# Typescript

## TypeScript 中如何定义和使用泛型？

TypeScript 中的泛型提供了一种通过编写可处理不同类型的函数、类和接口来编写可重用且灵活的代码的方法。

泛型可以用来实现类型安全，提高代码的可读性和可维护性。

## interface 和 type 都用于定义类型，但它们有一些区别

### 相同点

1. 都可以描述一个对象或者函数
2. 都允许相互拓展属性，但是语法不同

interface extends type 关键词：extends

type extends type 关键词：&

type extends interface 关键词：&

总结只要是interface扩展属性无论是interface还是type都使用关键字extends;type的扩展属性无论是type还是interface都是使用关键字&

### 不同点

1. 语法差异：

interface 关键字用于声明接口，使用 interface 可以定义对象的形状、函数的签名等。

type 关键字用于声明类型别名，可以给一个类型起一个新的名字。

2. 合并能力：

interface 具有合并能力，即同名的接口会自动合并为一个接口，合并后的接口会继承所有同名接口的成员。

type 不具有合并能力，同名的类型别名会报错。

3. 实现能力：

interface 可以被类实现（使用 implements），用于类与接口的约束关系。

type 不能被类实现，它只是给类型起别名，无法用于类与类型的约束关系。

4. 扩展能力：

interface 可以通过 extends 关键字扩展其他接口或类，实现接口的继承。

type 可以使用交叉类型（&）或联合类型（|）组合多个类型来创建新的类型。

## JS 项目逐步迁移到 TS 项目，该如何做【热度: 870】

**关键词：TS 项目迁移**

在 JavaScript 项目迁移到 TypeScript 的过程中确实会出现大量 JS 和 TS 文件共存的情况。要配置项目以使它们兼容并顺利运行，你需要进行以下设置：

### 1. 初始化 TypeScript 配置

首先，创建tsconfig.json文件来配置 TypeScript 编译选项。可以通过运行npx tsc --init来自动生成一个基础的配置文件。为了使 JavaScript 和 TypeScript 文件共存，你需要确保tsconfig.json中包含以下配置：

{

"compilerOptions": {

"allowJs": true, // 允许编译JavaScript文件

"checkJs": false, // 禁用对JS文件的检查，使迁移更加平滑

"outDir": "./dist", // 指定输出目录

"target": "es5", // 目标编译版本

"module": "commonjs", // 模块化标准，根据项目情况调整

"strict": false, // 可以开始时设置为false，逐步提高严格性

"esModuleInterop": true

},

"include": [

"src/\*\*/\*" // 指定项目源代码目录

]

}

### 2. 配置构建工具

如果你的项目是基于 React 的，并且你希望在迁移过程中同时使用 TypeScript 和 Babel 来处理 JSX 和最新的 JavaScript 特性，以下是对上面 Webpack 示例的补充，以支持这些需求：

#### 2.1. 安装必要的包

首先，你需要安装与 React、TypeScript、Babel 相关的 npm 包：

npm install --save react react-dom

npm install --save-dev typescript @types/react @types/react-dom

npm install --save-dev webpack webpack-cli webpack-dev-server

npm install --save-dev ts-loader @babel/core @babel/preset-env @babel/preset-react babel-loader

#### 2.2. 配置 Babel

创建或更新项目根目录下的.babelrc或babel.config.json文件，以包含 React 的预设和对最新 ECMAScript 特性的支持：

{

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

}

如果你使用 TypeScript，你还可以在 Babel 配置中添加@babel/preset-typescript，这样 Babel 也可以直接处理.ts和.tsx文件：

{

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

}

#### 2.3. 配置 Webpack

更新webpack.config.js配置以使用babel-loader，并确保它能够正确处理.js和.jsx以及.ts和.tsx文件：

const path = require("path");

module.exports = {

entry: "./src/index.tsx", // 假设你的入口文件是一个TypeScript文件

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js",

},

resolve: {

// 添加.ts 和 .tsx 作为解析扩展名，确保导入时可以省略扩展名

extensions: [".tsx", ".ts", ".jsx", ".js"],

},

module: {

rules: [

{

test: /\.(ts|js)x?$/, // 同时匹配 TS(TSX) 和 JS(JSX) 文件

exclude: /node\_modules/,

use: {

loader: "babel-loader", // 使用Babel加载器处理

options: {

// 在此传递Babel预设也是可行的，但最好在Babel配置文件中统一配置

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

},

},

},

],

},

// 如果你需要的话，加入source map支持

devtool: "source-map",

// 配置 webpack-dev-server

devServer: {

contentBase: "./dist",

hot: true,

},

};

通过上述配置，Webpack 将能够正确地处理你的 React 项目中的.js、.jsx、.ts和.tsx文件。Babel 会负责转译 JSX 和 TypeScript，而 Webpack 会负责打包它们。

### 3. 逐步迁移

开始逐步将.js文件重命名为.ts文件，并解决任何类型错误。这可以逐个文件进行，以避免项目变得不可管理。一般建议先从项目的底层（即不依赖其他文件或依赖较少的文件）开始迁移，逐步向上。

### 4. Linting

为了保证代码质量，在项目中配置 ESLint 是个好主意。如果还没有配置 ESLint，你可以如下安装:

npm install eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin --save-dev

然后，在.eslintrc文件中配置 ESLint 来支持 TS：

{

"parser": "@typescript-eslint/parser",

"extends": ["eslint:recommended", "plugin:@typescript-eslint/recommended"],

"parserOptions": {

"ecmaVersion": 2020,

"sourceType": "module"

},

"rules": {

// 自定义规则

}

}

### 5. 运行和测试

确保你的运行和测试脚本兼容 TS 文件。可能需要配置或更新一些依赖，比如使用ts-node而不是node来运行 TS 文件，或者更新 Jest 配置以支持 TS。

通过上述步骤，你的项目应该能够在迁移到 TypeScript 的同时继续正常运行和构建。记住，这是一个逐步的过程，不需要急于一时完成所有迁移。