**typeScript**

1. **简介：**

TypeScript是一种由微软开发的自由和[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90)的编程语言。它是[JavaScript](https://baike.baidu.com/item/JavaScript)的一个超集，而且本质上向这个语言添加了可选的静态类型和基于类的[面向对象编程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%BC%96%E7%A8%8B)。[安德斯·海尔斯伯格](https://baike.baidu.com/item/%E5%AE%89%E5%BE%B7%E6%96%AF%C2%B7%E6%B5%B7%E5%B0%94%E6%96%AF%E4%BC%AF%E6%A0%BC)，[C#](https://baike.baidu.com/item/C%23)的首席架构师，已工作于TypeScript的开发。2012年十月份，微软发布了首个公开版本的TypeScript，2013年6月19日，在经历了一个预览版之后微软正式发布了正式版TypeScript 0.9，向未来的TypeScript 1.0版迈进了很大一步。

1. **基本使用：**
2. **typeScript全局安装进行文件的编译**
3. **安装typescript:**

cnpm install -g typescript

1. **编写第一个**greeter.**ts文件：**

class Student {

firstName : string;

lastName : string;

constructor(fiestName : string, lastName : string) {

this.firstName = fiestName;

this.lastName = lastName;

}

greeter() {

return "Hello，您好" + this.firstName + " " + this.lastName;

}

}

var user = new Student("王", "小明");

// document.body.innerHTML = user.greeter();

document.body.innerHTML = user.greeter();

1. **进行编译：**

**tsc greeter.ts**

1. **定义接口：（类型的定义可以在全局进行）**

interface Person{

name:string,

age:number

}

function greeter(person:Person){

return "Hello," +person.name+"-"+person.age

}

var user={name:"zhangsan",age:58}

document.body.innerHTML=greeter(user)

1. **定义类：（class，contructor，public）**

class Student {

fullName: string;

constructor(public firstName, public middleInitial, public lastName) {

this.fullName = firstName + " " + middleInitial + " " + lastName;

}

}

var user = new Student("Jane", "M.", "User");

1. **使用gulp进行自动化的构建：**
2. **基础功能：**

var gulp = require("gulp");

var ts = require("gulp-typescript"); //typescript基础包

var tsProject = ts.createProject("tsconfig.json");

gulp.task("default", function () {

return tsProject.src()

.pipe(tsProject())

.js.pipe(gulp.dest("dist"));

});

1. **进行模块化打包**

// 使用babel和es2015进行文件的打包和转码

var gulp = require("gulp");

var browserify = require("browserify"); //模块化打包

var source = require('vinyl-source-stream'); //改变文件名字

var tsify = require("tsify"); //用此包代替gulp-typescript进行解析

var watchify = require("watchify"); //监听服务

var uglify = require('gulp-uglify'); //js文件的打包压缩

var sourcemaps = require('gulp-sourcemaps'); //形成sourcemap方便调试

var buffer = require('vinyl-buffer');//帮助形成sourcemap的包

var webserver = require('gulp-webserver'); //web服务热启动

var paths = {

pages: ['src/\*.html']

};

gulp.task("copy-html", function () {

return gulp.src(paths.pages)

.pipe(gulp.dest("dist"));

});

// 创建监听服务

var watchedBrowserify =watchify(browserify({ //监听服务

basedir: '.',

debug: true,

entries: ['src/main.ts'],

cache: {},

packageCache: {}

}).plugin(tsify)

)

// 压缩等操作

var build=function(){

return watchedBrowserify

.bundle() //watchedBrowserify自带方法

.pipe(source('bundle.js'))

// 下面进行代码压缩与形成sourcemap

.pipe(buffer())

.pipe(sourcemaps.init({loadMaps: true}))

.pipe(uglify())

.pipe(sourcemaps.write('./'))

.pipe(gulp.dest("dist"));

}

gulp.task("build",["copy-html"],build)

gulp.task("server",["build"],function(){

gulp.src('./dist')

.pipe(webserver({

// livereload: true,

directoryListing: true,

open: "index.html"

}))

})

gulp.task("default", ["server"]);

watchedBrowserify.on("update",build) //监听更新任务，重新进行js的打包

watchedBrowserify.on("log",build)

1. **使用webpack手动搭建ts项目：（模块化开发）**
2. **需要的依赖有：**

"devDependencies": {

"html-webpack-plugin": "^2.30.1",

"source-map-loader": "^0.2.3",

"ts-loader": "^3.1.1",

"typescript": "^2.6.1",

"webpack": "^3.8.1",

"webpack-dev-server": "^2.9.4"

}

1. **配置tsconfig.json文件：**

{

"compilerOptions": {

"outDir": "./dist", //文件输出目标文件夹，和webpack的output路径一致

"allowJs": true,

"target": "es5" //解析成es5

},

"include": [

"./src/\*\*/\*" //需要解析的文件夹

]

}

1. **配置webpack.config.js文件：**

const HtmlWebpackPlugin = require("html-webpack-plugin")

var webpack=require("webpack")

const path=require("path")

module.exports = {

entry: "./src/index.ts",

output: {

filename: "./dist/bundle.js",

},

devtool: "source-map",

resolve: {

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

},

devServer: {

contentBase: path.join(\_\_dirname, "dist"),

compress: true,

port: 9000,

hot: true,

inline: true,

watchContentBase: true,

host: 'localhost',

setup: function(app) {

app.get('/some/path', function(req, res) {

console.log(11)

res.json({ custom: 'response' });

});

}

},

module: {

rules: [

{ test: /\.tsx?$/, loader: "ts-loader" },

{ test: /\.js$/,enforce: "pre", loader: "source-map-loader" }

]

},

plugins:[

new HtmlWebpackPlugin({

// filename: 'test.html',

template: "./index.html"

}),

new webpack.HotModuleReplacementPlugin()

]

};

1. **typeScript基本规范：**
2. **基础类型：**
3. **布尔值：**

let isDone: boolean = false;

1. **数字：全部为浮点型**

let decLiteral: number = 6;

1. **字符串：**

let name: string = `Gene`;

1. **数组：**

let list: number[] = [1, 2, 3];

数组泛型：

let list: Array<number> = [1, 2, 3];

1. **元组（tuple）：**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

let x: [string, number];

console.log(x[0].substr(1));

x[3] = 'world'; //设置值成功

x[6] = true; //设置值失败，这里越界元素会使用联合类型代替；

1. **枚举：（js数据类型的补充）**

默认情况下，从0开始为元素编号。 你也可以手动的指定成员的数值

enum Color {Red=1, Green, Blue}

let c: Color = Color.Green;

let colorName: string = Color[2];

**console.log(c) //2**

**console.log(colorName) //green**

1. **any(在不确定类型的情况下，使用此字段来定义值)**

let notSure: any = 4;

notSure = "maybe a string instead";

notSure = false;

let list: any[] = [1, true, "free"];

list[1] = 100;

**注：any跟object有一定的区别，两者都可以进行任意类型的赋值，但是object却不能调用object对象上任意方法；**

1. **void:** 与any类型相反，它表示没有任何类型,一个函数没有返回值时，声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined和null：

function warnUser(): void {

alert("This is my warning message");

}

let unusable: void = undefined;

1. **Null 和 Undefined：**undefined和null两者各自有自己的类型分别叫做undefined和null默认情况下null和undefined是所有类型的子类型。 就是说你可以把 null和undefined赋值给number类型的变量：

let u: undefined = undefined;

let n: null = null;

1. **never：**never类型表示的是那些永不存在的值的类型，never类型是任何类型的子类型，也可以赋值给任何类型，*没有*类型是never的子类型或可以赋值给never类型（除了never本身之外）。 即使 any也不可以赋值给never

// 返回never的函数必须存在无法达到的终点 function error(message: string): never { throw new Error(message); } // 推断的返回值类型为never function fail() { return error("Something failed"); } // 返回never的函数必须存在无法达到的终点 function infiniteLoop(): never { while (true) { } }

1. **断言：**类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。 它没有运行时的影响，只是在编译阶段起作用。 TypeScript会假设你，程序员，已经进行了必须的检查。下面两种形式：

let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;

let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;

1. **变量声明：**
2. **解构过程中，如果指定相应模式的类型，如下：**

let {a, b}: {a: string, b: number} = o;

**默认值：**

function keepWholeObject(wholeObject: { a: string, b?: number }) {

let { a, b = 1001 } = wholeObject;

}

1. **函数声明：**

type C = { a: string, b?: number }

function f({ a, b }: C): void {

// ... }

1. **接口：使用interface声明接口**
2. **普通接口**

interface LabelledValue {

label: string; //定义必填选项，且类型必须为string

}

1. **可选接口：**可选属性的好处之一是可以对可能存在的属性进行预定义，好处之二是可以捕获引用了不存在的属性时的错误。 比如，我们故意将 createSquare里的color属性名拼错，就会得到一个错误提示

interface SquareConfig {

color?: string;

width?: number;

}

function createSquare(config: SquareConfig): { color: string; area: number } {

let newSquare = {color: "white", area: 100};

if (config.color) {

// Error: Property 'clor' does not exist on type 'SquareConfig'

newSquare.color = config.clor;

}

if (config.width) {

newSquare.area = config.width \* config.width;

}

return newSquare;

}

let mySquare = createSquare({color: "black"})

1. **只读属性：**一些对象属性只能在对象刚刚创建的时候修改其值。 你可以在属性名前用 readonly来指定只读属性

interface Point {

readonly x: number;

readonly y: number;

}

数组的只读：

let a: number[] = [1, 2, 3, 4];

let ro: ReadonlyArray<number> = a;

ro[0] = 12; // error!

ro.push(5); // error! ro.length = 100; // error!

a = ro; // error! //整个ReadonlyArray赋值到一个普通数组也是不可以的

a = ro as number[] //使用断言重写；

1. **额外属性检查：**对象字面量会被特殊对待而且会经过 *额外属性检查*，当将它们赋值给变量或作为参数传递的时候。 如果一个对象字面量存在任何“目标类型”不包含的属性时，你会得到一个错误：

interface SquareConfig {

color?: string;

width?: number;

}

function createSquare(config: SquareConfig): { color: string; area: number } {

// ...

}

let mySquare = createSquare({ colour: "red", width: 100 });

**断言解决额外属性的检查：**

let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);

//如果为额外属性定义一定的类型，采用如下写法：

interface SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

**跳过检查：**如果跳过这些检查的方式，将这个对象赋值给一个另一个变量： 因为squareOptions不会经过额外属性检查，所以编译器不会报错

let squareOptions = { colour: "red", width: 100 };

let mySquare = createSquare(squareOptions);

1. **函数类型：** 为了使用接口表示函数类型，我们需要给接口定义一个调用签名。 它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型：

interface SearchFunc {

(source: string, subString: string): boolean;

}

let mySearch: SearchFunc;

mySearch = function(source: string, subString: string) {

let result = source.search(subString);

return result > -1;

}

1. **可索引的类型：**描述那些能够“通过索引得到”的类型，描述了对象索引的类型，还有相应的索引返回值类型

interface StringArray {

[index: number]: string;

}

let myArray: StringArray;

myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];

**如下错误的定义：**

interface NumberDictionary {

[index: string]: number;

length: number; // 可以，length是number类型

name: string // 错误，`name`的类型与索引类型返回值的类型不匹配

}

**将索引签名设置为只读：**

interface ReadonlyStringArray {

readonly [index: number]: string;

}

let myArray: ReadonlyStringArray = ["Alice", "Bob"];

myArray[2] = "Mallory"; // error!

1. **类类型：**

interface ClockInterface {

currentTime: Date;

setTime(d: Date);

}

class Clock implements ClockInterface {

currentTime: Date;

setTime(d: Date) {

this.currentTime = d;

}

constructor(h: number, m: number) { }

}

1. **继承接口**

interface Shape {

color: string;

}

interface PenStroke {

penWidth: number;

}

interface Square extends Shape, PenStroke {

sideLength: number;

}

let square = <Square>{};

square.color = "blue";

square.sideLength = 10;

square.penWidth = 5.0;

1. **混合类型：**一个对象可以同时做为函数和对象使用，并带有额外的属性。

interface Counter {

(start: number): string;

interval: number;

reset(): void;

}

function getCounter(): Counter {

let counter = <Counter>function (start: number) { };

counter.interval = 123;

counter.reset = function () { };

return counter;

}

let c = getCounter();

c(10);

c.reset();

c.interval = 5.0;