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

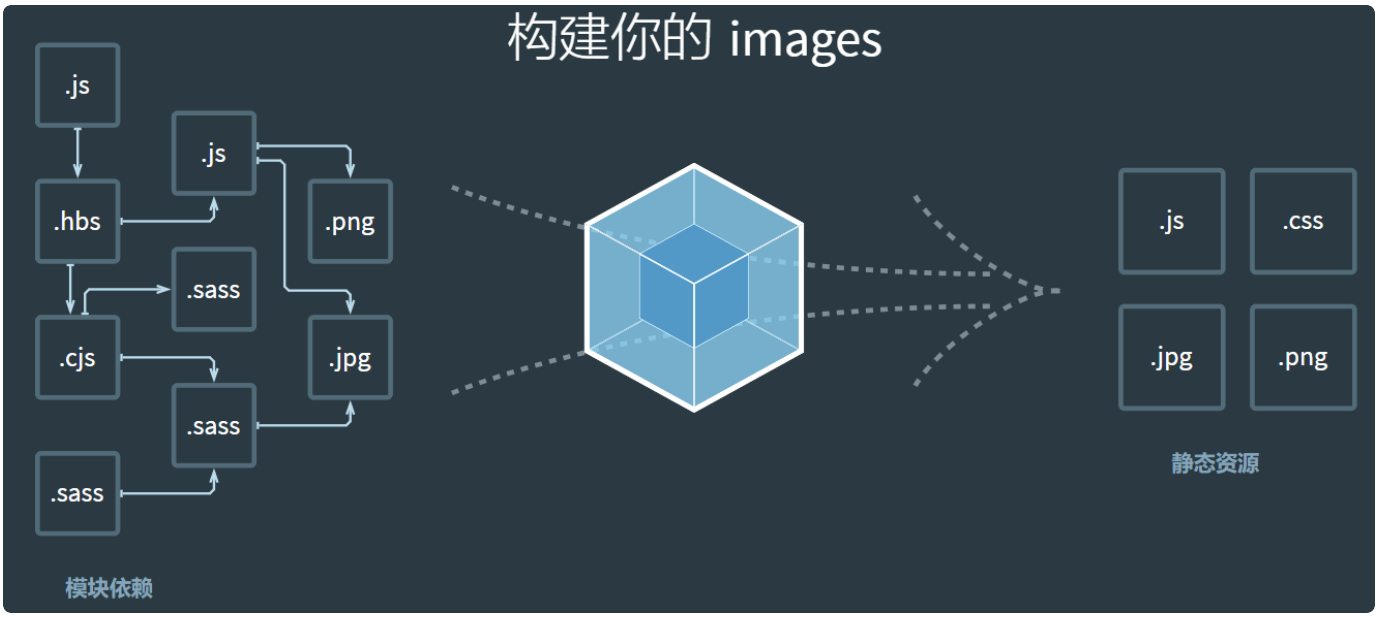
1.3. 是什么

`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

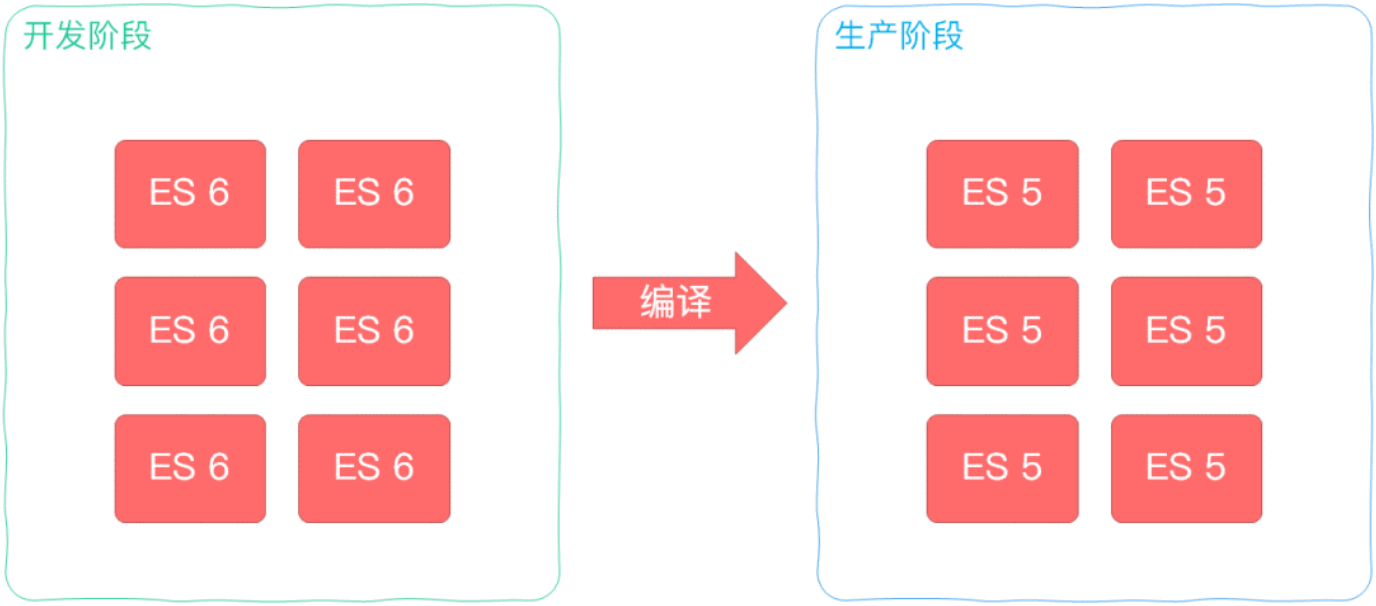
这里的静态模块指的是开发阶段，可以被 `webpack` 直接引用的资源（可以直接被获取打包进 `bundle.js` 的资源）

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`

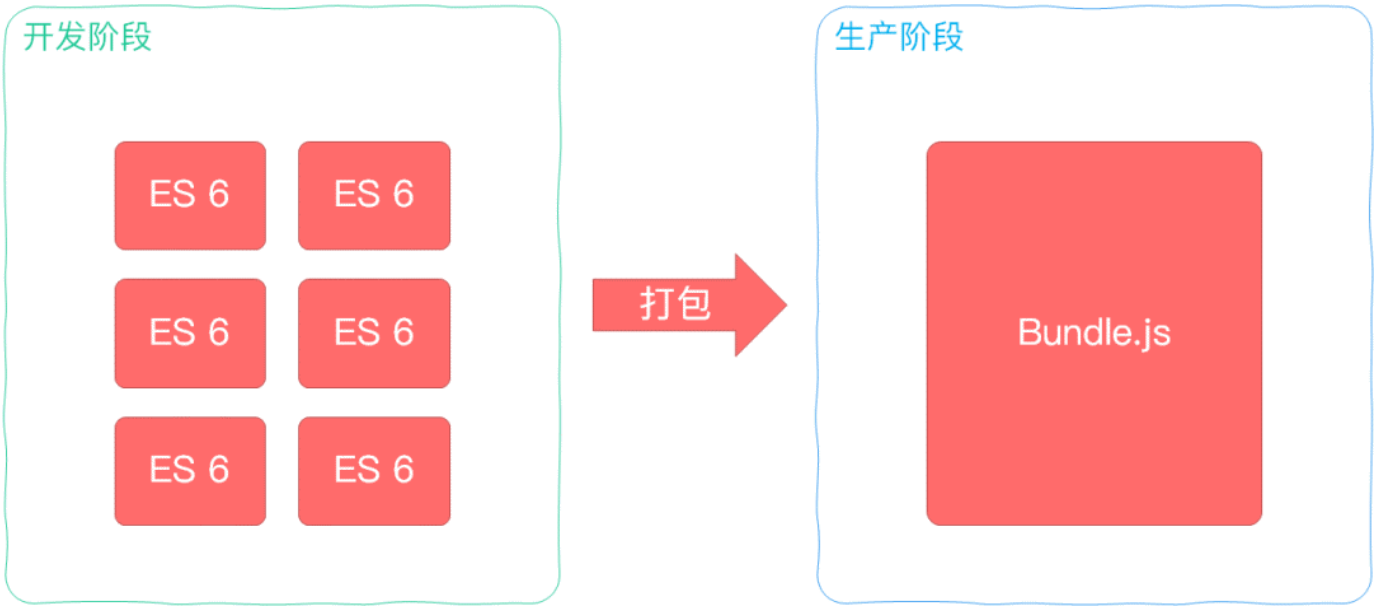


1.3.1. `webpack` 的能力：

编译代码能力，提高效率，解决浏览器兼容问题

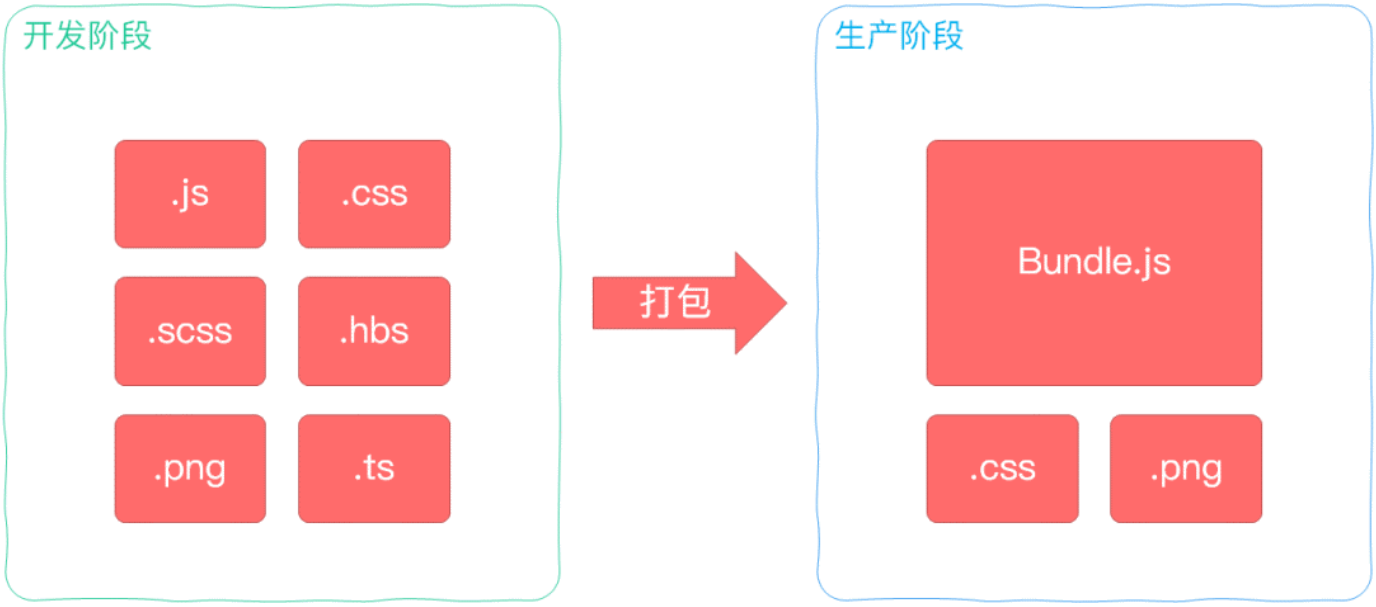


模块整合能力，提高性能，可维护性，解决浏览器频繁请求文件的问题

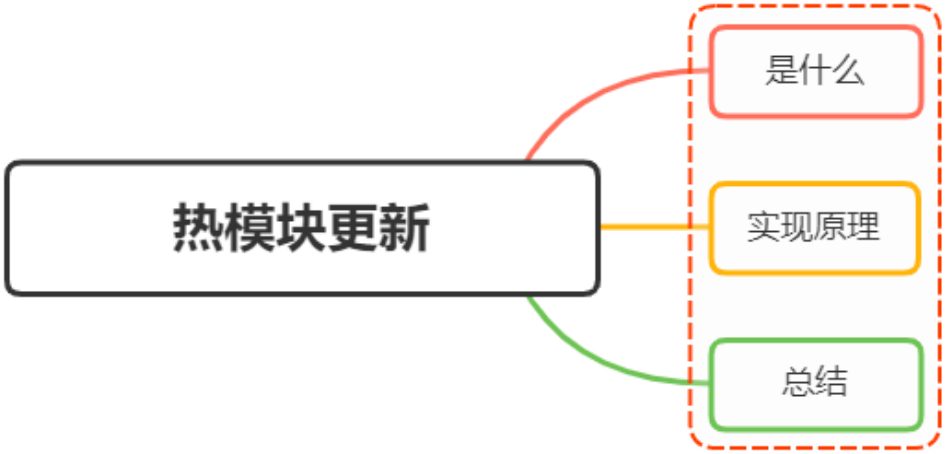


万物皆可模块能力，项目维护性增强，支持不同种类的前端模块类型，统一的模块化方案，所有资源文

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：

JavaScript | 复制代码

```
const webpack = require('webpack')
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

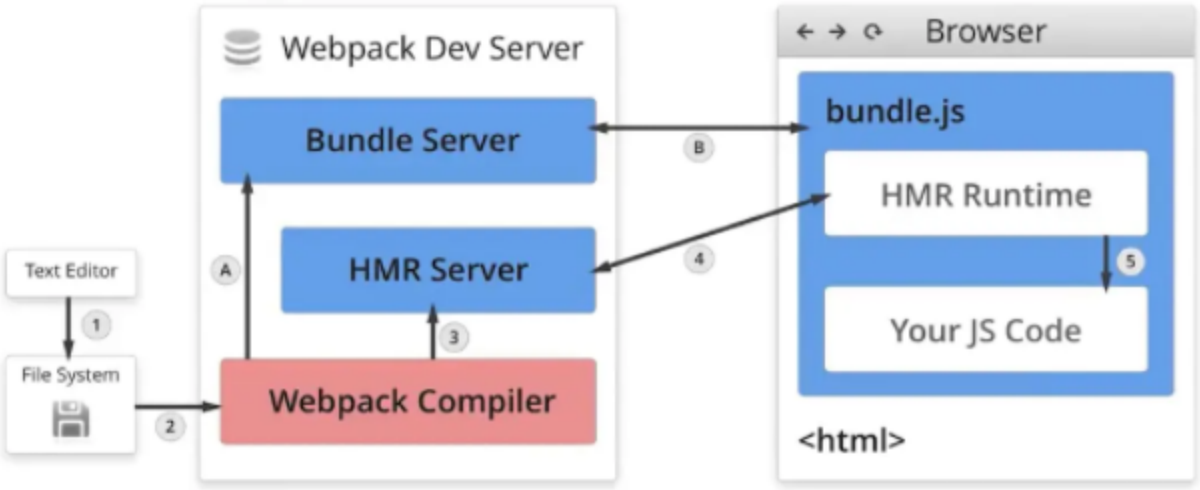
我们需要去指定哪些模块发生更新时进行 `HRM` ，如下代码：

JavaScript | 复制代码

```
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

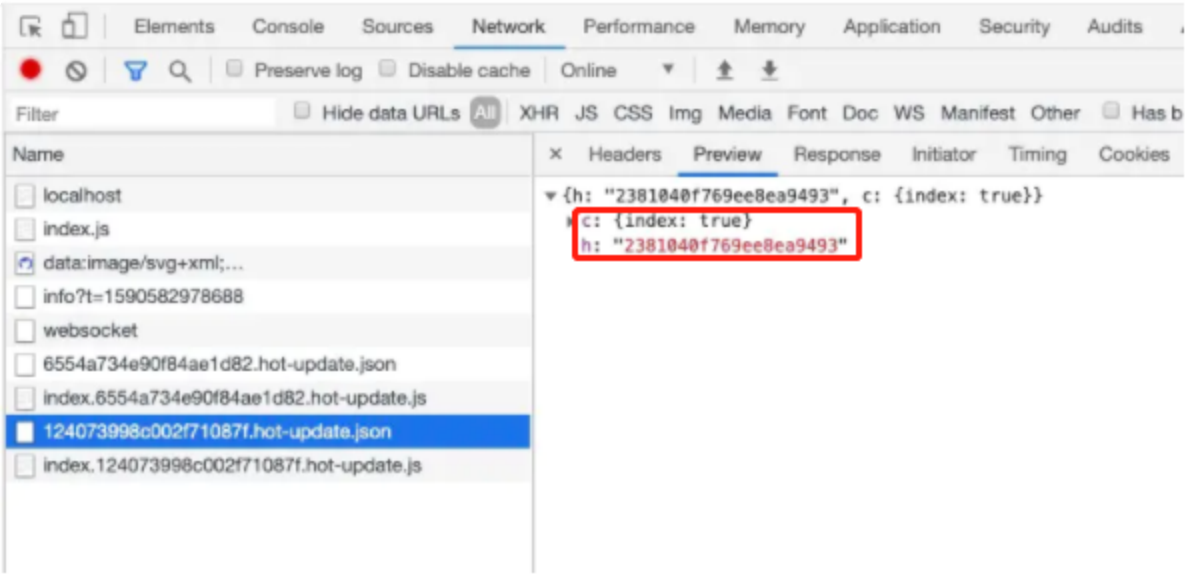
在编写未经过 `webpack` 打包的源代码后， `Webpack Compile` 将源代码和 `HMR Runtime` 一起编译成 `bundle` 文件，传输给 `Bundle Server` 静态资源服务器

- 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

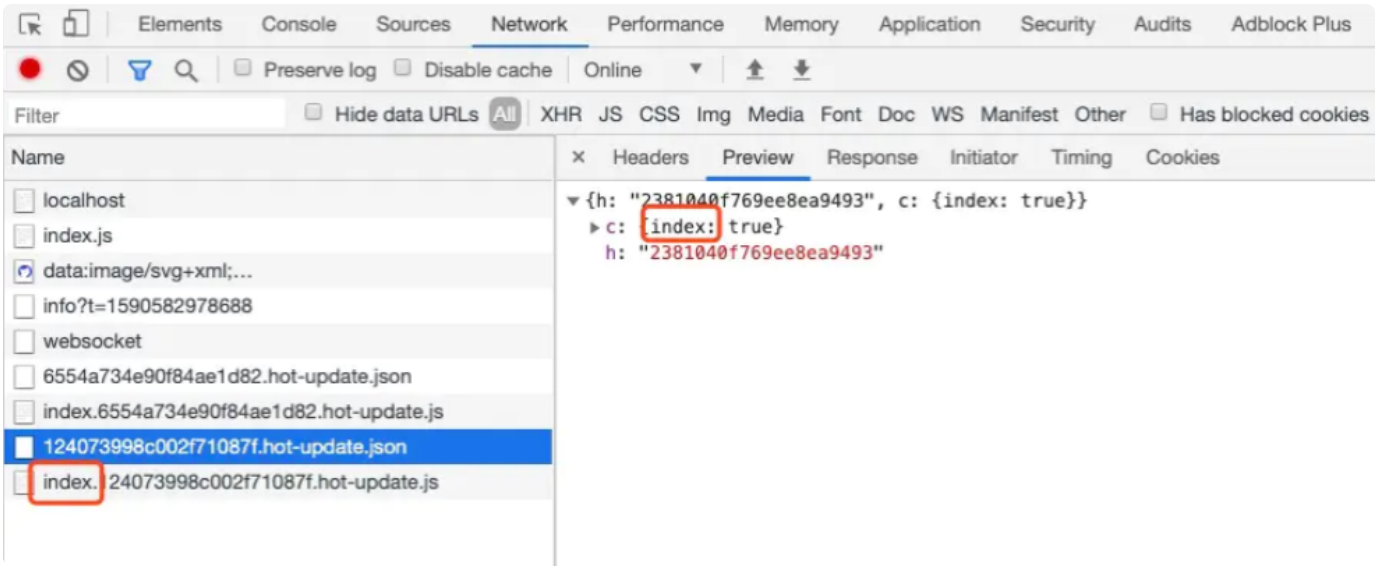
由于 `socket` 服务器在 `HMR Runtime` 和 `HMR Server` 之间建立 `websocket` 链接，当文件发生改动的时候，服务端会向浏览器推送一条消息，消息包含文件改动后生成的 `hash` 值，如下图的 `h` 属性，作为下一次热更细的标识



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

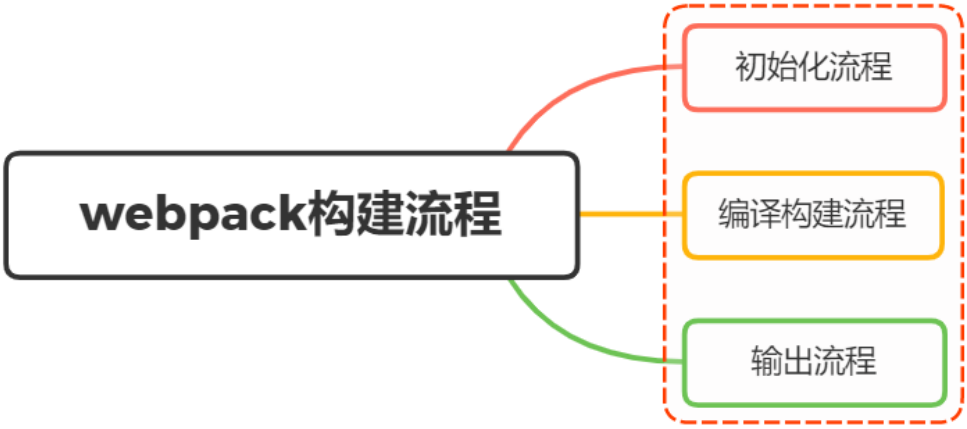
关于 `webpack` 热模块更新的总结如下：

- 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?



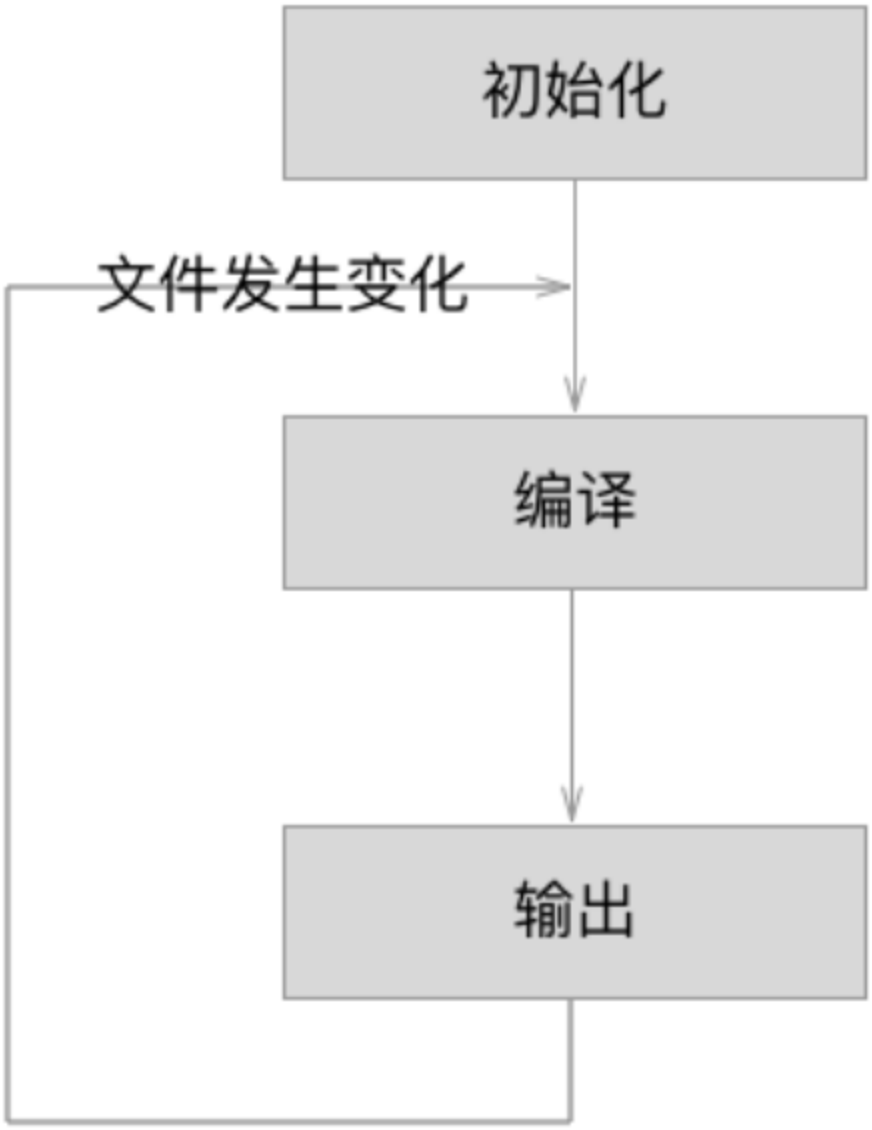
3.1. 运行流程

`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：

JavaScript | 复制代码

```
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

```JavaScript
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

```JavaScript
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：

JavaScript | 复制代码

```
_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

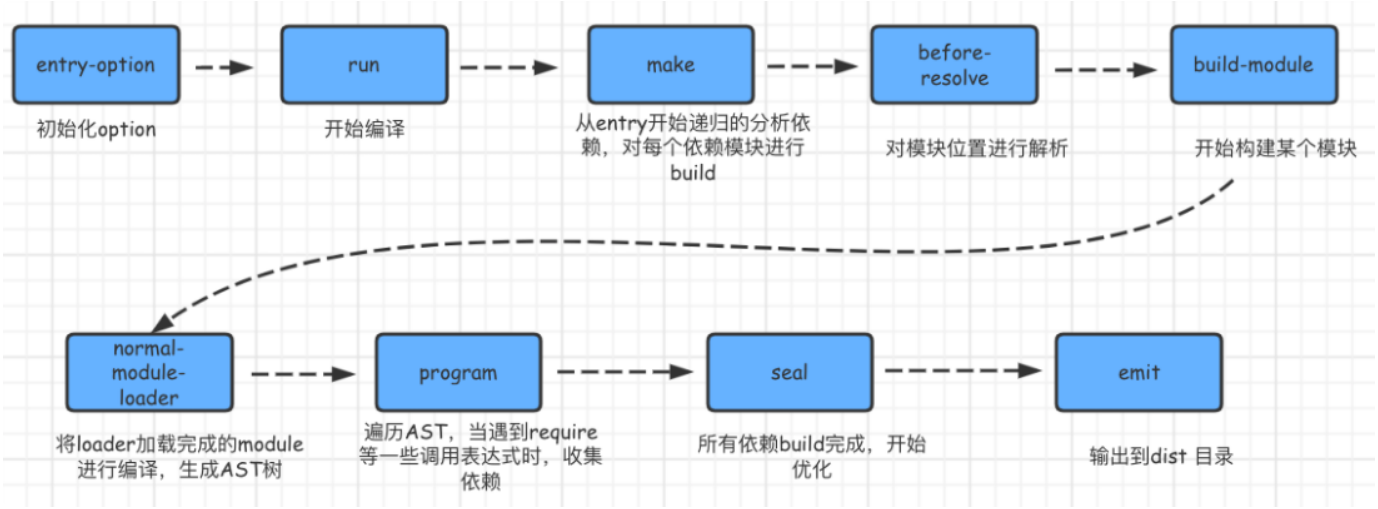
JavaScript | 复制代码

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

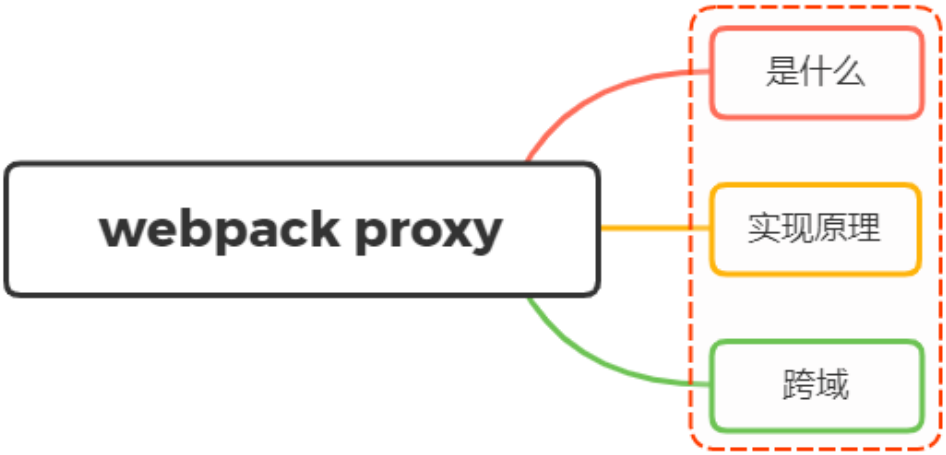
在 `Compiler` 开始生成文件前，钩子 `emit` 会被执行，这是我们修改最终文件的最后一个机会

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：

JavaScript | 复制代码

```
// ./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'
            }
        }
        // ...
    }
}
```

`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

JavaScript | 复制代码

```
const express = require('express');
const proxy = require('http-proxy-middleware');

const app = express();

app.use('/api', proxy({target: 'http://www.example.org', changeOrigin: true
}));
app.listen(3000);

// http://localhost:3000/api/foo/bar -> http://www.example.org/api/foo/bar
```

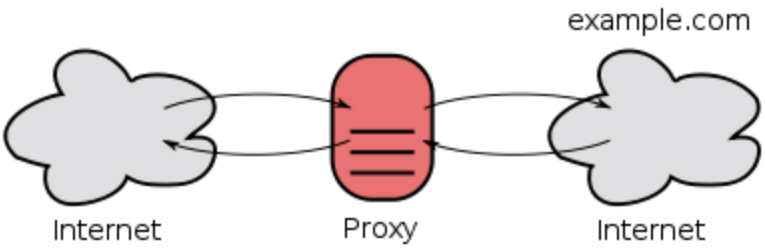
4.3. 跨域

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

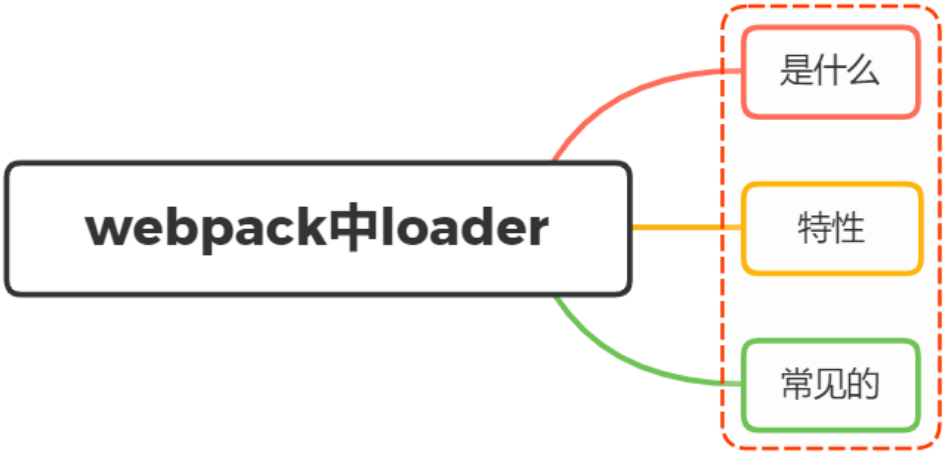
当本地发送请求的时候，代理服务器响应该请求，并将请求转发到目标服务器，目标服务器响应数据后再将数据返回给代理服务器，最终再由代理服务器将数据响应给本地



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**

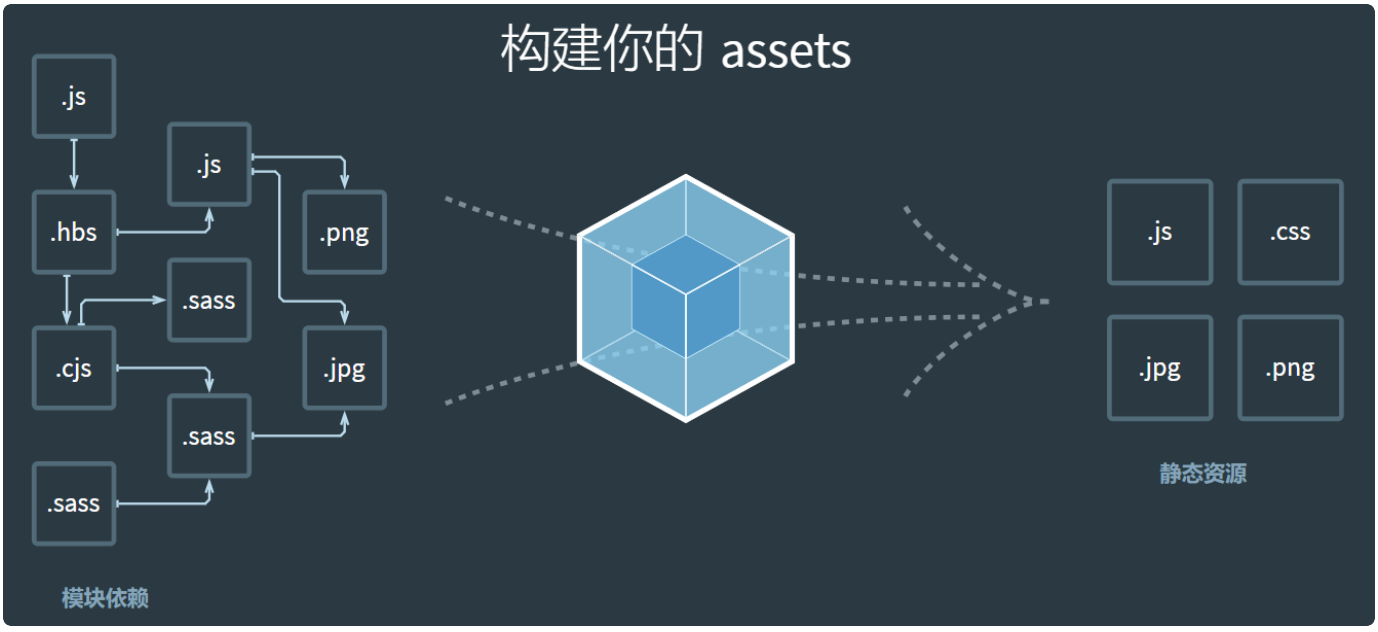
5. 说说webpack中常见的Loader？解决了什么问题？



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：

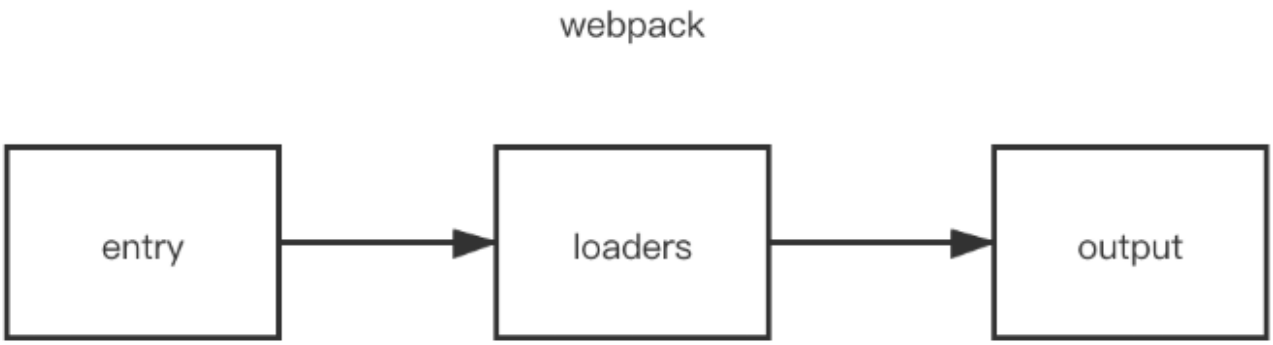


在 `webpack` 内部中，任何文件都是模块，不仅仅只是 `js` 文件

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：

JavaScript | 复制代码

```javascript
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`

Bash | 复制代码

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

```JavaScript
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

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

```JavaScript
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`

Plain Text | 复制代码

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

JavaScript | 复制代码

```
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

Bash | 复制代码

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

然后在 webpack.config.js 中进行配置

```JavaScript
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)

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

```JavaScript
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。

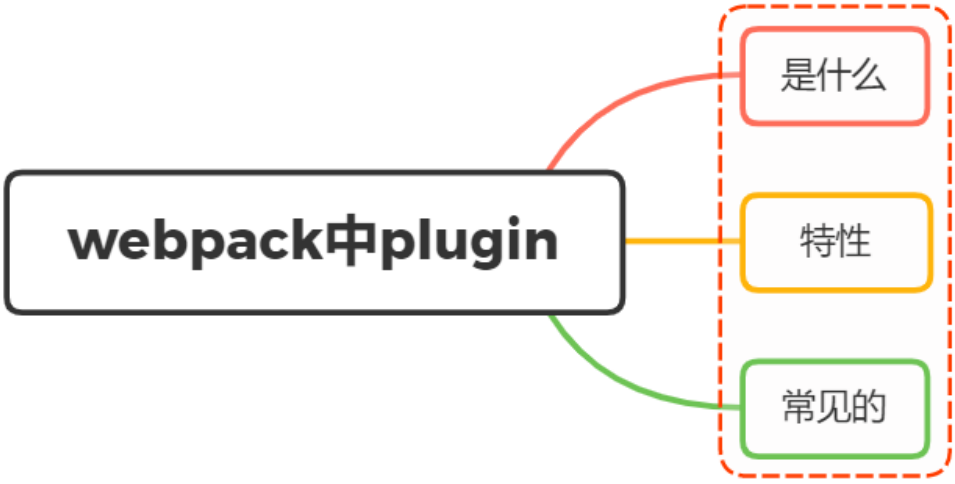
Bash | 复制代码

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

JavaScript | 复制代码

```javascript
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？

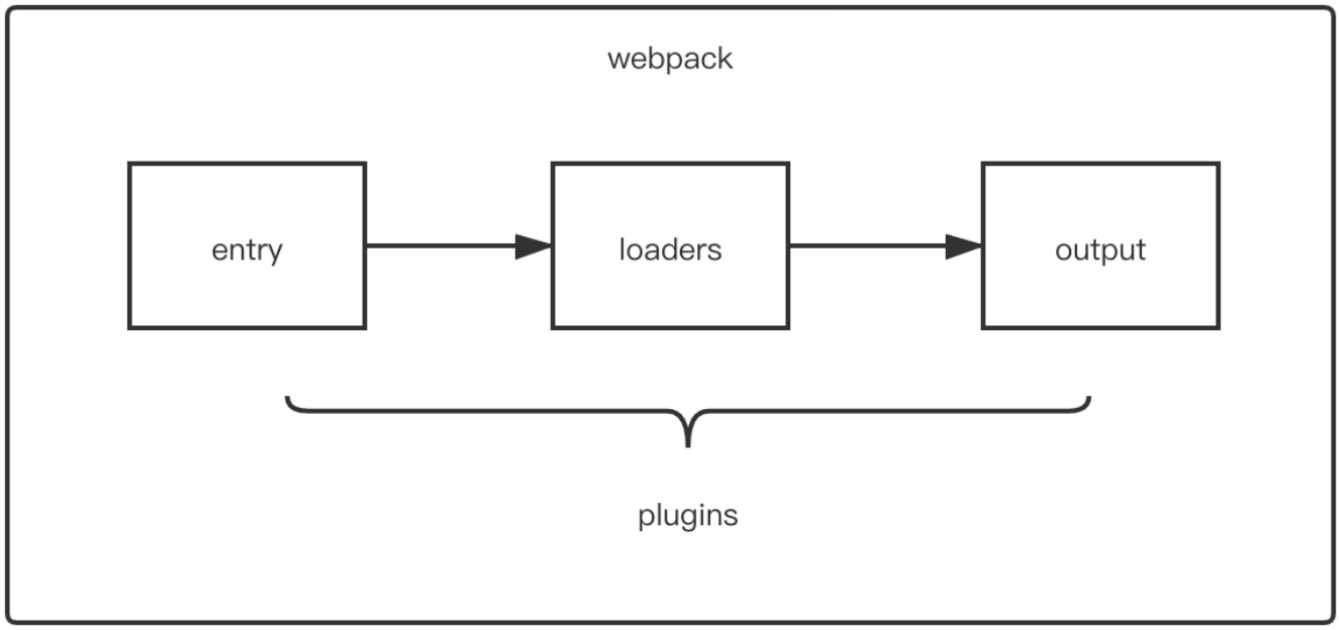


6.1. 是什么

`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