# Babel

## 理解babel

**一、安装**

npm install --save-dev @babel/core @babel/cli @babel/preset-env

npm install --save @babel/polyfill

**二、核心库**

Babel的核心库功能包含在@babel/core模块中

**三、CLI命令行工具**

@babel/cli是一个能够从终端（命令行）使用的工具。

./node\_modules/.bin/babel src --out-dir lib

这将解析src目录下的所有JavaScript文件，

1. 插件和预设

代码转换功能能以插件的形式出现，插件是小型的JavaScript程序，用于指导Babel如何对代码进行转换。你甚至可以编写自己的插件将你所需要的任何代码转换功能应用到你的代码上。

例如将ES6语法转换为ES5语法，我们可以使用诸如@babel/plugin-transform-arrow-functions之类的官方插件：

npm install --save-dev @babel/plugin-transform-arrow-functions ./node\_modules/.bin/babel src --out-dir lib --plugins=@babel/plugin-transform-arrow-functions

现在，我们代码中的所有箭头函数（arrow functions）都被转换为ES5兼容的函数表达式了

但是我们的代码中仍然残留了其他的ES6的特性，我们希望对它们也进行转换。

我们不需要一个接一个的添加所有需要的插件，我们可以使用一个“preset”（即一组预先设定的插件）。

## Babel处理三步骤：parse解析、transform转换、generate生成

<https://zhuanlan.zhihu.com/p/158211262>

* 解析过程基于@babel/parser做词法分析、语法分析。

词法分析将源程序解析成令牌流；语法分析将立牌流转换成AST。

* 转化过程基于@babel/traverse遍历AST，对节点进行添加、更新及移除等操作。此时插件介入工作。
* 生成过程基于@babel/generator深度遍历AST，将其转换成字符串形式的代码，并生成source maps源码映射。

# Webpack

## 四个核心概念（entry、output、loader、plugins）

入口（entry）：指示webpack应该使用哪个模块，来作为构建其内部依赖图的开始；

输出（output）：在哪里输出它所创建的bundles；

Loader：让webpack能够去处理那些非JavaScript文件；

插件（plugins）：用于执行范围更广的任务。

## 常用loader

**css-loader和style-loader**

css-loader用来解释@import和url();

style-loader用来将css-loader生成的样式表通过<style>标签，插入到页面中去。

**sass-loader、 less-loader和postcss-loader**

**postcss-loader**

主要功能：

* 把css解析成JS可以操作的抽象语法树AST
* 调用插件来处理AST并得到结果

配合autoprefixer插件，打包后的css自动加上-moz、-ms、-webkit等浏览器私有前缀

**babel-**loader

配合babel将高版本的ES6甚至ES7转化为ES5

**file-loader和url-loader**

都是用来处理图片、字体图标等文件

分两种情况：

1. 当文件大小小于limit参数，url-loader将文件转化为base-64编码，用于减少http请求；
2. 当文件大小大于limit参数，调用file-loader进行处理

vue-loader

## webpack构建过程

webpack的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和shell语句中读取与合并参数，得到最终的参数。
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找到所有的入口文件。
4. 编译模块：从入口文件出发，调用该所有配置的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，得到了每个模块翻译后的最终内容以及它们之间的依赖关系。
6. 输入资源：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转化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：再确定好输出内容以后，根据配置确认输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，webpack会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用webpack提供的API改变webpack的运行结果。

## 热更新HMR原理

1. 在启动webpack的时候，sockjs在服务器和浏览器端建立一个WebSocket长连接。
2. Webpack-dev-server调用webpack api监听compiler的done事件。当compile完成后，webpack-dev-server将编译打包的新模块hash值通过WebSocket长连接发送给浏览器端。
3. 浏览器端webpack-client接收到type为hash的数据后将hash值暂存起来。
4. Webpack-dev-server通过WebSocket长连接发送error、ok、warning类型的信息给浏览器。
5. 浏览器端webpack-dev-client接收到type为ok或warning的消息后，根据hot配置决定是刷新浏览器还是对浏览器进行热更新。

如果是热更新，基于node events的emit和on方法，将步骤3中获取到的hash提交给webpack，然后将控制权交给webpack客户端代码。

1. Webpack客户端client先判断模块是否有更新，即判断服务器传过来的最新hash值和客户端上一次缓存的hash是否一致。
2. 不一致的话，代表模块已更新，则webpack客户端client通过AJAX向服务器请求最新的文件，如果有将返回更新的文件hash列表。
3. webpack客户端client通过jsonp请求最新的代码块。
4. 新的代码块替换旧的模块代码。
5. 替换后，我们的业务代码并不知道代码已经发生变化，所以我们需要在业务代码中调用HMR的accept方法，添加模块更新后的处理函数

Module.hot.accept

我们手动在业务代码中添加容易犯错，也很麻烦，目前已经有很多的loader可以帮忙处理。

## 模块加载

* webpack定义自己的模块加载方法\_\_webpack\_require\_\_
* \_\_webpack\_require\_\_方法接受一个moduleId作为参数
* moduleId是一个文件路径的字符串
* webpack定义一个用来缓存模块的对象installedModules
* 判断缓存对象是否已缓存指定moduleId的模块
* 有缓存直接从对象中获取属性值的exports方法
* 没有缓存则定义一个对象module，该对象有以下属性：

1. exports对象类型
2. parents
3. children 数组类型

* 将模块对象缓存到installedModules对象中
* 执行moduleId对应的文件
* 返回module.exports

总结 ：每个文件只会执行一遍；

## 模块循环加载

a.js 加载 b.js , b.js 加载 a.js

1、首先加载a.js并执行, a.js，生成moduleA，moduleA的exports属性是一个对象；该对象中包含的是a导出的属性和方法，并放入缓存对象中；

2、执行a.js文件中import b from ‘./b.js’ 语句时，加载b.js并执行，生成moduleB，并放入缓存对象中；

3、执行b.js中import a from ‘./a.js’语句时，直接从缓存对象中返回moduleA.exports对应的对象，不会再进入a.js文件中执行import b from ‘/b.js’语句。所以不会引起死循环。

## devtool