构建工具

Webpack

webpack如何实现 让我在一个前端项目中引用一个不在这个项目中的js代码

1. 将外部代码打包成一个独立的 library。
2. 安装打包好的 library，例如通过 npm 安装。
3. 在项目中引用安装的 library。

第一步：打包外部代码

在打包外部代码的过程中，你需要使用 `libraryTarget` 选项。

以下是一个 Webpack 配置文件的示例：

```
const path = require('path');

module.exports = {
  entry: './path/to/your/library.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'library.js',
    library: 'library',
    libraryTarget: 'umd'
  }
};
```

第二步：安装打包好的 library

执行以下命令即可安装打包好的 library：

```
npm install path/to/your/library
```

第三步：引用安装的 library

在项目中引用 library 可以使用以下代码

```
const library = require('library');

//

import library from 'library';
```

如果要打包的文件不是一个js文件而是一个文件夹，文件夹中有很多js文件怎么办

如果要打包的文件不是一个 JavaScript 文件，而是一个文件夹，你可以使用 `entry` 选项将该文件夹作为入口文件，Webpack 会根据该文件夹下的所有 JavaScript 文件自动构建一个 bundle。

以下是一个 Webpack 配置文件的示例：

```
const path = require('path');

module.exports = {
  entry: './path/to/your/folder',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'library.js',
    library: 'library',
    libraryTarget: 'umd'
  }
};
```

该配置文件会将 `./path/to/your/folder` 下的所有 JavaScript 文件作为入口文件，并打包成一个名为 `library.js` 的文件。

最后，执行以下命令即可打包文件：

```
webpack
```

执行此命令后，Webpack 会根据配置文件的设置将文件夹中的所有 JavaScript 文件打包成一个可导出的 library。

举例：

```
const path = require('path');

module.exports = {
    mode: 'production',
    resolve: {
        alias: {
          "@": path.resolve('src'),
        },
      },
    entry: {
        AccountController: "./src/store/controller/AccountController",
        AccountTypeController: "./src/store/controller/AccountTypeController",
        GroupController: "./src/store/controller/GroupController",
        InvitationController: "./src/store/controller/InvitationController",
        LoginController: "./src/store/controller/LoginController",
        MyAccountController: "./src/store/controller/MyAccountController",
        MyActionController: "./src/store/controller/MyActionController",
        MyRobotController: "./src/store/controller/MyRobotController",
        SCPController: "./src/store/controller/SCPController",
        ServicesController: "./src/store/controller/ServicesController",
        TransferController:"./src/store/controller/TransferController",
        UserController: "./src/store/controller/UserController",
        Util: "./src/util/Util",
      },
      output: {
        path: path.join(__dirname, "FD"),
        filename: '[name].js',
        library: 'FD',
        libraryTarget: 'umd'
      }
};
```

官方文档：webpack中libraries的用法

在entry中有一个是ts文件其他是js文件 output中如何表示

如果在 Webpack 的 `entry` 中有一个或多个 TypeScript 文件（`.ts` 文件）以及其它 JavaScript 文件（`.js` 文件），则你需要在 Webpack 的配置中将 TypeScript 文件转换成 JavaScript 文件，以便能够被 Webpack 处理。

你可以使用 `ts-loader` 来将 TypeScript 文件转换成 JavaScript 文件。以下是一个示例 Webpack 配置文件，该文件将一个名为 `app` 的 TypeScript 文件和一个名为 `vendor` 的 JavaScript 文件打包到一个名为 `bundle.js` 的文件中：

```
const path = require('path');

module.exports = {
  entry: {
    app: './src/app.ts',
    vendor: './src/vendor.js'
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.ts$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: ['.ts', '.js']
  }
};
```

在上面的配置中，我们通过 `entry` 属性指定了两个入口文件：一个是 `./src/app.ts`，一个是 `./src/vendor.js`。在 `module` 配置中，我们指定了一个 `ts-loader` 规则，用于将所有匹配 `.ts` 后缀的文件转换成 JavaScript 文件。最后，在 `resolve` 配置中，我们指定了模块的扩展名，包括 `.ts` 和 `.js`。

对于输出（`output`）配置，你可以直接将 JavaScript 文件和 TypeScript 文件打包在一起，因为它们最终都会被转换成 JavaScript 文件。所以在 `output` 配置中，你只需要指定输出文件的名称和路径即可。在上面的示例中，我们将输出文件的名称设置为 `bundle.js`，并将其输出到 `dist` 目录中。

注意：如果你在 TypeScript 文件中使用了 ES6 模块语法，例如 `import` 和 `export`，那么你需要在 Webpack 的配置中启用 `babel-loader` 来将 ES6 模块语法转换成 CommonJS 模块语法。

tsconfig 应该如何配置 如何配合webpack使用

`tsconfig.json` 是 TypeScript 的配置文件，用于配置 TypeScript 编译器的选项。你可以在这个文件中指定编译选项，如编译输出目录、使用的 ECMAScript 版本、是否启用严格模式等等。

以下是一些常见的 `tsconfig.json` 配置选项：

```
{
  "compilerOptions": {
    "target": "es6", //  JavaScript
    "module": "commonjs", //
    "outDir": "./dist", //
    "strict": true, //
    "esModuleInterop": true, //  CommonJS  ES6
    "allowSyntheticDefaultImports": true, //
    "sourceMap": true, //  source map
    "declaration": true, //  (*.d.ts)
    "resolveJsonModule": true, //  .json
    "lib": ["es6", "dom"] //
  },
  "include": ["src/**/*"], //
  "exclude": ["node_modules"] //
}
```

这是一个基本的配置示例，你可以根据自己的需要进行调整。通常情况下，你只需要配置 `compilerOptions`，而 `include` 和 `exclude` 可以根据需要进行调整。

在 webpack 中使用 TypeScript 时，可以将 `ts-loader` 的 `options` 配置选项中的 `configFile` 属性指向 `tsconfig.json` 文件的路径。例如：

```
{
  test: /\.ts$/,
  use: {
    loader: 'ts-loader',
    options: {
      configFile: 'tsconfig.json'
    }
  }
}
```

这样，webpack 就会根据 `tsconfig.json` 文件中的配置来编译 TypeScript 文件了。

typeRoots 是什么意思

`typeRoots` 是 TypeScript 的一个编译选项，用于指定自定义的类型声明文件的路径。

默认情况下，TypeScript 会在 `node_modules/@types` 文件夹中查找类型声明文件。如果你想使用自定义的类型声明文件，可以在 `tsconfig.json` 文件中配置 `typeRoots` 选项。该选项可以是一个数组，每个元素是一个文件夹路径，用于指定类型声明文件的存放路径。例如：

```
{
  "compilerOptions": {
    "typeRoots": ["./typings"]
  }
}
```

上述配置会告诉 TypeScript 编译器，在项目根目录下的 `./typings` 文件夹中查找类型声明文件。如果该文件夹中存在与项目中使用的模块对应的类型声明文件，TypeScript 编译器会自动使用这些声明文件。

可以将 `typeRoots` 和 `types` 选项一起使用，以确保 TypeScript 编译器能够找到项目中使用的所有类型声明文件。如果你使用了第三方库，并且该库没有提供类型声明文件，可以将自己编写的类型声明文件存放在 `typeRoots` 指定的文件夹中，从而告诉 TypeScript 编译器在编译过程中使用这些类型声明文件。

noEmit是什么意思

`noEmit` 是 TypeScript 的一个编译选项，用于指示 TypeScript 编译器是否生成 JavaScript 代码。

默认情况下，TypeScript 编译器会将 TypeScript 代码编译成 JavaScript 代码，并将其保存到指定的输出目录中。如果你希望 TypeScript 编译器不生成 JavaScript 代码，可以将 `noEmit` 选项设置为 `true`。例如：

```
{
  "compilerOptions": {
    "noEmit": true
  }
}
```

上述配置会告诉 TypeScript 编译器不生成任何 JavaScript 代码。这通常用于在构建过程中进行类型检查或者生成类型声明文件，而不需要生成 JavaScript 代码。

需要注意的是，如果将 `noEmit` 设置为 `true`，则其他与代码生成相关的编译选项将会被忽略，例如 `outDir`、`outFile` 等。同时，如果在命令行中使用 `--watch` 选项进行实时编译，`noEmit` 选项将不会生效，编译器仍会生成 JavaScript 代码。

总之，`noEmit` 选项可以帮助你控制 TypeScript 编译器是否生成 JavaScript 代码，从而使得 TypeScript 编译更加灵活和高效。

不想将TS的代码编译成js，想让编译后的代码存在ts与js两种文件

如果你希望将 TypeScript 代码编译成 JavaScript 代码，并且同时保留 TypeScript 代码的副本，可以通过 `outDir` 选项指定输出目录，然后将 TypeScript 代码和生成的 JavaScript 代码保存在该目录下。例如：

```
{
  "compilerOptions": {
    "outDir": "./dist",
    "module": "commonjs",
    "target": "es5",
    "lib": ["es6", "dom"],
    "sourceMap": true,
    "declaration": true,
    "declarationMap": true
  }
}
```

上述配置会将编译后的 JavaScript 代码和类型声明文件保存在 `./dist` 文件夹下。同时，`sourceMap` 选项会生成 `.js.map` 文件，用于调试和源码映射，`declaration` 选项会生成 `.d.ts` 类型声明文件。

在使用上述配置进行编译后，你可以同时拥有 TypeScript 代码和生成的 JavaScript 代码，从而既能在开发阶段使用 TypeScript 进行调试，也能在生产环境中使用 JavaScript 运行代码。