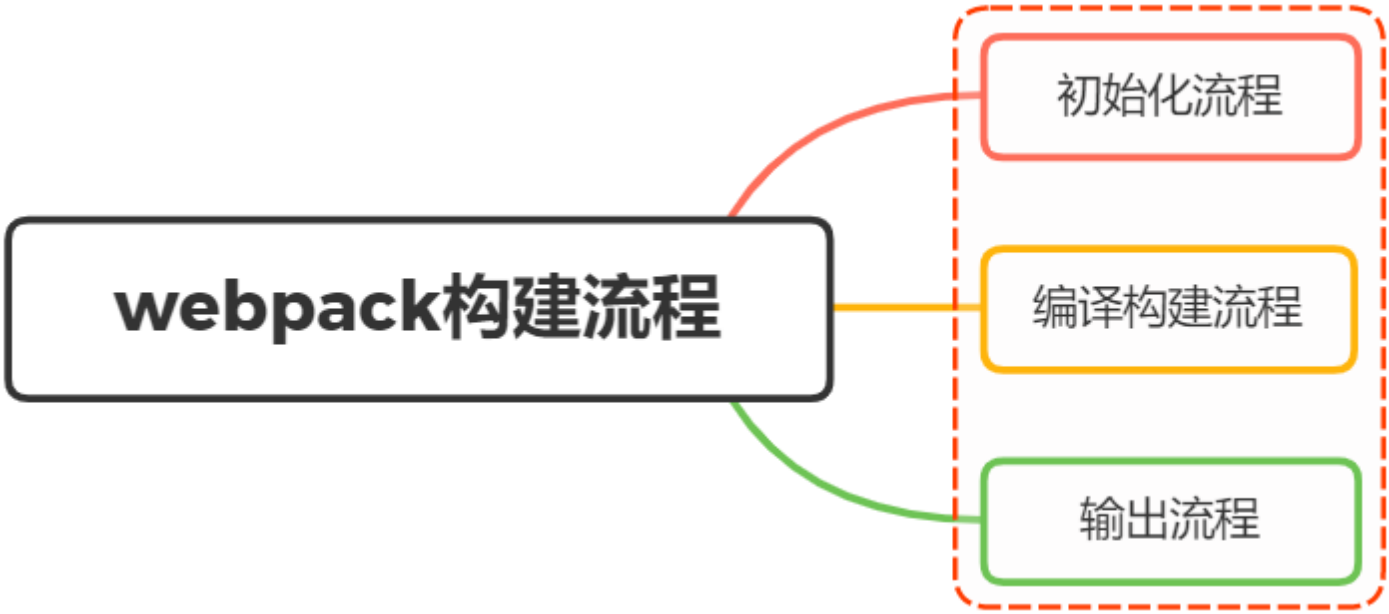


2.3. 总结

关于 `webpack` 热模块更新的总结如下：

- 通过 `webpack-dev-server` 创建两个服务器：提供静态资源的服务（express）和Socket服务
- express server 负责直接提供静态资源的服务（打包后的资源直接被浏览器请求和解析）
- socket server 是一个 websocket 的长连接，双方可以通信
- 当 socket server 监听到对应的模块发生变化时，会生成两个文件.json（manifest文件）和.js文件（update chunk）
- 通过长连接，socket server 可以直接将这两个文件主动发送给客户端（浏览器）
- 浏览器拿到两个新的文件后，通过HMR runtime机制，加载这两个文件，并且针对修改的模块进行更新

3. 说说webpack的构建流程?



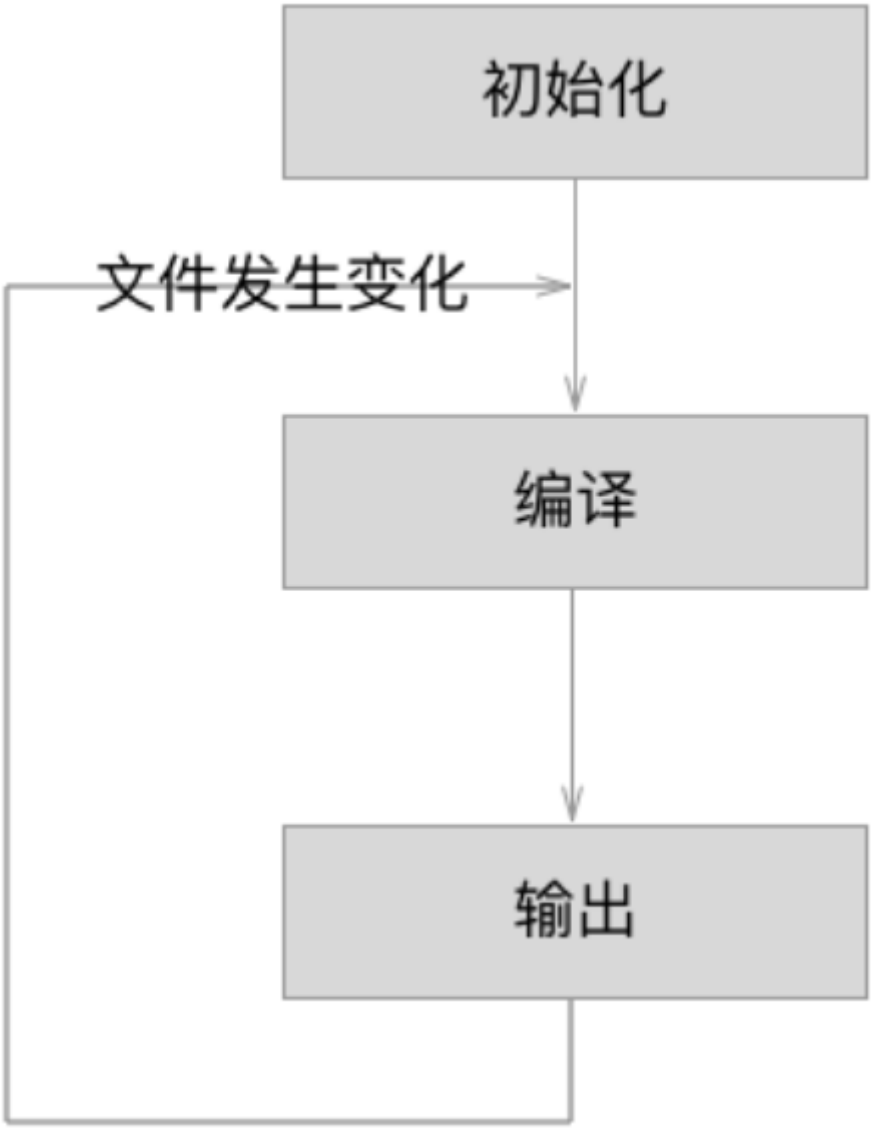
3.1. 运行流程

`webpack` 的运行流程是一个串行的过程，它的工作流程就是将各个插件串联起来

在运行过程中会广播事件，插件只需要监听它所关心的事件，就能加入到这条 `webpack` 机制中，去改变 `webpack` 的运作，使得整个系统扩展性良好

从启动到结束会依次执行以下三大步骤：

- 初始化流程：从配置文件和 `Shell` 语句中读取与合并参数，并初始化需要使用的插件和配置插件等执行环境所需要的参数
- 编译构建流程：从 Entry 发出，针对每个 Module 串行调用对应的 Loader 去翻译文件内容，再找到该 Module 依赖的 Module，递归地进行编译处理
- 输出流程：对编译后的 Module 组合成 Chunk，把 Chunk 转换成文件，输出到文件系统



3.1.1. 初始化流程

从配置文件和 `Shell` 语句中读取与合并参数，得出最终的参数

配置文件默认下为 `webpack.config.js` ，也或者通过命令的形式指定配置文件，主要作用是用于激活 `webpack` 的加载项和插件

关于文件配置内容分析，如下注释：

```
var path = require('path');
var node_modules = path.resolve(__dirname, 'node_modules');
var pathToReact = path.resolve(node_modules, 'react/dist/react.min.js');
module.exports = {
  // 入口文件，是模块构建的起点，同时每一个入口文件对应最后生成的一个 chunk。
  entry: './path/to/my/entry/file.js',
  // 文件路径指向(可加快打包过程)。
  resolve: {
    alias: {
      'react': pathToReact
```

```
    }
  },
  // 生成文件，是模块构建的终点，包括输出文件与输出路径。
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: '[name].js'
  },
  // 这里配置了处理各模块的 loader ，包括 css 预处理 loader ，es6 编译 loader，图片处理 loader。
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ],
    noParse: [pathToReact]
  },
  // webpack 各插件对象，在 webpack 的事件流中执行对应的方法。
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ]
};
```

`webpack` 将 `webpack.config.js` 中的各个配置项拷贝到 `options` 对象中，并加载用户配置的 `plugins`

完成上述步骤之后，则开始初始化 `Compiler` 编译对象，该对象掌控者 `webpack` 声明周期，不执行具体的任务，只是进行一些调度工作

```
class Compiler extends Tapable {
    constructor(context) {
        super();
        this.hooks = {
            beforeCompile: new AsyncSeriesHook(["params"]),
            compile: new SyncHook(["params"]),
            afterCompile: new AsyncSeriesHook(["compilation"]),
            make: new AsyncParallelHook(["compilation"]),
            entryOption: new SyncBailHook(["context", "entry"])
            // 定义了很多不同类型的钩子
        };
        // ...
```

```
    }
}
function webpack(options) {
  var compiler = new Compiler();
  ...// 检查options,若watch字段为true,则开启watch线程
  return compiler;
}
...
```

`Compiler` 对象继承自 `Tapable`，初始化时定义了很多钩子函数

3.1.2. 编译构建流程

根据配置中的 `entry` 找出所有的入口文件

```
module.exports = {
  entry: './src/file.js'
}
```

初始化完成后会调用 `Compiler` 的 `run` 来真正启动 `webpack` 编译构建流程，主要流程如下：

- `compile` 开始编译
- `make` 从入口点分析模块及其依赖的模块，创建这些模块对象
- `build-module` 构建模块
- `seal` 封装构建结果
- `emit` 把各个chunk输出到结果文件

3.1.2.1. compile 编译

执行了 `run` 方法后，首先会触发 `compile`，主要是构建一个 `Compilation` 对象

该对象是编译阶段的主要执行者，主要会依次下述流程：执行模块创建、依赖收集、分块、打包等主要任务的对象

3.1.2.2. make 编译模块

当完成了上述的 `compilation` 对象后，就开始从 `Entry` 入口文件开始读取，主要执行 `_addModuleChain()` 函数，如下：

```
_addModuleChain(context, dependency, onModule, callback) {
    ...
    // 根据依赖查找对应的工厂函数
    const Dep = /** @type {DepConstructor} */ (dependency.constructor);
```

```
    const moduleFactory = this.dependencyFactories.get(Dep);

    // 调用工厂函数NormalModuleFactory的create来生成一个空的NormalModule对象
    moduleFactory.create({
        dependencies: [dependency]
        ...
    }, (err, module) => {
        ...
        const afterBuild = () => {
         this.processModuleDependencies(module, err => {
          if (err) return callback(err);
          callback(null, module);
            });
     };

        this.buildModule(module, false, null, null, err => {
            ...
            afterBuild();
        })
    })
}
```

过程如下：

`_addModuleChain` 中接收参数 `dependency` 传入的入口依赖，使用对应的工厂函数 `NormalModuleFactory.create` 方法生成一个空的 `module` 对象

回调中会把此 `module` 存入 `compilation.modules` 对象和 `dependencies.module` 对象中，由于是入口文件，也会存入 `compilation.entries` 中

随后执行 `buildModule` 进入真正的构建模块 `module` 内容的过程

3.1.2.3. build module 完成模块编译

这里主要调用配置的 `loaders` ，将我们的模块转成标准的 `JS` 模块

在用 `Loader` 对一个模块转换完后，使用 `acorn` 解析转换后的内容，输出对应的抽象语法树（`AST`），以方便 `Webpack` 后面对代码的分析

从配置的入口模块开始，分析其 `AST` ，当遇到 `require` 等导入其它模块语句时，便将其加入到依赖的模块列表，同时对新找出的依赖模块递归分析，最终搞清所有模块的依赖关系

3.1.3. 输出流程

3.1.3.1. seal 输出资源

`seal` 方法主要是要生成 `chunks` ，对 `chunks` 进行一系列的优化操作，并生成要输出的代码

`webpack` 中的 `chunk` ，可以理解为配置在 `entry` 中的模块，或者是动态引入的模块

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

3.1.3.2. emit 输出完成

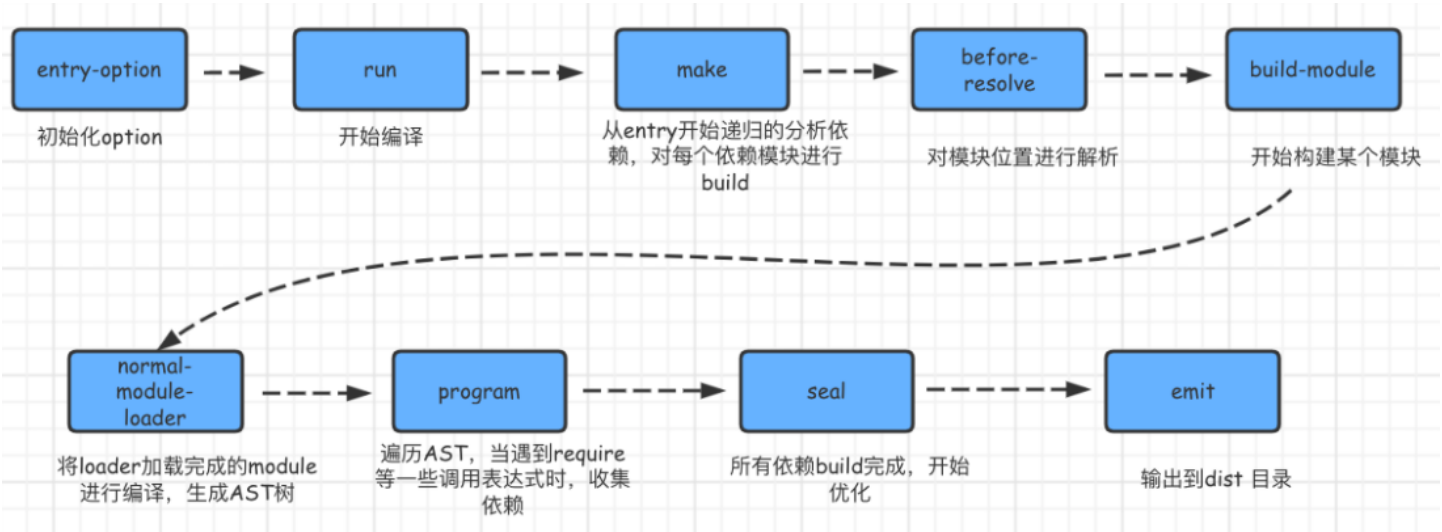
在确定好输出内容后，根据配置确定输出的路径和文件名

```
output: {
    path: path.resolve(__dirname, 'build'),
        filename: '[name].js'
}
```

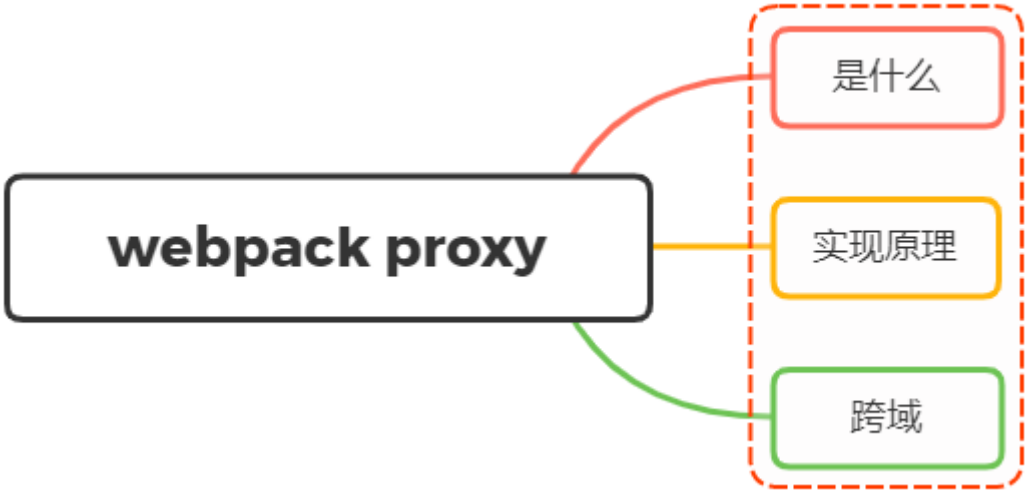
在 `Compiler` 开始生成文件前，钩子 `emit` 会被执行，这是我们修改最终文件的最后一个机会

从而 `webpack` 整个打包过程则结束了

3.1.4. 小结



4. 说说webpack proxy工作原理？为什么能解决跨域？



4.1. 是什么

`webpack proxy`，即`webpack`提供的代理服务

基本行为就是接收客户端发送的请求后转发给其他服务器

其目的是为了便于开发者在开发模式下解决跨域问题（浏览器安全策略限制）

想要实现代理首先需要一个中间服务器，`webpack`中提供服务器的工具为`webpack-dev-server`

4.1.1. webpack-dev-server

`webpack-dev-server`是`webpack`官方推出的一款开发工具，将自动编译和自动刷新浏览器等一系列对开发友好的功能全部集成在了一起

目的是为了提高开发者日常的开发效率，**只适用在开发阶段**

关于配置方面，在`webpack`配置对象属性中通过`devServer`属性提供，如下：

```
// ./webpack.config.js
const path = require('path')
module.exports = {
    // ...
    devServer: {
        contentBase: path.join(__dirname, 'dist'),
        compress: true,
        port: 9000,
        proxy: {
            '/api': {
                target: 'https://api.github.com'
            }
        }
        // ...
    }
```

```
16 }
```

`devServetr` 里面 `proxy` 则是关于代理的配置，该属性为对象的形式，对象中每一个属性就是一个代理的规则匹配

属性的名称是需要被代理的请求路径前缀，一般为了辨别都会设置前缀为 `/api` ，值为对应的代理匹配规则，对应如下：

- target：表示的是代理到的目标地址
- pathRewrite：默认情况下，我们的 /api-hy 也会被写入到URL中，如果希望删除，可以使用 pathRewrite
- secure：默认情况下不接收转发到https的服务器上，如果希望支持，可以设置为false
- changeOrigin：它表示是否更新代理后请求的 headers 中host地址

4.2. 工作原理

`proxy` 工作原理实质上是利用 `http-proxy-middleware` 这个 `http` 代理中间件，实现请求转发给其他服务器

举个例子：

在开发阶段，本地地址为 `http://localhost:3000` ，该浏览器发送一个前缀带有 `/api` 标识的请求到服务端获取数据，但响应这个请求的服务器只是将请求转发到另一台服务器中

```
1 const express = require('express');
2 const proxy = require('http-proxy-middleware');
3 const app = express();
4 app.use('/api', proxy({target: 'http://www.example.org', changeOrigin: true}));
5 app.listen(3000);
6 // http://localhost:3000/api/foo/bar -> http://www.example.org/api/foo/bar
```

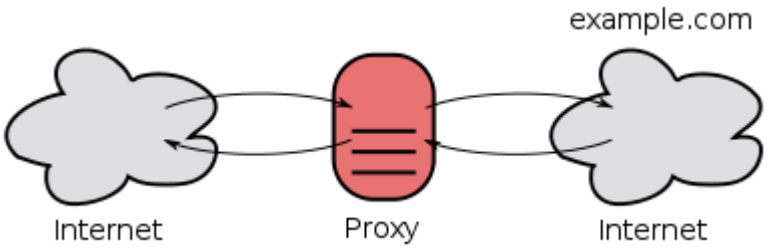
4.3. 跨域

在开发阶段， `webpack-dev-server` 会启动一个本地开发服务器，所以我们的应用在开发阶段是独立运行在 `localhost` 的一个端口上，而后端服务又是运行在另外一个地址上

所以在开发阶段中，由于浏览器同源策略的原因，当本地访问后端就会出现跨域请求的问题

通过设置 `webpack proxy` 实现代理请求后，相当于浏览器与服务端中添加一个代理者

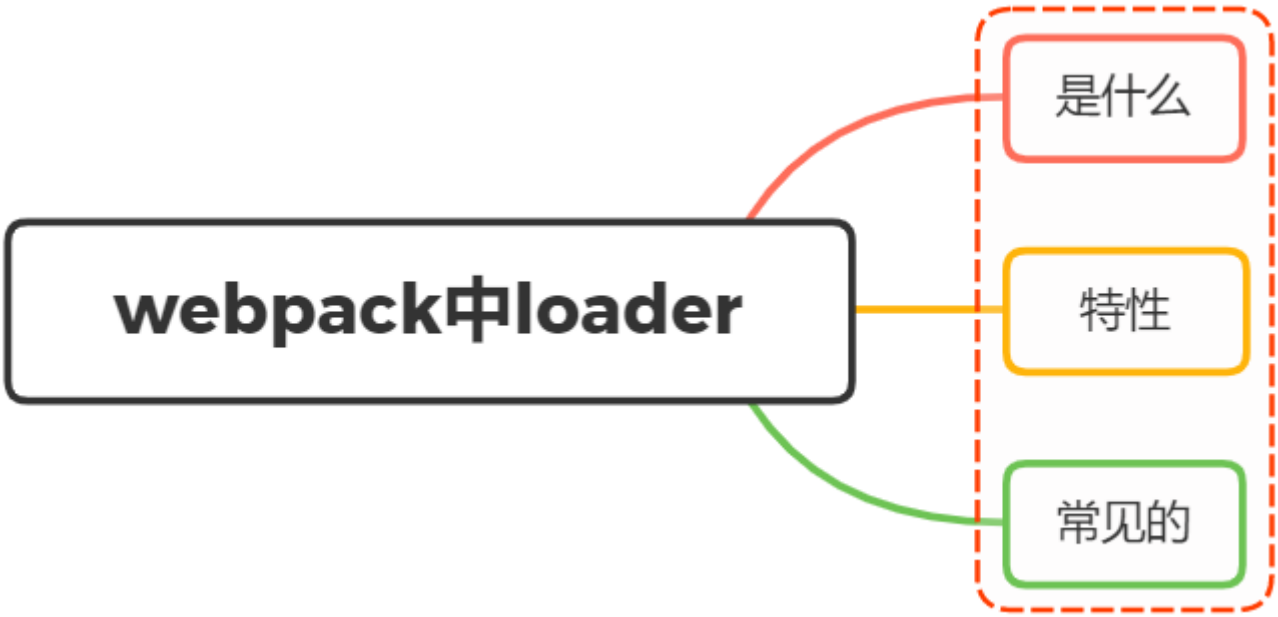
当本地发送请求的时候，代理服务器响应该请求，并将请求转发到目标服务器，目标服务器响应数据后再将数据返回给代理服务器，最终再由代理服务器将数据响应给本地



在代理服务器传递数据给本地浏览器的过程中，两者同源，并不存在跨域行为，这时候浏览器就能正常接收数据

注意：**服务器与服务器之间请求数据并不会存在跨域行为，跨域行为是浏览器安全策略限制**

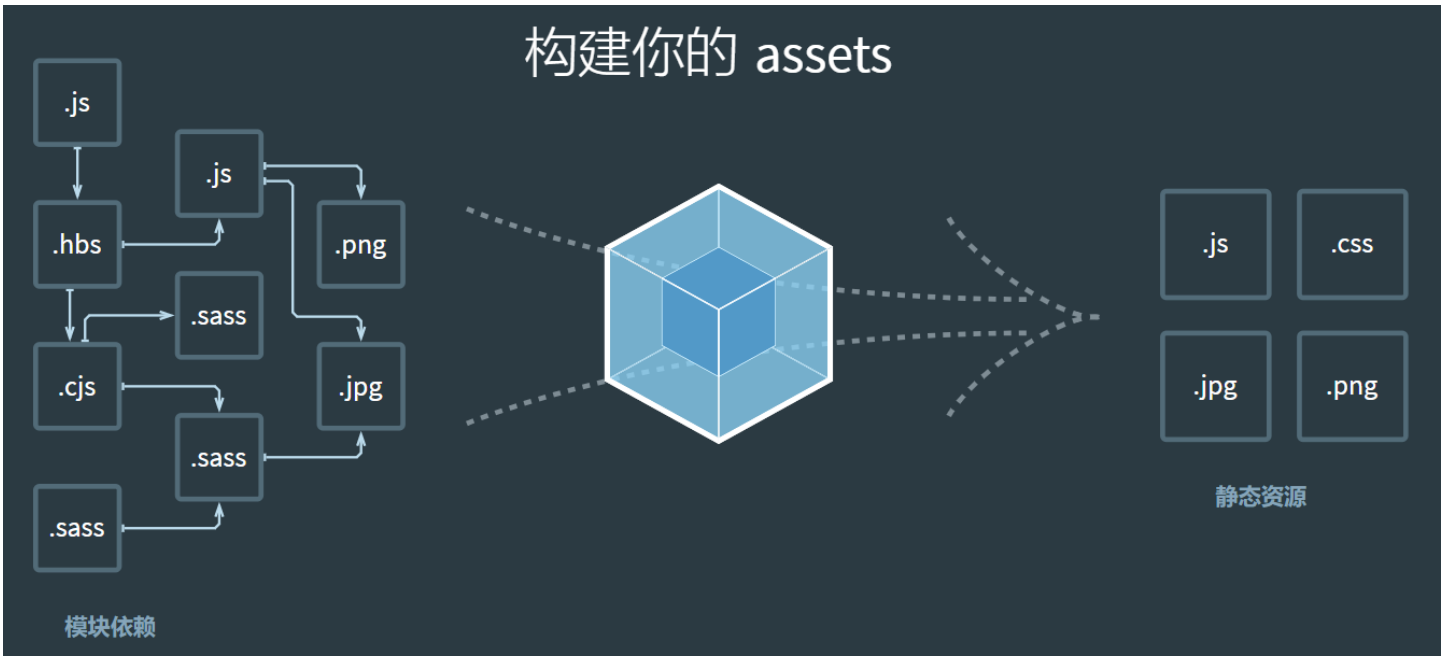
5. 说说webpack中常见的Loader？解决了什么问题？



5.1. 是什么

`loader` 用于对模块的"源代码"进行转换，在 `import` 或"加载"模块时预处理文件

`webpack` 做的事情，仅仅是分析出各种模块的依赖关系，然后形成资源列表，最终打包生成到指定的文件中。如下图所示：

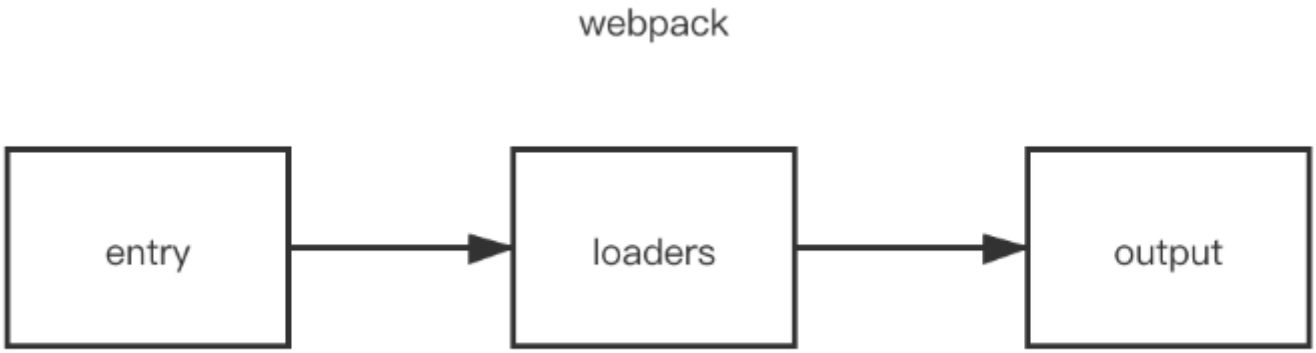


在 `webpack` 内部中，任何文件都是模块，不仅仅只是 `js` 文件

默认情况下，在遇到 `import` 或者 `require` 加载模块的时候， `webpack` 只支持对 `js` 和 `json` 文件打包

像 `css` 、 `sass` 、 `png` 等这些类型的文件的时候， `webpack` 则无能为力，这时候就需要配置对应的 `loader` 进行文件内容的解析

在加载模块的时候，执行顺序如下：



当 `webpack` 碰到不识别的模块的时候， `webpack` 会在配置的中查找该文件解析规则

关于配置 `loader` 的方式有三种：

- 配置方式（推荐）：在 webpack.config.js文件中指定 loader
- 内联方式：在每个 import 语句中显式指定 loader
- CLI 方式：在 shell 命令中指定它们

5.1.1. 配置方式

关于 `loader` 的配置，我们是写在 `module.rules` 属性中，属性介绍如下：

- `rules` 是一个数组的形式，因此我们可以配置很多个 `loader`

- 每一个 `loader` 对应一个对象的形式，对象属性 `test` 为匹配的规则，一般情况为正则表达式
- 属性 `use` 针对匹配到文件类型，调用对应的 `loader` 进行处理

代码编写，如下形式：

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          { loader: 'style-loader' },
          {
            loader: 'css-loader',
            options: {
              modules: true
            }
          },
          { loader: 'sass-loader' }
        ]
      }
    ]
  }
};
```

5.2. 特性

这里继续拿上述代码，来讲讲 `loader` 的特性

从上述代码可以看到，在处理 `css` 模块的时候， `use` 属性中配置了三个 `loader` 分别处理 `css` 文件

因为 `loader` 支持链式调用，链中的每个 `loader` 会处理之前已处理过的资源，最终变为 `js` 代码。顺序为相反的顺序执行，即上述执行方式为 `sass-loader` 、 `css-loader` 、 `style-loader`

除此之外， `loader` 的特性还有如下：

- loader 可以是同步的，也可以是异步的
- loader 运行在 Node.js 中，并且能够执行任何操作
- 除了常见的通过 `package.json` 的 `main` 来将一个 npm 模块导出为 loader，还可以在 module.rules 中使用 `loader` 字段直接引用一个模块
- 插件(plugin)可以为 loader 带来更多特性
- loader 能够产生额外的任意文件

可以通过 loader 的预处理函数，为 JavaScript 生态系统提供更多能力。用户现在可以更加灵活地引入细粒度逻辑，例如：压缩、打包、语言翻译和更多其他特性

5.3. 常见的loader

在页面开发过程中，我们经常性加载除了 `js` 文件以外的内容，这时候我们就需要配置响应的 `loader` 进行加载

常见的 `loader` 如下：

- style-loader: 将css添加到DOM的内联样式标签style里
- css-loader :允许将css文件通过require的方式引入，并返回css代码
- less-loader: 处理less
- sass-loader: 处理sass
- postcss-loader: 用postcss来处理CSS
- autoprefixer-loader: 处理CSS3属性前缀，已被弃用，建议直接使用postcss
- file-loader: 分发文件到output目录并返回相对路径
- url-loader: 和file-loader类似，但是当文件小于设定的limit时可以返回一个Data Url
- html-minify-loader: 压缩HTML
- babel-loader :用babel来转换ES6文件到ES

下面给出一些常见的 `loader` 的使用：

5.3.1. css-loader

分析 `css` 模块之间的关系，并合成一个 `css`

```
npm install --save-dev css-loader
```

```
rules: [
  ...,
 {
  test: /\.css$/,
    use: {
      loader: "css-loader",
      options: {
     // 启用/禁用 url() 处理
     url: true,
     // 启用/禁用 @import 处理
     import: true,
```

```
        // 启用/禁用 Sourcemap
        sourceMap: false
      }
    }
  }
]
```

如果只通过 `css-loader` 加载文件，这时候页面代码设置的样式并没有生效

原因在于， `css-loader` 只是负责将 `.css` 文件进行一个解析，而并不会将解析后的 `css` 插入到页面中

如果我们希望再完成插入 `style` 的操作，那么我们还需要另外一个 `loader` ，就是 `style-loader`

5.3.2. style-loader

把 `css-loader` 生成的内容，用 `style` 标签挂载到页面的 `head` 中

```
npm install --save-dev style-loader
```

```
rules: [
  ...,
 {
  test: /\.css$/,
    use: ["style-loader", "css-loader"]
 }
]
```

同一个任务的 `loader` 可以同时挂载多个，处理顺序为：从右到左，从下往上

5.3.3. less-loader

开发中，我们也常常会使用 `less` 、 `sass` 、 `stylus` 预处理器编写 `css` 样式，使开发效率提高，这里需要使用 `less-loader`

```
npm install less-loader -D
```

```
rules: [
```

```
  ...,
 {
  test: /\.css$/,
    use: ["style-loader", "css-loader","less-loader"]
 }
]
```

5.3.4. raw-loader

在 `webpack` 中通过 `import` 方式导入文件内容，该 `loader` 并不是内置的，所以首先要安装

```
npm install --save-dev raw-loader
```

然后在 webpack.config.js 中进行配置

```
module.exports = {
  ...,
  module: {
      rules: [
      {
        test: /\.(txt|md)$/,
        use: 'raw-loader'
     }
    ]
 }
}
```

5.3.5. file-loader

把识别出的资源模块，移动到指定的输出目目录，并且返回这个资源在输出目录的地址(字符串)

```
npm install --save-dev file-loader
```

```
rules: [
  ...,
 {
  test: /\.(png|jpe?g|gif)$/,
    use: {
      loader: "file-loader",
```

```
      options: {
        // placeholder 占位符 [name] 源资源模块的名称
        // [ext] 源资源模块的后缀
        name: "[name]_[hash].[ext]",
        //打包后的存放位置
        outputPath: "./images",
        // 打包后文件的 url
        publicPath: './images',
      }
    }
  }
]
```

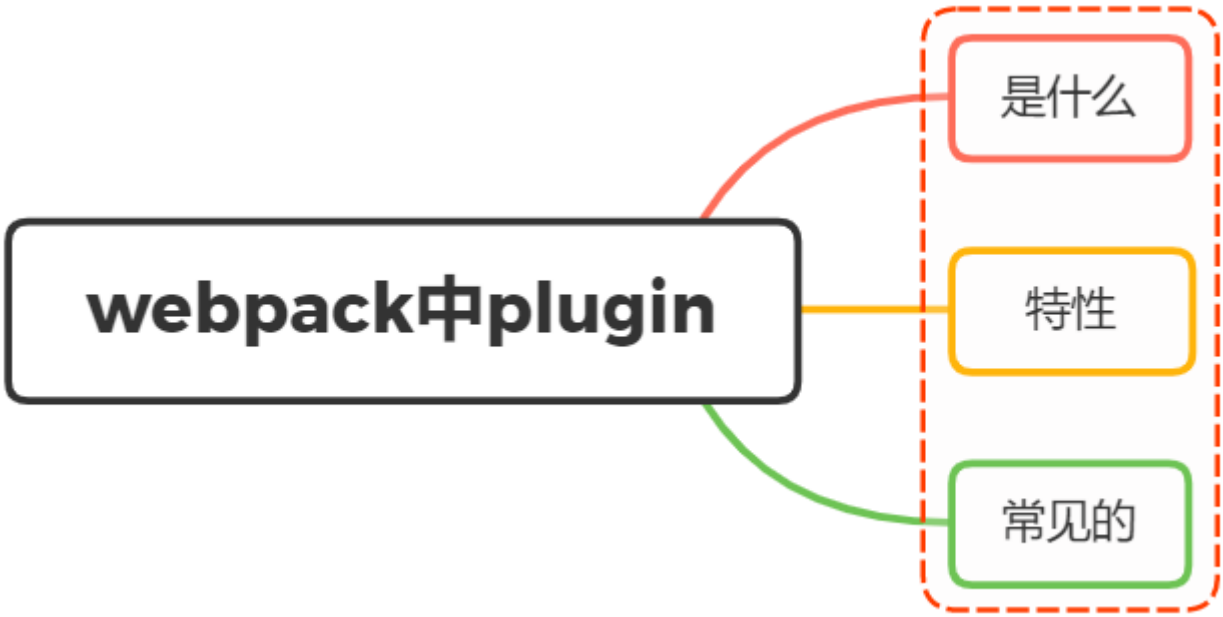
5.3.6. url-loader

可以处理理 `file-loader` 所有的事情，但是遇到图片格式的模块，可以选择性的把图片转成 `base64` 格式的字符串，并打包到 `js` 中，对小体积的图片比较合适，大图片不合适。

```
npm install --save-dev url-loader
```

```
rules: [
  ...,
 {
  test: /\.(png|jpe?g|gif)$/,
    use: {
      loader: "url-loader",
      options: {
        // placeholder 占位符 [name] 源资源模块的名称
        // [ext] 源资源模块的后缀
        name: "[name]_[hash].[ext]",
        //打包后的存放位置
        outputPath: "./images"
        // 打包后文件的 url
        publicPath: './images',
        // 小于 100 字节转成 base64 格式
        limit: 100
      }
    }
  }
]
```

6. 说说webpack中常见的Plugin？解决了什么问题？

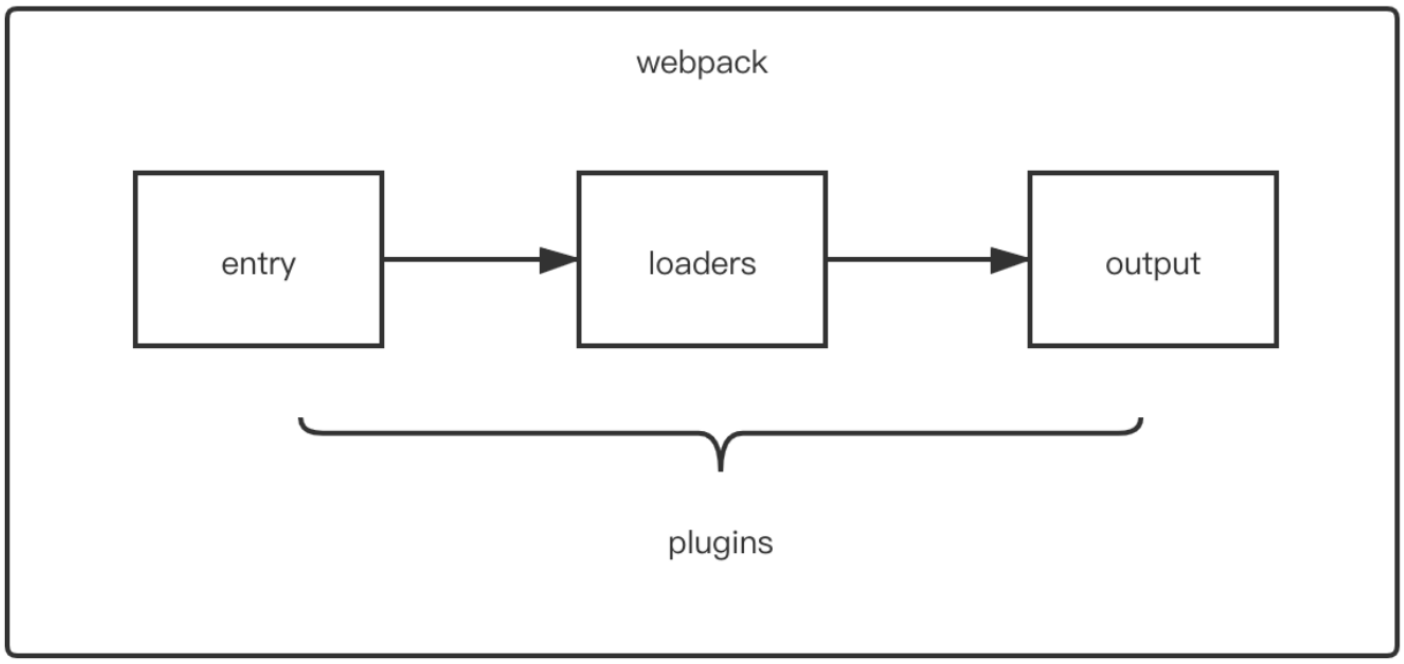


6.1. 是什么

`Plugin` （Plug-in）是一种计算机应用程序，它和主应用程序互相交互，以提供特定的功能

是一种遵循一定规范的应用程序接口编写出来的程序，只能运行在程序规定的系统下，因为其需要调用原纯净系统提供的函数库或者数据

`webpack` 中的 `plugin` 也是如此， `plugin` 赋予其各种灵活的功能，例如打包优化、资源管理、环境变量注入等，它们会运行在 `webpack` 的不同阶段（钩子 / 生命周期），贯穿了 `webpack` 整个编译周期



目的在于解决 `loader` 无法实现的其他事

6.1.1. 配置方式

这里讲述文件的配置方式，一般情况，通过配置文件导出对象中 `plugins` 属性传入 `new` 实例对象。如下所示：

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装
const webpack = require('webpack'); // 访问内置的插件
module.exports = {
  ...
  plugins: [
    new webpack.ProgressPlugin(),
    new HtmlWebpackPlugin({ template: './src/index.html' }),
  ],
};
```

6.2. 特性

其本质是一个具有 `apply` 方法 `javascript` 对象

`apply` 方法会被 `webpack compiler` 调用，并且在整个编译生命周期都可以访问 `compiler` 对象

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';
class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    compiler.hooks.run.tap(pluginName, (compilation) => {
      console.log('webpack 构建过程开始! ');
    });
  }
}
module.exports = ConsoleLogOnBuildWebpackPlugin;
```

`compiler hook` 的 `tap` 方法的第一个参数，应是驼峰式命名的插件名称

关于整个编译生命周期钩子，有如下：

- entry-option ： 初始化 option
- run
- compile： 真正开始的编译，在创建 compilation 对象之前
- compilation ： 生成好了 compilation 对象
- make 从 entry 开始递归分析依赖，准备对每个模块进行 build
- after-compile： 编译 build 过程结束

- emit：在将内存中 assets 内容写到磁盘文件夹之前
- after-emit：在将内存中 assets 内容写到磁盘文件夹之后
- done：完成所有的编译过程
- failed：编译失败的时候

6.3. 三、常见的Plugin

常见的 `plugin` 有如图所示：

`AggressiveSplittingPlugin`	将原来的 chunk 分成更小的 chunk
`BabelMinifyWebpackPlugin`	使用 babel-minify进行压缩
`BannerPlugin`	在每个生成的 chunk 顶部添加 banner
`CommonsChunkPlugin`	提取 chunks 之间共享的通用模块
`CompressionWebpackPlugin`	预先准备的资源压缩版本，使用 Content-Encoding 提供访问服务
`ContextReplacementPlugin`	重写 `require` 表达式的推断上下文
`CopyWebpackPlugin`	将单个文件或整个目录复制到构建目录
`DefinePlugin`	允许在编译时(compile time)配置的全局常量
`DllPlugin`	为了极大减少构建时间，进行分离打包
`EnvironmentPlugin`	`DefinePlugin` 中 `process.env` 键的简写方式。
`ExtractTextWebpackPlugin`	从 bundle 中提取文本（CSS）到单独的文件
`HotModuleReplacementPlugin`	启用模块热替换(Enable Hot Module Replacement - HMR)
`HtmlWebpackPlugin`	简单创建 HTML 文件，用于服务器访问
`I18nWebpackPlugin`	为 bundle 增加国际化支持
`IgnorePlugin`	从 bundle 中排除某些模块
`LimitChunkCountPlugin`	设置 chunk 的最小/最大限制，以微调和控制 chunk
`LoaderOptionsPlugin`	用于从 webpack 1 迁移到 webpack 2
`MinChunkSizePlugin`	确保 chunk 大小超过指定限制
`NoEmitOnErrorsPlugin`	在输出阶段时，遇到编译错误跳过
`NormalModuleReplacementPlugin`	替换与正则表达式匹配的资源

下面介绍几个常用的插件用法：

6.3.1. HtmlWebpackPlugin

在打包结束后，自动生成一个 `html` 文文件，并把打包生成的 `js` 模块引入到该 `html` 中

```
npm install --save-dev html-webpack-plugin
```

```
// webpack.config.js
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
 ...
  plugins: [
     new HtmlWebpackPlugin({
       title: "My App",
       filename: "app.html",
       template: "./src/html/index.html"
     })
  ]
};
```

```
<!--./src/html/index.html-->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title><%=htmlWebpackPlugin.options.title%></title>
</head>
<body>
    <h1>html-webpack-plugin</h1>
</body>
</html>
```

在 `html` 模板中，可以通过 `<%=htmlWebpackPlugin.options.XXX%>` 的方式获取配置的值

更多的配置可以自寻查找

6.3.2. clean-webpack-plugin

删除（清理）构建目录

```
npm install --save-dev clean-webpack-plugin
```

```
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
module.exports = {
 ...
  plugins: [
    ...,
    new CleanWebpackPlugin(),
    ...
  ]
}
```

6.3.3. mini-css-extract-plugin

提取 `CSS` 到一个单独的文件中

```
npm install --save-dev mini-css-extract-plugin
```

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
module.exports = {
 ...,
  module: {
   rules: [
    {
     test: /\.s[ac]ss$/,
     use: [
      {
       loader: MiniCssExtractPlugin.loader
     },
          'css-loader',
          'sass-loader'
        ]
   }
   ]
 },
  plugins: [
    ...,
    new MiniCssExtractPlugin({
     filename: '[name].css'
    }),
    ...
```

```
  ]
}
```

6.3.4. DefinePlugin

允许在编译时创建配置的全局对象，是一个 `webpack` 内置的插件，不需要安装

```
const { DefinePlugun } = require('webpack')
module.exports = {
 ...
    plugins:[
        new DefinePlugin({
            BASE_URL:'"./"'
        })
    ]
}
```

这时候编译 `template` 模块的时候，就能通过下述形式获取全局对象

```
<link rel="icon" href="<%= BASE_URL%>favicon.ico>"
```

6.3.5. copy-webpack-plugin

复制文件或目录到执行区域，如 `vue` 的打包过程中，如果我们将一些文件放到 `public` 的目录下，那么这个目录会被复制到 `dist` 文件夹中

```
npm install copy-webpack-plugin -D
```

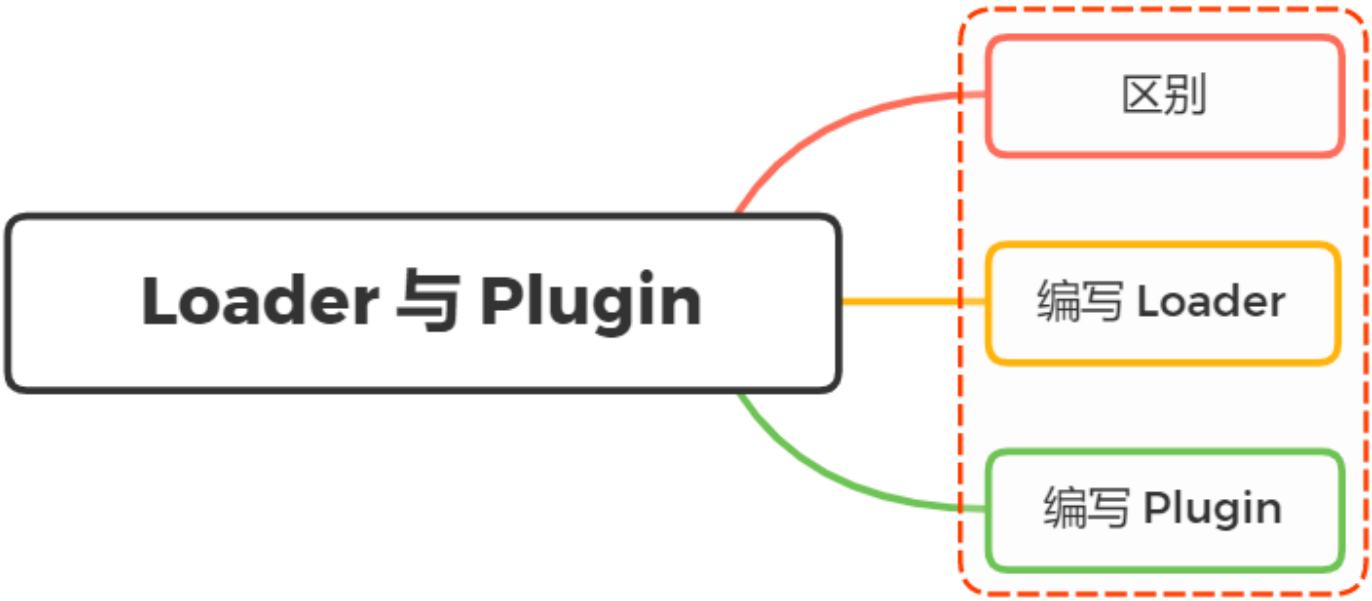
```
new CopyWebpackPlugin({
    parrerns:[
        {
            from:"public",
            globOptions:{
                ignore:[
                    '**/index.html'
                ]
            }
        }
    ]
```

```
12 })
```

复制的规则在 `patterns` 属性中设置：

- from：设置从哪一个源中开始复制
- to：复制到的位置，可以省略，会默认复制到打包的目录下
- globOptions：设置一些额外的选项，其中可以编写需要忽略的文件

7. 说说Loader和Plugin的区别？编写Loader，Plugin的思路？

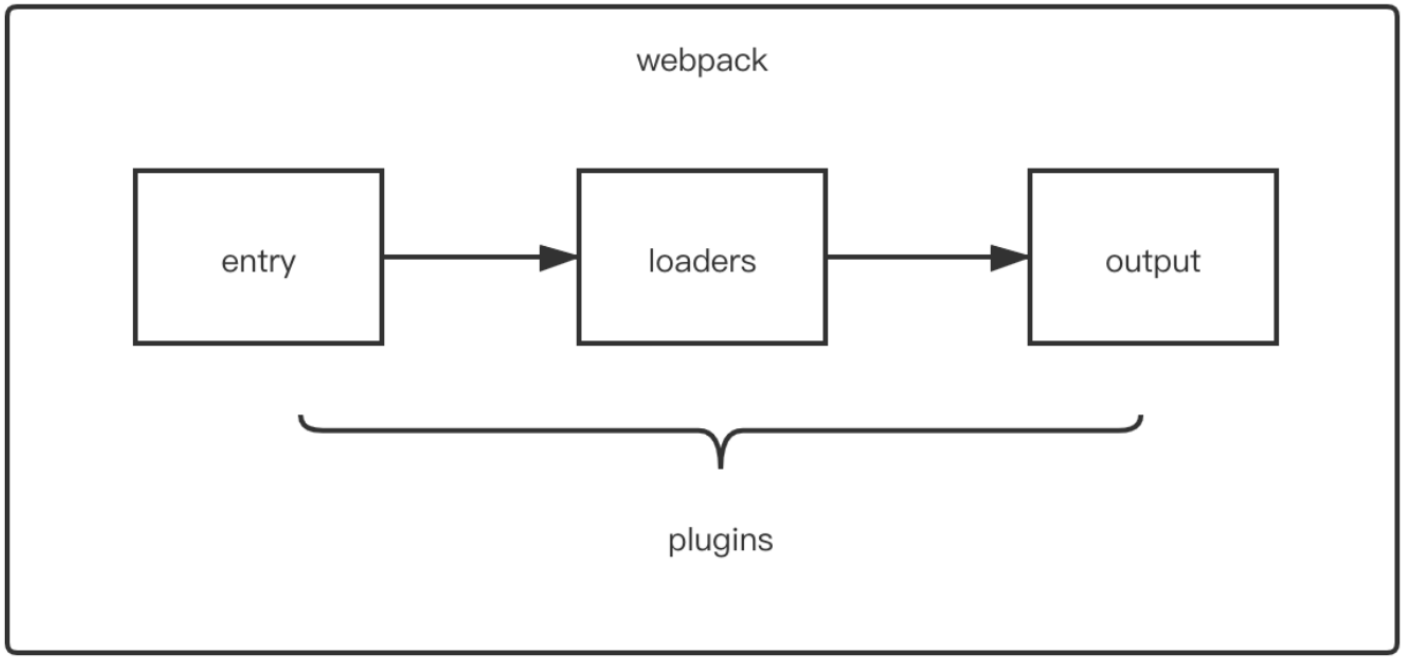


7.1. 区别

前面两节我们有提到 `Loader` 与 `Plugin` 对应的概念，先来回顾下

- loader 是文件加载器，能够加载资源文件，并对这些文件进行一些处理，诸如编译、压缩等，最终一起打包到指定的文件中
- plugin 赋予了 webpack 各种灵活的功能，例如打包优化、资源管理、环境变量注入等，目的是解决 loader 无法实现的其他事

从整个运行时机上来看，如下图所示：



可以看到，两者在运行时机上的区别：

- loader 运行在打包文件之前
- plugins 在整个编译周期都起作用

在 `Webpack` 运行的生命周期中会广播出许多事件， `Plugin` 可以监听这些事件，在合适的时机通过 `Webpack` 提供的 `API` 改变输出结果

对于 `loader` ，实质是一个转换器，将A文件进行编译形成B文件，操作的是文件，比如将 `A.scss` 或 `A.less` 转变为 `B.css` ，单纯的文件转换过程

7.2. 编写loader

在编写 `loader` 前，我们首先需要了解 `loader` 的本质

其本质为函数，函数中的 `this` 作为上下文会被 `webpack` 填充，因此我们不能将 `loader` 设为一个箭头函数

函数接受一个参数，为 `webpack` 传递给 `loader` 的文件源内容

函数中 `this` 是由 `webpack` 提供的对象，能够获取当前 `loader` 所需要的各种信息

函数中有异步操作或同步操作，异步操作通过 `this.callback` 返回，返回值要求为 `string` 或者 `Buffer`

代码如下所示：

```
// 导出一个函数，source为webpack传递给loader的文件源内容
module.exports = function(source) {
    const content = doSomeThing2JsString(source);

    // 如果 loader 配置了 options 对象，那么this.query将指向 options
```

```
    const options = this.query;

    // 可以用作解析其他模块路径的上下文
    console.log('this.context');

    /*
     * this.callback 参数：
     * error：Error | null，当 loader 出错时向外抛出一个 error
     * content：String | Buffer，经过 loader 编译后需要导出的内容
     * sourceMap：为方便调试生成的编译后内容的 source map
     * ast：本次编译生成的 AST 静态语法树，之后执行的 loader 可以直接使用这个 AST，进而省去重复生成 AST 的过程
     */
    this.callback(null, content); // 异步
    return content; // 同步
}
```

一般在编写 `loader` 的过程中，保持功能单一，避免做多种功能

如 `less` 文件转换成 `css` 文件也不是一步到位，而是 `less-loader` 、 `css-loader` 、 `style-loader` 几个 `loader` 的链式调用才能完成转换

7.3. 编写plugin

由于 `webpack` 基于发布订阅模式，在运行的生命周期中会广播出许多事件，插件通过监听这些事件，就可以在特定的阶段执行自己的插件任务

在之前也了解过， `webpack` 编译会创建两个核心对象：

- compiler：包含了 webpack 环境的所有的配置信息，包括 options，loader 和 plugin，和 webpack 整个生命周期相关的钩子
- compilation：作为 plugin 内置事件回调函数的参数，包含了当前的模块资源、编译生成资源、变化的文件以及被跟踪依赖的状态信息。当检测到一个文件变化，一次新的 Compilation 将被创建

如果自己要实现 `plugin` ，也需要遵循一定的规范：

- 插件必须是一个函数或者是一个包含 `apply` 方法的对象，这样才能访问 `compiler` 实例
- 传给每个插件的 `compiler` 和 `compilation` 对象都是同一个引用，因此不建议修改
- 异步的事件需要在插件处理完任务时调用回调函数通知 `Webpack` 进入下一个流程，不然会卡住

实现 `plugin` 的模板如下：

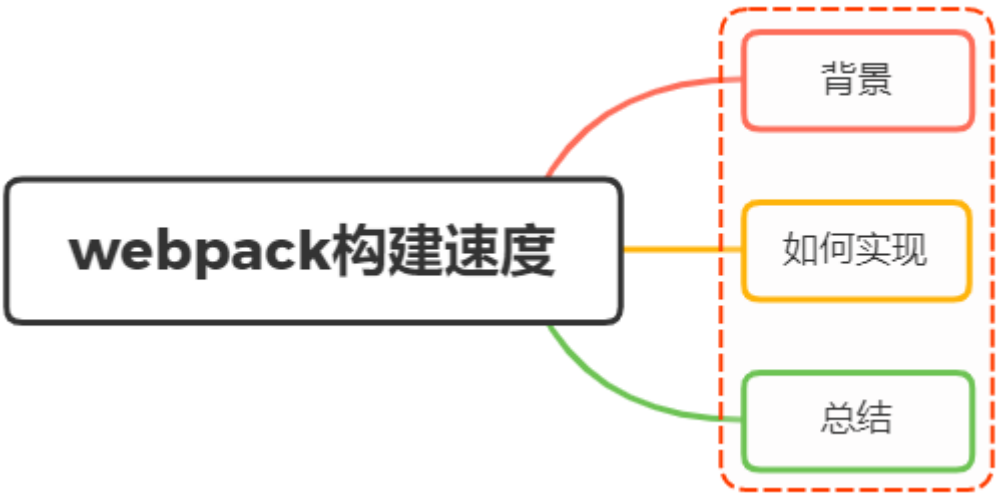
```
class MyPlugin {
    // Webpack 会调用 MyPlugin 实例的 apply 方法给插件实例传入 compiler 对象
  apply (compiler) {
    // 找到合适的事件钩子，实现自己的插件功能
```

```
    compiler.hooks.emit.tap('MyPlugin', compilation => {
        // compilation: 当前打包构建流程的上下文
        console.log(compilation);

        // do something...
    })
  }
}
```

在 `emit` 事件发生时，代表源文件的转换和组装已经完成，可以读取到最终将输出的资源、代码块、模块及其依赖，并且可以修改输出资源的内容

8. 如何提高webpack的构建速度?



8.1. 背景

随着我们的项目涉及到页面越来越多，功能和业务代码也会随着越多，相应的 `webpack` 的构建时间也会越来越久

构建时间与我们日常开发效率密切相关，当我们本地开发启动 `devServer` 或者 `build` 的时候，如果时间过长，会大大降低我们的工作效率

所以，优化 `webpack` 构建速度是十分重要的环节

8.2. 如何优化

常见的提升构建速度的手段有如下：

- 优化 loader 配置
- 合理使用 resolve.extensions
- 优化 resolve.modules

- 优化 resolve.alias
- 使用 DLLPlugin 插件
- 使用 cache-loader
- terser 启动多线程
- 合理使用 sourceMap

8.2.1. 优化loader配置

在使用 `loader` 时，可以通过配置 `include`、`exclude`、`test` 属性来匹配文件，接触 `include`、`exclude` 规定哪些匹配应用 `loader`

如采用 ES6 的项目为例，在配置 `babel-loader` 时，可以这样：

```
module.exports = {
  module: {
    rules: [
      {
        // 如果项目源码中只有 js 文件就不要写成 /\.jsx?
$/，提升正则表达式性能
        test: /\.js$
/,
        // babel-loader 支持缓存转换出的结果，通过 cacheDirectory 选项开启
        use: ['babel-loader?cacheDirectory'],
        // 只对项目根目录下的 src 目录中的文件采用 babel-loader
        include: path.resolve(__dirname, 'src'),
      },
    ]
  },
};
```

8.2.2. 合理使用 resolve.extensions

在开发中我们会有各种各样的模块依赖，这些模块可能来自于自己编写的代码，也可能来自第三方库， `resolve` 可以帮助 `webpack` 从每个 `require/import` 语句中，找到需要引入到合适的模块代码

通过 `resolve.extensions` 是解析到文件时自动添加拓展名，默认情况如下：

```
module.exports = {
    ...
    extensions:[".warm",".mjs",".js",".json"]
}
```

当我们引入文件的时候，若没有文件后缀名，则会根据数组内的值依次查找

当我们配置的时候，则不要随便把所有后缀都写在里面，这会调用多次文件的查找，这样就会减慢打包速度

8.2.3. 优化 resolve.modules

`resolve.modules` 用于配置 `webpack` 去哪些目录下寻找第三方模块。默认值为 `['node_modules']`，所以默认会从 `node_modules` 中查找文件
当安装的第三方模块都放在项目根目录下的 `./node_modules` 目录下时，所以可以指明存放第三方模块的绝对路径，以减少寻找，配置如下：

```
module.exports = {
  resolve: {
    // 使用绝对路径指明第三方模块存放的位置，以减少搜索步骤
    // 其中 __dirname 表示当前工作目录，也就是项目根目录
    modules: [path.resolve(__dirname, 'node_modules')]
  },
};
```

8.2.4. 优化 resolve.alias

`alias` 给一些常用的路径起一个别名，特别当我们的项目目录结构比较深的时候，一个文件的路径可能是 `./../../` 的形式

通过配置 `alias` 以减少查找过程

```
module.exports = {
    ...
    resolve:{
        alias:{
            "@":path.resolve(__dirname,'./src')
        }
    }
}
```

8.2.5. 使用 DLLPlugin 插件

`DLL` 全称是 动态链接库，是为软件在winodw种实现共享函数库的一种实现方式，而Webpack也内置了DLL的功能，为的就是可以共享，不经常改变的代码，抽成一个共享的库。这个库在之后的编译过程中，会被引入到其他项目的代码中

使用步骤分成两部分：

- 打包一个 DLL 库
- 引入 DLL 库

8.2.5.1. 打包一个 DLL 库

`webpack` 内置了一个 `DllPlugin` 可以帮助我们打包一个DLL的库文件

```
module.exports = {
    ...
    plugins:[
        new webpack.DllPlugin({
            name:'dll_[name]',
            path:path.resolve(__dirname,"./dll/[name].mainfest.json")
        })
    ]
}
```

8.2.5.2. 引入 DLL 库

使用 `webpack` 自带的 `DllReferencePlugin` 插件对 `mainfest.json` 映射文件进行分析，获取要使用的 `DLL` 库

然后再通过 `AddAssetHtmlPlugin` 插件，将我们打包的 `DLL` 库引入到 `Html` 模块中

```
module.exports = {
    ...
    new webpack.DllReferencePlugin({
        context:path.resolve(__dirname,"./dll/dll_react.js"),
        mainfest:path.resolve(__dirname,"./dll/react.mainfest.json")
    }),
    new AddAssetHtmlPlugin({
        outputPath:"./auto",
        filepath:path.resolve(__dirname,"./dll/dll_react.js")
    })
}
```

8.2.6. 使用 cache-loader

在一些性能开销较大的 `loader` 之前添加 `cache-loader` ，以将结果缓存到磁盘里，显著提升二次构建速度

保存和读取这些缓存文件会有一些时间开销，所以请只对性能开销较大的 `loader` 使用此 `loader`

```
module.exports = {
    module: {
        rules: [
            {
                test: /\.ext$/,
                use: ['cache-loader', ...loaders],
                include: path.resolve('src'),
            },
        ],
    },
};
```

8.2.7. terser 启动多线程

使用多进程并行运行来提高构建速度

```
module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        parallel: true,
      }),
    ],
  },
};
```

8.2.8. 合理使用 sourceMap

打包生成 `sourceMap` 的时候，如果信息越详细，打包速度就会越慢。对应属性取值如下所示：

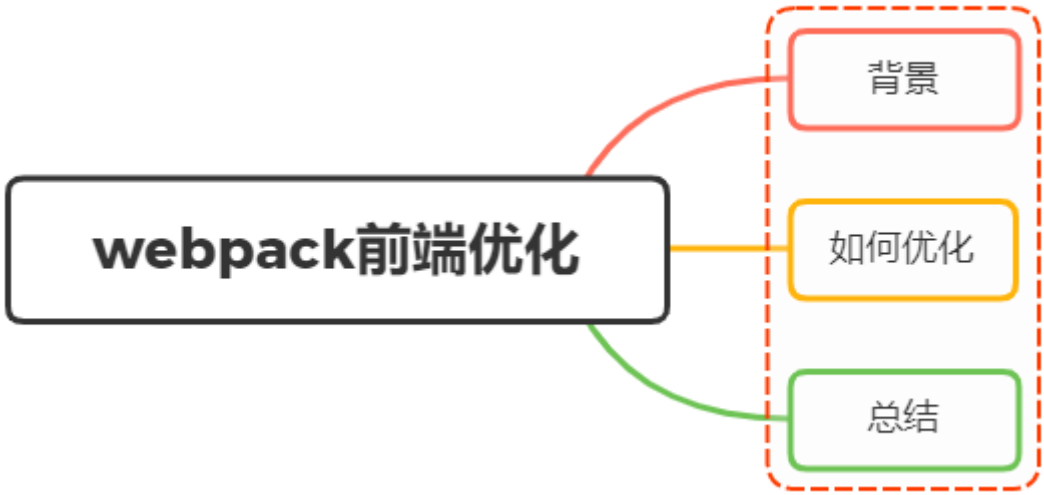
devtool	构建速度	重新构建速度	生产环境	品质(quality)
(none)	+++	+++	yes	打包后的代码
eval	+++	+++	no	生成后的代码
cheap-eval-source-map	+	++	no	转换过的代码（仅限行）
cheap-module-eval-source-map	o	++	no	原始源代码（仅限行）
eval-source-map	--	+	no	原始源代码
cheap-source-map	+	o	yes	转换过的代码（仅限行）
cheap-module-source-map	o	-	yes	原始源代码（仅限行）
inline-cheap-source-map	+	o	no	转换过的代码（仅限行）
inline-cheap-module-source-map	o	-	no	原始源代码（仅限行）
source-map	--	--	yes	原始源代码
inline-source-map	--	--	no	原始源代码
hidden-source-map	--	--	yes	原始源代码
nosources-source-map	--	--	yes	无源代码内容

+++ 非常快速 ++ 快速 + 比较快 o 中等 - 比较慢 -- 慢

8.3. 总结

可以看到，优化 `webpack` 构建的方式有很多，主要可以从优化搜索时间、缩小文件搜索范围、减少不必要的编译等方面入手

9. 说说如何借助webpack来优化前端性能?



9.1. 背景

随着前端的项目逐渐扩大，必然会带来的一个问题就是性能

尤其在大型复杂的项目中，前端业务可能因为一个小小的数据依赖，导致整个页面卡顿甚至奔溃

一般项目在完成后，会通过 `webpack` 进行打包，利用 `webpack` 对前端项目性能优化是一个十分重要的环节

9.2. 如何优化

通过 `webpack` 优化前端的手段有：

- JS代码压缩
- CSS代码压缩
- Html文件代码压缩
- 文件大小压缩
- 图片压缩
- Tree Shaking
- 代码分离
- 内联 chunk

9.2.1. JS代码压缩

`terser` 是一个 `JavaScript` 的解释、绞肉机、压缩机的工具集，可以帮助我们压缩、丑化我们的代码，让 `bundle` 更小

在 `production` 模式下， `webpack` 默认就是使用 `TerserPlugin` 来处理我们的代码的。如果想要自定义配置它，配置方法如下：

```
1 const TerserPlugin = require('terser-webpack-plugin')
```

```
module.exports = {
    ...
    optimization: {
        minimize: true,
        minimizer: [
            new TerserPlugin({
                parallel: true // 电脑cpu核数-1
            })
        ]
    }
}
```

属性介绍如下

- extractComments：默认值为true，表示会将注释抽取到一个单独的文件中，开发阶段，我们可设置为 false ，不保留注释
- parallel：使用多进程并发运行提高构建的速度，默认值是true，并发运行的默认数量：os.cpus().length - 1
- terserOptions：设置我们的terser相关的配置：
- compress：设置压缩相关的选项，mangle：设置丑化相关的选项，可以直接设置为true
- mangle：设置丑化相关的选项，可以直接设置为true
- toplevel：底层变量是否进行转换
- keep_classnames：保留类的名称
- keep_fnames：保留函数的名称

9.2.2. CSS代码压缩

`CSS` 压缩通常是去除无用的空格等，因为很难去修改选择器、属性的名称、值等

CSS的压缩我们可以使用另外一个插件：`css-minimizer-webpack-plugin`

```
npm install css-minimizer-webpack-plugin -D
```

配置方法如下：

```
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin')
module.exports = {
    // ...
    optimization: {
        minimize: true,
```

```
        minimizer: [
            new CssMinimizerPlugin({
                parallel: true
            })
        ]
    }
}
```

9.2.3. Html文件代码压缩

使用 `HtmlWebpackPlugin` 插件来生成 `HTML` 的模板时候，通过配置属性 `minify` 进行 `html` 优化

```
module.exports = {
    ...
    plugin:[
        new HtmlwebpackPlugin({
            ...
            minify:{
                minifyCSS:false, // 是否压缩css
                collapseWhitespace:false, // 是否折叠空格
                removeComments:true // 是否移除注释
            }
        })
    ]
}
```

设置了 `minify` ，实际会使用另一个插件 `html-minifier-terser`

9.2.4. 文件大小压缩

对文件的大小进行压缩，减少 `http` 传输过程中宽带的损耗

```
npm install compression-webpack-plugin -D
```

```
new ComepressionPlugin({
    test:/\.(css|js)$/,  // 哪些文件需要压缩
    threshold:500, // 设置文件多大开始压缩
    minRatio:0.7, // 至少压缩的比例
    algorithm:"gzip", // 采用的压缩算法
})
```

9.2.5. 图片压缩

一般来说在打包之后，一些图片文件的大小是远远要比 `js` 或者 `css` 文件要来的大，所以图片压缩较为重要

配置方法如下：

```
module: {
  rules: [
    {
      test: /\.(png|jpg|gif)$/,
      use: [
        {
          loader: 'file-loader',
          options: {
            name: '[name]_[hash].[ext]',
            outputPath: 'images/',
          }
        },
        {
          loader: 'image-webpack-loader',
          options: {
            // 压缩 jpeg 的配置
            mozjpeg: {
              progressive: true,
              quality: 65
            },
            // 使用 imagemin**-optipng 压缩 png，enable: false 为关闭
            optipng: {
              enabled: false,
            },
            // 使用 imagemin-pngquant 压缩 png
            pngquant: {
              quality: '65-90',
              speed: 4
            },
            // 压缩 gif 的配置
            gifsicle: {
              interlaced: false,
            },
            // 开启 webp，会把 jpg 和 png 图片压缩为 webp 格式
            webp: {
              quality: 75
            }
          }
```

```
        }
      ]
    },
  ]
}
```

9.2.6. Tree Shaking

`Tree Shaking` 是一个术语，在计算机中表示消除死代码，依赖于 `ES Module` 的静态语法分析（不执行任何的代码，可以明确知道模块的依赖关系）

在 `webpack` 实现 `Trss shaking` 有两种不同的方案：

- usedExports：通过标记某些函数是否被使用，之后通过Terser来进行优化的
- sideEffects：跳过整个模块/文件，直接查看该文件是否有副作用

两种不同的配置方案， 有不同的效果

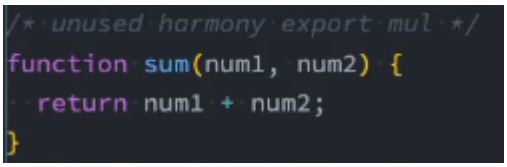
9.2.6.1. usedExports

配置方法也很简单，只需要将 `usedExports` 设为 `true`

```
module.exports = {
    ...
    optimization:{
        usedExports
    }
}
```

使用之后，没被用上的代码在 `webpack` 打包中会加入 `unused harmony export mul` 注释，用来告知 `Terser` 在优化时，可以删除掉这段代码

如下面 `sum` 函数没被用到， `webpack` 打包会添加注释， `terser` 在优化时，则将该函数去掉



9.2.6.2. sideEffects

`sideEffects` 用于告知 `webpack compiler` 哪些模块时有副作用，配置方法是在 `package.json` 中设置 `sideEffects` 属性

如果 `sideEffects` 设置为false，就是告知 `webpack` 可以安全的删除未用到的 `exports`

如果有些文件需要保留，可以设置为数组的形式

```
"sideEffecis":[
    "./src/util/format.js",
    "*.css" // 所有的css文件
]
```

上述都是关于 `javascript` 的 `tree shaking`， `css` 同样也能够实现 `tree shaking`

9.2.6.3. css tree shaking

`css` 进行 `tree shaking` 优化可以安装 `PurgeCss` 插件

```
npm install purgecss-plugin-webpack -D
```

```
const PurgeCssPlugin = require('purgecss-webpack-plugin')
module.exports = {
    ...
    plugins:[
        new PurgeCssPlugin({
            path:glob.sync(
${path.resolve('./src')}/**/*
), {nodir:true}// src里面的所有文件
            satelist:function(){
                return {
                    standard:["html"]
                }
            }
        })
    ]
}
```

- paths：表示要检测哪些目录下的内容需要被分析，配合使用glob
- 默认情况下，Purgecss会将我们的html标签的样式移除掉，如果我们希望保留，可以添加一个safelist的属性

9.2.7. 代码分离

将代码分离到不同的 `bundle` 中，之后我们可以按需加载，或者并行加载这些文件

默认情况下，所有的 `JavaScript` 代码（业务代码、第三方依赖、暂时没有用到的模块）在首页全部都加载，就会影响首页的加载速度

代码分离可以分出出更小的 `bundle` ，以及控制资源加载优先级，提供代码的加载性能

这里通过 `splitChunksPlugin` 来实现，该插件 `webpack` 已经默认安装和集成，只需要配置即可

默认配置中，chunks仅仅针对于异步（async）请求，我们可以设置为initial或者all

```
module.exports = {
    ...
    optimization:{
        splitChunks:{
            chunks:"all"
        }
    }
}
```

`splitChunks` 主要属性有如下：

- Chunks，对同步代码还是异步代码进行处理
- minSize： 拆分包的大小, 至少为minSize，如何包的大小不超过minSize，这个包不会拆分
- maxSize： 将大于maxSize的包，拆分为不小于minSize的包
- minChunks：被引入的次数，默认是1

9.2.8. 内联chunk

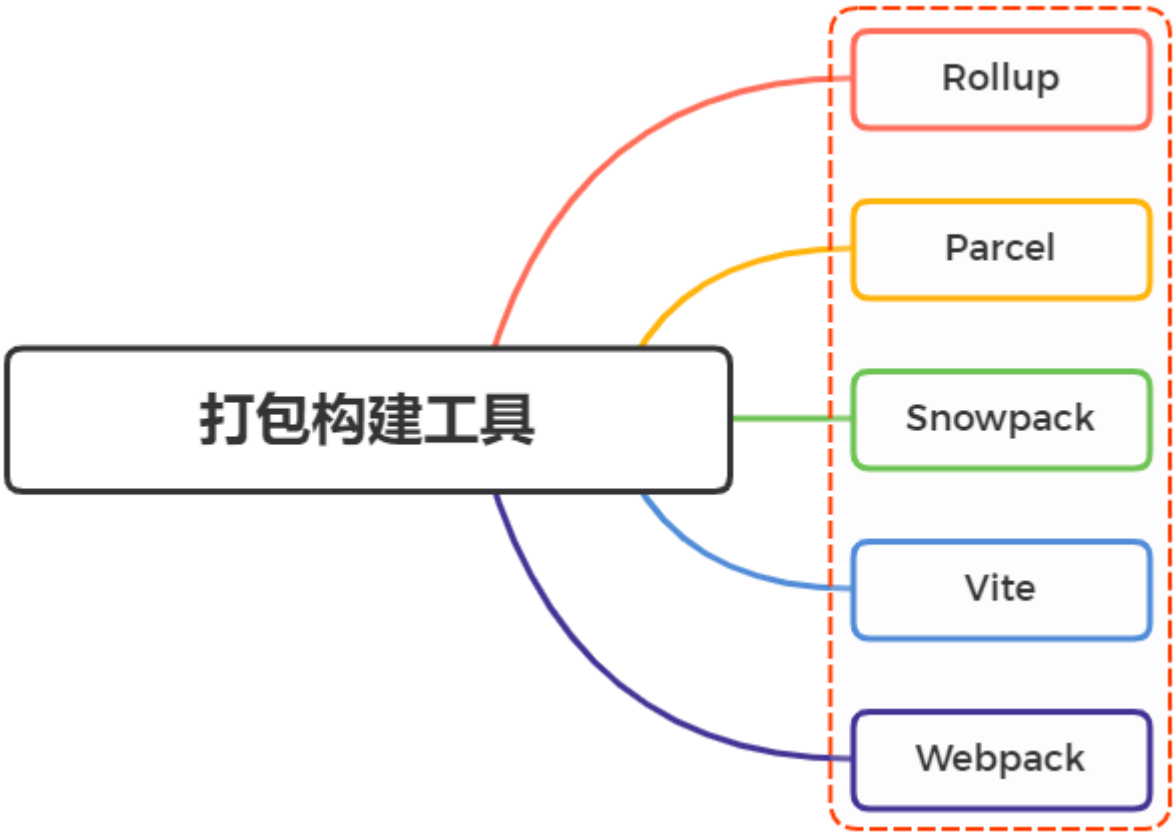
可以通过 `InlineChunkHtmlPlugin` 插件将一些 `chunk` 的模块内联到 `html` ，如 `runtime` 的代码（对模块进行解析、加载、模块信息相关的代码），代码量并不大，但是必须加载的

```
const InlineChunkHtmlPlugin = require('react-dev-utils/InlineChunkHtmlPlugin')
const HtmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
    ...
    plugin:[
        new InlineChunkHtmlPlugin(HtmlWebpackPlugin,[/runtime.+\.js/]
}
```

9.3. 总结

关于 `webpack` 对前端性能的优化，可以通过文件体积大小入手，其次还可通过分包的形式、减少http请求次数等方式，实现对前端性能的优化

10. 与webpack类似的工具还有哪些？区别？



10.1. 模块化工具

模块化是一种处理复杂系统分解为更好的可管理模块的方式

可以用来分割，组织和打包应用。每个模块完成一个特定的子功能，所有的模块按某种方法组装起来，成为一个整体(`bundle`)

在前端领域中，并非只有 `webpack` 这一款优秀的模块打包工具，还有其他类似的工具，例如 `Rollup` 、 `Parcel` 、 `snowpack` ，以及最近风头无两的 `Vite`

通过这些模块打包工具，能够提高我们的开发效率，减少开发成本

这里没有提及 `gulp` 、 `grunt` 是因为它们只是定义为构建工具，不能类比

10.1.1. Rollup

`Rollup` 是一款 `ES Modules` 打包器，从作用上来看， `Rollup` 与 `Webpack` 非常类似。不过相比于 `Webpack` ， `Rollup` 要小巧的多

现在很多我们熟知的库都都使用它进行打包，比如： `Vue` 、 `React` 和 `three.js` 等

举个例子：

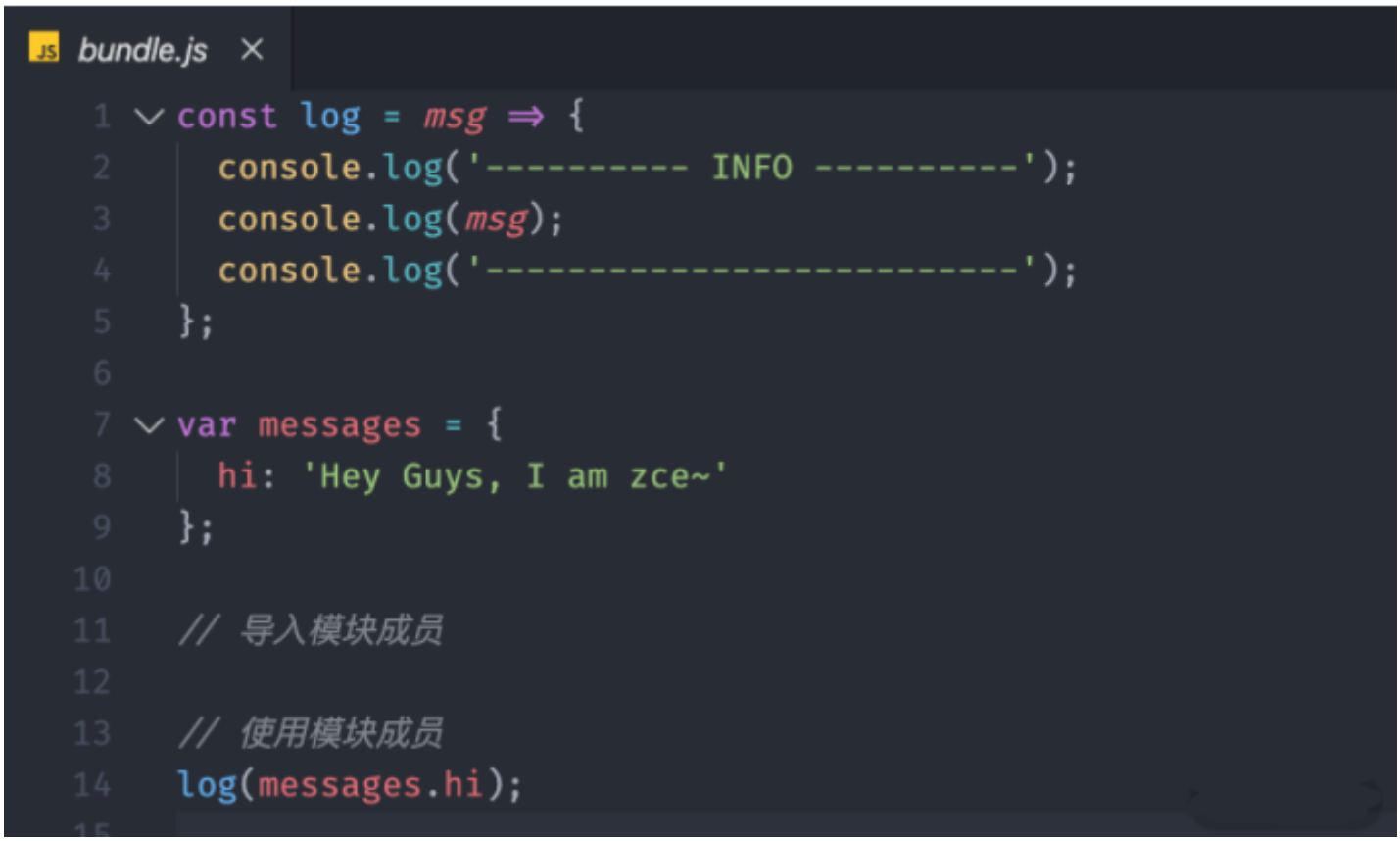
```
// ./src/messages.js
export default {
  hi: 'Hey Guys, I am zce~'
}
// ./src/logger.js
```

```
export const log = msg => {
  console.log('---------- INFO ----------')
  console.log(msg)
  console.log('--------------------------')
}
export const error = msg => {
  console.error('---------- ERROR ----------')
  console.error(msg)
  console.error('---------------------------')
}
// ./src/index.js
import { log } from './logger'
import messages from './messages'
log(messages.hi)
```

然后通过 `rollup` 进行打包

```
$ npx rollup ./src/index.js --file ./dist/bundle.js
```

打包结果如下图



可以看到，代码非常简洁，完成不像 `webpack` 那样存在大量引导代码和模块函数

并且 `error` 方法由于没有被使用，输出的结果中并无 `error` 方法，可以看到， `rollup` 默认开始 `Tree-shaking` 优化输出结果

因此，可以看到 `Rollup` 的优点：

- 代码效率更简洁、效率更高
- 默认支持 Tree-shaking

但缺点也十分明显，加载其他类型的资源文件或者支持导入 `CommonJS` 模块，又或是编译 `ES` 新特性，这些额外的需求 `Rollup` 需要使用插件去完成

综合来看， `rollup` 并不适合开发应用使用，因为需要使用第三方模块，而目前第三方模块大多数使用 `CommonJs` 方式导出成员，并且 `rollup` 不支持 `HMR` ，使开发效率降低

但是在用于打包 `JavaScript` 库时， `rollup` 比 `webpack` 更有优势，因为其打包出来的代码更小、更快，其存在的缺点可以忽略

10.1.2. Parcel

Parcel ，是一款完全零配置的前端打包器，它提供了 “傻瓜式” 的使用体验，只需了解简单的命令，就能构建前端应用程序

`Parcel` 跟 `Webpack` 一样都支持以任意类型文件作为打包入口，但建议使用 `HTML` 文件作为入口，该 `HTML` 文件像平时一样正常编写代码、引用资源。如下所示：

```
<!-- ./src/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Parcel Tutorials</title>
</head>
<body>
  <script src="main.js"></script>
</body>
</html>
```

main.js文件通过 `ES Moudle` 方法导入其他模块成员

```
// ./src/main.js
import { log } from './logger'
log('hello parcel')
// ./src/logger.js
export const log = msg => {
  console.log('---------- INFO ----------')
```

```
7   console.log(msg)
8 }
```

运行之后，使用命令打包

```
1 npx parcel src/index.html
```

执行命令后，`Parcel` 不仅打包了应用，同时也启动了一个开发服务器，跟 `webpack Dev Server` 一样

跟 `webpack` 类似，也支持模块热替换，但用法更简单

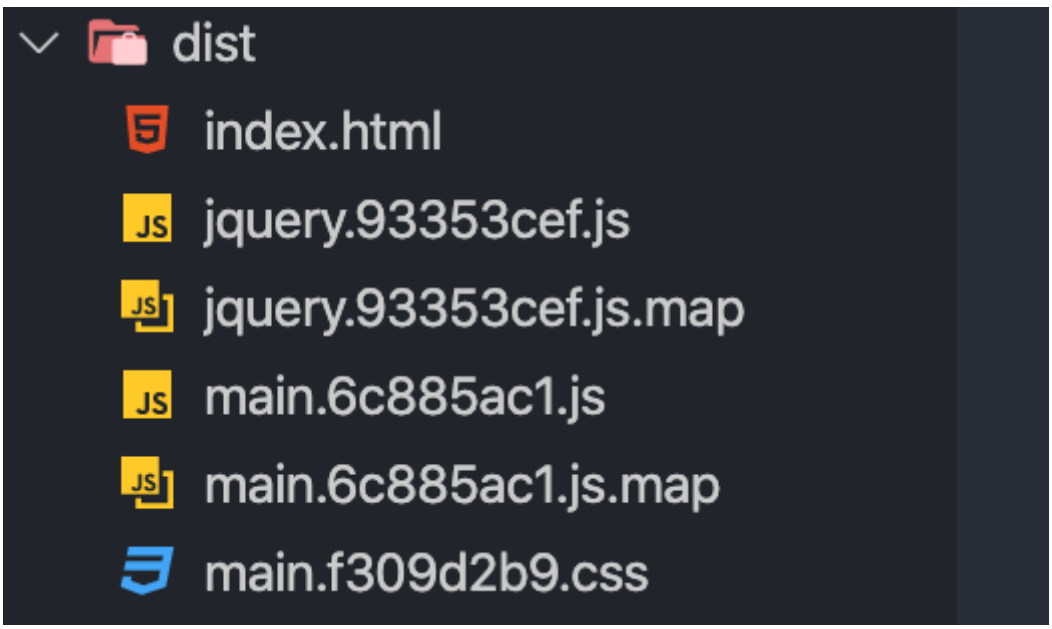
同时，`Parcel` 有个十分好用的功能：支持自动安装依赖，像 `webpack` 开发阶段突然使用安装某个第三方依赖，必然会终止 `dev server` 然后安装再启动。而 `Parcel` 则免了这繁琐的工作流程

同时，`Parcel` 能够零配置加载其他类型的资源文件，无须像 `webpack` 那样配置对应的 `loader`

打包命令如下：

```
1 npx parcel src/index.html
```

由于打包过程是多进程同时工作，构建速度会比 `Webpack` 快，输出文件也会被压缩，并且样式代码也会被单独提取到单个文件中



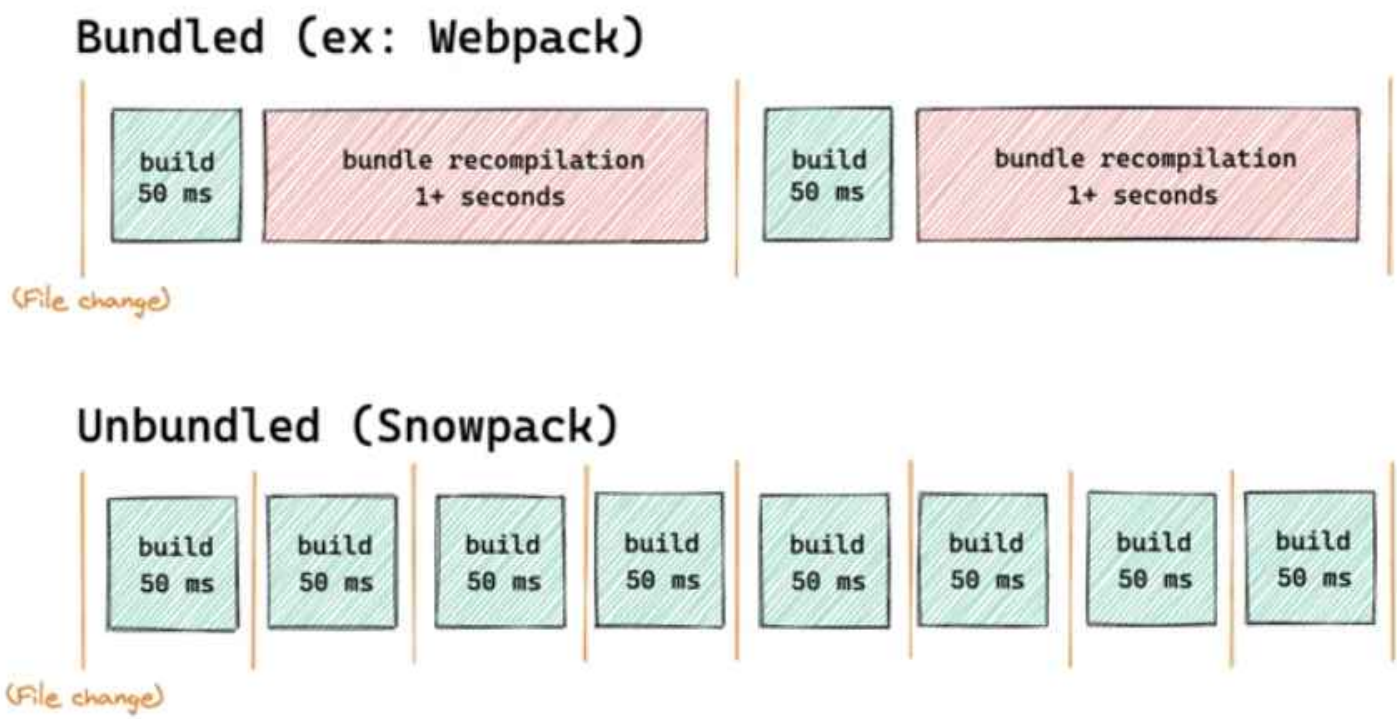
可以感受到，`Parcel` 给开发者一种很大的自由度，只管去实现业务代码，其他事情用 `Parcel` 解决

10.1.3. Snowpack

Snowpack，是一种闪电般快速的前端构建工具，专为现代`Web`设计，较复杂的打包工具（如`Webpack`或`Parcel`）的替代方案，利用`JavaScript`的本机模块系统，避免不必要的工作并保持流畅的开发体验

开发阶段，每次保存单个文件时，`Webpack`和`Parcel`都需要重新构建和重新打包应用程序的整个`bundle`。而`Snowpack`为你的应用程序每个文件构建一次，就可以永久缓存，文件更改时，`Snowpack`会重新构建该单个文件

下图给出`webpack`与`snowpack`打包区别：



在重新构建每次变更时没有任何的时间浪费，只需要在浏览器中进行HMR更新

10.1.4. Vite

vite，是一种新型前端构建工具，能够显著提升前端开发体验

它主要由两部分组成：

- 一个开发服务器，它基于 原生 ES 模块 提供了丰富的内建功能，如速度快到惊人的 [模块热更新 HMR
- 一套构建指令，它使用 Rollup打包你的代码，并且它是预配置的，可以输出用于生产环境的优化过的静态资源

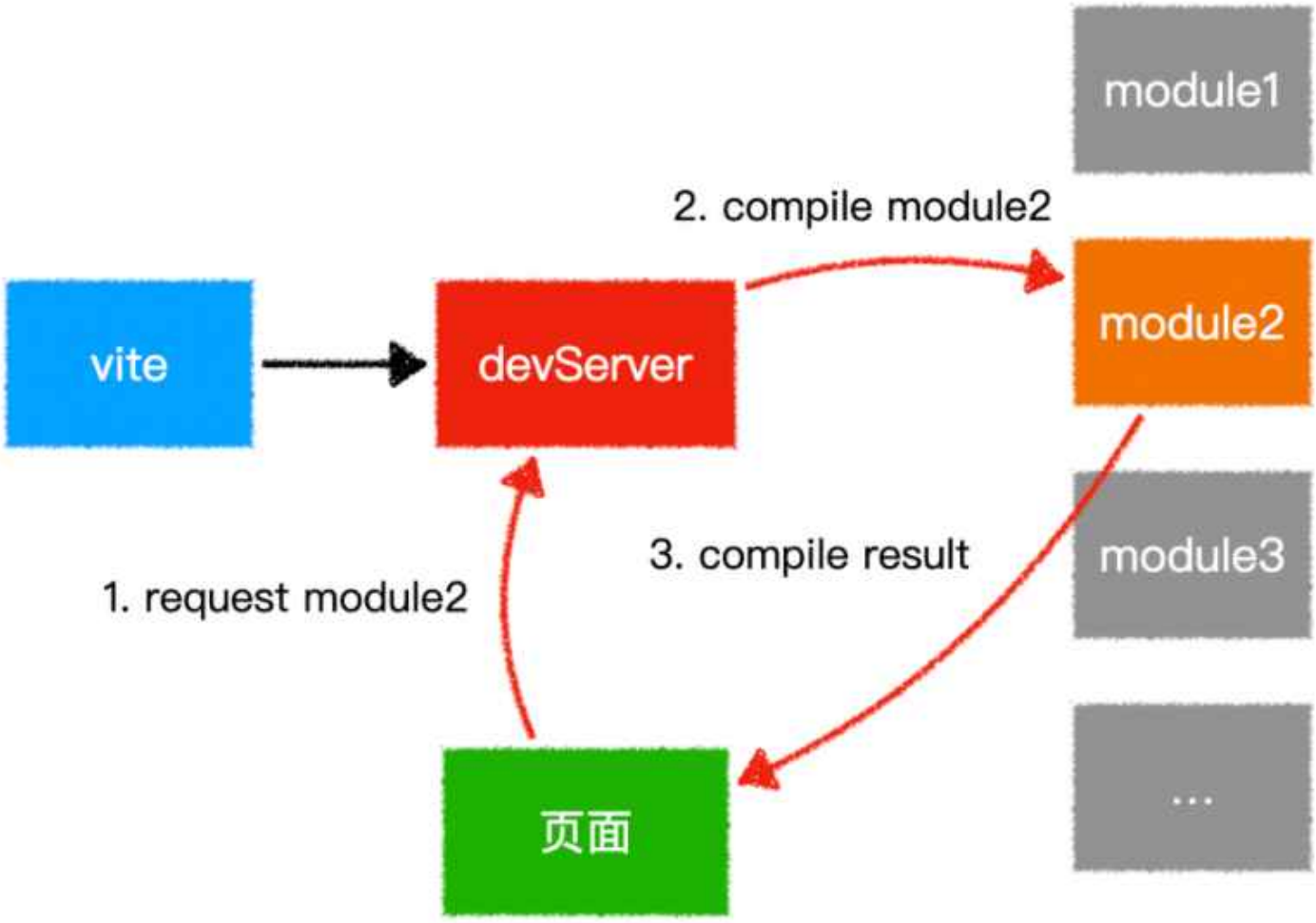
其作用类似`webpack` + `webpack-dev-server`，其特点如下：

- 快速的冷启动
- 即时的模块热更新
- 真正的按需编译

`vite` 会直接启动开发服务器，不需要进行打包操作，也就意味着不需要分析模块的依赖、不需要编译，因此启动速度非常快

利用现代浏览器支持 `ES Module` 的特性，当浏览器请求某个模块的时候，再根据需要对模块的内容进行编译，这种方式大大缩短了编译时间

原理图如下所示：



在热模块 `HMR` 方面，当修改一个模块的时候，仅需让浏览器重新请求该模块即可，无须像 `webpack` 那样需要把该模块的相关依赖模块全部编译一次，效率更高

10.1.5. webpack

相比上述的模块化工具， `webpack` 大而全，很多常用的功能做到开箱即用。有两大最核心的特点：**一切皆模块**和**按需加载**

与其他构建工具相比，有如下优势：

- 智能解析：对 CommonJS 、 AMD 、ES6 的语法做了兼容
- 万物模块：对 js、css、图片等资源文件都支持打包
- 开箱即用：HRM、Tree-shaking等功能
- 代码分割：可以将代码切割成不同的 chunk，实现按需加载，降低了初始化时间

- 插件系统，具有强大的 Plugin 接口，具有更好的灵活性和扩展性
- 易于调试：支持 SourceUrls 和 SourceMaps
- 快速运行：webpack 使用异步 IO 并具有多级缓存，这使得 webpack 很快且在增量编译上更加快
- 生态环境好：社区更丰富，出现的问题更容易解决