# 工程搭建

工程搭建

前端工程的基础概念

**by eric wen**

# 工程文件结构

- src

-- assets

-- components

--- Head

--- .....

-- Form

--- input

--- select

-- layout

--- home-layout

--- default-layout

-- pages

--- home

--- cart

--- .....

-- store

-- utils

# webpack

思考一个问题：

打包时，自动去掉代码里的console

打包时，自动去掉代码里的debbuger

# 是什么webpack

Webpack 是一个模块打包工具，主要用于JavaScript 程序的构建。它将各种资源（如 JavaScript 文件、样式表、图片等）视为模块，并通过依赖关系图将这些模块打包成一个或多个最终的输出文件。

# 为什么需要webpack

至今为止，javascript 并没有真正支持文件的模块化。

组件化是将视图和逻辑封装成一个函数或者类。

而模块化，是将这段代码放在一个文件中，实现组件-文件，1v1的效果，方便维护和开发。

是现代前端工程化的基础。

webpack解决了这一问题。

# webpack的基础常识

(1) 入口 (Entry)：指示 Webpack 应该从哪个文件开始打包。这个文件通常被称为入口点，是应用程序的起点。

(2) 输出 (Output)：定义了 Webpack 打包后的文件存放路径及其名称。默认情况下，WebPack 将所有打包好的文件放入 dist 目录中。

(3) 加载器 (Loaders)：允许你转换非 JavaScript 模块（例如，将 SASS 转换为 CSS 或将 TypeScript 转换为 JavaScript）。加载器本质上是用于处理不同类型的文件并将其转换为有效的模块作为依赖项。

(4) 插件 (Plugins)：可以完成加载器无法完成的任务。它们可以执行更广泛的任务，包括优化包文件、管理环境变量、压缩文件等等。

# Webpack常用插件

1、HtmlWebpackPlugin

作用：简化 HTML 文件的创建，用于自动生成包含打包后资源的 HTML 文件。它会自动将生成的 JavaScript 和 CSS 文件注入到 HTML 中。

使用场景：当你需要自动生成 index.html 文件，并自动引入打包后的资源时。

2. MiniCssExtractPlugin

作用：将 CSS 从 JavaScript 中提取出来，生成单独的 CSS 文件，而不是将 CSS 内嵌到 JavaScript 中。

使用场景：在生产环境中，为了优化性能，通常会将 CSS 提取为独立的文件。

3、HotModuleReplacementPlugin

作用：启用模块热替换功能，允许在运行时更新模块而无需完全刷新页面。

使用场景：在开发环境中，用于提高开发效率，修改代码后无需刷新页面即可看到更新。

1、TerserWebpackPlugin

作用：用于压缩和优化 JavaScript 代码，移除未使用的代码、缩短变量名等。

使用场景：在生产环境中，用于减小 JavaScript 文件的体积，提升加载速度。

2、ESLintWebpackPlugin

作用：将 ESLint 集成到 Webpack 构建流程中，在打包时自动检查代码规范。

使用场景：在开发或构建过程中，确保代码符合 ESLint 的规范，避免潜在的错误。

6、CleanWebpackPlugin

作用：在每次构建前清理输出目录（如 dist 文件夹），避免旧文件残留。

使用场景：确保每次构建时输出目录是干净的。

7、ImageMinimizerWebpackPlugin

作用：压缩图片资源，减小图片体积。

使用场景：优化图片加载性能。 **OptimizeChunkAssetsPlugin**

作用：分析和优化 JavaScript 文件的体积，以加快加载速度。

使用场景：在打包过程中对代码进行进一步优化，以减少资源加载时间。

# 常用loaders

JavaScript 相关

1、babel-loader 用于将 ES6+ 代码转换为 ES5，兼容旧版浏览器。 通常与 @babel/core 和 @babel/preset-env 一起使用。

2、ts-loader 用于编译 TypeScript 文件。

3、eslint-loader 在打包前对 JavaScript 代码进行 lint 检查。

样式相关

1、css-loader 解析 CSS 文件，处理 @import 和 url() 等语法。

2、sass-loader 编译 SASS/SCSS 文件为 CSS，通常与 sass 或 node-sass 一起使用。

3、less-loader 编译 Less 文件为 CSS。

4、postcss-loader 用于处理 CSS 的 PostCSS 插件（如自动添加浏览器前缀、压缩 CSS 等）。

# 解决问题

1、去掉console

2、去掉debugger

# Babel

Babel 是一个广泛使用的 JavaScript 编译器，主要用于将现代 JavaScript（ES6+）代码转换为向后兼容的版本（如 ES5），以便在旧版浏览器或环境中运行。它在前端开发和 Node.js 开发中非常流行，帮助开发者使用最新的 JavaScript 特性，而无需担心兼容性问题。

## Babel 的主要功能：

1. **语法转换**
   将新语法（如箭头函数、解构赋值、可选链操作符等）转换为旧版浏览器支持的语法。
2. **Polyfill**
   提供对新 API 的支持（如 `Promise`、`Array.prototype.includes` 等）。通过引入 polyfill，可以在旧环境中模拟这些新特性。

# 语法转换是什么

## 示例 1：箭头函数（Arrow Functions）

**原始代码：**

```
const add = (a, b) => a + b;
console.log(add(2, 3));
```

**Babel 转换后：**

```
var add = function add(a, b) {
  return a + b;
};
console.log(add(2, 3));
```

**解释：**

- 箭头函数被转换为传统的函数表达式。
- 这是为了兼容不支持箭头函数的旧版浏览器（如 IE11）。

## 示例 2：类（Classes）

**原始代码：**

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const person = new Person('Alice');
person.greet();
```

**Babel 转换后：**

```
function _classCallCheck(instance, Constructor) {
  if (!(instance instanceof Constructor)) {
    throw new TypeError("Cannot call a class as a function");
  }
}

function _defineProperties(target, props) {
  for (var i = 0; i < props.length; i++) {
    var descriptor = props[i];
    descriptor.enumerable = descriptor.enumerable || false;
    descriptor.configurable = true;
    if ("value" in descriptor) descriptor.writable = true;
    Object.defineProperty(target, descriptor.key, descriptor);
  }
}

function _createClass(Constructor, protoProps, staticProps) {
  if (protoProps) _defineProperties(Constructor.prototype, protoProps);
  if (staticProps) _defineProperties(Constructor, staticProps);
  return Constructor;
}

var Person = /*#__PURE__*/function () {
  function Person(name) {
    _classCallCheck(this, Person);
    this.name = name;
  }

  _createClass(Person, [{
    key: "greet",
    value: function greet() {
      console.log("Hello, my name is ".concat(this.name));
    }
  }]);
  return Person;
}();

var person = new Person('Alice');
person.greet();
```

**解释：**

- 类语法被转换为构造函数和原型方法的形式。
- 使用了辅助函数（如 `_classCallCheck` 和 `_createClass`）来模拟类的行为。

## 示例 3：解构赋值（Destructuring）

**原始代码：**

```
const user = { name: 'Bob', age: 25 };
const { name, age } = user;
console.log(name, age);
```

**Babel 转换后：**

```
var user = { name: 'Bob', age: 25 };
var name = user.name,
    age = user.age;
console.log(name, age);
```

**解释：**

- 解构赋值被转换为普通的属性访问形式。
- 这是为了兼容不支持解构语法的旧环境。

## 示例 4：可选链操作符（Optional Chaining）

**原始代码：**

```
const user = {};
console.log(user?.profile?.name ?? 'Unknown');
```

**Babel 转换后：**

```
var _user$profile;

var user = {};
console.log(
  (_user$profile = user.profile) === null || _user$profile === void 0
    ? void 0
    : _user$profile.name !== null && _user$profile.name !== void 0
    ? _user$profile.name
    : 'Unknown'
);
```

**解释：**

- 可选链操作符 `?.` 被转换为一系列条件判断。
- 空值合并操作符 `??` 被转换为三元运算符逻辑。

## 示例 5：模板字符串（Template Literals）

**原始代码：**

```
const name = 'Charlie';
console.log(`Hello, ${name}!`);
```

**Babel 转换后：**

```
var name = 'Charlie';
console.log("Hello, ".concat(name, "!"));
```

**解释：**

- 模板字符串被转换为使用 `String.prototype.concat` 的形式。
- 这是为了兼容不支持模板字符串的旧浏览器。

## 兼容性控制

https://browsersl.ist/