Babel的深入解析

王红元

coderwhy

实力IT教育

为什么需要babel？

- 事实上，在开发中我们很少直接去接触babel，但是babel对于前端开发来说，目前是不可缺少的一部分：
  - 开发中，我们想要使用ES6+的语法，想要使用TypeScript，开发React项目，它们都是离不开Babel的；
  - 所以，学习Babel对于我们理解代码从编写到线上的转变过程直观重要；
  - 了解真相，你才能获得真知的自由！
- 那么，Babel到底是什么呢？
  - Babel是一个工具链，主要用于旧浏览器或者缓解中将ECMAScript 2015+代码转换为向后兼容版本的JavaScript；
  - 包括：语法转换、源代码转换、Polyfill实现目标缓解缺少的功能等；

```
[1, 2, 3].map((n) => n + 1);

[1, 2, 3].map(function(n) {
  return n + 1;
});
```

Babel命令行使用

- babel本身可以作为一个独立的工具（和postcss一样），不和webpack等构建工具配置来单独使用。
- 如果我们希望在命令行尝试使用babel，需要安装如下库：
  - @babel/core：babel的核心代码，必须安装；
  - @babel/cli：可以让我们在命令行使用babel；

```
npm install @babel/cli @babel/core
```

- 使用babel来处理我们的源代码：
  - src：是源文件的目录；
  - --out-dir：指定要输出的文件夹dist；

```
npx babel src --out-dir dist
```

插件的使用

- 比如我们需要转换箭头函数，那么我们就可以使用箭头函数转换相关的插件：

```
npm install @babel/plugin-transform-arrow-functions -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-arrow-functions
```

- 查看转换后的结果：我们会发现 const 并没有转成 var
  - 这是因为 plugin-transform-arrow-functions，并没有提供这样的功能；
  - 我们需要使用 plugin-transform-block-scoping 来完成这样的功能；

```
npm install @babel/plugin-transform-block-scoping -D
```

```
npx babel src --out-dir dist --plugins=@babel/plugin-transform-block-scoping
                                      ,@babel/plugin-transform-arrow-functions
```

Babel的预设preset

- 但是如果要转换的内容过多，一个个设置是比较麻烦的，我们可以使用预设（preset）：
  - 后面我们再具体来讲预设代表的含义；
- 安装@babel/preset-env预设：

```
npm install @babel/preset-env -D
```

- 执行如下命令：

```
npx babel src --out-dir dist --presets=@babel/preset-env
```

Babel的底层原理

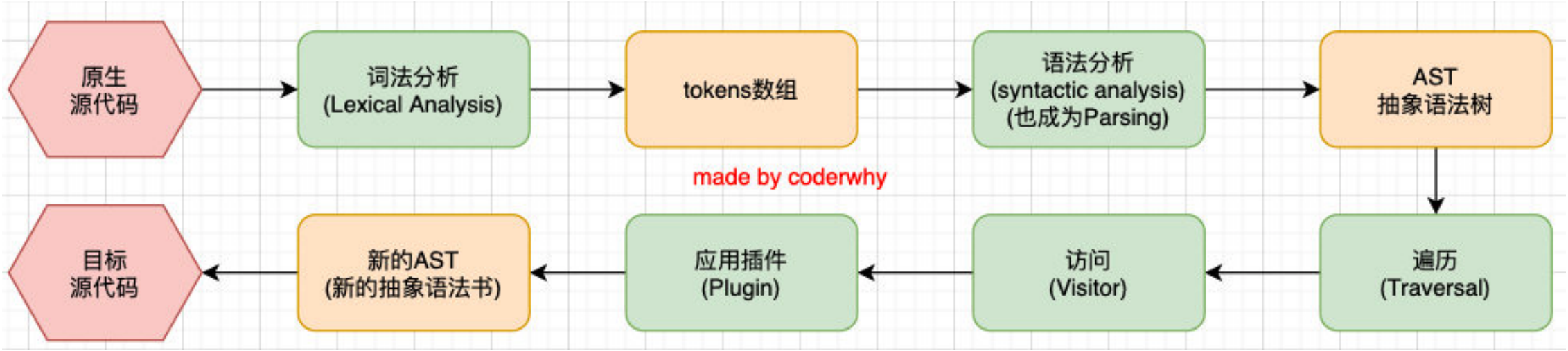
- babel是如何做到将我们的一段代码（ES6、TypeScript、React）转成另外一段代码（ES5）的呢？
  - 从一种源代码（原生语言）转换成另一种源代码（目标语言），这是什么的工作呢？
  - 就是**编译器**，事实上我们可以将babel看成就是一个编译器。
  - Babel编译器的作用就是将我们的源代码，转换成浏览器可以直接识别的另外一段源代码；
- Babel也拥有编译器的工作流程：
  - 解析阶段（Parsing）
  - 转换阶段（Transformation）
  - 生成阶段（Code Generation）

- https://github.com/jamiebuilds/the-super-tiny-compiler

babel编译器执行原理

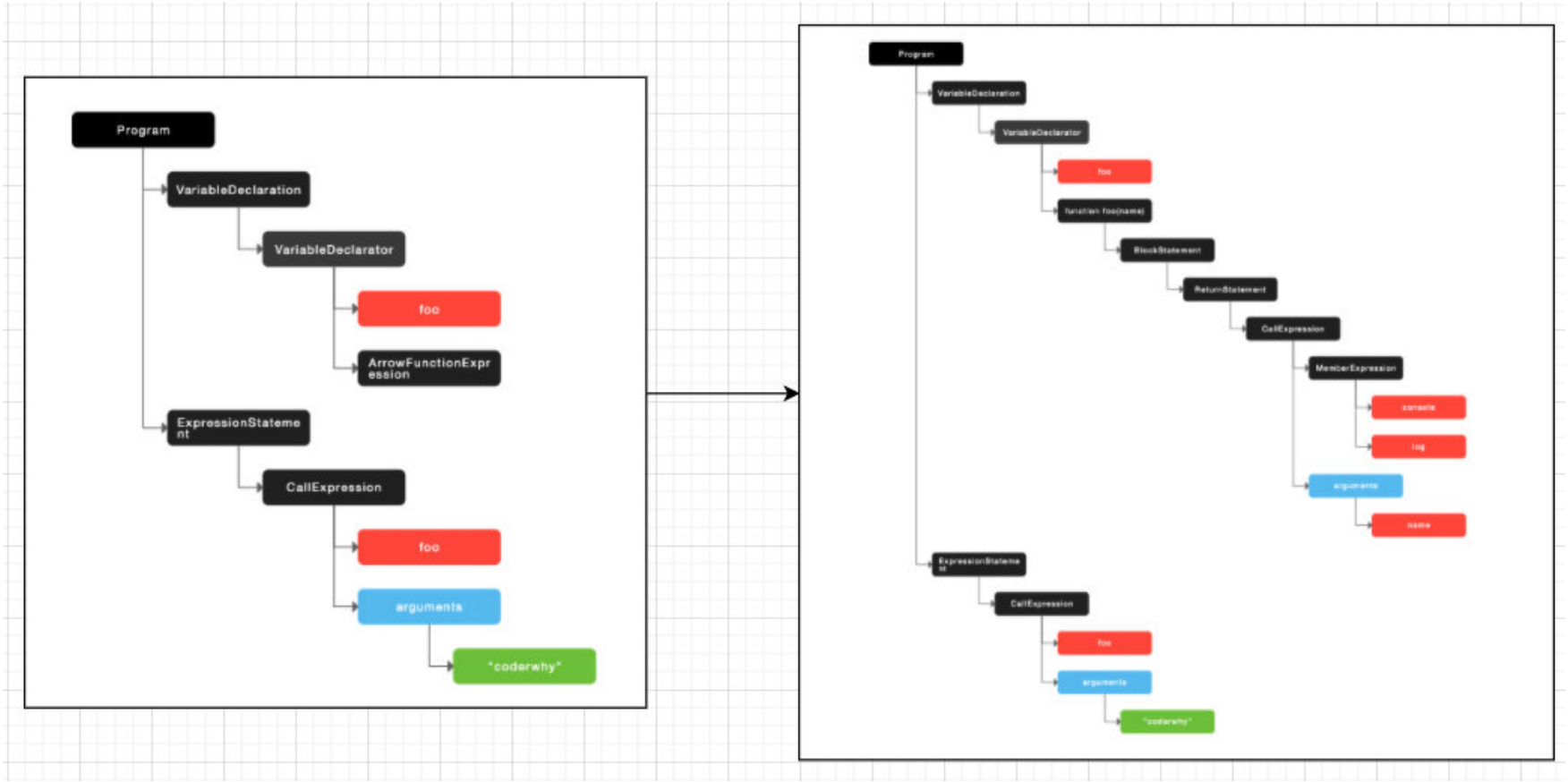
- Babel的执行阶段

- 当然，这只是一个简化版的编译器工具流程，在每个阶段又会有自己具体的工作：



中间产生的代码

```
const name = "coderwhy";
const foo = (name) => console.log(name);
foo(name);
```



babel-loader

- 在实际开发中，我们通常会在构建工具中通过配置babel来对其进行使用的，比如在webpack中。
- 那么我们就需要去安装相关的依赖：
  - 如果之前已经安装了@babel/core，那么这里不需要再次安装；

```
npm install babel-loader @babel/core
```

- 我们可以设置一个规则，在加载js文件时，使用我们的babel：

```
module: {
  rules: [
    {
      test: /\.m?js$/,
      use: {
        loader: "babel-loader"
      }
    }
  ]
},
```

指定使用的插件

- 我们必须指定使用的插件才会生效

```
{
  test: /\.m?js$/,
    use: {
      loader: "babel-loader",
      options: {
        plugins: [
          "@babel/plugin-transform-block-scoping",
          "@babel/plugin-transform-arrow-functions"
        ]
      }
  }
}
```

babel-preset

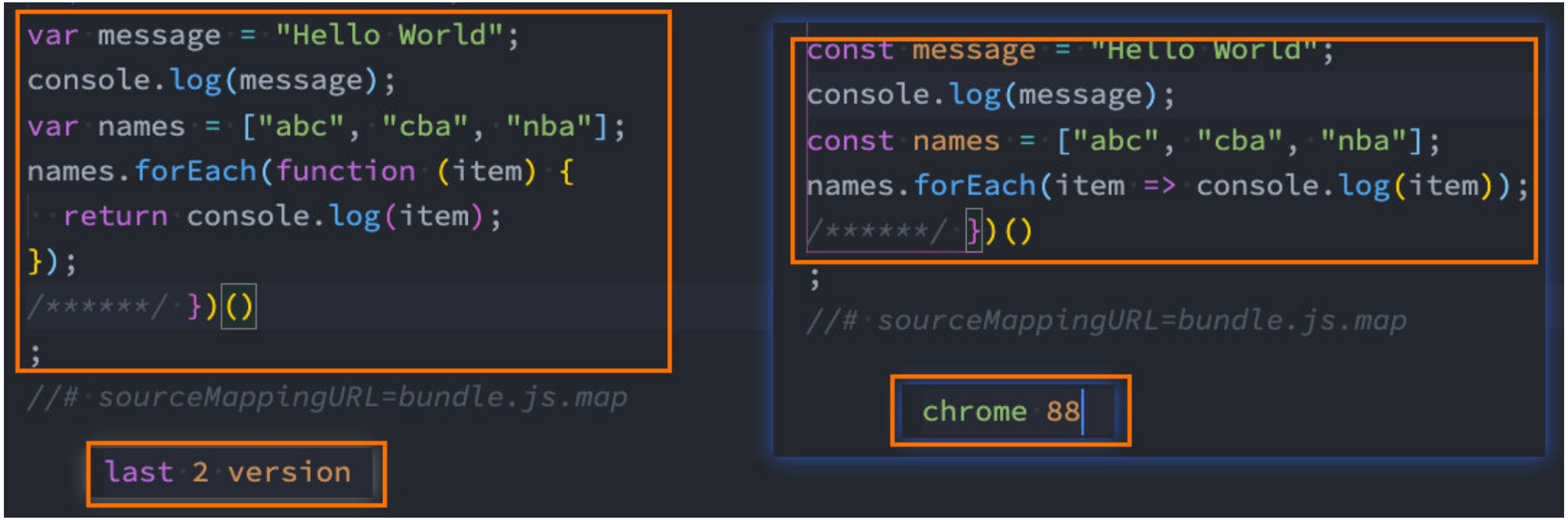
- 如果我们一个个去安装使用插件，那么需要手动来管理大量的babel插件，我们可以直接给webpack提供一个preset，webpack会根据我们的预设来加载对应的插件列表，并且将其传递给babel。
- 比如常见的预设有三个：
  - env
  - react
  - TypeScript
- 安装preset-env：

```
npm install @babel/preset-env
```

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

设置目标浏览器 browserslist

- 我们最终打包的JavaScript代码，是需要跑在目标浏览器上的，那么如何告知babel我们的目标浏览器呢？
  - browserslist工具
  - target属性
- 之前我们项目中已经使用了browserslist工具，我们可以对比一下不同的配置，打包的区别：



设置目标浏览器 targets

- 我们也可以通过targets来进行配置：

```
{
  test: /\.m?js$/,
  use: {
    loader: "babel-loader",
    options: {
      presets: [
        ["@babel/preset-env", {
          targets: "last 2 version"
        }]
      ]
    }
  }
}
```

- 那么，如果两个同时配置了，哪一个会生效呢？
  - 配置的targets属性会覆盖browserslist；
  - 但是在开发中，更推荐通过browserslist来配置，因为类似于postcss工具，也会使用browserslist，进行统一浏览器的适配；

Stage-X的preset

- 要了解Stage-X，我们需要先了解一下TC39的组织：
  - TC39是指技术委员会（Technical Committee）第 39 号；
  - 它是 ECMA 的一部分，ECMA 是 “ECMAScript” 规范下的 JavaScript 语言标准化的机构；
  - ECMAScript 规范定义了 JavaScript 如何一步一步的进化、发展；
- TC39 遵循的原则是：分阶段加入不同的语言特性，新流程涉及四个不同的 Stage
  - **Stage 0：**strawman（稻草人），任何尚未提交作为正式提案的讨论、想法变更或者补充都被认为是第 0 阶段的"稻草人"；
  - **Stage 1：**proposal（提议），提案已经被正式化，并期望解决此问题，还需要观察与其他提案的相互影响；
  - **Stage 2：**draft（草稿），Stage 2 的提案应提供规范初稿、草稿。此时，语言的实现者开始观察 runtime 的具体实现是否合理；
  - **Stage 3：**candidate（候补），Stage 3 提案是建议的候选提案。在这个高级阶段，规范的编辑人员和评审人员必须在最终规范上签字。Stage 3 的提案不会有太大的改变，在对外发布之前只是修正一些问题；
  - **Stage 4：**finished（完成），进入 Stage 4 的提案将包含在 ECMAScript 的下一个修订版中；

Babel的Stage-X设置

- 在babel7之前（比如babel6中），我们会经常看到这种设置方式：
  - 它表达的含义是使用对应的 babel-preset-stage-x 预设；
  - 但是从babel7开始，已经不建议使用了，建议使用preset-env来设置；

```
module.exports = {
  "presets": ["stage-0"]
}
```

Babel的配置文件

- 像之前一样，我们可以将babel的配置信息放到一个独立的文件中，babel给我们提供了两种配置文件的编写：
  - babel.config.json（或者.js，.cjs，.mjs）文件；
  - .babelrc.json（或者.babelrc，.js，.cjs，.mjs）文件；
- 它们两个有什么区别呢？目前很多的项目都采用了多包管理的方式（babel本身、element-plus、umi等）；
  - .babelrc.json：早期使用较多的配置方式，但是对于配置Monorepos项目是比较麻烦的；
  - babel.config.json（babel7）：可以直接作用于Monorepos项目的子包，更加推荐；

```
module.exports = {
  presets: [
    ["@babel/preset-env", {
      targets: "last 2 version"
    }]
  ]
}
```

认识polyfill

- Polyfill是什么呢？
  - 翻译：一种用于衣物、床具等的聚酯填充材料, 使这些物品更加温暖舒适；
  - 理解：更像是应该填充物（垫片），一个补丁，可以帮助我们更好的使用JavaScript；
- 为什么时候会用到polyfill呢？
  - 比如我们使用了一些语法特性（例如：Promise, Generator, Symbol等以及实例方法例如Array.prototype.includes等）
  - 但是某些浏览器压根不认识这些特性，必然会报错；
  - 我们可以使用polyfill来填充或者说打一个补丁，那么就会包含该特性了；

如何使用polyfill？

- babel7.4.0之前，可以使用 @babel/polyfill的包，但是该包现在已经不推荐使用了：

```
coderwhy@why 10_webpack中TypeScript % npm install @babel/polyfill --save
npm WARN deprecated @babel/polyfill@7.12.1: This package has been deprecated in favor of separate inclusion of a polyfill and
 regenerator-runtime (when needed). See the @babel/polyfill docs (https://babeljs.io/docs/en/babel-polyfill) for more informatio
n.
npm WARN deprecated core-js@2.6.12: core-js@<3 is no longer maintained and not recommended for usage due to the number of issues
. Please, upgrade your dependencies to the actual version of core-js@3.
```

- babel7.4.0之后，可以通过单独引入core-js和regenerator-runtime来完成polyfill的使用：

```
npm install core-js regenerator-runtime --save
```

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

配置babel.config.js

- **我们需要在babel.config.js文件中进行配置，给preset-env配置一些属性：**
- useBuiltIns：设置以什么样的方式来使用polyfill；
- corejs：设置corejs的版本，目前使用较多的是3.x的版本，比如我使用的是3.8.x的版本；
  - 另外corejs可以设置是否对提议阶段的特性进行支持；
  - 设置 proposals属性为true即可；

useBuiltIns属性设置

- useBuiltIns属性有三个常见的值
- 第一个值：false
  - 打包后的文件不使用polyfill来进行适配；
  - 并且这个时候是不需要设置corejs属性的；
- 第二个值：usage
  - 会根据源代码中出现的语言特性，自动检测所需要的polyfill；
  - 这样可以确保最终包里的polyfill数量的最小化，打包的包相对会小一些；
  - 可以设置corejs属性来确定使用的corejs的版本；

```
presets: [
  ["@babel/preset-env", {
    useBuiltIns: "usage",
    corejs: 3.8
  }]
]
```

useBuiltIns属性设置

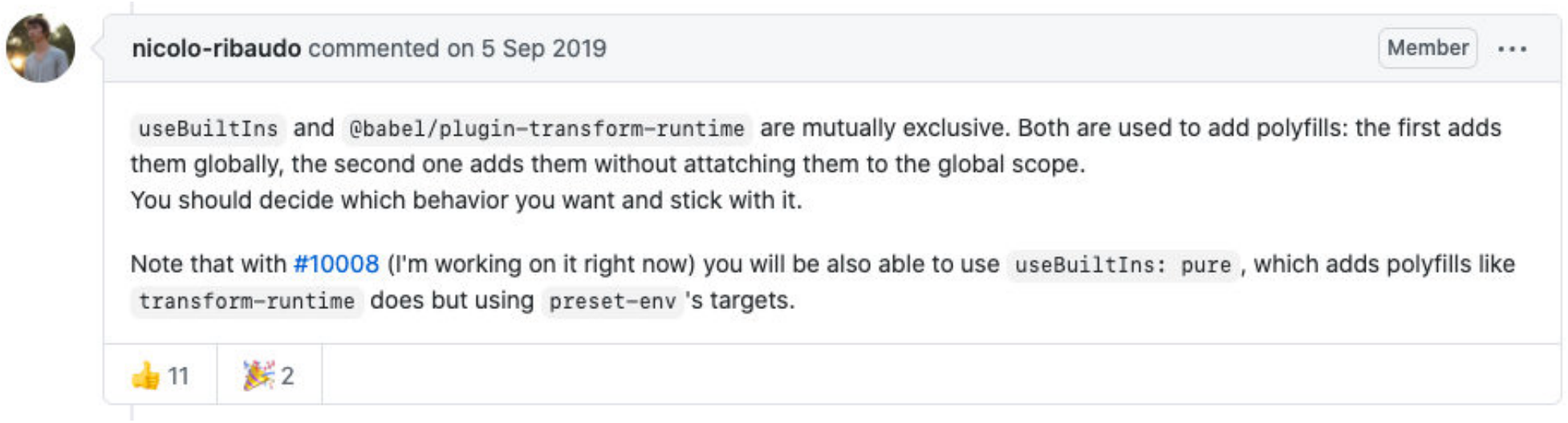
- 第三个值：entry
  - 如果我们依赖的某一个库本身使用了某些polyfill的特性，但是因为我们使用的是usage，所以之后用户浏览器可能会报错；
  - 所以，如果你担心出现这种情况，可以使用 entry；
  - 并且需要在入口文件中添加 **`import 'core-js/stable'; import 'regenerator-runtime/runtime';**
  - 这样做会根据 browserslist 目标导入所有的polyfill，但是对应的包也会变大；

```
presets: [
  ["@babel/preset-env", {
    useBuiltIns: "entry",
    corejs: 3.8
  }]
]
```

```
import "core-js/stable";
import "regenerator-runtime/runtime";
```

认识Plugin-transform-runtime（了解）

- 在前面我们使用的polyfill，默认情况是添加的所有特性都是全局的
  - 如果我们正在编写一个工具库，这个工具库需要使用polyfill；
  - 别人在使用我们工具时，工具库通过polyfill添加的特性，可能会污染它们的代码；
  - 所以，当编写工具时，babel更推荐我们使用一个插件： @babel/plugin-transform-runtime来完成polyfill的功能；



使用Plugin-transform-runtime

- 安装 @babel/plugin-transform-runtime：

```
npm install @babel/plugin-transform-runtime -D
```

- 使用plugins来配置babel.config.js：

```
module.exports = {
  presets: [
    ["@babel/preset-env", {
      // useBuiltIns: "usage",
      // corejs: 3.8
    }]
  ],
  plugins: [
    ["@babel/plugin-transform-runtime", {
      "corejs": 3,
      // 其他属性
    }]
  ]
}
```

注意：因为我们使用了corejs3，所以我们需要安装对应的库：

This option requires changing the dependency used to provide the necessary runtime helpers:

`corejs` option	Install command
`false`	`npm install --save @babel/runtime`
`2`	`npm install --save @babel/runtime-corejs2`
`3`	`npm install --save @babel/runtime-corejs3`

React的jsx支持

- 在我们编写react代码时，react使用的语法是jsx，jsx是可以直接使用babel来转换的。
- 对react jsx代码进行处理需要如下的插件：
  - @babel/plugin-syntax-jsx
  - @babel/plugin-transform-react-jsx
  - @babel/plugin-transform-react-display-name
- 但是开发中，我们并不需要一个个去安装这些插件，我们依然可以使用preset来配置：

```
npm install @babel/preset-react -D
```

```
presets: [
  ["@babel/preset-env", {
    // useBuiltIns: "usage",
    // corejs: 3.8
  }],
  ["@babel/preset-react"]
],
```

TypeScript的编译

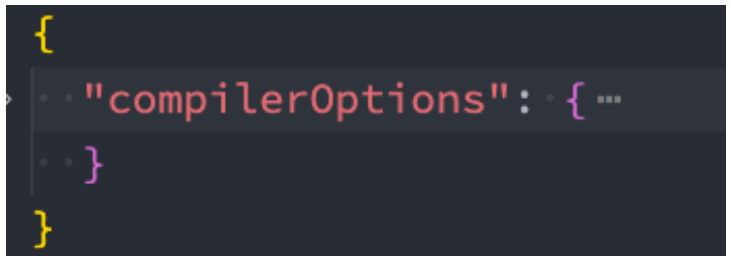
- 在项目开发中，我们会使用TypeScript来开发，那么TypeScript代码是需要转换成JavaScript代码。
- 可以通过TypeScript的compiler来转换成JavaScript：

```
npm install typescript -D
```

- 另外TypeScript的编译配置信息我们通常会编写一个tsconfig.json文件：

```
tsc --init
```

- 生成配置文件如下：



- 之后我们可以运行 npx tsc来编译自己的ts代码：

```
npx tsc
```

使用ts-loader

- 如果我们希望在webpack中使用TypeScript，那么我们可以使用ts-loader来处理ts文件：

```
npm install ts-loader -D
```

- 配置ts-loader：

```
{
  test: /\.ts$/,
  exclude: /node_modules/,
  use: [
    "ts-loader"
  ]
},
```

- 之后，我们通过npm run build打包即可。

使用babel-loader

- 除了可以使用TypeScript Compiler来编译TypeScript之外，我们也可以使用Babel：
  - Babel是有对TypeScript进行支持；
  - 我们可以使用插件： @babel/tranform-typescript；
  - 但是更推荐直接使用preset：@babel/preset-typescript；
- 我们来安装@babel/preset-typescript：

```
npm install @babel/preset-typescript -D
```

```
presets: [
  ["@babel/preset-env", {
    useBuiltIns: "usage",
    corejs: 3.8
  }],
  ["@babel/preset-react"],
  ["@babel/preset-typescript"]
],
```

```
{
  test: /\.ts$/,
  exclude: /node_modules/,
  use: [
    "babel-loader"
  ]
},
```

ts-loader和babel-loader选择

- **那么我们在开发中应该选择ts-loader还是babel-loader呢？**
- 使用ts-loader（TypeScript Compiler）
  - 来直接编译TypeScript，那么只能将ts转换成js；
  - 如果我们还希望在这个过程中添加对应的polyfill，那么ts-loader是无能为力的；
  - 我们需要借助于babel来完成polyfill的填充功能；
- 使用babel-loader（Babel）
  - 来直接编译TypeScript，也可以将ts转换成js，并且可以实现polyfill的功能；
  - 但是babel-loader在编译的过程中，不会对类型错误进行检测；
- 那么在开发中，我们如何可以同时保证两个情况都没有问题呢？

编译TypeScript最佳实践

- 事实上TypeScript官方文档有对其进行说明：

Babel vs `tsc` for TypeScript

When making a modern JavaScript project, you might ask yourself what is the right way to convert files from TypeScript to JavaScript?

A lot of the time the answer is *"it depends"*, or *"someone may have decided for you"* depending on the project. If you are building your project with an existing framework like tsdx, Angular, NestJS or any framework mentioned in the Getting Started then this decision is handled for you.

However, a useful heuristic could be:

- Is your build output mostly the same as your source input files? Use `tsc`
- Do you need a build pipeline with multiple potential outputs? Use `babel` for transpiling and `tsc` for type checking

- 也就是说我们使用Babel来完成代码的转换，使用tsc来进行类型的检查。
- 但是，如何可以使用tsc来进行类型的检查呢？
  - 在这里，我在scripts中添加了两个脚本，用于类型检查；
  - 我们执行 npm run type-check可以对ts代码的类型进行检测；
  - 我们执行 npm run type-check-watch可以实时的检测类型错误；

```
"scripts": {
  "build": "webpack --config wk.config.js",
  "type-check": "tsc --noEmit",
  "type-check-watch": "npm run type-check -- --watch"
},
```

认识ESLint

- **什么是ESLint呢？**
- ESLint是一个静态代码分析工具（Static program analysis，在没有任何程序执行的情况下，对代码进行分析）；
- ESLint可以帮助我们在项目中建立统一的团队代码规范，保持正确、统一的代码风格，提高代码的可读性、可维护性；
- 并且ESLint的规则是可配置的，我们可以自定义属于自己的规则；
- 早期还有一些其他的工具，比如JSLint、JSHint、JSCS等，目前使用最多的是ESLint。

使用ESLint？

- 首先我们需要安装ESLint：

```
npm install eslint -D
```

- 创建ESLint的配置文件：

```
npx eslint --init
```

- 选择想要使用的ESLint：

```
? How would you like to use ESLint? …
  To check syntax only
  To check syntax and find problems
❯ To check syntax, find problems, and enforce code style
```

- 执行检测命令：

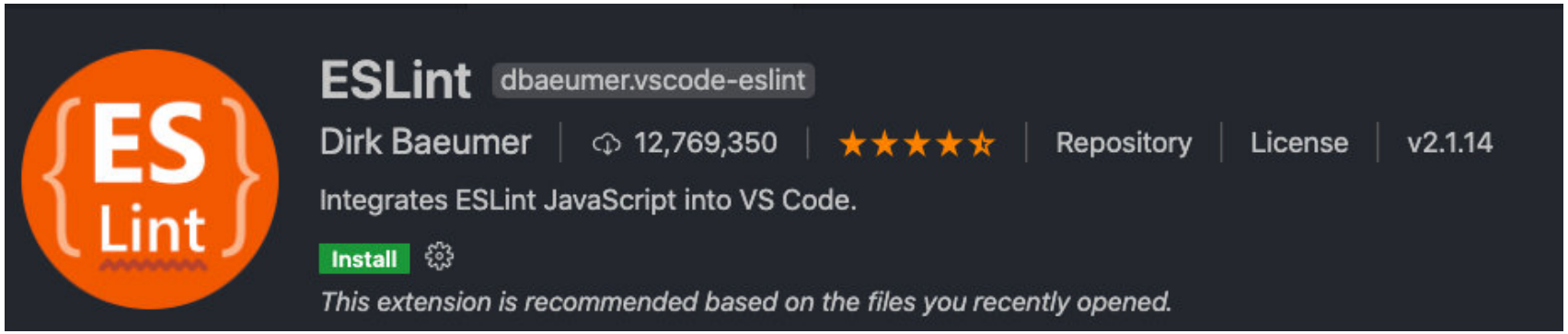
```
npx eslint ./src/main.js
```

ESLint的文件解析

- **默认创建的环境如下：**
- env：运行的环境，比如是浏览器，并且我们会使用es2021（对应的ecmaVersion是12）的语法；
- extends：可以扩展当前的配置，让其继承自其他的配置信息，可以跟字符串或者数组（多个）；
- parserOptions：这里可以指定ESMAScript的版本、sourceType的类型
  - parser：默认情况下是espree（也是一个JS Parser，用于ESLint），但是因为我们需要编译TypeScript，所以需要指定对应的解释器；
- plugins：指定我们用到的插件；
- rules：自定义的一些规则；

VSCode的ESLint插件

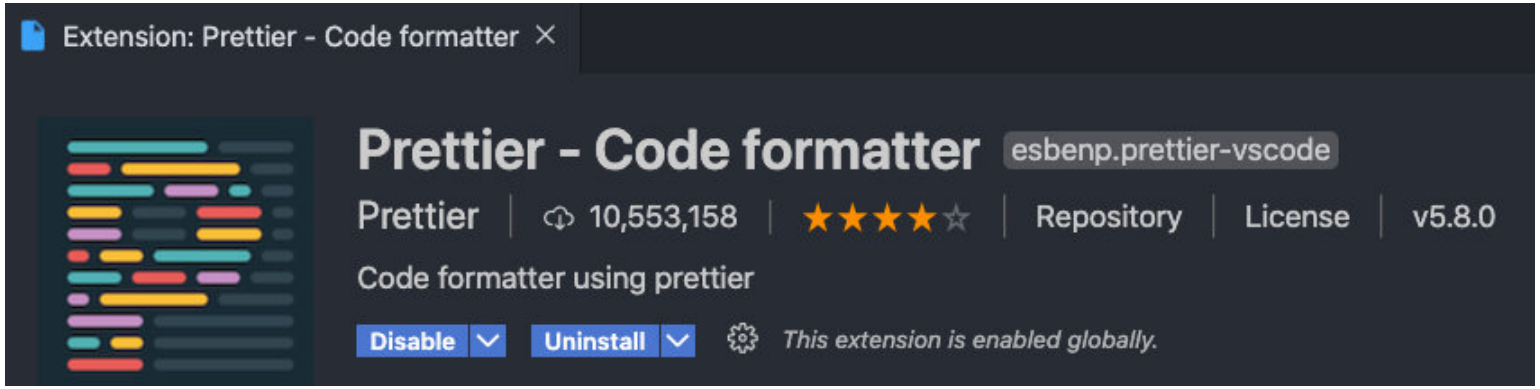
- 但是如果每次校验时，都需要执行一次npm run eslint就有点麻烦了，所以我们可以使用一个VSCode的插件：ESLint

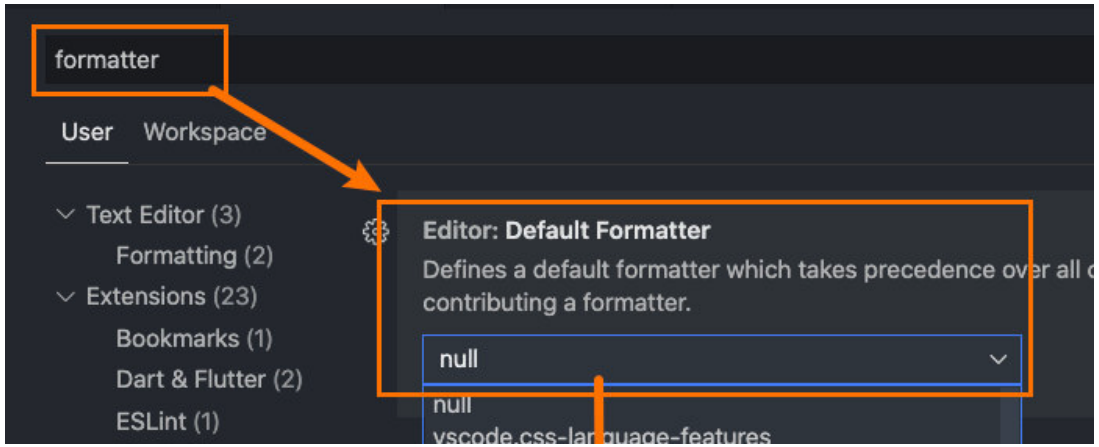


- 一方面我们可以修改代码来修复错误，另外我们可以通过配置规则：
  - 格式是： 配置的规则名称：对应的值值可以是数字、字符串、数组：
    - ✓ 字符串对应有三个值： off、warn、error；
    - ✓ 数字对应有三个值： 0、1、2（分别对应上面的值）；
    - ✓ 数组我们可以告知对应的提示以及希望获取到的值：比如 ['error', 'double']

VSCode的Prettier插件

- ESLint会帮助我们提示错误（或者警告），但是不会帮助我们自动修复：
  - 在开发中我们希望文件在保存时，可以自动修复这些问题；
  - 我们可以选择使用另外一个工具：prettier；





ESLint-Loader的使用

- 事实上，我们在编译代码的时候，也希望进行代码的eslint检测，这个时候我们就可以使用eslint-loader来完成了：

```
npm install eslint-loader -D
```

```
{
  test: /\.m?js$/,
  exclude: /node_modules/,
  use: [
    'babel-loader',
    'eslint-loader'
  ]
},
```

加载Vue文件

- 编写vue相关的代码：

```
new Vue({
  render: h => h(App)
}).$mount("#app");
```

```
<template>
  <div>
    <h2 class="title">{message}</h2>
  </div>
</template>

<script>
export default {
  props: {
    info: String
  },
  data: () => {
    return {
      message: "Hello Vue"
    }
  }
};
</script>
```

```
<style scoped>
  .title {
    color: red;
  }
</style>
```

Webpack中配置vue加载

- 安装相关的依赖：

```
npm install vue-loader -D
```

```
npm install vue-template-compiler -D
```

- 配置webpack

```
{
  test: /\.vue$/,
  use: "vue-loader"
}
```

```
const VueLoaderPlugin = require('vue-loader/lib/plugin');
```

```
plugins: [
  new CleanWebpackPlugin(),
  new HtmlWebpackPlugin({
    title: "coderwhy webpack",
    template: "./index.html"
  }),
  new VueLoaderPlugin()
]
```

```
{
  test: /\.less$/,
  use: [
    "style-loader",
    {
      loader: "css-loader",
      options: {
        importLoaders: 2
      }
    },
    "postcss-loader",
    "less-loader",
  ]
},
```

为什么要搭建本地服务器？

- 目前我们开发的代码，为了运行需要有两个操作：
  - 操作一：npm run build，编译相关的代码；
  - 操作二：通过live server或者直接通过浏览器，打开index.html代码，查看效果；
- 这个过程经常操作会影响我们的开发效率，我们希望可以做到，当文件发生变化时，可以自动的完成 编译 和 展示；
- 为了完成自动编译，webpack提供了几种可选的方式：
  - webpack watch mode；
  - webpack-dev-server；
  - webpack-dev-middleware
- 接下来，我们一个个来学习一下它们；

Webpack watch

- webpack给我们提供了watch模式：
  - 在该模式下，webpack依赖图中的所有文件，只要有一个发生了更新，那么代码将被重新编译；
  - 我们不需要手动去运行 npm run build指令了；
- 如何开启watch呢？两种方式：
  - 方式一：在导出的配置中，添加 watch: true；
  - 方式二：在启动webpack的命令中，添加 --watch的标识；
- 这里我们选择方式二，在package.json的 scripts 中添加一个 watch 的脚本：

```
"scripts": {
  "build": "webpack --config wk.config.js",
  "watch": "webpack --watch",
  "type-check": "tsc --noEmit",
  "type-check-watch": "npm run type-check -- --watch"
},
```

webpack-dev-server

- 上面的方式可以监听到文件的变化，但是事实上它本身是没有自动刷新浏览器的功能的：
  - 当然，目前我们可以在VSCode中使用live-server来完成这样的功能；
  - 但是，我们希望在不适用live-server的情况下，可以具备live reloading（实时重新加载）的功能；
- 安装webpack-dev-server

```
npm install --save-dev webpack-dev-server
```

- 修改配置文件，告知 dev server，从什么位置查找文件：

```
devServer: {
  contentBase: "./build"
},
```

```
target: "web",
```

```
"serve": "webpack serve --config wk.config.js",
```

- webpack-dev-server 在编译之后不会写入到任何输出文件。而是将 bundle 文件保留在内存中：
  - 事实上webpack-dev-server使用了一个库叫memfs（memory-fs webpack自己写的）

webpack-dev-middleware

- webpack-dev-middleware 是一个封装器(wrapper)，它可以把 webpack 处理过的文件发送到一个 server。
  - webpack-dev-server 在内部使用了它，然而它也可以作为一个单独的 package 来使用，以便根据需求进行更多自定义设置；
  - 我们可以搭配一个服务器来使用它，比如express；

```
npm install --save-dev express webpack-dev-middleware
```

```
const express = require("express");
const webpack = require("webpack");
const webpackDevMiddleware = require("webpack-dev-middleware");

const app = express();
const config = require("./wk.config");
const compiler = webpack(config);

app.use(webpackDevMiddleware(compiler, {
  publicPath: config.output.publicPath
}));

app.listen(3000, function() {
  console.log("运行在3000端口上");
})
```

node server.js 运行代码

publicPath

- Output中有两个很重要的属性：path和publicPath
  - path：用于指定文件的输出路径（比如打包的html、css、js等），是一个绝对路径；
  - publicPath：默认是一个空字符串，它为我们项目中的资源指定一个公共的路径（publicPath）；
- 这个publicPath很不容易理解，其实就是给我们打包的资源，给它一个路径：

```
资源的路径 = output.publicPath + 打包资源的路径（比如"js/[name].bundle.js"）
```

- 比较常设置的是两个值：
  - ./：本地环境下可以使用这个相对路径；
  - /：服务器部署时使用，服务器地址 + /js/[name].bundle.js；