

Webpack配置和css处理

王红元
coderwhy

webpack默认打包

- 我们可以通过webpack进行打包，之后运行打包之后的代码

- 在目录下直接执行 webpack 命令

```
webpack
```

- 生成一个dist文件夹，里面存放一个main.js的文件，就是我们打包之后的文件：

- 这个文件中的代码被压缩和丑化了；
 - 我们暂时不关心他是如何做到的，后续我讲webpack实现模块化原理时会再次讲到；
 - 另外我们发现代码中依然存在ES6的语法，比如箭头函数、const等，这是因为默认情况下webpack并不清楚我们打包后的文件是否需要转成ES5之前的语法，后续我们需要通过babel来进行转换和设置；

- 我们发现是可以正常进行打包的，但是有一个问题，webpack是如何确定我们的入口的呢？

- 事实上，当我们运行webpack时，webpack会查找当前目录下的 src/index.js作为入口；
 - 所以，如果当前项目中没有存在src/index.js文件，那么会报错；

- 当然，我们也可以通过配置来指定入口和出口

```
npx webpack --entry ./src/main.js --output-path ./build
```

Webpack配置文件

- 在通常情况下，webpack需要打包的项目是非常复杂的，并且我们需要一系列的配置来满足要求，默认配置必然是不可以的。
- 我们可以在根目录下创建一个webpack.config.js文件，来作为webpack的配置文件：

```
const path = require('path');  
  
// 导出配置信息  
module.exports = {  
  entry: './src/main.js',  
  output: {  
    filename: 'bundle.js',  
    path: path.resolve(__dirname, './dist')  
  }  
}
```

继续执行webpack命令，依然可以正常打包

webpack

■ 但是如果我们的配置文件并不是webpack.config.js的名字，而是其他的名字呢？

□ 比如我们将webpack.config.js修改成了 wk.config.js；

□ 这个时候我们可以通过 --config 来指定对应的配置文件；

```
webpack --config wk.config.js
```

■ 但是每次这样执行命令来对源码进行编译，会非常繁琐，所以我们可以package.json中增加一个新的脚本：

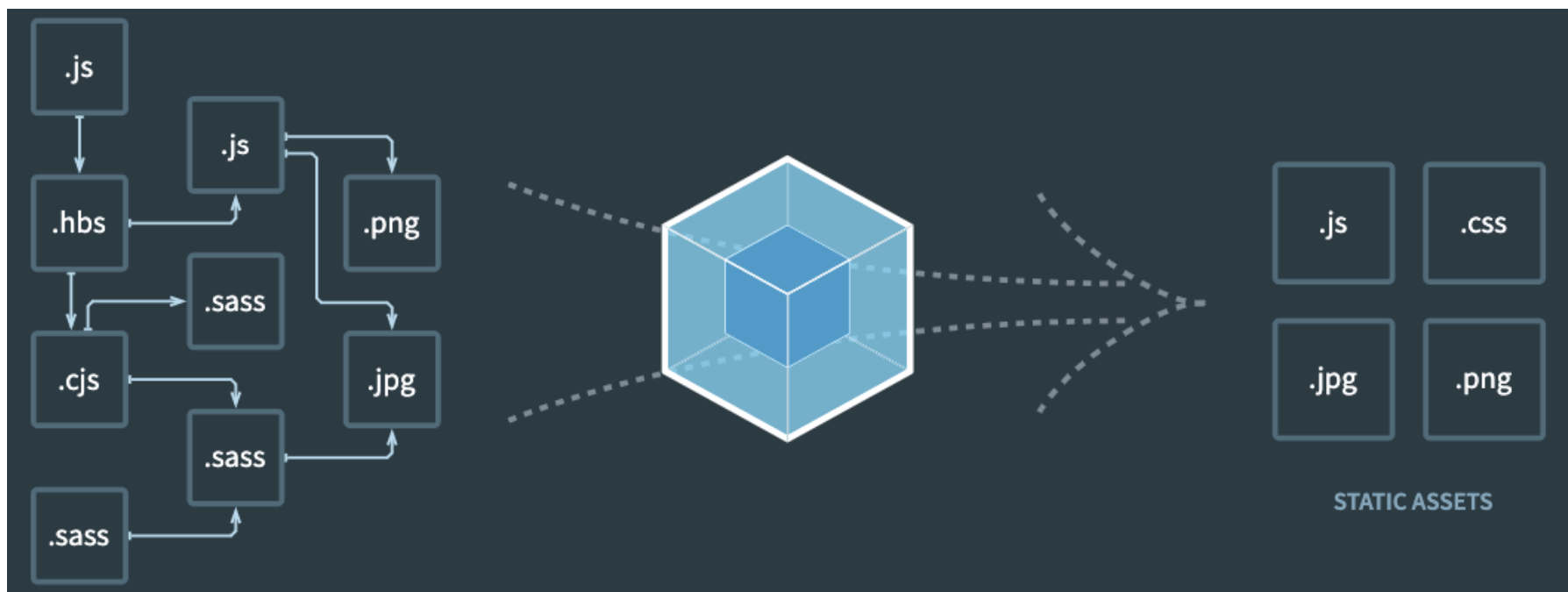
```
{
  ▶ Debug
  "scripts": {
    "build": "webpack --config wk.config.js"
  },
  "devDependencies": {
    "webpack": "^5.14.0",
    "webpack-cli": "^4.3.1"
  }
}
```

之后我们执行 `npm run build` 来打包即可。

Webpack依赖图

■ webpack到底是如何对我们的项目进行打包的呢？

- 事实上webpack在处理应用程序时，它会根据命令或者配置文件找到入口文件；
- 从入口开始，会生成一个 **依赖关系图**，这个**依赖关系图**会包含应用程序中所需的所有模块（比如.js文件、css文件、图片、字体等）；
- 然后遍历图结构，打包一个个模块（根据文件的不同使用不同的loader来解析）；



■ 我们创建一个component.js

□ 通过JavaScript创建了一个元素，并且希望给它设置一些样式；

```
import "../css/style.css";

function component() {
  const element = document.createElement('div');

  element.innerHTML = ["Hello", "Webpack"].join(" ");
  element.className = "content";

  return element;
}

document.body.appendChild(component());
```

```
.content {
  color: red;
}
```

继续编译命令npm run build

```
ERROR in ./src/css/style.css 1:0
Module parse failed: Unexpected token (1:0) 模块解析失败
You may need an appropriate loader to handle this file type, currently no
pack.js.org/concepts#loaders 你需要一个合适的loader来处理这个文件类型
> .content {
|   color: red;
| }
@ ./src/js/component.js 1:0-26
@ ./src/main.js 3:0-24
```

■ 上面的错误信息告诉我们需要一个loader来加载这个css文件，但是**loader**是什么呢？

□ loader 可以用于对**模块的源代码**进行转换；

□ 我们可以**将css文件也看成是一个模块**，我们是**通过import来加载这个模块的**；

□ 在加载这个模块时，**webpack其实并不知道如何对其进行加载**，我们必须制定对应的loader来完成这个功能；

■ 那么我们需要一个什么样的loader呢？

□ 对于加载css文件来说，我们需要一个可以读取css文件的loader；

□ 这个loader最常用的是**css-loader**；

■ css-loader的安装：

```
npm install css-loader -D
```

■ 如何使用这个loader来加载css文件呢？有三种方式：

- 内联方式；
- CLI方式（webpack5中不再使用）；
- 配置方式；

■ 内联方式：内联方式使用较少，因为不方便管理；

- 在引入的样式前加上使用的loader，并且使用!分割；

```
import "css-loader!../css/style.css";
```

■ CLI方式

- 在webpack5的文档中已经没有了`--module-bind`；
- 实际应用中也比较少使用，因为不方便管理；

- 配置方式表示的意思是在我们的webpack.config.js文件中写明配置信息：
 - module.rules中允许我们配置多个loader（因为我们也会继续使用其他的loader，来完成其他文件的加载）；
 - 这种方式可以更好的表示loader的配置，也方便后期的维护，同时也让你对各个Loader有一个全局的概览；
- module.rules的配置如下：
- rules属性对应的值是一个数组：**[Rule]**
- 数组中存放的是一个个的Rule，Rule是一个对象，对象中可以设置多个属性：
 - **test属性**：用于对 resource（资源）进行匹配的，通常会设置成正则表达式；
 - **use属性**：对应的值是一个数组：**[UseEntry]**
 - ✓ UseEntry是一个对象，可以通过对象的属性来设置一些其他属性
 - **loader**：必须有一个 loader 属性，对应的值是一个字符串；
 - **options**：可选的属性，值是一个字符串或者对象，值会被传入到loader中；
 - **query**：目前已经使用options来替代；
 - ✓ **传递字符串（如：use: ['style-loader']）是 loader 属性的简写方式（如：use: [{ loader: 'style-loader' }]）；**
 - **loader属性**：Rule.use: [{ loader }] 的简写。

Loader的配置代码

```
// 导出配置信息
module.exports = {
  mode: "development",
  entry: "./src/main.js",
  output: {
    filename: "bundle.js",
    path: path.resolve(__dirname, "./dist")
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        // loader: "css-loader" // 写法一
        // use: ["css-loader"] // 写法二
        // 写法三
        use: [
          { loader: "css-loader" }
        ]
      }
    ]
  }
}
```

- 我们已经可以通过css-loader来加载css文件了

- 但是你会发现这个css在我们的代码中并没有生效（页面没有效果）。

- 这是为什么呢？

- 因为css-loader只是负责将.css文件进行解析，并不会将解析之后的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；

- 安装style-loader：

```
npm install style-loader -D
```

■ 那么我们应该如何使用style-loader：

- 在配置文件中，添加style-loader；
- 注意：因为loader的执行顺序是从右向左（或者说从下到上，或者说从后到前的），所以我们需要将style-loader写到css-loader的前面；

```
use: [
  // 注意: style-loader在css-loader的前面
  { loader: "style-loader" },
  { loader: "css-loader" }
]
```

■ 重新执行编译npm run build，可以发现打包后的css已经生效了：

- 当前目前我们的css是通过页内样式的方式添加进来的；
- 后续我们也会讲如何将css抽取到单独的文件中，并且进行压缩等操作；

如何处理less文件？

- 在我们开发中，我们可能会使用less、sass、stylus的预处理器来编写css样式，效率会更高。
- 那么，如何可以让我们的环境支持这些预处理器呢？
 - 首先我们需要确定，less、sass等编写的css需要通过工具转换成普通的css；
- 比如我们编写如下的less样式：

```
@fontSize: 30px;
@fontWeight: 700;

.content {
  font-size: @fontSize;
  font-weight: @fontWeight;
}
```

- 我们可以使用less工具来完成它的编译转换：

```
npm install less -D
```

- 执行如下命令：

```
npx less ./src/css/title.less > title.css
```

■ 但是在项目中我们会编写大量的css，它们如何可以自动转换呢？

□ 这个时候我们就可以使用less-loader，来自动使用less工具转换less到css；

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

■ 配置webpack.config.js

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

执行npm run build
less就可以自动转换成css，并且页面也会生效了

■ 我们来思考一个问题：开发中，浏览器的兼容性问题，我们应该如何去解决和处理？

□ 当然这个问题很笼统，这里我说的兼容性问题不是指屏幕大小的变化适配；

□ 我这里指的兼容性是针对不同的浏览器支持的特性：比如css特性、js语法，之间的兼容性；

■ 我们知道市面上有大量的浏览器：

□ 有Chrome、Safari、IE、Edge、Chrome for Android、UC Browser、QQ Browser等等；

□ 它们的市场占有率是多少？我们要不要兼容它们呢？

■ 其实在很多的脚手架配置中，都能看到类似于这样的配置信息：

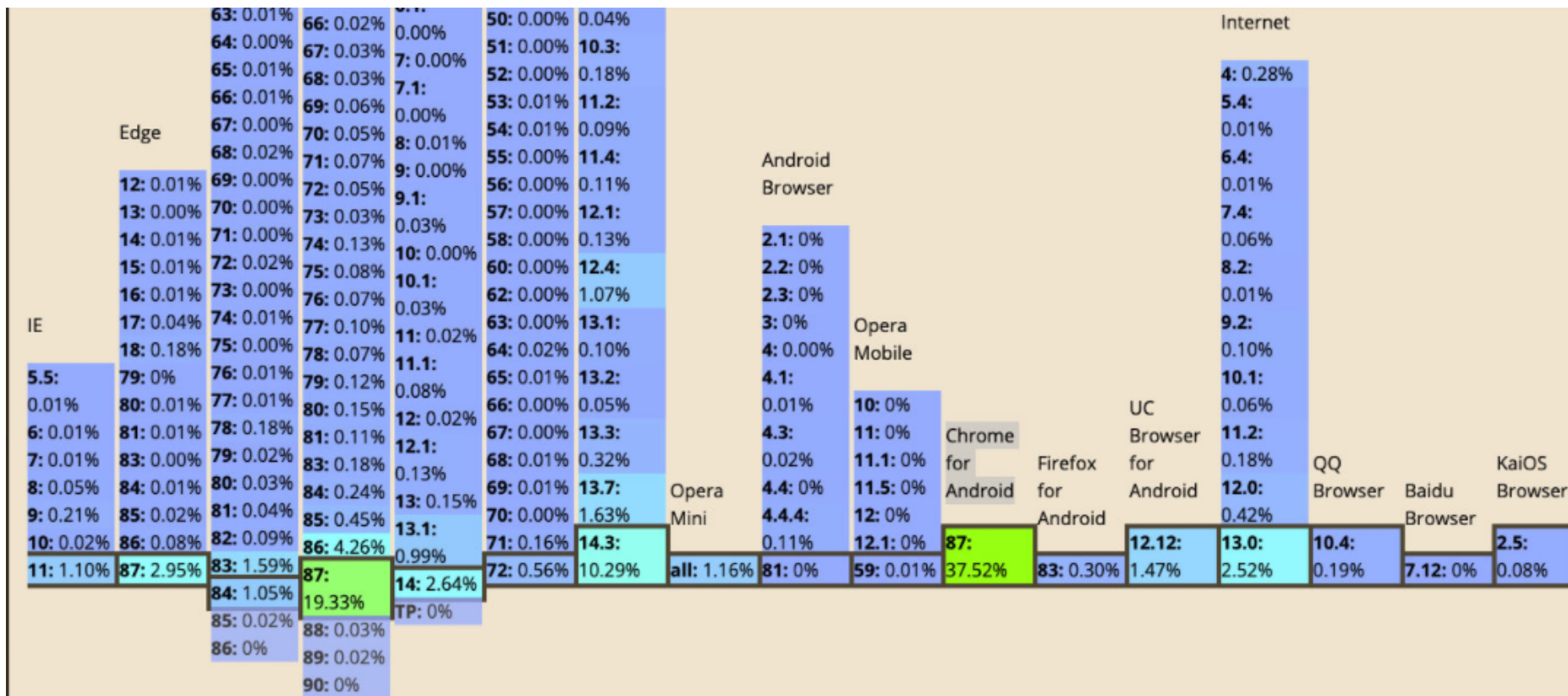
□ 这里的百分之一，就是指市场占有率

```
> 1%  
last 2 versions  
not dead
```


■ 但是在哪里可以查询到浏览器的市场占有率呢？

□ 这个最好用的网站，也是我们工具通常会查询的一个网站就是caniuse；

□ <https://caniuse.com/usage-table>



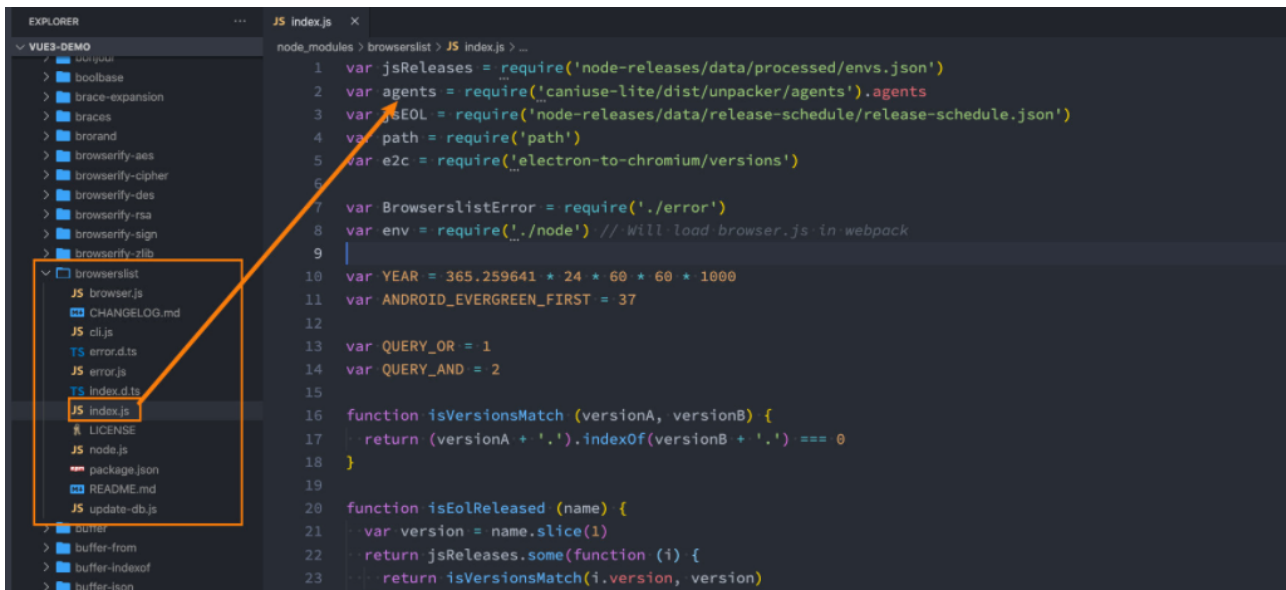
- 但是有一个问题，我们如何可以在css兼容性和js兼容性下共享我们配置的兼容性条件呢？
 - 就是当我们设置了一个条件： $> 1\%$ ；
 - 我们表达的意思是css要兼容市场占有率大于1%的浏览器，js也要兼容市场占有率大于1%的浏览器；
 - 如果我们是通过工具来达到这种兼容性的，比如后面我们会讲到的postcss-preset-env、babel、autoprefixer等
- 如何可以让他们共享我们的配置呢？
 - 这个问题的答案就是Browserslist；
- Browserslist是什么？Browserslist是一个在不同的前端工具之间，共享目标浏览器和Node.js版本的配置：
 - [Autoprefixer](#)
 - [Babel](#)
 - [postcss-preset-env](#)
 - [eslint-plugin-compat](#)
 - [stylelint-no-unsupported-browser-features](#)
 - [postcss-normalize](#)
 - [obsolete-webpack-plugin](#)

- 我们可以编写类似于这样的配置：

```
> 1%  
last 2 versions  
not dead
```

- 那么之后，这些工具会根据我们的配置来获取相关的浏览器信息，以方便决定是否需要进行兼容性的支持：

- 条件查询使用的是caniuse-lite的工具，这个工具的数据来自于caniuse的网站上；



Browserslist编写规则一：

- 那么在开发中，我们可以编写的条件都有哪些呢？（加粗部分是最常用的）
- **defaults**：Browserslist的默认浏览器（> 0.5%, last 2 versions, Firefox ESR, not dead）。
- **5%**：通过全局使用情况统计信息选择的浏览器版本。>=，<和<=工作过。
 - 5% in US：使用美国使用情况统计信息。它接受两个字母的国家/地区代码。
 - > 5% in alt-AS：使用亚洲地区使用情况统计信息。有关所有区域代码的列表，请参见caniuse-lite/data/regions
 - > 5% in my stats：使用自定义用法数据。
 - > 5% in browserslist-config-mycompany stats：使用 来自的自定义使用情况数据browserslist-config-mycompany/browserslist-stats.json。
 - cover 99.5%：提供覆盖率的最受欢迎的浏览器。
 - cover 99.5% in US：与上述相同，但国家/地区代码由两个字母组成。
 - cover 99.5% in my stats：使用自定义用法数据。
- **dead**：24个月内没有官方支持或更新的浏览器。现在是IE 10，IE_Mob 11，BlackBerry 10，BlackBerry 7，Samsung 4和OperaMobile 12.1。
- **last 2 versions**：每个浏览器的最后2个版本。
 - last 2 Chrome versions：最近2个版本的Chrome浏览器。
 - last 2 major versions或last 2 iOS major versions：最近2个主要版本的所有次要/补丁版本。

Browserslist编写规则二：

- node 10和node 10.4：选择最新的Node.js10.x.x 或10.4.x版本。
 - ❑ current node：Browserslist现在使用的Node.js版本。
 - ❑ maintained node versions：所有Node.js版本，仍由 Node.js Foundation维护。
- iOS 7：直接使用iOS浏览器版本7。
 - ❑ Firefox > 20：Firefox的版本高于20 >=，<并且<=也可以使用。它也可以与Node.js一起使用。
 - ❑ ie 6-8：选择一个包含范围的版本。
 - ❑ Firefox ESR：最新的[Firefox ESR]版本。
 - ❑ PhantomJS 2.1和PhantomJS 1.9：选择类似于PhantomJS运行时的Safari版本。
- extends browserslist-config-mycompany：从browserslist-config-mycompanynpm包中查询。
- supports es6-module：支持特定功能的浏览器。es6-module这是“我可以使用的”页面feat的URL上的参数。有关所有可用功能的列表，请参见 caniuse-lite/data/features
- browserslist config：在Browserslist配置中定义的浏览器。在差异服务中很有用，可用于修改用户的配置，例如 browserslist config and supports es6-module。
- since 2015或last 2 years：自2015年以来发布的所有版本（since 2015-03以及since 2015-03-10）。
- unreleased versions或unreleased Chrome versions：Alpha和Beta版本。
- not ie <= 8：排除先前查询选择的浏览器。

命令行使用browserslist

- 我们可以直接通过命令来查询某些条件所匹配到的浏览器：

```
npx browserslist ">1%, last 2 version, not dead"
```

```
coderwhy@why 02_webpack打包css资源 % npx browserslist ">1%, last 2 version, not dead"
and_chr 87
and_ff 83
and_qq 10.4
and_uc 12.12
android 81
baidu 7.12
chrome 87
chrome 86
edge 87
edge 86
firefox 84
firefox 83
ie 11
ios_saf 14.0-14.3
```

■ 我们如何可以配置browserslist呢？两种方案：

□ 方案一：在package.json中配置；

□ 方案二：单独的一个配置文件.browserslistrc文件；

■ 方案一：package.json配置：

```
"browserslist": [  
  "last 2 version",  
  "not dead",  
  "> 0.2%"  
]
```

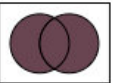

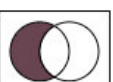
■ 方案二：.browserslistrc文件

```
webpack打包css资源 > .browserslistrc  
1 > 0.5%  
2 last 2 version  
3 not dead
```

- 如果没有配置，那么也会有一个默认配置：

```
// Default browsers query  
browserslist.defaults = [  
  '> 0.5%',  
  'last 2 versions',  
  'Firefox ESR',  
  'not dead'  
]
```

- 我们编写了多个条件之后，多个条件之间是什么关系呢？

| Query combiner type | Illustration | Example |
|---------------------------------------|---|--|
| or / , combiner (union) |  | <pre>> .5% or last 2 versions > .5%, last 2 versions</pre> |
| and combiner (intersection) |  | <pre>> .5% and last 2 versions</pre> |
| not combiner (relative complement) |  | <p>All those three are equivalent to the first one</p> <pre>> .5% and not last 2 versions > .5% or not last 2 versions > .5%, not last 2 versions</pre> |

■ 什么是PostCSS呢？

- PostCSS是一个通过JavaScript来转换样式的**工具**；
- 这个工具可以帮助我们进行一些CSS的转换和适配，比如自动添加浏览器前缀、css样式的重置；
- 但是实现这些工具，我们需要借助于PostCSS对应的插件；

■ 如何使用PostCSS呢？主要就是两个步骤：

- 第一步：查找PostCSS在构建工具中的扩展，比如webpack中的postcss-loader；
- 第二步：选择可以添加你需要的PostCSS相关的插件；

■ 当然，我们能不能也直接在终端使用PostCSS呢？

□ 也是可以的，但是我们需要单独安装一个工具postcss-cli；

■ 我们可以安装一下它们：postcss、postcss-cli

```
npm install postcss postcss-cli -D
```

■ 我们编写一个需要添加前缀的css：

□ <https://autoprefixer.github.io/>

□ 我们可以在上面的网站中查询一些添加css属性的样式；

```
:fullscreen {  
  color: red;  
}  
  
.content {  
  user-select: none;  
}
```

插件autoprefixer

- 因为我们需要添加前缀，所以要安装autoprefixer：

```
npm install autoprefixer -D
```

- 直接使用使用postcss工具，并且制定使用autoprefixer

```
npx postcss --use autoprefixer -o end.css ./src/css/style.css
```

- 转化之后的css样式如下：

```
:-ms-fullscreen {  
}  
  
:fullscreen {  
}  
  
.content {  
  -webkit-user-select: none;  
  -moz-user-select: none;  
  -ms-user-select: none;  
  user-select: none;  
}
```

■ 真实开发中我们必然不会直接使用命令行工具来对css进行处理，而是可以借助于构建工具：

□ 在webpack中使用postcss就是使用postcss-loader来处理的；

■ 我们来安装postcss-loader：

```
npm install postcss-loader -D
```

■ 我们修改加载css的loader：（配置文件已经过多，给出一部分了）

□ 注意：因为postcss需要有对应的插件才会起效果，所以我们需要配置它的plugin；

```
{
  loader: "postcss-loader",
  options: {
    postcssOptions: {
      plugins: [
        require('autoprefixer')
      ]
    }
  }
}
```

单独的postcss配置文件

■ 当然，我们也可以将这些配置信息放到一个单独的文件中进行管理：

□ 在根目录下创建postcss.config.js

```
module.exports = {  
  ...  
  plugins: [  
    require("autoprefixer")  
  ],  
}
```

postcss-preset-env

- 事实上，在配置postcss-loader时，我们配置插件并不需要使用autoprefixer。
- 我们可以使用另外一个插件：postcss-preset-env
 - postcss-preset-env也是一个postcss的插件；
 - 它可以帮助我们将一些现代的CSS特性，转成大多数浏览器认识的CSS，并且会根据目标浏览器或者运行时环境添加所需的polyfill；
 - 也包括会自动帮助我们添加autoprefixer（所以相当于已经内置了autoprefixer）；
- 首先，我们需要安装postcss-preset-env：

```
npm install postcss-preset-env -D
```

- 之后，我们直接修改掉之前的autoprefixer即可：

```
plugins: [  
  require("postcss-preset-env"),  
],
```

注意：我们在使用某些postcss插件时，也可以直接传入字符串

```
module.exports = {  
  plugins: [  
    "postcss-preset-env",  
  ],  
}
```

■ 我们举一个例子：

- 我们这里在使用十六进制的颜色时设置了8位；
- 但是某些浏览器可能不认识这种语法，我们最好可以转成RGBA的形式；
- 但是autoprefixer是不会帮助我们转换的；
- 而postcss-preset-env就可以完成这样的功能；

```
.content {  
  color: □ #12345678;  
}
```