# Babel

**对babel的理解?【热度: 551】**

关键词：babel 作用、为何要使用 babel

Babel 是一个非常流行的 JavaScript 编译器，用于将最新版本的 ECMAScript 代码转换为向后兼容的 JavaScript 代码，以便在旧版浏览器或环境中运行。

以下是对 Babel 的理解：

* 语法转换：Babel 可以将使用了最新 ECMAScript 标准的代码转换为被广泛支持的旧版 JavaScript 代码。例如，将使用了箭头函数、解构赋值等语法的代码转换为使用 function 关键字和传统赋值的等效代码。
* Polyfill：Babel 可以通过添加 Polyfill 来支持新的全局函数、方法和对象，以确保代码在旧版本的浏览器中正常运行。通过使用 Polyfill，Babel 可以在浏览器中模拟缺失的特性，使旧版浏览器能够运行使用了这些特性的代码。
* 插件系统：Babel 的核心功能可以通过插件进行扩展和定制。Babel 提供了众多的插件，用于实现不同的转换和功能。开发者可以根据自己的需求选择和配置相应的插件，以便将代码转换为特定的目标环境。
* 预设（Presets）：Babel 提供了预设的概念，可以一次性地配置一组插件，以实现特定的转换目标。预设是一组插件的集合，可以根据应用程序的需要进行选择和配置。常见的预设包括 "@babel/preset-env"（根据目标环境自动选择转换规则）和 "@babel/preset-react"（用于处理 React 相关的代码）。
* 与构建工具的集成：Babel 可以与构建工具（如 webpack、Rollup 等）无缝集成，作为其转换代码的一部分。通过配置构建工具，可以让 Babel 在构建过程中自动处理源代码，将其转换为目标代码。

总之，Babel 是一个功能强大的 JavaScript 编译器，可以将使用最新 ECMAScript 标准的代码转换为向后兼容的 JavaScript 代码，从而在旧版浏览器或环境中运行。通过插件和预设的配置，Babel 提供了高度的灵活性，使开发者能够根据项目需求定制转换规则。

**追问：为何要使用 babel**

使用 Babel 的主要原因是为了**解决 JavaScript 代码的兼容性问题**。以下是一些使用 Babel 的主要理由：

* 兼容旧版浏览器：不同的浏览器对 JavaScript 的支持程度不同，特别是旧版浏览器可能不支持最新的 ECMAScript 标准。通过使用 Babel，可以将使用了最新语法的代码转换为等效的旧版 JavaScript 语法，使代码能够在旧版浏览器中正常运行。
* 支持新特性：JavaScript 不断演进，每年都会发布新的 ECMAScript 标准，引入了很多有用的特性和语法糖。使用 Babel 可以提前使用这些新特性，而不用等待浏览器的支持。Babel 可以将这些新特性转换为旧版 JavaScript 语法，以便在当前的浏览器环境中使用。
* 模块化支持：Babel 可以转换模块化导入和导出的语法，使得开发者可以在浏览器中使用模块化的方式组织和管理代码。这样可以提高代码的可维护性和重用性。

**babel核心库有哪些?【热度: 35】**

关键词：babel 核心库

Babel 是一个 JavaScript 编译器，主要用于将 ES6 及以上版本的代码转换为向后兼容的 JavaScript 语法，以便在当前和旧版浏览器或环境中执行。核心的 Babel 库主要包括：

**@babel/core:**

这是 Babel 编译器的核心包，提供了 Babel 的主要转换引擎。它包含了解析、转换和生成代码的主要功能。几乎所有的 Babel 操作都需要这个模块作为基础。

**@babel/cli:**

这是 Babel 的命令行接口，通过它可以在终端或命令提示符中运行 Babel。它允许你执行转编译操作，如将 ES6 代码转换为 ES5。

**@babel/preset-env:**

这是一个智能预设，允许你使用最新的 JavaScript，而不必管理语法转换。@babel/preset-env会根据你的目标环境（比如特定版本的浏览器或 Node.js），自动决定使用哪些 Babel 插件和 polyfills。

**@babel/polyfill (现在已经被废弃，推荐使用 core-js 和 regenerator-runtime):**

早期 Babel 版本中用于模拟完整的 ES2015+环境的包。它的目的是在全局范围内添加填充以模拟较新的环境。从 Babel 7.4.0 开始，建议直接包括 core-js 和 regenerator-runtime，因为这提供了更好的模块化和按需加载功能。

**babel-loader:**

这是 Babel 的一个 webpack 插件，可以将 Babel 集成到 webpack 构建过程中，使得你可以使用 webpack 来处理和打包使用了新版 JavaScript 语法的文件。

**@babel/plugin-transform-runtime:**

这个插件用于复用 Babel 注入的辅助代码，以节省代码大小，并能够在不污染全局环境的情况下使用新语言特性的 polyfills。

除了这些核心库外，还有许多可用的 Babel 插件，以支持各种 JavaScript 语法和特性（比如装饰器、类属性等）。这些插件可以按需引入，配置在 Babel 的配置文件（通常是.babelrc或babel.config.js）中。这些插件的命名通常遵循 @babel/plugin- 的格式。

**在Babel里， stage0、stage1、stage2和stage3分别代表什么含义?[了解即可]**

Babel是⼀个流⾏的JavaScript编译器，它允许开发者使⽤新的语⾔特性，然后将它们编译成可以在当前和低版本的浏览器或环境中运⾏的代码。

在Babel⾥，stage0、stage1、stage2和stage3这些术语指的是ECMAScript提案的不同阶段。ECMAScript是JavaScript语⾔的标准化规范，新的特性进⼊标准之前会通过⼏个阶段的提案。

这些阶段表⽰了⼀个特性在正式成为ECMAScript标准的⼀部分之前的成熟度。这个过程有⼀个官⽅的5个阶段流程，即从Stage0（strawman）到Stage4（finished）。下⾯是这些阶段的含义：

• Stage0-Strawman（稻草⼈阶段）:初始阶段，任何尚未被TC39（ECMAScript的标准化组织）官⽅审议的提案都属于这⾥。这些都是某个委员或者社区成员提交的想法，还不算是正式的提案。

• Stage1-Proposal（提案阶段）:这个阶段的特性是值得进⼀步探讨的。它们需要有⼀个形式化的提案和⼀个负责⼈。在这个阶段，主要是确定问题和解决⽅案，以及进⾏初步探讨。

• Stage2-Draft（草案阶段）:⼀旦⼀个提案到达这个阶段，它就被认为是初步规格的草案。特性的描述应该⾜够具体和详细，并且有初步的实现。这个阶段通常需要提案的规格⽂本和⾄少⼀种实验性实现。

• Stage3-Candidate（候选阶段）:在候选阶段，提案的规格已经基本完成，并且需要更多的⽤户反馈来发现潜在问题。通常在这个阶段，实现者和开发者开始在⽣产环境中尝试使⽤这些特性，发现问题并提出改善建议。

• Stage4-Finished（完成阶段）:当⼀个提案达到这个阶段，它已经准备好被集成到下⼀个版本的ECMAScript标准中了。这意味着它已经获得了多个独⽴环境的实现，通过了综合的可⾏性和稳定性测试，并且已经被TC39委员会接受。

开发者们可以根据特性的稳定性和⾃⼰的需求，选择使⽤Babel的哪个阶段的预设。然⽽，请注意，使⽤较低阶段的提案特性在⽣产环境中是有⻛险的，因为它们还没有被完全确定并可能会在将来发⽣变更。

## 如何编写一个babel插件【热度: 1,062】

关键词：babel插件、babel插件api、babel插件代码示例

编写一个 babel 插件的基本步骤

编写一个 Babel 插件可以让你自定义转换、分析或操作 JavaScript 代码。下面是编写 Babel 插件的基本步骤：

1.安装 Babel：首先，确保你已经安装了 Babel 的相关工具和依赖。可以使用 npm 或 yarn 安装 @babel/core、@babel/preset-env 和 @babel/plugin-syntax-plugin-name。

2.创建插件文件：在项目中创建一个新的 JavaScript 文件，用于编写自定义插件的代码。命名约定是以 babel-plugin- 开头，例如 babel-plugin-custom-plugin.js。

3.导出插件函数：在插件文件中，导出一个函数作为你的插件。这个函数将接收一个 Babel 的 babel 对象作为参数，包含了一些 Babel 的工具方法，如 types 和 template。

module.exports = function(babel) {

// 插件代码

};

4.实现插件逻辑：在插件函数内部，实现你的插件逻辑。可以使用 babel.types 对象提供的方法来操作抽象语法树（AST）节点，例如 babel.types.VariableDeclaration、babel.types.CallExpression 等。

module.exports = function(babel) {

const { types: t } = babel;

return {

visitor: {

Identifier(path) {

// 对每个 Identifier 节点进行处理

const name = path.node.name;

path.node.name = name.toUpperCase();

}

}

};

};

5.导出插件配置：为了让 Babel 可以识别你的插件，需要在插件函数中返回一个配置对象，其中 visitor 属性指定了你的插件要处理的 AST 节点类型和对应的处理函数。

module.exports = function(babel) {

const { types: t } = babel;

return {

visitor: {

// ...

}

};

};

6.配置 Babel：在项目的 .babelrc 或 babel.config.js 文件中，将你的插件添加到 Babel 的插件列表中。

{

"plugins": ["babel-plugin-custom-plugin"]

}

7.使用插件：运行 Babel，它将根据你的配置和代码中的语法，应用插件并对代码进行转换。

以上是编写 Babel 插件的基本步骤，可以根据具体需求和场景，实现各种自定义的转换、分析和操作逻辑。

### babel 编写插件的时候有哪些核心方法

在编写 Babel 插件时，可以使用以下核心方法来操作抽象语法树（AST）节点：

1.types 对象： Babel 的 types 对象是你在插件中最常用的工具之一。它提供了一系列用于创建、访问和操作 AST 节点的方法。

types.identifier(name): 创建一个标识符节点，表示一个变量或函数的名称。

types.stringLiteral(value): 创建一个字符串字面量节点，表示一个字符串值。

types.numericLiteral(value): 创建一个数值字面量节点，表示一个数字值。

types.booleanLiteral(value): 创建一个布尔字面量节点，表示一个布尔值。

types.objectExpression(properties): 创建一个对象表达式节点，表示一个对象字面量。

types.arrayExpression(elements): 创建一个数组表达式节点，表示一个数组字面量。

types.callExpression(callee, arguments): 创建一个函数调用表达式节点，表示一个函数的调用。

types.memberExpression(object, property): 创建一个成员表达式节点，表示一个对象的成员访问。

这些方法可以帮助你构建新的 AST 节点或访问现有的 AST 节点。

2.path 对象： Babel 的 path 对象代表 AST 中的一个路径，你可以通过该对象访问和操作 AST 节点。在插件的处理函数中，你将会经常使用 path 对象。

path.node: 访问当前路径对应的节点。

path.parent: 访问当前路径的父路径。

path.scope: 访问当前路径的作用域。

path.traverse(visitor): 遍历当前路径的子路径，使用指定的访问者函数。

path.replaceWith(node): 替换当前路径的节点。

path.remove(): 移除当前路径的节点。

这些方法可以帮助你在遍历 AST 树时对节点进行修改、替换或删除。

3.traverse 方法： babel-traverse 是 Babel 提供的一个独立的模块，用于遍历和操作 AST。在插件中，你可以使用 traverse 方法来遍历 AST 树并应用你的插件逻辑。

traverse(ast, visitor): 使用指定的访问者函数遍历给定的 AST 树。

visitor 是一个对象，其中包含了处理不同类型节点的方法。通过在 visitor 对象中定义相应类型节点的处理函数，你可以在遍历过程中针对特定类型的节点执行你的插件逻辑。

4.babel.template 方法： babel-template 是 Babel 提供的一个独立模块，用于根据字符串模板生成 AST 节点。你可以使用 babel.template 方法来创建包含特定模板结构的 AST 节点。

babel.template(code, options): 根据指定的代码模板生成 AST 节点。

code 参数是一个包含要生成的代码模板的字符串，而 options 参数可以指定一些配置选项，如 preserveComments 来保留注释。该方法将返回一个函数，调用该函数并传入替换模板中的变量值，即可生成对应的 AST 节点。

通过使用 babel.template 方法，你可以更方便地创建复杂的 AST 节点结构，尤其在需要生成大量相似结构的节点时非常有用。

5.babel.transform 方法： babel-transform 是 Babel 提供的一个独立模块，用于将 JavaScript 代码转换为 AST 或将 AST 转换回 JavaScript 代码。在编写插件时，你可以使用 babel.transform 方法来进行代码转换操作。

babel.transform(code, options): 将指定的代码转换为 AST 或将 AST 转换回代码。

code 参数是一个包含要转换的 JavaScript 代码的字符串，而 options 参数可以指定一些配置选项，如 plugins 来指定要应用的插件。该方法将返回一个包含 ast 和 code 属性的对象，ast 属性表示生成的 AST 树，code 属性表示转换后的代码。

通过使用 babel.transform 方法，你可以在插件内部对代码进行转换和处理，将代码转换为 AST 进行修改，然后再将修改后的 AST 转换回代码。

### 编写一个去除代码里面 console.log 的 babel 插件

以下是一个简单的 Babel 插件示例，用于去除代码中的 console.log 语句：

// babel-plugin-remove-console.js

module.exports = function ({ types: t }) {

return {

visitor: {

// 处理函数调用表达式

CallExpression(path) {

const { callee } = path.node;

// 如果函数调用的名称是 console.log

if (

t.isMemberExpression(callee) &&

t.isIdentifier(callee.object, { name: 'console' }) &&

t.isIdentifier(callee.property, { name: 'log' })

) {

// 移除该函数调用

path.remove();

}

}

}

};

};

该插件会遍历代码中的函数调用表达式，如果发现是 console.log，则会移除该函数调用。

要使用该插件，可以在项目中安装并配置它。例如，创建一个 .babelrc 文件，并将该插件添加到 Babel 的插件列表中：

{

"plugins": ["./path/to/babel-plugin-remove-console.js"]

}

然后运行 Babel 命令或构建工具，它将应用该插件，并从代码中去除所有的 console.log 语句。

请注意，这只是一个简单的示例插件，仅适用于演示目的。在实际开发中，你可能需要更复杂的逻辑来处理不同的情况和要求。

## Babel Polyfill了解多少【热度: 200】

关键词：Babel Polyfill 原理、Babel Polyfill 作用、Babel Polyfill 使用、Babel Polyfill 按需加载

### Babel Polyfill 作用是啥

Babel Polyfill 的作用是在旧版本浏览器中提供对新的JavaScript特性和API的支持。当使用Babel进行代码转换时，它只会转换语法，而不会转换新的API和全局对象（如Promise、Map、Set等）。

旧版本的浏览器可能不支持这些新的API和全局对象，因此在运行使用这些特性的代码时会抛出错误。为了解决这个问题，可以使用Babel Polyfill来填充缺失的功能，以确保代码在旧版本浏览器中正常运行。

Babel Polyfill通过修改全局对象和原型链，添加缺失的方法和属性，使得代码能够在不支持这些功能的浏览器中运行。它会检测当前环境的特性支持情况，并根据需要自动加载所需的Polyfill代码。

使用Babel Polyfill可以让开发人员在编写代码时不必过多考虑浏览器的兼容性，而专注于使用最新的JavaScript特性和API。它提供了一种简单方便的方式来填充浏览器的功能差异，确保代码在各种浏览器环境中具有一致的行为。

### 如何使用

要使用 Babel Polyfill，需要按照以下步骤进行设置：

1.安装依赖：首先，确保你的项目已经安装了 Babel 相关的依赖包。这包括 @babel/core、@babel/preset-env 和 @babel/polyfill。你可以使用 npm 或者 yarn 进行安装：

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

2.配置 Babel：在项目根目录下创建一个 .babelrc 文件，并添加以下配置：

{

"presets": ["@babel/preset-env"]

}

这样的配置将告诉 Babel 使用 @babel/preset-env 预设来进行转换。

3.导入 Polyfill：在你的入口文件（通常是项目的主 JavaScript 文件）中导入 Babel Polyfill。你可以使用 import 语句或者 require 来导入 Polyfill：

使用 import（适用于 ES6 模块）：

import '@babel/polyfill';

使用 require（适用于 CommonJS 模块）：

require('@babel/polyfill');

导入 Polyfill 的位置很重要，通常应该在你的应用程序代码之前导入，以确保 Polyfill 在应用程序代码之前被加载和执行。

4.配置目标浏览器：为了让 Babel Polyfill 根据目标浏览器进行特性填充，你可以在 .babelrc 文件中的 @babel/preset-env 配置中指定目标浏览器的选项。例如，你可以在配置中添加 targets 属性：

{

"presets": [

[

"@babel/preset-env",

{

"targets": {

"browsers": ["last 2 versions", "ie >= 11"]

}

}

]

]

}

这样，Polyfill 将根据所选的目标浏览器填充相应的功能。

完成以上步骤后，Babel Polyfill 将根据配置在目标浏览器中填充所需的功能，以确保你的代码在旧版本浏览器中正常运行。请注意，Polyfill 会增加你的应用程序的大小，因此请考虑仅填充所需的功能，以减小文件大小并优化性能。

### 按需加载 Polyfill

Babel Polyfill 默认会填充所有缺失的功能，但如果你只需要按需加载特定功能，可以使用 core-js 库的按需加载特性。下面是按需加载 Babel Polyfill 的步骤：

1.安装依赖：确保你的项目已经安装了必要的依赖。除了之前提到的 Babel 相关依赖外，你还需要安装 core-js。

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

2.配置 Babel：在 .babelrc 文件中，添加以下配置：

{

"presets": [

[

"@babel/preset-env",

{

"useBuiltIns": "usage",

"corejs": 3

}

]

]

}

useBuiltIns 选项设置为 "usage" 表示按需加载特性，而 "corejs": 3 指定了使用的 core-js 版本。

3.导入 Polyfill：在需要使用特定功能的文件中，按需导入所需的 Polyfill。例如，如果你需要填充 Promise 和 Array.prototype.includes，你可以按如下方式导入：

import 'core-js/features/promise';

import 'core-js/features/array/includes';

这样只会加载和填充所需的功能，而不会加载整个 Polyfill 库。你可以根据具体的功能需求进行按需导入。

请注意，使用按需加载的方式可以减小应用程序的文件大小，并且只填充需要的功能，但需要确保在使用相关功能之前已经导入了相应的 Polyfill。

## babel的工作流程是如何的?

Babel 是一个 JavaScript 编译器，它的主要功能是将新版本的 JavaScript 代码转换成向后兼容的代码。Babel 的工作流程可以简单概括为以下几个步骤：

1. 解析：将 JavaScript 代码解析成 AST（抽象语法树）。
2. 转换：对 AST 进行遍历，进行代码转换。
3. 生成：将转换后的 AST 生成 JavaScript 代码。

具体来说，Babel 的工作流程如下：

1. Babel 使用 babylon 解析器将 JavaScript 代码解析成 AST，babylon 是一个基于 AST 的 JavaScript 解析器。
2. Babel 使用 babel-traverse 遍历器对 AST 进行遍历，找到需要转换的节点，进行转换。
3. Babel 使用 babel-core 转换器将 AST 转换成 JavaScript 代码。babel-core 是 babel 的核心模块，它包含了所有的转换器和插件。
4. Babel 使用 babel-generator 生成器将转换后的 AST 生成 JavaScript 代码。babel-generator 是一个将 AST 转换成 JavaScript 代码的工具。

在整个流程中，Babel 还会使用 babel-preset-env、babel-plugin-transform-runtime、babel-polyfill 等插件和工具来完成更加复杂的任务，如将 ES6 模块转换成 CommonJS 模块，使用 Polyfill 来实现一些新的 API 等。

需要注意的是，Babel 的转换过程是有损的，转换后的代码不一定与原始代码完全相同，也可能存在性能问题。因此，在使用 Babel 进行转换时，需要谨慎选择转换的规则和插件，以确保转换后的代码正确、高效。

## [Webpack] 通过 babel-loader 来编译 tsx 文件， 应该如何配置呢？【热度: 221】

如果你想使用 babel-loader 来编译 TypeScript 文件（.tsx），你需要在 babel.config.js 和 webpack.config.js 两个文件中进行相应的配置。

首先，在 babel.config.js 文件中，你需要添加 @babel/preset-typescript 预设，以便兼容 TypeScript：

module.exports = {

presets: ['@babel/preset-env', '@babel/preset-react', '@babel/preset-typescript']

};

接下来，在 webpack.config.js 文件中，你需要对 .tsx 文件使用 babel-loader：

module.exports = {

// ...其他配置项

module: {

rules: [

// ...其他规则

{

test: /\.(ts|tsx)$/,

exclude: /node\_modules/,

use: 'babel-loader',

},

],

},

};

这样，当 webpack 执行构建时，babel-loader 将会使用 babel.config.js 中配置的预设来编译 .tsx 文件。同时，除了 TypeScript 文件，你还可以使用该配置来编译 JavaScript 文件（.js）和 React JSX 文件（.jsx）。

### 追问 '@babel/preset-env', '@babel/preset-typescript', '@babel/preset-react' 这三个插件的作用是什么？

@babel/preset-env：它是 Babel 的一个预设，用于根据目标环境（浏览器、Node.js 等）自动确定需要转译的 JavaScript 特性，并将其转换为目标环境所支持的代码。它将根据你配置的目标环境和浏览器的使用情况，智能地选择需要转译的特性，以减小转译后的代码体积。它也包含了一些插件，例如转换箭头函数、解构赋值、模板字符串等。

@babel/preset-typescript：它是 Babel 的一个预设，用于将 TypeScript 代码转译为 JavaScript 代码，以便在不支持 TypeScript 的环境中运行。它包含了一些插件，例如转换 TypeScript 的类型注解、类成员修饰符、泛型类型等。

@babel/preset-react：它是 Babel 的一个预设，用于将 React 的 JSX 语法转译为普通的 JavaScript 代码，以便在不支持 JSX 的环境中运行。它也包含了一些插件，例如转换 JSX 语法、处理 React 的内置组件等。

## babel-runtime 作用是啥【热度: 200】

关键词：babel-runtime 作用

babel-runtime 是一个**包含 babel 模块化运行时助手的库**。

在使用 babel 进行代码转换时，有时会注入一些在多个文件中相同且可能被重复使用的代码。例如，使用类转换（无松散模式）时，每个包含类的文件都会重复出现类似 \_classcallcheck 这样的函数。

babel-runtime 的主要作用就是将这些可能被重用的代码抽取成单独的模块，以避免在每个文件中重复出现相同的代码。它通过模块导入的方式引入这些功能，从而避免了对全局作用域的修改或污染。

具体来说，babel-runtime 包含了诸如 core-js（提供 JavaScript 内置库的垫片，如 array、json、math、promise、symbol 等）、regenerator-runtime（实现了 generator/yield、async/await）以及一些语法转换的辅助函数（如 es5 与 es6 的继承转换等）。

使用 babel-runtime 通常需要配合 babel-plugin-transform-runtime 插件一起使用。babel-plugin-transform-runtime 插件会进行一些处理，例如自动导入 babel-runtime/core-js，并将全局静态方法、全局内置对象映射到对应的模块；将内联的工具函数移除，改成通过 babel-runtime/helpers 模块进行导入；如果使用了 async/generator 函数，则自动导入 babel-runtime/regenerator 模块等。

这样，在代码中如果需要使用特定的功能，只需从 babel-runtime 相应的模块中导入即可，而不是直接使用全局的对象或函数。

例如，如果代码中使用了 promise，可以这样导入：

import promise from "babel-runtime/core-js/promise";

总的来说，babel-runtime 更像是一种按需加载的实现方式，适用于开发库、工具等场景，可避免对全局作用域的污染，同时减少重复代码。