目标

1. 企业级 ES6 核心技能详解与实践
2. 企业级Typescript 核心概念详解与实践
3. 企业级Typescript 工程配置详解与实践

企业级 ES6 核心技能详解与实践

ES6， 全称 ECMAScript 6.0 ，是 JavaScript 的下一个版本标准，2015.06 发版。ES6 主要是为了解决 ES5 的先天不足，比如 JavaScript 里并没有类的概念，但是目前浏览器的 JavaScript 是 ES5 版本，大多数高版本的浏览器也支持 ES6，不过只实现了 ES6 的部分特性和功能。

1. 块级绑定

ES2015(ES6) 新增加了两个重要的 JavaScript 关键字: **let** 和 **const**。let 声明的变量只在 let 命令所在的代码块内有效。const 声明一个只读的常量，一旦声明，常量的值就不能改变。

基本例子

```
{
    let a = 0;
    var b = 1;
    const c = 2;
}
a; // 报错，let 是在代码块内有效，var 是在全局范围内有效
b; // 1
c; // 报错，const 是在代码块内有效，var 是在全局范围内有效
```

let只能声明一次 var 可以声明多次

```
let a = 1;
let a = 2;
var b = 3;
var b = 4;
a; // Identifier 'a' has already been declared
b; // 4
```

经典的for循环闭包问题可以使用let 解决

```
for (var i = 0; i < 10; i++) {
  setTimeout(function(){
    console.log(i);
  })
}
// 输出十个 10
for (let j = 0; j < 10; j++) {
  setTimeout(function(){
    console.log(j);
  })
}
// 输出 0123456789
```

变量 i 是用 var 声明的，在全局范围内有效，所以全局中只有一个变量 i, 每次循环时，setTimeout 定时器里面的 i 指的是全局变量 i ，而循环里的十个 setTimeout 是在循环结束后才执行，所以此时的 i 都是 10。变量 j 是用 let 声明的，当前的 j 只在本轮循环中有效，每次循环的 j 其实都是一个新的变量，所以 setTimeout 定时器里面的 j 其实是不同的变量。

不存在变量提升

let 不存在变量提升，var 会变量提升:

```
console.log(a);  //ReferenceError: a is not defined
let a = "apple";

console.log(b);  //undefined
var b = "banana";
```

暂时性死区

```
var A = "a";
if(true){
  console.log(A);  // ReferenceError: A is not defined
  const A = "3";
}
```

ES6 明确规定，代码块内如果存在 let 或者 const，代码块会对这些命令声明的变量从块的开始就形成一个封闭作用域。代码块内，在声明变量 PI 之前使用它会报错。

全局块绑定

let 与 const 不同于 var 的另一个方面是在全局作用域上的表现。当在全局作用域上使 用 var 时，它会创建一个新的全局变量，并成为全局对象（在浏览器中是 window ）的一 个属性，然而若你在全局作用域上使用 let 或 const ，虽然在全局作用域上会创建新的绑定，但不 会有任何属性被添加到全局对象上。

```
// 在浏览器中
var RegExp = "Hello!"; console.log(window.RegExp); // "Hello!"
var ncz = "Hi!"; console.log(window.ncz); // "Hi!"
```

```
// 在浏览器中
let RegExp = "Hello!"; console.log(RegExp); // "Hello!"
console.log(window.RegExp === RegExp); // false
const ncz = "Hi!"; console.log(ncz); // "Hi!" console.log("ncz" in window); //
false
```

块级绑定的最佳实践

在 ES6 的发展阶段，被广泛认可的变量声明方式是：默认情况下应当使用 let 而不是 var 。对于多数 JS 开发者来说， let 的行为方式正是 var 本应有的方式，因此直接用 let 替代 var 更符合逻辑。在这种情况下，你应当对需要受到保护的变量使用 const 。 然而，随着更多的开发者迁移到 ES6 上，一种替代方案变得更为流行，那就是在默认情况下 使用 const 、并且只在知道变量值需要被更改的情况下才使用 let 。其理论依据是大部分 变量在初始化之后都不应当被修改，因为预期外的改动是 bug 的源头之一。这种理念有着足 够强大的吸引力，在你采用 ES6 之后是值得在代码中照此进行探索实践的。

2. 解构：更方便的数据访问

解构赋值是对赋值运算符的扩展，是一种针对数组或者对象进行模式匹配，然后对其中的变量进行赋值。

在代码书写上简洁且易读，语义更加清晰明了；也方便了复杂对象中数据字段获取。在解构中，有下面两部分参与：解构的源，解构赋值表达式的右边部分。解构的目标，解构赋值表达式的左边部分。

数组的解构

```
// 基本
let [a, b, c] = [1, 2, 3];
// a = 1
// b = 2
// c = 3
// 嵌套
let [a, [[b], c]] = [1, [[2], 3]];
// a = 1
// b = 2
// c = 3
// 剩余参数
let [a, ...b] = [1, 2, 3];
//a = 1
//b = [2, 3]
```

对象的解构

```
// 基本
let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
// foo = 'aaa'
// bar = 'bbb'

let { baz : foo } = { baz : 'ddd' };
// foo = 'ddd'
// 剩余运算符
let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40};
// a = 10
// b = 20
// rest = {c: 30, d: 40}
// 解构默认值
let {a = 10, b = 5} = {a: 3};
// a = 3; b = 5;
let {a: aa = 10, b: bb = 5} = {a: 3};
// aa = 3; bb = 5;
```

参数解构

解构还有一个特别有用的场景，即在传递函数参数时。当 JS 的函数接收大量可选参数时，一 个常用模式是创建一个 options 对象，其中包含了附加的参数，就像这样：

```
function setObj(name, type, { age = 123 }) {
    console.log(name, type, age);
}
   setObj(1, 1, { age: 345 }); // 正常
// setObj(1, 1);// 报错
```

参数解构有一个怪异点：默认情况下调用函数时未给参数解构传值会抛出错误。

若你让解构的参数作为必选参数，那么上述行为并不会令人困扰。但若你要求它是可选的，

可以给解构的参数提供默认值来处理这种行为，就像这样：

```
function setObj(name, type, { age = 123 } = {}) {
    console.log(name, type, age);
}
//   setObj(1, 1, { age: 345 });
setObj(1, 1);
```

3.符号-Symbol

ES6 引入了一种新的原始数据类型 Symbol ，表示独一无二的值，最大的用法是用来定义对象的唯一属性名。

Symbol 函数栈不能用 new 命令，因为 Symbol 是原始数据类型，不是对象。可以接受一个字符串作为参数，为新创建的 Symbol 提供描述，用来显示在控制台或者作为字符串的时候使用，便于区分。

```
let sy = Symbol("kk");
console.log(sy);   // Symbol(kk)
typeof(sy);        // "symbol"

// 相同参数 Symbol() 返回的值不相等
let sy1 = Symbol("kk");
sy === sy1;       // false
```

使用场景

作为属性名

```
let sy = Symbol("key");
let syObject = {};
syObject[sy] = "kk";
console.log(syObject);    // {Symbol(key): "kk"}
```

定义常量

```
const RED = Symbol("red");
const YELLOW = Symbol("yellow");
const BLUE = Symbol("blue");
```

在不同代码中共享symbol值

Symbol.for() 类似单例模式，首先会在全局搜索被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值，如果有即返回该 Symbol 值，若没有则新建并返回一个以该字符串参数为名称的 Symbol 值，并登记在全局环境中供搜索。

```
let red = Symbol("Red"); // 不是注册在全局中的
let red1 = Symbol.for("Red");
red === red1;      // false
let red2 = Symbol.for("Red");
red1 === red2;     // true
```

利用Symbol.keyFor 返回一个已登记的 Symbol 类型值的 key

```
let red1 = Symbol.for("Red");
Symbol.keyFor(red1);    // "Red"
```

4. 字符串

模板字符串相当于加强版的字符串，用反引号 `,除了作为普通字符串，还可以用来定义多行字符串，还可以在字符串中加入变量和表达式。

基本用法

```
let string = `Hello'\n'world`;
console.log(string);
// "Hello'
// 'world"
```

多行字符串

```
let string1 =  `Hey,
can you stop angry now?`;
console.log(string1);
// Hey,
// can you stop angry now?
```

字符串插入变量和表达式。

变量名写在 ${} 中，${} 中可以放入 JavaScript 表达式。

```
let name = "Thomas";
let age = 25;
let info = `My Name is ${name},I am ${age+1} years old next year.`
console.log(info);
// My Name is Mike,I am 28 years old next year.
```

5. 对象

对象字面量

属性简写

ES6允许对象的属性直接写变量，这时候属性名是变量名，属性值是变量值。

```
const age = 12;
const name = "Thomas";
const person = {age, name};
person    //{age: 12, name: "Amy"}
//等同于
const person = {age: age, name: name}
```

方法名简写

```
const person = {
  sayHello(){
    console.log("Hello");
  }
}
person.sayHello();  //"Hello"
//等同于
const person = {
  sayHello:function(){
    console.log("Hello");
  }
}
person.sayHello();//"Hello"
```

属性名表达式

ES6允许用表达式作为属性名，但是一定要将表达式放在方括号内。

```
const obj = {
 ["he"+"llo"](){
   return "Hello";
  }
}
obj.hello();  //"Hello"
```

对象的拓展运算符

拓展运算符（...）用于取出参数对象所有可遍历属性然后拷贝到当前对象。

```
let person = {name: "Thomas", age: 16};
let someone = { ...person };
someone;  //{name: "Thomas", age: 16}
```

可用于合并两个对象

```
let age = {age: 16};
let name = {name: "Thomas"};
let person = {...age, ...name};
person;  //{age: 16, name: "Thomas"}
```

6. 箭头函数

箭头函数提供了一种更加简洁的函数书写方式。基本语法是

```
参数 => 函数体
```

基本用法

```
var f = v => v;
```

注意： 箭头函数体中的 this 对象，是定义函数时的对象，而不是使用函数时的对象。

```
function fn(){
  setTimeout(()=>{
    // 定义时，this 绑定的是 fn 中的 this 对象
    console.log(this.a);
  },0)
}
var a = 30;
// fn 的 this 对象为 {a: 20}
fn.call({a: 20});  // 20
```

不可以作为构造函数，也就是不能使用 new 命令，否则会报错

适合使用的场景

ES6 之前，JavaScript中经常看到 var that= this 这样的代码，为了将外部 this 传递到回调函数中，那么有了箭头函数，就不需要这样做了，直接使用 箭头函数就行。

```
// 回调函数
var Person = {
    'age': 20,
    'sayHello': function () {
      setTimeout(function () {
        console.log(this.age);
      });
    }
};
var age = 30;
Person.sayHello();  // 20

var Person1 = {
    'age': 50,
    'sayHello': function () {
      setTimeout(()=>{
        console.log(this.age);
      });
    }
};
var age = 20;
Person1.sayHello();  // 50
```

7. ES 6 中的 class

在ES6中，class (类)作为对象的模板被引入，可以通过 class 关键字定义类。class 的本质是 function。

它可以看作一个语法糖，让对象原型的写法更加清晰、更像面向对象编程的语法。

声明类

```
class Example {
    constructor(a) {
        this.a = a;
    }
}
```

类实例化

class 的实例化必须通过 new 关键字。

```
let example = new Example()
```

类的继承

通过 extends 实现类的继承。 子类 constructor 方法中必须有 super ， 且必须出现在 this 之前。

```
class Father {
    constructor() {}
}
class Child extends Father {
    constructor() {
        super()
    }
}
let child = new Child();
```

8.ES6 中的模块

在 ES6 前， 实现模块化使用的是 RequireJS 或者 seaJS（分别是基于 AMD 规范的模块化库， 和基于 CMD 规范的模块化库）。ES6 引入了模块化，其设计思想是在编译时就能确定模块的依赖关系，以及输入和输出的变量。

ES6 的模块化分为导出（export） @与导入（import）两个模块。

模块导入导出各种类型的变量，如字符串，数值，函数，类。

methods.js

```
let foo = 1;
let bar = 2;
export { foo, bar };
```

test.js

```
import { foo, bar } from "methods.js";
```

as 用法

```
import { foo as name1 } from "./methods.js";
```

export default 命令

- 在一个文件或模块中，export、import 可以有多个，export default 仅有一个。
- export default 中的 default 是对应的导出接口变量。
- 通过 export 方式导出，在导入时要加{ }，export default 则不需要。
- export default 向外暴露的成员，可以使用任意变量来接收。

```
var a = "My name is Thomas!";
export default a; // 仅有一个
```

另外一个文件

```
import b from "./xxx.js"; // 不需要加{}， 使用任意变量接收
```

9. Promise、async和await

Promise

是异步编程的一种解决方案。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。

Promise 异步操作有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。除了异步操作的结果，任何其他操作都无法改变这个状态。Promise 对象只有：从 pending 变为 fulfilled 和从 pending 变为 rejected 的状态改变。只要处于 fulfilled 和 rejected ，状态就不会再变了即 resolved（已定型）。

```
const p1 = new Promise(function (resolve, reject) {
    resolve('success1');
});
const p2 = new Promise(function (resolve, reject) {
    reject('reject');
});
p1.then(function (value) {
    console.log(value); // success1
});
// 链式调用
p2.catch(function (value) {
    console.log(value); // success3
    return 123;
}).then((res) => console.log(123));
```

Async

async 是 ES7 才有的与异步操作有关的关键字， 配合await，可以让异步代码变同步。

async 函数中可能会有 await 表达式，async 函数执行时，如果遇到 await 就会先暂停执行 ，等到触发的异步操作完成后，恢复 async 函数的执行并返回解析值。await 关键字仅在 async function 中有效。如果在 async function 函数体外使用 await ，你只会得到一个语法错误。

```
function testAwait (x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function helloAsync() {
  var x = await testAwait ("hello world");
  console.log(x);
}
helloAsync ();
// hello world
```

企业级Typescript 核心概念详解与实践

什么是TypeScript

TypeScript 是JavaScript 类型的超集，它可以编译成纯JavaScript. TypeScript可以编译出纯净、 简洁的JavaScript代码，并且可以运行在任何浏览器上、Node.js环境中和任何支持ECMAScript 3（或更高版本）的JavaScript引擎中。

这里需要注意三个要点：

1. 类型检查，TypeScript 会在编译代码时，进行严格的类型检查，这以为这你可以在编码阶段发现潜在的隐患。
2. 语言拓展，Typescript 中会包括ES6包括未来体验中的特性，比如异步操作和装饰器，也会从其它语言借鉴某些特性，比如接口和抽象类。
3. 工具属性，Typescript可以编译成标准的Javascript,这意味着可以在任何的浏览器、操作系统上运行，从这个角度上讲TypeScript更像是一个工具。

类型基础

强类型和弱类型语言

强类型语言： 是一种总是强制类型定义的语言，要求变量的使用要严格符合定义，所有变量都必须先定义后使用,如Java,C++。

弱类型语言： 弱类型语言也称为弱类型定义语言，与强类型定义相反， 数据类型易出错， 如Javscript。

编写你的第一个Typscript程序

1. 使用vscode打开文件夹,初始化文件

```
npm init -y
```

2. 全局安装typescript

```
npm i typescript -g
```

3. 初始化配置文件,配置文件后面详细讲解

```
tsc --init
```

4. 在根目录下新建 src/index.ts,填入

```
let hello:string = "hello"
```

5. 运行

```
tsc ./src/index.ts
```

6. 可以看到 在 src 下生成了一个 index.js 文件，里面内容如下所示

```
var hello = "hello";
```

TypeScript中的数据类型

- Boolean
- Number
- String
- Array
- Function
- Object
- Symbol
- undefined
- null
- void
- any
- never
- 元组
- 枚举
- 高级类型

类型注解

作用：相当于强类型语言中的类型声明

语法：(变量/函数):type

基本类型

```
// 原始类型
let bool: boolean = true
let num: number | undefined | null = 123
let str: string = 'abc'
// str = 123

// 数组
let arr1: number[] = [1, 2, 3]
// 联合类型
let arr2: Array<number | string> = [1, 2, 3, '4']

// 元组，特殊数组
let tuple: [number, string] = [0, '1']
// tuple.push(3)
// console.log(tuple)
// tuple[2]

// 函数
let add = (x: number, y: number) => x + y
// 定义一个函数类型
let myfunc: (x: number, y: number) => number
myfunc = (a, b) => a + b

// 对象
let obj: { x: number, y: number } = { x: 1, y: 2 }
obj.x = 3

// symbol
let s1: symbol = Symbol()
let s2 = Symbol()
// console.log(s1 === s2)
```

```
// undefined, null
let un: undefined = undefined
let nu: null = null
num = undefined
num = null

// void 一个没有返回值的类型
let noReturn = () => {}

// any 没有类型注解
let x
x = 1
x = []
x = () => {}

// never 永远不会有返回值的类型
let error = () => {
    throw new Error('error')
}
let endless = () => {
    while(true) {}
}
```

枚举类型

使用枚举我们可以定义一些带名字的常量。 使用枚举可以清晰地表达意图或创建一组有区别的用例。TypeScript支持数字的和基于字符串的枚举。

```
// 数字枚举
enum Role {
    Reporter = 1,
    Developer,
    Maintainer,
    Owner,
    Guest
}
// console.log(Role.Reporter)
// console.log(Role)

// 字符串枚举
enum Message {
    Success = '恭喜你，成功了',
    Fail = '抱歉，失败了'
}
```

接口

TypeScript的核心原则之一是对值所具有的*结构*进行类型检查。 它有时被称做“鸭式辨型法”或“结构性子类型化”。 在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

接口定义变量

```
interface List {
    readonly id: number;// readony 表示制度属性
```

```
    name: string;
    // [x: string]: any; // 字符串索引签名 任意属性
    age?: number; // ? 表示可选
}
interface Result {
    data: List[]
}
function render(result: Result) {
    result.data.forEach((value) => {
        console.log(value.id, value.name)
    })
}
let result = {
    data: [
        {id: 1, name: 'A', sex: 'male'},
        {id: 2, name: 'B', age: 10}
    ]
}
render(result)
```

接口定义函数

```
// interface Add {
//     (x: number, y: number): number
// }
type Add = (x: number, y: number) => number
let add: Add = (a: number, b: number) => a + b
```

类

传统的JavaScript程序使用函数和基于原型的继承来创建可重用的组件，但对于熟悉使用面向对象方式的程序员来讲就有些棘手，因为他们用的是基于类的继承并且对象是由类构建出来的。 从ECMAScript 2015，也就是ECMAScript 6开始，JavaScript程序员将能够使用基于类的面向对象的方式。 使用TypeScript，我们允许开发者现在就使用这些特性，并且编译后的JavaScript可以在所有主流浏览器和平台上运行，而不需要等到下个JavaScript版本。

```
// 抽象类，只能被继承，不能被实例化
abstract class Animal {
    eat() {
        console.log('eat')
    }
    abstract sleep(): void
}
// let animal = new Animal()

class Dog extends Animal {
    constructor(name: string) {
        super()
        this.name = name
        this.pri()
    }
    public name: string = 'dog' // 共有成员，可以被子类和实例调用
    run() {}
    private pri() {} // 私有成员，不能被子类和实例调用
    protected pro() {}  // 只能在类或者子类中访问
    readonly legs: number = 4 // 只读属性
```

```
    static food: string = '骨头' // 类的静态成员
    sleep() {
        console.log('狗睡了')
    }
}
// console.log(Dog.prototype)
let dog = new Dog('汪汪')
console.log(Dog.food)
dog.eat()
```

泛型

软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。 组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

在像C#和Java这样的语言中，可以使用`泛型`来创建可重用的组件，一个组件可以支持多种类型的数据。 这样用户就可以以自己的数据类型来使用组件。

```
function identity<T>(arg: T): T {
    return arg;
}
let output = identity<string>("myString");  // type of output will be 'string'
```

类型推断

TypeScript里，在有些没有明确指出类型的地方，类型推论会帮助提供类型。如下面的例子

```
let x = 3;
```

变量`x`的类型被推断为数字。 这种推断发生在初始化变量和成员，设置默认参数值和决定函数返回值时。

类型兼容性

TypeScript里的类型兼容性是基于结构子类型的。 结构类型是一种只使用其成员来描述类型的方式。 它正好与名义（nominal）类型形成对比，如下面的例子。

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

高级类型

交叉类型（Intersection Types）

交叉类型是将多个类型合并为一个类型。 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。 例如， `Person & Serializable & Loggable` 同时是 `Person` *和* `Serializable` *和* `Loggable`。 就是说这个类型的对象同时拥有了这三种类型的成员。

我们大多是在混入（mixins）或其它不适合典型面向对象模型的地方看到交叉类型的使用。 （在JavaScript里发生这种情况的场合很多！） 下面是如何创建混入的一个简单例子：

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) { }
}
interface Loggable {
    log(): void;
}
class ConsoleLogger implements Loggable {
    log() {
        // ...
    }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```

联合类型

联合类型表示一个值可以是几种类型之一。 我们用竖线（`|`）分隔每个类型，所以 `number | string | boolean` 表示一个值可以是 `number`， `string`，或 `boolean`。

```
interface Bird {
    fly();
    layEggs();
}

interface Fish {
    swim();
    layEggs();
}

function getSmallPet(): Fish | Bird {
    // ...
}

let pet = getSmallPet();
```

```
pet.layEggs(); // okay
pet.swim();    // errors
```

索引类型（Index types）

使用索引类型，编译器就能够检查使用了动态属性名的代码。 例如，一个常见的JavaScript模式是从对象中选取属性的子集。

```
function pluck(o, names) {
    return names.map(n => o[n]);
}
```

下面是如何在TypeScript里使用此函数，通过 **索引类型查询**和 **索引访问**操作符：

```
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {
  return names.map(n => o[n]);
}

interface Person {
    name: string;
    age: number;
}
let person: Person = {
    name: 'Jarid',
    age: 35
};
let strings: string[] = pluck(person, ['name']); // ok, string[]
```

映射类型

这在JavaScript里经常出现，TypeScript提供了从旧类型中创建新类型的一种方式 — **映射类型**。 在映射类型里，新类型以相同的形式去转换旧类型里每个属性。 例如，你可以令每个属性成为 `readonly` 类型或可选的。 下面是一些例子：

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
}
type Partial<T> = {
    [P in keyof T]?: T[P];
}
```

像下面这样使用：

```
type PersonPartial = Partial<Person>;
type ReadonlyPerson = Readonly<Person>;
```

条件类型

先看一下条件类型是什么

> T extends U ? X : Y
> 上面的类型表示：若 T 能够分配（赋值）给 U，那么类型是 X，否则为 Y，有点类似于JavaScript中的三元条件运算符。

上文说到只有类型系统中给出 充足的条件 之后,它才会根据条件推断出类型结果，如果判断条件不足，则会得到第三种结果，即 推迟 条件判断，等待充足条件。

```
interface Foo {
    A: boolean;
    B: boolean;
}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {
    // 因为 "x" 未知，因此判断条件不足，不能确定条件分支，推迟条件判断直到 "x" 明确，
    // 推迟过程中，"a" 的类型为分支条件类型组成的联合类型，
    // string | number
    let a = f(x);
    // ok
    let b: string | number = a;
}
```

企业级Typescript 工程配置详解与实践

概述

如果一个目录下存在一个`tsconfig.json`文件，那么它意味着这个目录是TypeScript项目的根目录。`tsconfig.json`文件中指定了用来编译这个项目的根文件和编译选项。 一个项目可以通过以下方式之一来编译：

使用tsconfig.json

- 不带任何输入文件的情况下调用`tsc`，编译器会从当前目录开始去查找`tsconfig.json`文件，逐级向上搜索父目录。
- 不带任何输入文件的情况下调用`tsc`，且使用命令行参数`--project`（或`-p`）指定一个包含`tsconfig.json`文件的目录。

当命令行上指定了输入文件时，`tsconfig.json`文件会被忽略。

示例

`tsconfig.json`示例文件:

- 使用`"files"`属性

```
{
    "compilerOptions": {
        "module": "commonjs",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    },
    "files": [
        "core.ts",
        "sys.ts",
```

```
        "types.ts",
        "scanner.ts",
        "parser.ts",
        "utilities.ts",
        "binder.ts",
        "checker.ts",
        "emitter.ts",
        "program.ts",
        "commandLineParser.ts",
        "tsc.ts",
        "diagnosticInformationMap.generated.ts"
    ]
}
```

- 使用`"include"`和`"exclude"`属性

```
{
    "compilerOptions": {
        "module": "system",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```

细节

`"compilerOptions"`可以被忽略，这时编译器会使用默认值。在这里查看完整的编译器选项列表。

`"files"`指定一个包含相对或绝对文件路径的列表。`"include"`和`"exclude"`属性指定一个文件glob匹配模式列表。 支持的glob通配符有：

- `*` 匹配0或多个字符（不包括目录分隔符）
- `?` 匹配一个任意字符（不包括目录分隔符）
- `**/` 递归匹配任意子目录

如果一个glob模式里的某部分只包含`*`或`.*`，那么仅有支持的文件扩展名类型被包含在内（比如默认`.ts`，`.tsx`，和`.d.ts`， 如果`allowJs`设置能`true`还包含`.js`和`.jsx`）。

如果`"files"`和`"include"`都没有被指定，编译器默认包含当前目录和子目录下所有的TypeScript文件（`.ts`, `.d.ts`和`.tsx`），排除在`"exclude"`里指定的文件。JS文件（`.js`和`.jsx`）也被包含进来如果`allowJs`被设置成`true`。 如果指定了`"files"`或`"include"`，编译器会将它们结合一并包含进来。 使用`"outDir"`指定的目录下的文件永远会被编译器排除，除非你明确地使用`"files"`将其包含进来（这时就算用`exclude`指定也没用）。

使用"include"引入的文件可以使用"exclude"属性过滤。 然而，通过"files"属性明确指定的文件却总是会被包含在内，不管"exclude"如何设置。 如果没有特殊指定，"exclude"默认情况下会排除node_modules，bower_components，jspm_packages和``目录。

任何被"files"或"include"指定的文件所引用的文件也会被包含进来。 A.ts引用了B.ts，因此B.ts不能被排除，除非引用它的A.ts在"exclude"列表中。

需要注意编译器不会去引入那些可能做为输出的文件；比如，假设我们包含了index.ts，那么index.d.ts和index.js会被排除在外。 通常来讲，不推荐只有扩展名的不同来区分同目录下的文件。

tsconfig.json文件可以是个空文件，那么所有默认的文件（如上面所述）都会以默认配置选项编译。

在命令行上指定的编译选项会覆盖在tsconfig.json文件里的相应选项。

@types, typeRoots和types

默认所有*可见的*"@types"包会在编译过程中被包含进来。 node_modules/@types文件夹下以及它们子文件夹下的所有包都是*可见的*； 也就是说，./node_modules/@types/，../node_modules/@types/和../../node_modules/@types/等等。

如果指定了typeRoots，*只有*typeRoots下面的包才会被包含进来。 比如：

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

这个配置文件会包含*所有*./typings下面的包，而不包含./node_modules/@types里面的包。

如果指定了types，只有被列出来的包才会被包含进来。 比如：

```
{
   "compilerOptions": {
        "types" : ["node", "lodash", "express"]
   }
}
```

这个tsconfig.json文件将*仅会*包含./node_modules/@types/node，./node_modules/@types/lodash和./node_modules/@types/express。/@types/。 node_modules/@types/*里面的其它包不会被引入进来。

指定"types": []来禁用自动引入@types包。

注意，自动引入只在你使用了全局的声明（相反于模块）时是重要的。 如果你使用import "foo"语句，TypeScript仍然会查找node_modules和node_modules/@types文件夹来获取foo包。

使用extends继承配置

tsconfig.json文件可以利用extends属性从另一个配置文件里继承配置。

extends是tsconfig.json文件里的顶级属性（与compilerOptions，files，include，和exclude一样）。 extends的值是一个字符串，包含指向另一个要继承文件的路径。

在原文件里的配置先被加载，然后被来至继承文件里的配置重写。 如果发现循环引用，则会报错。

来至所继承配置文件的`files`， `include`和`exclude`*覆盖*源配置文件的属性。

配置文件里的相对路径在解析时相对于它所在的文件。

比如：

`configs/base.json`：

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

`tsconfig.json`：

```
{
  "extends": "./configs/base",
  "files": [
    "main.ts",
    "supplemental.ts"
  ]
}
```

`tsconfig.nostrictnull.json`：

```
{
  "extends": "./tsconfig",
  "compilerOptions": {
    "strictNullChecks": false
  }
}
```

`compileOnSave`

在最顶层设置`compileOnSave`标记，可以让IDE在保存文件的时候根据`tsconfig.json`重新生成文件。

```
{
    "compileOnSave": true,
    "compilerOptions": {
        "noImplicitAny" : true
    }
}
```

要想支持这个特性需要Visual Studio 2015， TypeScript1.8.4以上并且安装atom-typescript插件。

编译选项

项	类型	默认值	描述
`--allowJs`	`boolean`	`false`	允许编译javascript文件。
`--allowSyntheticDefaultImports`	`boolean`	`module === "system"` 或设置了 `--esModuleInterop` 且 `module` 不为 `es2015` / `esnext`	允许从没有设置默认导出的模块中默认导入。这并不影响代码的输出，仅为了类型检查。
`--allowUnreachableCode`	`boolean`	`false`	不报告执行不到的代码错误。
`--allowUnusedLabels`	`boolean`	`false`	不报告未使用的标签错误。
`--alwaysStrict`	`boolean`	`false`	以严格模式解析并为每个源文件生成 `"use strict"` 语句
`--baseUrl`	`string`		解析非相对模块名的基准目录。查看 模块解析文档了解详情。
`--charset`	`string`	`"utf8"`	输入文件的字符集。
`--checkJs`	`boolean`	`false`	在 `.js` 文件中报告错误。与 `--allowJs` 配合使用。
`--declaration -d`	`boolean`	`false`	生成相应的 `.d.ts` 文件。
`--declarationDir`	`string`		生成声明文件的输出路径。
`--diagnostics`	`boolean`	`false`	显示诊断信息。
`--disableSizeLimit`	`boolean`	`false`	禁用JavaScript工程体积大小的限制
`--emitBOM`	`boolean`	`false`	在输出文件的开头加入BOM头（UTF-8 Byte Order Mark）。
`--emitDecoratorMetadata` [1]	`boolean`	`false`	给源码里的装饰器声明加上设计类型元数据。查看 issue #2577了解更多信息。
`--experimentalDecorators` [1]	`boolean`	`false`	启用实验性的ES装饰器。
`--extendedDiagnostics`	`boolean`	`false`	显示详细的诊段信息。
`--forceConsistentCasingInFileNames`	`boolean`	`false`	禁止对同一个文件的不一致的引用。
`--help -h`			打印帮助信息。
`--importHelpers`	`string`		从 `tslib` 导入辅助工具函数（比如 `__extends`， `__rest`等）
`--inlineSourceMap`	`boolean`	`false`	生成单个sourcemaps文件，而不是将每sourcemaps生成不同的文件。
`--inlineSources`	`boolean`	`false`	将代码与sourcemaps生成到一个文件中，要求同时设置了 `--inlineSourceMap` 或 `--sourceMap` 属性。
`--init`			初始化TypeScript项目并创建一个 `tsconfig.json` 文件。
`--isolatedModules`	`boolean`	`false`	将每个文件作为单独的模块（与“ts.transpileModule”类似）。
`--jsx`	`string`	`"Preserve"`	在 `.tsx` 文件里支持JSX： `"React"` 或 `"Preserve"`。查看 JSX。
`--jsxFactory`	`string`	`"React.createElement"`	指定生成目标为react JSX时，使用的JSX工厂函数，比如 `React.createElement` 或 `h`。

项	类型	默认值	描述
`--lib`	`string[]`		编译过程中需要引入的库文件的列表。 可能的值为： ▶ `ES5` ▶ `ES6` ▶ `ES2015` ▶ `ES7` ▶ `ES2016` ▶ `ES2017` ▶ `ES2018` ▶ `ESNext` ▶ `DOM` ▶ `DOM.Iterable` ▶ `WebWorker` ▶ `ScriptHost` ▶ `ES2015.Core` ▶ `ES2015.Collection` ▶ `ES2015.Generator` ▶ `ES2015.Iterable` ▶ `ES2015.Promise` ▶ `ES2015.Proxy` ▶ `ES2015.Reflect` ▶ `ES2015.Symbol` ▶ `ES2015.Symbol.WellKnown` ▶ `ES2016.Array.Include` ▶ `ES2017.object` ▶ `ES2017.Intl` ▶ `ES2017.SharedMemory` ▶ `ES2017.String` ▶ `ES2017.TypedArrays` ▶ `ES2018.Intl` ▶ `ES2018.Promise` ▶ `ES2018.RegExp` ▶ `ESNext.AsyncIterable` ▶ `ESNext.Array` ▶ `ESNext.Intl` ▶ `ESNext.Symbol` 注意：如果`--lib`没有指定默认注入的库的列表。默认注入的库为： ▶ 针对于`--target ES5`： `DOM，ES5，ScriptHost` ▶ 针对于`--target ES6`： `DOM，ES6，DOM.Iterable，ScriptHost`
`--listEmittedFiles`	`boolean`	`false`	打印出编译后生成文件的名字。
`--listFiles`	`boolean`	`false`	编译过程中打印文件名。
`--locale`	`string`	*(platform specific)*	显示错误信息时使用的语言，比如：en-us。
`--mapRoot`	`string`		为调试器指定指定sourcemap文件的路径，而不是使用生成时的路径。当 `.map` 文件是在运行时指定的，并不同于 `js` 文件的地址时使用这个标记。指定的路径会嵌入到 `sourceMap` 里告诉调试器到哪里去找它们。
`--maxNodeModuleJsDepth`	`number`	`0`	node_modules依赖的最大搜索深度并加载JavaScript文件。仅适用于 `--allowJs`。
`--module` `-m`	`string`	`target === "ES6" ? "ES6" : "commonjs"`	指定生成哪个模块系统代码： `"None"`， `"CommonJS"`， `"AMD"`， `"System"`， `"UMD"`， `"ES6"`或 `"ES2015"`。 ▶ 只有 `"AMD"`和 `"System"`能和 `--outFile`一起使用。 ▶ `"ES6"`和 `"ES2015"`可使用在目标输出为 `"ES5"`或更低的情况下。
`--moduleResolution`	`string`	`module === "AMD" or "System" or "ES6" ? "Classic" : "Node"`	决定如何处理模块。或者是`"Node"`对于Node.js/io.js，或者是`"Classic"`（默认）。查看模块解析了解详情。
`--newLine`	`string`	*(platform specific)*	当生成文件时指定行结束符: `"crlf"`（windows）或 `"lf"`（unix）。
`--noEmit`	`boolean`	`false`	不生成输出文件。
`--noEmitHelpers`	`boolean`	`false`	不在输出文件中生成用户自定义的帮助函数代码，如 `__extends`。
`--noEmitOnError`	`boolean`	`false`	报错时不生成输出文件。
`--noErrorTruncation`	`boolean`	`false`	不截短错误消息。
`--noFallthroughCasesInSwitch`	`boolean`	`false`	报告switch语句的fallthrough错误。（即，不允许switch的case语句贯穿）
`--noImplicitAny`	`boolean`	`false`	在表达式和声明上有隐含的 `any`类型时报错。
`--noImplicitReturns`	`boolean`	`false`	不是函数的所有返回路径都有返回值时报错。
`--noImplicitThis`	`boolean`	`false`	当 `this`表达式的值为 `any`类型的时候，生成一个错误。
`--noImplicitUseStrict`	`boolean`	`false`	模块输出中不包含 `"use strict"`指令。
`--noLib`	`boolean`	`false`	不包含默认的库文件（ `lib.d.ts`）。

项	类型	默认值	描述
`--noResolve`	`boolean`	`false`	不把 `///`或模块导入的文件加到编译文件列表。
`--noStrictGenericChecks`	`boolean`	`false`	禁用在函数类型里对泛型签名进行严格检查。
`--noUnusedLocals`	`boolean`	`false`	若有未使用的局部变量则抛错。
`--noUnusedParameters`	`boolean`	`false`	若有未使用的参数则抛错。
`--out`	`string`		弃用。使用 `--outFile` 代替。
`--outDir`	`string`		重定向输出目录。
`--outFile`	`string`		将输出文件合并为一个文件。合并的顺序是根据传入编译器的文件顺序和 `///`和 `import`的文件顺序决定的。查看输出文件顺序文件了解详情。
`paths` [2]	`Object`		模块名到基于 `baseUrl`的路径映射的列表。查看 模块解析文档了解详情。
`--preserveConstEnums`	`boolean`	`false`	保留 `const`和 `enum`声明。查看 const enums documentation了解详情。
`--preserveSymlinks`	`boolean`	`false`	不把符号链接解析为其真实路径；将符号链接文件视为真正的文件。
`--preserveWatchOutput`	`boolean`	`false`	保留watch模式下过时的控制台输出。
`--pretty` [1]	`boolean`	`false`	给错误和消息设置样式，使用颜色和上下文。
`--project` `-p`	`string`		编译指定目录下的项目。这个目录应该包含一个 `tsconfig.json`文件来管理编译。查看 tsconfig.json文档了解更多信息。
`--reactNamespace`	`string`	`"React"`	当目标为生成 `"react"` JSX时，指定 `createElement`和 `__spread`的调用对象
`--removeComments`	`boolean`	`false`	删除所有注释，除了以 `/!*`开头的版权信息。
`--rootDir`	`string`	*(common root directory is computed from the list of input files)*	仅用来控制输出的目录结构 `--outDir`。
`rootDirs` [2]	`string[]`		*根（root）*文件夹列表，表示运行时组合工程结构的内容。查看 模块解析文档了解详情。
`--skipDefaultLibCheck`	`boolean`	`false`	忽略 库的默认声明文件的类型检查。
`--skipLibCheck`	`boolean`	`false`	忽略所有的声明文件（ `*.d.ts`）的类型检查。
`--sourceMap`	`boolean`	`false`	生成相应的 `.map`文件。
`--sourceRoot`	`string`		指定TypeScript源文件的路径，以便调试器定位。当TypeScript文件的位置是在运行时指定时使用此标记。路径信息会被加到 `sourceMap`里。
`--strict`	`boolean`	`false`	启用所有严格类型检查选项。 启用 `--strict`相当于启用 `--noImplicitAny`, `--noImplicitThis`, `--alwaysStrict`， `--strictNullChecks`和 `--strictFunctionTypes`和`--strictPropertyInitialization`。
`--strictFunctionTypes`	`boolean`	`false`	禁用函数参数双向协变检查。
`--strictPropertyInitialization`	`boolean`	`false`	确保类的非`undefined`属性已经在构造函数里初始化。若要令此选项生效，需要同时启用`--strictNullChecks`。
`--strictNullChecks`	`boolean`	`false`	在严格的 `null`检查模式下， `null`和 `undefined`值不包含在任何类型里，只允许用它们自己和 `any`来赋值（有个例外，`undefined`可以赋值到 `void`）。
`--stripInternal` [1]	`boolean`	`false`	不对具有 `/** @internal */` JSDoc注解的代码生成代码。

项	类型	默认值	描述
`--suppressExcessPropertyErrors` [1]	`boolean`	`false`	阻止对对象字面量的额外属性检查。
`--suppressImplicitAnyIndexErrors`	`boolean`	`false`	阻止 `--noImplicitAny` 对缺少索引签名的索引对象报错。查看 issue #1232了解详情。
`--target` `-t`	`string`	`"ES3"`	指定ECMAScript目标版本 `"ES3"`（默认），`"ES5"`，`"ES6"`/`"ES2015"`，`"ES2016"`，`"ES2017"`或`"ESNext"`。 注意：`"ESNext"`最新的生成目标列表为 ES proposed features
`--traceResolution`	`boolean`	`false`	生成模块解析日志信息
`--types`	`string[]`		要包含的类型声明文件名列表。查看 @types，--typeRoots和--types章节了解详细信息。
`--typeRoots`	`string[]`		要包含的类型声明文件路径列表。查看 @types，--typeRoots和--types章节了解详细信息。
`--version` `-v`			打印编译器版本号。
`--watch` `-w`			在监视模式下运行编译器。会监视输出文件，在它们改变时重新编译。监视文件和目录的具体实现可以通过环境变量进行配置。详情请看配置Watch。

- [1] 这些选项是试验性的。
- [2] 这些选项只能在 `tsconfig.json` 里使用，不能在命令行使用。