Webpack-Day4

Babel处理ES6

官方网站：https://babeljs.io/

中文网站：https://www.babeljs.cn/

Babel是JavaScript编译器，能将ES6代码转换成ES5代码，让我们开发过程中放心使用JS新特性而不用担心兼容性问题。并且还可以通过插件机制根据需求灵活的扩展。

Babel在执行编译的过程中，会从项目根目录下的`.babelrc`JSON文件中读取配置。没有该文件会从loader的options地方读取配置。

测试代码

```
//index.js
const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

Env 是面向未来的一种规范 会成长的 已经发布了的

不需要我们去关心当前需要转换的语法，处在什么阶段

垫片！polyfill -> 我们自己定义了一套ES6+新特性的语法库，掉版本浏览器引入了这个库，就会解决不知道如何处理es6+新特性的问题

按需加载

tc39精简了提案流程

es2020

前端技术委员会 提出一个新特性，支持laohan函数

Stage0 只是一个想法

stage1 这个想法不错，值得跟进

Stage2 尝试制定这个特性的规范，

Stage3 进入到候选名单，不会太大的改变了，对外界发布一些信息，我们有可能会在下一个版本支持laohan函数

Stage4 完成阶段 发布在下一代版本 是确定 不可修改的

babel 来进行语法转换的时候，需要通过预设插件机制文成

env 官方推荐

安装

```
npm i babel-loader @babel/core @babel/preset-env -D
```

`1.babel-loader是webpack 与 babel的通信桥梁，不会做把es6转成es5的工作，这部分工作需要用到@babel/preset-env来做`

`2.@babel/preset-env里包含了es，6，7，8转es5的转换规则`

Ecma 5 6 7 8... 草案（评审通过的，还有未通过的）

面向未来的

env是babel7之后推行的预设插件

env{

ecma 5

ecma 6

ecma 7

ecma 8

。。。

}

Webpack.config.js

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  use: {
    loader: "babel-loader",
    options: {
      presets: ["@babel/preset-env"]
    }
  }
}


  "browserslist": [
    "last 2 version",
    "> 1%",
    "not ie < 11",
    "cover 99.5%",
    "dead" ”两年不维护“
  ]

  "browserslist": [
    "last 2 version",
    ">1 %",
    "not ie < 11",
    "cover 99.5%",
    "ie"
    "chrome",
    "Baidu",
```

```
  ]
```

通过上面的几步 还不够，默认的Babel只支持let等一些基础的特性转换，Promise等一些还有转换过来，这时候需要借助@babel/polyfill，把es的新特性都装进来，来弥补低版本浏览器中缺失的特性

@babel/polyfill

以全局变量的方式注入进来的。windows.Promise，它会造成全局对象的污染

```
npm install --save @babel/polyfill
```

```
//index.js 顶部
import "@babel/polyfill";
```

按需加载，减少冗余

会发现打包的体积大了很多，这是因为polyfill默认会把所有特性注入进来，假如我想我用到的es6+，才会注入，没用到的不注入，从而减少打包的体积，可不可以呢

当然可以

修改Webpack.config.js

```
options: {
          presets: [
            [
              "@babel/preset-env",
              {
                targets: {
                  edge: "17",
                  firefox: "60",
                  chrome: "67",
                  safari: "11.1"
                },
                corejs: 2,//新版本需要指定核心库版本
                useBuiltIns: "entry"//按需注入
              }
            ]
          ]
```

```
        }
```

`useBuiltIns`选项是`babel 7`的新功能，这个选项告诉`babel`如何配置`@babel/polyfill`。 它有三个参数可以使用： ①entry: 需要在`webpack`的入口文件里`import "@babel/polyfill"`一次。`babel`会根据你的使用情况导入垫片，没有使用的功能不会被导入相应的垫片。 ②usage: 不需要`import`，全自动检测，但是要安装`@babel/polyfill`。（试验阶段） ③false: 如果你`import "@babel/polyfill"`，它不会排除掉没有使用的垫片，程序体积会庞大。(不推荐)

请注意： usage 的行为类似 babel-transform-runtime，不会造成全局污染，因此也会不会对类似 Array.prototype.includes() 进行 polyfill。

扩展：

babelrc文件：

新建.babelrc文件，把options部分移入到该文件中，就可以了

```
//.babelrc

{
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        corejs: 2, //新版本需要指定核心库版本
        useBuiltIns: "usage" //按需注入
      }
    ]
  ]
}


//webpack.config.js

{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader"
}
```

暗号：做人嘛，最重要的是开心

作业：实现text-webpack-plugin 自定义插件

要求：提交代码截图 ，在emit阶段，往资源列表里插入一个新的txt文档，文档的内容和体积不限

配置React打包环境

安装

```
npm install react react-dom --save
```

解析插件

转换插件

编写react代码：

```
//index.js
import React, { Component } from "react";
import ReactDom from "react-dom";

class App extends Component {
  render() {
    return <div>hello world</div>;
  }
}

ReactDom.render(<App />, document.getElementById("app"));
```

安装babel与react转换的插件：

```
npm install --save-dev @babel/preset-react
```

在babelrc文件里添加：

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
```

```
            "edge": "17",
            "firefox": "60",
            "chrome": "67",
            "safari": "11.1",
            "Android":"6.0"
          },
          "useBuiltIns": "usage", //按需注入
        }
      ],
      "@babel/preset-react"
    ]
}
```

暗号：做人嘛，最重要的是开心

如何自己编写一个Plugin

webpack 在编译代码过程中，会触发一系列 Tapable 钩子事件，插件所做的，就是找到相应的钩子，往上面挂上自己的任务，也就是注册事件，这样，当 webpack 构建的时候，插件注册的事件就会随着钩子的触发而执行了。

Plugin: 开始打包，在某个时刻，帮助我们处理一些什么事情的机制

plugin要比loader稍微复杂一些，在webpack的源码中，用plugin的机制还是占有非常大的场景，可以说plugin是webpack的灵魂

设计模式

事件驱动

发布订阅

plugin是一个类，里面包含一个apply函数，接受一个参数，compiler

案例：

- 创建copyright-webpack-plugin.js

```
class CopyrightWebpackPlugin {
  constructor() {
  }

  //compiler：webpack实例
  apply(compiler) {

  }
}
module.exports = CopyrightWebpackPlugin;
```

- 配置文件里使用

```
const CopyrightWebpackPlugin = require("./plugin/copyright-webpack-plugin");

plugins: [new CopyrightWebpackPlugin()]
```

- 如何传递参数

```
//webpack配置文件
plugins: [
    new CopyrightWebpackPlugin({
      name: "开课吧"
    })
  ]

//copyright-webpack-plugin.js
class CopyrightWebpackPlugin {
  constructor(options) {
    //接受参数
    console.log(options);
  }

  apply(compiler) {}
}
module.exports = CopyrightWebpackPlugin;
```

- 配置plugin在什么时刻进行

```
class CopyrightWebpackPlugin {
```

```
  constructor(options) {
    // console.log(options);
  }

  apply(compiler) {
    //hooks.emit 定义在某个时刻
    compiler.hooks.emit.tapAsync(
      "CopyrightWebpackPlugin",
      (compilation, cb) => {
        compilation.assets["copyright.txt"] = {
          source: function() {
            return "hello copy";
          },
          size: function() {
            return 20;
          }
        };
        cb();
      }
    );

    //同步的写法
    //compiler.hooks.compile.tap("CopyrightWebpackPlugin", compilation => {
    //  console.log("开始了");
    //});
  }
}
module.exports = CopyrightWebpackPlugin;
```