NodeJs&Webpack&TypeScript

无15 向明义

Node.js

简介

问题的产生：Javascript 简单好用，能否给它加上更丰富的API和更强大的功能，并且让它脱离浏览器，独立运行在服务端?

Node.js 为 Javascript 提供了一个运行环境，让其可以脱离浏览器运行，还添加了诸如文件系统访问、网络编程等功能，让其能够处理前后端的各种需求甚至开发桌面软件，使得 Javascript 有了质的飞跃。

> 官网：Node.js 中文网 (nodejs.cn)
>
> 下载：下载 | Node.js 中文网 (nodejs.cn)

```
sudo apt-get install nodejs
```

Node.js 使用非阻塞式I/O，它是单线程的，通过事件队列的方式实现并发。

当网络或文件请求发生时，Node.js 会注册一个回调函数，然后就去处理下一个请求了。当之前的操作完成后，会触发之前注册的回调函数，进而响应之前的请求。基于这个特性，node的许多操作都是异步的。

> createFile.js 示例

JS模块化

要想使用丰富的功能，必然要对不同的程序进行模块化。Node.js从诞生以后便经久不衰，离不开它成熟的模块化实现。

在讲解具体的实现方式之前，可以先大概了解一下原始的html引用js片段可能有的一些问题：

```
<script src='a.js'></script>
<script src='b.js'></script>
<script src='c.js'></script>
```

- 全局变量污染

```
//a.js
var name = 'aaa';
//b.js
var name = 'bbb';
```

- 数据不安全

```
//a.js
var a_name = 'aaa';
//b.js
a_name = 'bbb';
```

- 模块依赖关系难以手动维护

为了解决这些问题，Node.js在运行时会将每个文件视为单独的模块，其中的变量有自己的作用域，若无特殊声明，每个文件里定义的变量、函数等都是私有的，对其他文件不可见。而模块系统的具体实现方式，则有 `CommonJS` 和 `ECMAScript` 两种。

CommonJS

CommonJS 模块 | Node.js v20.4.0 文档 (nodejs.cn)

CommonJS 模块是为 Node.js 打包 JavaScript 代码的原始方式，在不加任何设置的情况下，Node.js 默认采用此方式加载模块。

CommonJS的关键是 `module.exports` 和 `require`。

每个模块内部都有一个 `module` 变量，是对本身模块的引用。`module.exports` 保存了当前模块可以被其他模块访问的变量或接口。当需要访问其他模块时，使用 `require` 加载模块：

```
// a.js
var name = 'morrain'
var age = 18
module.exports.name = name
module.exports.getAge = function(){
    return age
}
//b.js
var a = require('./a.js')
console.log(a.name) // 'morrain'
console.log(a.getAge())// 18
```

ECMAScript

ECMAScript 模块 | Node.js v20.4.0 文档 (nodejs.cn)

ECMAScript （ES）模块是如今的官方标准格式，采用 `import` 和 `export` 来导入导出：

```
import transform from './transform.js' /* default import */
import { var1 } from './consts.js'     /* import a specific item */
import('http://example.com/example-module.js').then(() => {console.log('loaded')})
export const MODE = 'production'       /* exported const */
```

> tip：相对路径要加`./`，不然会在`node_modules`里面找。更具体的包寻找方式参见官网。

可以通过配置，也可以通过文件后缀名 `.cjs` 或 `.mjs` 来让 nodejs 确定以何种方式加载模块。

npm & yarn

https://www.npmjs.com/

Yarn中文文档(yarnpkg.cn)

为了使用更多的第三方模块，我们需要一款包管理工具。npm(node package manager)和yarn都是包管理工具，它们可以实现：

- 下载别人编写的模块到本地使用
- 安装别人编写的命令行程序到本地使用
- 将自己编写的模块或命令行程序上传

安装

npm一般会随着nodejs一起安装，但在某些linux发行版中仍然需要 `sudo apt-get install npm`

ps: 官方推荐使用 `NVM` 来管理node和npm的版本，但日常测试不太需要。

安装完成后可以用 `npm -v` 来检查版本。

可以修改 npm 下载包的镜像源地址为淘宝镜像，加快下载速度：

```
npm config set registry https://registry.npm.taobao.org
```

之后使用 `npm install [package name]` 就可以安装各种包了。

yarn是由facebook发布的一款较新的包管理工具，弥补了npm的一些缺陷。较高版本的node自带安装yarn， 其他版本可通过 `npm install -g yarn` 来安装。

设置 yarn 为淘宝源：

```
yarn config set registry https://registry.npm.taobao.org -g
yarn config set sass_binary_site http://cdn.npm.taobao.org/dist/node-sass -g
```

使用

这里仅介绍yarn的使用。所有的命令行操作都可以在上面给出的链接中的CLI部分查看。

- 初始化

  `yarn init`

  相关信息可以直接回车省略。执行完成后，文件夹内会生成 `package.json` 文件。

  > **package.json**
  >
  > npm和yarn都使用package.json确定其依赖版本。（yarn会默认增加yarn.lock以确定更精确的版本）
  >
  > 在package.json中，各依赖的版本在"dependencies"属性中通过以下方式列出：
  >
  > ```
  > "5.0.3" // 表示指定安装了5.0.3版本
  > "~5.0.3" // 表示安装了5.0.X中的最新版
  > "^5.0.3" // 表示安装了5.X.X中的最新版
  > ```
  >
  > 除了依赖，package.json中还定义了一些别的配置，如scripts。
  >
  > 我们可以在scripts中定义一组可以运行的node脚本，省下重复输入大量代码的时间。
  >
  > 更多信息参见：Manifest fields (yarnpkg.cn)

  > **yarn.lock**
  >
  > 这个文件已经把依赖模块的版本号全部锁定，当你执行yarn install的时候，yarn会读取这个文件获得依赖的版本号，然后依照这个版本号去安装对应的依赖模块，这样依赖就会被锁定，以后再也不用担心版本号的问题了。其他人或者其他环境下使用的时候，把这个yarn.lock拷贝到相应的环境项目下再装包即可。
  >
  > 这个文件不需要手动修改，yarn会自动更新yarn.lock。

- 安装 package.json 中的包

`yarn install`（或简写为`yarn`）。如果package.json指定的依赖版本是一个范围，yarn会读取yarn.lock中的精确版本号。

- 添加新的包

  `yarn add [package]`。会自动更新`package.json`和`yarn.lock`。

  `yarn add [package]@[version]`安装特定版本。

  可以设置add的参数，使其出现在package.json的不同位置。

  > **全局依赖**
  >
  > `npm install [package] -g`和`yarn global add [package]`都可以实现某些全局依赖的安装。它将在所有工作目录下均可见。
  >
  > 对于绝大多数包来说，全局依赖并不是值得推荐的，因为全局依赖是隐性的，难以察觉的。更好的方式是项目中所有的依赖都采用本地依赖的方式，这样更明确，也能保证任何人使用你的项目会得到跟你一样的依赖（从依赖版本到依赖结构）。

  > **开发环境依赖**
  >
  > 有些包只需要在开发时使用，部署到服务器上的应用只提供服务功能。
  >
  > `npm install --save-dev`和`yarn add --dev`会将包添加到`package.json`的`devDependencies`而非`dependencies`下，当执行`yarn install --production`或指定了环境变量为`NODE_ENV=production`，那么`devDependencies`下的依赖不会被安装，减小最终应用的大小。

- 移除依赖包

  `yarn remove`

Webpack

简介

当你要构建一个大型前端应用时，各种模块的依赖和资源的引用可能变得难以维护。传统的方式中，HTML，css，JS三者是分离的，在生产环境中要思考怎么把它们结合在一起，同时要在更改时修改对应的文件，这并不是一个很轻松的活。

所幸，只要你在编写时遵循一定的结构，并做好一些配置，这些都可以交给webpack自动帮你解决。

概念 | webpack 中文文档 (webpackjs.com)

本质上，webpack 是一个现代 JavaScript 应用程序的静态模块打包器(module bundler)。当webpack处理应用程序时，它会递归地构建一个依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundles。

webpack还有一些强大的特性，比如在js文件中导入图片和css文件，仅打包使用到的代码而不是加载整个包，有很好的性能和加载时间...

安装

webpack通常只在开发时使用，因此安装时使用`--dev`参数：

```
yarn add --dev webpack webpack-cli
```

配置

webpack的默认配置文件为 `webpack.config.js`。当没有参数指定别的配置文件名时，默认使用该文件，在webpack4.0.0以上的版本，可以没有配置文件，入口将默认为 ./src/index.js，输出到dist目录。

核心概念

- **入口 (entry)**

  指定由哪个文件作为起始构建依赖图。

- **出口 (output)**

  告诉 webpack 在哪里输出它所创建的 *bundle*，以及如何命名这些文件。主要输出文件的默认值是 `./dist/main.js`，其他生成文件(图片、css等资源)默认放置在 `./dist` 文件夹中。

- **加载器 (loader)**

  webpack 只能理解 JavaScript 和 JSON 文件，这是 webpack 开箱可用的自带能力。**loader** 让 webpack 能够去处理其他类型的文件（比如图片、css文件），并将它们转换为有效模块，使得js可以import这些文件，这是webpack所独有的特点。

- **插件 (plugin)**

  插件目的在于解决 loader 无法实现的其他事，想要使用一个插件，你只需要 require() 它，然后把它添加到配置文件的 plugins 数组中。

> 项目演示，参考教程：起步 | webpack 中文文档 (webpackjs.com)

TypeScript

TypeScript 中文网 (nodejs.cn)

JavaScript 是一个不具有强类型的动态语言，这赋予了它极大的灵活性，但也带来了开发和生产上可能存在的问题。TypeScript 是 JavaScript 的超集，使得 JavaScript 中的每一个变量和函数都具有和 C 一样的类型定义。

将“Type”放在名字当中，可见TypeScript中“类型”的重要性。

你可以利用 TypeScript 在编译期进行类型检查，提前发现错误。我们在使用ts的时候，最终还是会将其编译为js代码，但是在编译的时候会进行静态检查如果发现有错误，编译的时候就会报错。

比如在 JS 中，可能会出现这样的情况：

```
if ("" == 0) {
  // true!
}
if (1 < x < 3) {
  // True for *any* value of x! (null, undefined)
}
console.log(1/[]); //Infinity
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth; //area = NaN
```

所有这些类型不匹配的操作都会在编译期被查出。

安装

安装TypeScript编译器：

```
npm install -g typescript
或
yarn global add typescript
```

使用 `tsc -v` 检查是否安装成功。（注意命令行是缩写）

使用 `tsc <file>` 即可运行编译器，生成一个编译后的 Javascript 文件。

类型注解

TypeScript中的类型注解用于对变量添加约束，可以使用冒号 `:` 来添加类型注解，直接初始化变量相当于隐式添加类型注解(ts会自行推断)。 编译器会在编译TS时检查变量类型是否符合注解。

```
function greeter(person: string) {
    return "Hello, " + person;
}
let user = [0, 1, 2];
greeter(user);
//greeter.ts(7,26): error TS2345: Argument of type 'number[]' is not assignable to
parameter of type 'string'.
```

TypeScript中的类型

```
//布尔值
let isDone: boolean = false;
//数字
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
let binaryLiteral: number = 0b1010;
let octalLiteral: number = 0o744;
//字符串
let name: string = "bob";
name = "smith";
let sentence: string = `Hello, my name is ${ name }.`
//数组
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
//元组,表示一个已知元素数量和类型的数组，各元素的类型不必相同
//声明元组
let x: [string, number];
// 正确初始化
x = ['hello', 10]; // OK
// 错误初始化
x = [10, 'hello']; // Error
//枚举
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
//Void，表示一个函数没有返回值
function warnUser(): void {
console.log("This is my warning message");
}
```

```
//null 和 undefined
let u: undefined = undefined;
let n: null = null;
// 注意：null和undefined是所有类型的子类型
// 这样不会报错
let num: number = undefined;
```

any类型

any（任意值）用来表示允许赋值为任意类型，并且**可以对其访问任何属性，调用任何方法。**它主要用于为那些在编程阶段还不清楚类型的变量指定一个类型。同时，**对于未赋初值且未指定类型的变量，ts会将其自动识别为any类**，any会在不太合规的TypeScript开发中大量出现。

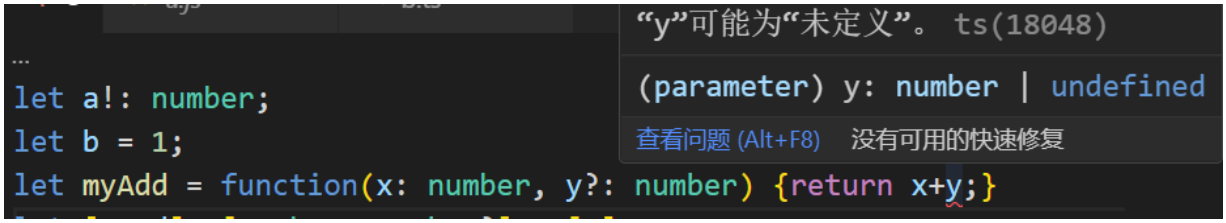
函数类型

函数本身也是"数"的一种，因此我们同样可以对function进行类型注解。指定其参数类型，返回值类型

```
function(x: number, y: number): number { return x + y; };

let myAdd: (x: number, y: number) => number =
function(x: number, y: number): number { return x + y; }; //完整写法

let myAdd = function(x: number, y: number): number { return x + y; }; //可推断

let myAdd: (baseValue: number, increment: number) => number =
function(x, y) { return x + y; }; //可推断
```

可选属性与非空断言

参数后面加 `?` 表示该项可选，后面加 `!` 表示在类型检查时忽略该项没有赋初值的问题。




```
let myAdd = function(x: number, y?: number) {return x+(y||0);}
//js的||运算符：a||b，如果a为真，返回a，否则返回b。注意不是返回0/1
//js中NaN，空字符串，0，undefined都会被判定为false。
```

interface

顾名思义，就是用接口规定类型，用法可以类比C的结构体。

```
interface Person {
    name: string;
    age: number;
}
let tom: Person = {
    name: 'Tom',
    age: 25
};
//当缺失字段时会报错
let jack: Person = {
    name: 'Jack'
} // Property 'age' is missing in type '{ name: string; }'.
```

这个问题可以通过给接口加上可选属性解决：

```
interface Person {
    name: string;
    age?: number;
}
let jack: Person = {
    name: 'Jack'
};
```

类型断言

通过类型断言可以覆盖编译器的推断，告诉编译器，“相信我，我知道自己在干什么，请不要再发出错误”。

类型断言有两种方式：

```
let someValue: any = "string";
let strLength1: number = (<string>someValue).length;      //方式一
let strLength2: number = (someValue as string).length;    //方式二
```

用例：

```
const foo = {};
foo.bar = 123; // Error: 'bar' 属性不存在于 ‘{}’
foo.bas = 'hello'; // Error: 'bas' 属性不存在于 '{}'

interface Foo {
    bar: number;
    bas: string;
}
const foo = {} as Foo; // 类型断言
foo.bar = 123;
foo.bas = 'hello';
```

断言其实就是把ts的类型检查去掉，回到js的状态。如果例子中foo没有给bar赋值，那么访问它的时候结果会是 undefined。

联合类型

联合类型（Union Types）表示取值可以为多种类型中的一种。联合类型使用 | 分隔每个类型。

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们**只能访问此联合类型的所有类型里共有的属性或方法。**

```
function getLength(something: string | number): number {
    return something.length;
}
// index.ts(2,22): error TS2339: Property 'length' does not exist on type
'string | number'.
// Property 'length' does not exist on type 'number'.
```

类型别名

使用 `type` 我们可以创建类型别名，常用于联合类型。

```
type NoS = number | string;
```

模块化

全局脚本

在默认情况下，当你开始在一个新的 TypeScript 文件中写下代码时，它处于全局命名空间中。如在foo.ts里定义的变量 `const foo = 1`。是可以被同一目录下的另一个文件bar.ts访问的。

模块

如果 TypeScript 文件含有 import 或者 export ，那么该文件会被视为一个模块，里面创建的变量若非export，均为私有的，与 ES 模块相同。

导出

- 可以在声明变量、函数、类、接口等时直接导出：

  ```
  export const numberRegexp = /^[0-9]+$/;
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }
  ```

- 也可以在声明之后导出，此时可以重命名：

```
class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

导入

对于以上两种导出方式，导入时需加大括号，且名称需一致，可重命名：

```
import { ZipCodeValidator } from "./ZipCodeValidator";
//or
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
```

默认导出

每个模块可以有一个 `default` 导出。对于 default 导出，在导入的时候不必加大括号，而且可以直接重命名。

```
//a.ts
export default "123";
//b.ts
import num from "./a.ts";
```

这里并没有列出导入导出的所有方式，更完整的介绍详见：TypeScript 中文网: 文档 - 模块 (nodejs.cn)

Babel

简介

我们现在有了ts，但ts的新语法并不能直接在nodejs环境运行，要经过tsc编译器转译为普通的js语法。

另外一个问题是，js语言本身在不断发展，ES6,ES7等新标准不一定能被所有的浏览器所支持，因此要让代码能够更通用，我们需要将用新语法写出的js转译为低标准也支持的语法。

要实现这些转译功能，我们有一个通用的工具：Babel。

什么是 Babel？ · Babel 中文网 (nodejs.cn)

> Babel 是一个 JavaScript 编译器，一个工具链，主要用于将 ECMAScript 2015+ (ES6+) 代码转换为当前和旧版浏览器或环境中向后兼容的 JavaScript 版本。 以下是 Babel 可以为你做的主要事情：
>
> - 转换语法
> - 补充目标环境中缺少的功能
> - 以及更多...

学习 ES2015 · Babel 中文网 (nodejs.cn)是ES6的新特性，它们通常在浏览器中运行通常需要转译。

运行方式

babel 本身不具有任何转换功能，它把转换的功能都分解到一个个 plugin 里面。因此当我们不配置任何插件时，经过 babel 的代码和输入是相同的。

插件分为两种：

- 语法插件 ，使得 babel 能够解析更多的语法。

  举个简单的例子，当我们定义或者调用方法时，最后一个参数之后是不允许增加逗号的，如 `callFoo(param1, param2,)` 就是非法的。如果源码是这种写法，经过 babel 之后就会提示语法错误。

  但最近的 JS 提案中已经允许了这种新的写法。为了避免 babel 报错，就需要增加语法插件 `babel-plugin-syntax-trailing-function-commas`

- 转译插件 ，对源码进行转换

  比如箭头函数 `(a) => a` 就会转化为 `function (a) {return a}` 。完成这个工作的插件叫做 `babel-plugin-transform-es2015-arrow-functions`。

如果我们使用了转译插件，babel 会自动使用语法插件。只有极少数情况需要自行添加语法插件。

安装babel核心与plugin

```
yarn add --dev @babel/core @babel/cli
```

`@babel/core` 是 babel 的核心功能库。`@babel/cli` 是允许你从命令行使用 babel 的工具。

```
./node_modules/.bin/babel src --out-dir lib
```

该命令将解析 `src` 目录中的所有 JavaScript 文件，应用我们告诉它的任何转换，并将每个文件输出到 `lib` 目录。`./node_modules/.bin/babel` 可用 `npx babel` 代替。

然后就是添加插件了，要使用一个插件，分为两步：

1. 将插件的名字添加到配置文件中 (根目录下创建 .babelrc 或 babel.package.json，添加plugin字段)
2. 使用 `npm install babel-plugin-xxx` 进行安装

（插件的使用方式很多，可以通过不同的配置文件以及命令行设置，这里只做一种演示）

使用preset插件组

ES6有许多特性需要转译，babel 提供了预制好的插件组合，这些组合被称为预设 preset，`preset-env` 是一个常用的编译ES6+的预设。

安装预设：

```
yarn add --dev @babel/preset-env
```

然后添加到配置文件：

```
{
    "presets": [
        "preset1",
        "preset2",
        ["preset3", {options}]
    ]
}
```

补丁

旧浏览器可能无法实现一些新语法的功能，我们可以引入 `core-js` 包来补充这些功能。

在 Babel 7.4.0 之前，我们通过引入 Polyfill 集成包来代替，在 babel 的`targets` 配置项中添加 `"useBuiltIns": "usage"` 可以在最终编译出的结果文件中只引入需要的模块。7.4.0之后，改为在文件中直接引入 `corejs/stable`。

```
yarn add core-js@3
```

该包需要在生产环境中预先加载，所以不使用 `--dev` 参数。

例：将TypeScript转为Javascript

安装所需的包：

```
yarn add @babel/core @babel/cli @babel/preset-env @babel/preset-typescript
@babel/node --dev
# @babel/core是Babel的核心库
# @babel/cli是Babel的命令行工具
# @babel/preset-env是一组将最新JavaScript语法转化的预设插件集
# @babel/preset-typescript是一组将TypeScript语法转化的预设插件集
# @babel/node可以应用所选的Babel工具并像node一样运行代码
```

修改配置文件：

```
{
    "presets": [
        "@babel/preset-env",
        "@babel/preset-typescript"
    ]
}
```

运行：

```
npx babel src -d lib  # -d是--out-dir的缩写
```

Links

web 入门 - 学习 Web 开发 | MDN (mozilla.org)

Node.js 中文网 (nodejs.cn)

https://www.npmjs.com/

Yarn中文文档(yarnpkg.cn)

webpack | webpack 中文文档 | webpack 中文网 (webpackjs.com)

TypeScript中文网 · TypeScript——JavaScript的超集 (tslang.cn)

什么是 Babel? · Babel 中文网 (nodejs.cn)