TypeScript

基础

目的：限制变量的类型。

分类：类型系统根据类型检查的机制，可分为动态类型和静态类型。

- 动态类型：在运行阶段进行类型检查，如 JS 解释型语言，没有编译阶段。
- 静态类型：在编译阶段进行类型检查，如 TS 在运行之前需要先编译为 JS。

注意：所有合法的 JS 代码，都是 TS 代码，意味着合法的 JS 代码都可以直接搬到 TS 中使用，因为 TS 是 JS 的扩展和延伸。

扩展：

TypeScript 和 VSCode 都是由微软开发的，并且 VSCode 内部是使用 TS 语言来开发的。所以，VSCode 对 TS 的支持非常好，也因此 VSCode 对书写的 TS 代码拥有很强大的语法检查能力，以及语法提示和语法矫正功能。例如：在使用 VSCode 书写 TS 代码的过程中，VSCode 就能够实时地提示出语法错误（将鼠标悬停到提示错误的代码上时，还会显示详细的错误信息）。

编译：在 node 中可使用 tsc 指令将指定的 ts 文件编译为 js 文件，如：tsc index.ts。如果 ts 文件中有语法错误，则会在控制台打印错误信息。默认情况下，即使在编译过程中报出了错误，最终仍然会在同级目录下转换出一个同名的 js 文件，如：index.js。

语言类型：根据是否允许变量自动发生类型转换，可以将语言分为强类型语言和弱类型语言。

- 弱类型：变量可以根据环境变化自动进行转换，即：允许隐式类型转换，如：JS，TS，PHP 等。
- 强类型：不会隐式转换类型，除非强制类型转换，否则永远保持该类型，如：Java，C++ 等。

注释：TS 是完全兼容 JS 的，因为 TS 的核心理念就是在完全保留 JS 运行行为的基础上，去引入静态类型系统，来提高代码的可维护性以及减少 bug。因此，TS 也是一门弱类型语言，即：TS 允许隐式类型转换。

使用

安装

在 NPM 官网，找到 TypeScript 的工具包，建议安装 4.9.4 版本，并且全局安装。

```
npm install -g typescript@4.9.4
```

检查版本号：安装好之后，通过 tsc -v 指令检查安装的版本号。【tsc：是由 TS 工具包提供的指令】

编译

新建 `ts` 文件，在命令行窗口中通过 `tsc` 指令将指定的 `ts` 文件编译成 `js` 文件。

```
tsc index.ts
```

如果被编译的 `ts` 文件中有语法错误，则会在控制台中打印错误信息。默认情况下，无论是否在编译过程中报出了错误，最终都会在同级目录下转换并生成一个同名的 `js` 文件，如：`index.js`。

执行

要查看 `ts` 执行的结果，可以通过执行编译生成的 `js` 文件，将其引入 `html` 文件中或者直接在命令行工具中通过 `node` 指令执行它。

```
node index.js
```

`VSCode` 终端运行：按 `F5` 可选择在何处运行当前文件，可选择将 `ts` 文件在 `Node.js` 环境下运行。
快捷键：`ctrl + ~`（反引号键）

语法

限定变量类型

在声明变量时，可以在变量名后紧跟一个 `:` 以及一个数据类型，以此来实现对变量类型的提前限定。

```
let age:number = 14; // 官方称其为：类型注解
```

冒号前后可以有空格，也可以都省略。但更应值得注意的是，严格模式下，如果变量声明时未初始化，那么使用它之前必须为其赋值。

```
// 默认在严格模式下
// 未初始化变量
let num: number; // 在JS中默认将其赋值为：undefined

// 任何一个变量在使用前，必须先为其赋值，否则会报错。
console.log(num); // Variable 'num' is used before being assigned.

num = 10;

console.log(num); // 10
-------------------------------------------------------------------------------
---------------------------

// 在非严格模式下：允许在为赋值之前使用未初始化的变量
let str: string;

console.log(str); // undefined
```

误区：

- `TS` 的类型系统限制的是变量的类型，而不是值的类型。通过限制变量类型来限制值。
- 限制变量的类型，是为了在赋值时，保证赋给该变量的值应该符合此变量的类型要求。

- 所以，值的类型通常都一定符合变量的类型要求。但也有变量误存其他类型值的情况。

```
let num: number = 0;
let an: any = 'any';

num = an;

console.log(typeof num, num); // 'string' 'any'
```

语法判断干扰

当在 `VSCode` 中，同时打开 `ts` 文件及其编译后的 `js` 文件时，由于两个文件中的变量都相同，所以，`VSCode` 会误认为开发者在 `ts` 文件中声明了两个同名变量，因此会出现语法报错的提示。此时，只需要关闭其编译的 `js` 文件即可。

简化运行步骤

每次写完 `ts` 代码，运行它都需要经过如下两步：先通过 `tsc` 指令编译指定的 `ts` 文件，然后通过 `node` 指令执行其编译的 `js` 文件。

为了简化这两个步骤，可以在 `npm` 中下载 `ts-node` 工具包，建议全局安装。

```
npm install -g ts-node
```

当安装好之后，该包会提供一个 `ts-node` 的命令。使用该指令就可以直接运行 `ts` 文件了。

```
ts-node index.ts
```

`ts-node` 工具会先将 `ts` 文件进行编译，然后运行其编译后的 `js` 文件。相当于是对原先两个步骤的封装，但它并不会生成 `js` 文件。

`ts` 配置文件

通过 `tsc --init` 可以得到 `ts` 初始化的默认配置文件。在项目根目录下会生成该 `tsconfig.json` 配置文件。

指定目标版本

在 `tsconfig.json` 配置文件中，可以通过 `target` 根节点将 `ts` 文件编译成指定版本的 `js` 文件。默认为 `es2016`，即：`ES5` 语法。

```
"target": "es2020", // 编译成es2020版本的js文件，以支持使用es2020语法书写的代码。
```

`ts` 数据类型

`ts` 的数据类型：

- `JS` 原有类型：
  - 原始类型：`nunber`、`string`、`boolean`、`null`、`undefined`、`symbol`、`bigint`。
  - 对象类型：`object`（对象、数组、函数、集合……）
- `TS` 新增类型：联合类型、接口、元组、枚举、`any`、`void` ……

注释：由于 ts 相关的文件已经使用了一些变量名，因此要避免使用这些变量，以防止变量冲突。例如：在 lib.dom.d.ts 文件中已经声明过了 name 变量，因此，不能在 ts 文件中再使用该变量，可使用 myName 等变量代替。

```
// 在lib.dom.d.ts文件中，有如下声明：
declare const name: void;
```

另外，虽然 bigint 类型和 number 类型都是数字类型，但是在 ts 中两者的变量不能互换值。它们本身就是两种原始的数据类型。

```
let myAge:number = 14;

myAge = 1000n; // 提示报错：不能将类型“bigint”分配给类型“number”。
```

void 类型

ts 新增的数据类型 void 类型，表示空值类型，其变量除了可以被赋值为 void 类型外，还可以被手动赋值为 undefined 或 null。

```
let v1:void = undefined,
    v2:void = null; // 仅限非严格模式下
```

不过，在严格模式下，将 void 类型的变量赋值为 null 会发生语法错误。此时，可以在 ts 的默认配置文件中，将 strict 根节点的值改为 false，表示关闭严格模式。

undefined 类型和 null 类型：这两种类型在 ts 中，一般用处不大，因为它们只能被赋值为 undefined 或 null 值。但是，在 ts 的非严格模式下，这两种类型可以被看作是所有类型的子类型，这意味着它们可以作为任何类型的初始值。但是 void 类型不可以。

```
// 仅限非严格模式下
let num:number = undefined,
    str:string = null;
```

数组类型

在 ts 中，限制数组类型其实是在限制（最里层）数组中元素的类型。主要有两种限制方法。

```
// 一维数组
// 1、类型[]的常用表示法
let arr1:number[] = [1, 2, 3];
let arr2:string[] = ['1', '2', '3'];

// 2、泛型表示法
let arr3:Array<string> = ['1', '2', '3'];

// 多维数组：在类型后的中括号数量，就是该数组的维度。
// 1、常用表示法
// 二维数组
let arr4: number[][] = [[1], [2], [3]];
// 三维数组
let arr5: number[][][] = [[[1]], [[2]], [[3]]];
```

```
// 2、泛型表示法
// 二维数组
let arr6: Array<Array<number>> = [[1], [2], [3]];
// 三维数组
let arr7: Array<Array<Array<number>>> = [[[1]], [[2]], [[3]]];
```

从泛型表示法中，可以直观地看出：限制数组类型，就是在限制最里层的数组元素的类型。并且支持更复杂的类型约束。

第一种在表示一维数组时比较直观且方便好用，因此是常用的限制一维数组元素类型的方法。但是，这样只能在数组中填充单一的数据类型，如果要填充多种类型，可以使用 `ts` 新增的联合数据类型。

联合类型

由多种数据类型组成的数据类型，值可以是指定的数据类型中的任意一种。使用一个竖线 `|` 来连接多种数据类型，表示一种并集。

```
let a: number | string = '1';    // 可以理解为：a变量能够存储number值和/或string值。
let b: number | string = 2;      // 可以理解为：b变量能够存储number值和/或string值。

// 在数组中的使用
let arr1: number[] | string[] = [1, 2, 3]; // 元素要么都是number，要么都是string，元素类型单一。
let arr2: number[] | string[] = ['1', '2', '3'];
// 上面这种方式，可将数组元素限制为number或者string类型的值，但应用这样限制的数组只接受单一类型的元素，它是一种单类型数组。
// 如果要在数组中允许多种类型的元素，可以使用如下的语法。
let arr3: (number | string)[] = [1, '2', 3]; // 元素既可以是number，也可以是string，元素类型多样。
```

另外，当定义多维数组的类型时，使用泛型表示法会更加方便和直观。

```
// 二维数组
let arr: Array<Array<number | string> | boolean> = [[1], [2], true];
```

类型别名

类型别名：使用 `type` 关键字给某个类型自定义一个别名，主要用在复杂的数据类型中，如：联合类型。

```
// 当联合类型过长，或者需要多次使用时，可以给这种类型起一个别名。
let arr1: number[] | string[] | boolean[] = [1, 2, 3];
let arr2: (number | string | boolean)[] = [1, '2', 3];

// 定义类型别名
type SingleArray_nsb = number[] | string[] | boolean[];
type Array_nsb = (number | string | boolean)[];
// 使用类型别名
let arr3: SingleArray_nsb = [1, 2, 3];
let arr4: Array_nsb = [1, '2', false];
```

关于类型别名的规范：建议将首字母大写，且在末尾体现数据类型，尽量做到见名知意，如 `CustomArray`。

注意：类型别名看起来像是创建了一种新的数据类型，但它并不没有创建新的数据类型，只是给某种类型起了一个方便的别名而已。

类型推论

在 TS 中，当没有明确指定某个变量的类型时，TS 会根据变量的值推断出该变量的类型，并将其隐式限定为该类型。

```
let a = 'wanzi';

a = 14; // 语法错误：不能将类型“number”分配给类型“string”。
```

由于类型推论机制的存在，就可以像 js 一样直接给变量初始化一个值，而省略类型注解了，因为变量会被自动限制为其初始值的类型。

```
let num = 10; // 将num变量自动限制为number类型。
```

如果只声明了变量而未将其初始化，那么该变量将被推断为 any 类型，即：任意类型的值都可以赋给它。

```
let a; // 变量a为any类型，可以存储任意类型的值

a = 1;
a = '0';
```

在 ts 中，通常不建议将变量指定为 any 类型，因为这样就与 js 无异了。而类型注解的意义就在于：给暂时不初始化的变量手动指定一个类型，避免类型推论机制将其推断为 any 类型。

```
let a:number; // 注解该变量的数据类型

a = 1;
```

any 类型

any 类型表示任意类型。一旦定义为了 any 类型，那么它就失去了 TS 的类型检查（或保护）机制，它就与使用 js 没有区别了。因此，通常情况下，当提示某个值“隐式具有 any 类型”时，就可能会在编译时报错。

没有 any 类型的字面量，所以也就没有 any 类型的值。TS 只有 any 类型的变量，这种变量表示可以在任何时候存储任意类型的值。与 JS 的变量极其相似，在运算过程中也会首先确定值的类型，但变量始终为 any 型。

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。 这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。 这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。 那么我们可以使用 any 类型来标记这些变量。在对现有代码进行改写的时候，any 类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。

小结

特性：类型系统（避免不必要的语法错误，提高开发效率），适用于任何规模的项目（尤其是中大型项目）。

与标准同步发展：

从 `2015` 年开始，`ECMA` 组织在每年的 6 月份左右都会发布一个最新版本的 `ES` 语法，每一版中都会带来一些新的语法。任何人都可以向标准委员会（又称 `TC39` 委员会）提案，要求修改语言标准。

一种新的语法从提案到变成正式标准，需要经历五个阶段。每个阶段的变动都需要由 `TC39` 委员会批准。

1. `Stage 0`：`Strawman`（展示阶段）
2. `Stage 1`：`Proposal`（征求意见阶段）
3. `Stage 2`：`Draft`（草案阶段）
4. `Stage 3`：`Candidate`（候选人阶段）
5. `Stage 4`：`Finished`（定案阶段）

一个提案只要能进入 `Stage 2`，就差不多肯定会包括在以后的正式标准里面。`ECMAScript` 当前的所有提案，可以在 `TC39` 的官方网站 https://tc39.es/ecma262 中查看。而当进入 `Stage 3` 时，其语法就可能已经得到部分浏览器的支持了，并成为实验性语法。

而一个新的 `ES` 语法只要到了草案阶段，`TS` 就可以支持该语法了，不需要等到正式发布。几乎与标准同步发展，不用担心兼容性。

优势：`TS` 相对于 `JS` 的优势

- 能够更早地发现代码的错误，提高开发效率。
- 大多数的 `IDE` 编辑器都已经对 `TS` 有了很好的支持，代码提示更全，开发体验更好。
- 类型系统提高代码的可维护性。
- 支持最新的 `ES` 语法。
- 类型推论机制，降低了成本（包括学习和开发）。

误区：`TS` 虽然很好用，但如果项目不需要类型验证，甚至使用 `TS` 反而会降低项目运行效率的话，就不必使用 `TS`。应该在需要它的时候，合理地运用它，而不是在任何时候都滥用它。

函数

函数类型

限定函数类型，其实是在限定其输入和输出的类型，即：参数和返回值。

限定参数

在 TS 中，必须限定参数的类型，否则，会提示错误。限定参数就是限定函数的形参，通过限定形参的类型，来间接地控制实参的输入。因为设置一个形参其实就是在函数的顶层作用域中声明一个局部变量，因此，限定形参类型的方式也与声明变量无异。

```
// 限制形参a为number型，形参b为string型
function fn(a: number, b: string) {} // 类型注解
```

注意：当限定了某个形参的类型之后，在调用函数时，就必须为该形参传入指定类型的值（没有隐式类型转换）。否则，会报错。

```
fn(1, '2'); // TS只对参数进行类型检验，不会发生类型转换。
```

另外，TS 对参数的数量也进行了严格的限制。实参必须与形参的数量相同，否则在编译时会报错。

```
fn(1, '2', 3); // Expected 2 arguments, but got 3.
```

限定返回值

限定返回值，要在函数声明的小括号后紧跟一个 : 以及一个数据类型。即：在小括号内限定参数类型，在小括号外限制返回值类型。

```
function fn1(a: number, b: string): string {
    return ""; // 必须返回一个string类型的值
}

let fn2 = (a: number, b: string): string => "";
```

注意：当限制了返回值的类型之后，就必须在函数内部返回一个该类型的值。否则，会因为函数默认返回 void 而提示语法错误。

默认返回值：在 TS 中，函数的默认返回值是 void，而不是 undefined。但是在执行 ts 文件时，默认返回 undefined，因为它已经被编译成了 js 文件，执行 js 文件，就会得到 undefined。

函数表达式

针对函数表达式，还有另外的限制方式。因为函数表达式是使用变量初始化的方式声明一个函数的，所以还可以给这个变量添加注解。

```
// 声明时限制变量fn的a、b参数以及返回值的类型，然后将一个被应用了同样类型约束的函数对象赋给它。
let fn: (a: number, b: number) => string = (a: number, b: number): string => "";

// 因为，两边应用了相同的类型约束，所以可以省略掉一边。
let fn = (a: number, b: number): string => "";
let fn: (a: number, b: number) => string = (a, b) => "";
```

注意：当限制左边时，使用 `=>` 来指定返回值的类型；而限制右边时，仍然使用 `:` 限制返回值的类型。

另外，当类型约束应用在两边时，以左边的限制为主；只应用在左边时，以左边为主；只应用在右边时，以右边为主。因为，限制左边变量的类型主要是为了使要赋值给它的函数对象必须符合它的赋值要求。而限制右边的函数对象则主要是为了使函数的实参和返回值符合函数对输入和输出的控制要求。

```
// 因此，当应用左边的类型约束时，右边的形参不一定要与左边的同名。
let fn: (a: number, b: number) => string = (x, y) => ""; // 约束自动应用到右边，右边只需要提供合规的形参和返回值即可。

let fn: (a: number, b: number) => string = (x: number, y: number): string => x + y + "";
```

注意：当在左边限定返回值的类型为：`{}`、`viod` 或 `any` 时，则相当于没有限定返回值的类型，此时会采用右边的约束规则。

```
let fn1: (a: number, b: number) => {} = (x, y): number => x + y;    // 返回值为number型
let fn2: (a: number, b: number) => viod = (x, y): number => x + y;  // 返回值为number型
let fn3: (a: number, b: number) => any = (x, y): number => x + y;   // 返回值为number型
```

函数返回值

因为 `TS` 函数的默认返回值是 `void`。因此，当只需要函数完成某些任务，而不需要返回值时，就可以将其返回值限定为 `void` 类型。

```
function fn(): void {} // 表示该函数没有返回值，或返回值没有意义。

// 函数表达式
let fn: (a: number, b: number) => void = (x, y) => {};
```

当不明确指定返回值的类型时，`TS` 会根据函数的实际返回值对其返回值进行类型推论，并将其限定为推论出来的类型。

另外，当在左边将返回值的类型指定为：`{}`、`viod` 或 `any` 时，也不会影响右边的函数对其返回值应用类型推论机制。

```
// 左边将返回值的类型指定为{}，但这并没有影响类型推论对它的判断。
let fn: (a: number, b: number) => {} = (x, y) => x + y;      // 被推论为number型

// 在左边将返回值的类型限定为viod，也不会影响类型推论对它的判断。
let fn: (a: number, b: number) => viod = (x, y) => x + y;    // 被推论为number型

// 在左边将返回值的类型限定为any，还是不会影响类型推论对它的判断。
let fn: (a: number, b: number) => any = (x, y) => x + y;     // 被推论为number型
```

注释：在左边将返回值的类型指定为：`{}`、`viod` 或 `any`，相当于没有限定返回值的类型，即：允许方法返回任意类型的值。但如果在右边限制返回值的类型（即：直接在函数上限制返回值类型）或者左边指定的返回值类型不是如上三者，那么返回值必须符合类型要求。

```
let fn: (a: string, b: number) => {} = (x, y) => x ? x : y;
// 其效果相当于：
let fn: (a: string, b: number) => number | string = (x, y) => x ? x : y;
let fn: (a: string, b: number) => number | string = (x, y): number | string => x
? x : y;
let fn: (a: string, b: number) => void = (x, y): number | string => x ? x : y;
let fn = (x: string, y: number): number | string => x ? x : y;
// 其实质接近于：
let fn = (x: string, y: number) => x ? x : y;

console.log(fn('f', 0));    // 'f'
console.log(fn('', 0));     // 0
```

可选参数

设置参数时，可以在参数后面紧跟一个 `?` 来将该参数设定为可选的参数。

```
function fn(a: number, b?: number) {
    // 注意：不可以不加判断地直接使用任何一个可选的参数，因为它们可能没有意义，这在ts中会提示语法错误。
    if(b) {
        return a + b;
    }
    return a;
}

console.log(fn(1, 2));  // 3
console.log(fn(1));     // 1
```

规定：

- 问号必须在冒号的前面，前后的空格可有可无。
- 可选参数必须放在所有必选参数的后面，相当于是一种对函数输入的扩展。
- 将参数设为可选，相当于将它设置为联合类型（在指定的类型上联合一个 `undefined` 型），如：`b: number | undefined`。
- 给可选的形参，只能传递 `undefined`（相当于不传值）或者指定类型的值，要么不传任何值。
- 在函数内部，必须判断每一个可选参数是否有意义，否则会提示语法错误。

```
// 可选参数，实际上是在参数的设定类型上又联合一个undefined类型。
function fn1(a: number, b?: number): void {}            // 第二个参数可传可不传
function fn2(a: number, b: number | undefined): void {} // 第二个参数必须传
// 它们并不等价，因为fn1的b参数是可选的，而fn2的b参数是必选的。
```

默认参数

默认参数，就是给参数设定一个默认值，用法和作用与 `js` 一样。默认参数可以放在任何位置，但一般放在末尾当作一种输入扩展使用。

```
let fn = (a: number, b: number = 10): number => {
    return a + b;
}

// 默认参数，因为具有了默认值，所以它会作为可选参数来使用，相当于：b?: number。
fn(1);
```

由于默认参数也是一种可选参数，因此它们可以写在可选参数的后面。不过，不允许将一个参数既设为可选又设定默认值。

```
let fn = (a: number, b?: number, c: number = 10): number => {
    return a + c;
}

// 错误的写法：不能给参数同时应用问号和初始化式
let fn = (a: number, b?: number = 0, c: number = 10): number => {
    return a + b;
}
```

另外，`TS` 不允许将首位空缺，如果要使用首位的默认值，则可以传入一个 `undefined`。

剩余参数

使用扩展运算符可以将调用时多余传入的实参统一收集到一个数组中，必须应用限制数组类型的语法去间接地控制实参的输入。

```
// TS要求必须限制每个参数的类型
function fn(a: number, ...args: (number | string)[]) {
    console.log(a);
    console.log(args);
}

fn(1, 2, '3', 4);
/*
> 1
> [2, '3', 4]
*/
```

注释：剩余参数只能作为最后一个参数使用，且每个函数只能有一个。

函数重载

原生的 `JS` 并没有真正的函数重载，只能通过 `if-else` 等语句进行分支执行，模拟函数重载的功能，即：根据传入参数的类型、顺序或个数的不同，输出不同的结果。

```
// 使数字或字符串翻转
function reverse(x) {
    if (typeof x === 'number') {
        return Number(x.toString().split("").reverse().join(""));
    } else if(typeof x === 'string') {
        return x.split("").reverse().join("");
    }
}
```

在 `TS` 中，可以改写为如下代码：

```
function reverse(x: number | string): number | string | void {
    if (typeof x === 'number') {
        return Number(x.toString().split("").reverse().join(""));
    } else if(typeof x === 'string') {
        return x.split("").reverse().join("");
    }
}
```

注意：将 `return` 语句放到块级作用域中之后，函数将检测不到它的存在。因此，可在函数作用域中 `return` 一个值，或令其为 `void`。

但是，上述代码在类型安全方面仍然存在威胁。因为，输入和输出的类型并没有进行严格地绑定。要实现严格绑定，就需要函数重载。

函数重载

函数重载是 `C++` 中的一个概念。重载函数是函数的一种特殊情况，为方便使用，`C++` 允许在同一范围中声明几个功能类似的同名函数，但是这些同名函数的形式参数必须不同（主要指参数的个数、类型或者顺序），也就是说可以使用同一个函数完成不同的功能，这就是重载函数。重载函数常用来实现功能类似而所处理的数据类型不同的问题。

但是，`JS` 和 `TS` 都不允许多个函数使用一个名称，因此 `TS` 的重载函数与 `C++` 有所不同，但与 `JS` 相似。

函数签名

在 `TS` 中，要使用函数重载的功能，首先要定义函数的重载签名，然后实现重载签名。

一个函数签名 （`Function Signature`，亦称：类型签名或方法签名）定义了函数或方法的输入与输出。

一个签名可以包括：

- 参数及参数的类型。
- 一个返回值及其类型。
- 可能被抛出或传回的异常。
- 有关面向对象程序中方法可用性的信息（例如：关键字 `public`、`static` 或 `prototype`）。

然而，JS 并没有函数签名。这是因为 JS 是一门动态型语言，它的参数实际上是使用一个 arguments 类数组来表示的，所谓的形式参数也只是为了提供便利而设计的，并不是必须要求的。在其他语言中，命名参数这块要求必须事先创建函数签名，而将来的调用也必须与该签名一致。JS 中没有这些条条框框，解析器不会验证命名参数，所以说 JS 没有函数签名。

重载签名

要实现将输入与输出的类型进行绑定，关键就是重载签名。重载签名时只需要指定函数输入和输出的类型即可，不需要函数体。

```
// 重载签名：绑定输入和输出的类型
function reverse(x: number): number // 输入number，输出number
function reverse(x: string): string // 输入string，输出string
```

重载签名中的签名主要指的是函数的输入和输出及其类型，即：参数和返回值及其类型。

实现签名

实现签名就是兼容重载签名的函数实现，它是实际的函数执行逻辑，与 JS 很相似。

当调用函数，传入的参数与任意一个重载的签名匹配时，都可以调用函数体来执行。

```
// 实现签名
function reverse(x: number | string): number | string | void {
    if (typeof x === 'number') {
        return Number(x.toString().split("").reverse().join(""));
    } else if(typeof x === 'string') {
        return x.split("").reverse().join("");
    }
}

reverse(123); // 首先检查实参的类型，然后与重载的签名参数进行匹配，只要有一个签名匹配成功，则就会使用该参数执行函数体。
```

注释：实际上，只需要保证重载签名与实现签名中的输入及其类型保持一致即可。重载中的输出类型只要不超出实现中的输出类型，是可以随意指定的，并没有强制绑定输出的类型。这意味着即使重载签名绑定了输入和输出的类型，实际的输出可能不一定是绑定的类型。

```
// 重载签名：严控输入，期望输出
function reverse(x: number): string // 当x的输入为number时，期望输出一个string

// 实现签名：须完全兼容重载签名
function reverse(x: number | string): number | string | void {
    if (typeof x === 'number') {
        return Number(x.toString().split("").reverse().join("")); // 实际输出
    } else if(typeof x === 'string') {
        return x.split("").reverse().join("");
    }
}

// 函数执行：输入参数，获得输出
reverse(123); // 尽管重载签名期望输出string，但实际上却输出了number。
```

由此，可以看出：函数重载主要是给开发者看的提示语，或者说是对输入和输出类型的一种期望，但并不是对实际输入和实际输出的严格绑定。因为函数的实际输入是好控制的，但实际输出是由内部的实现逻辑决定的，就算限定返回值的类型也只会将其限定在一个允许的范围中。因为想要根据不同的输入获得不同的输出，就必定要允许有多种类型的返回值。无论实际的输出如何，都必定是指定的类型之一。

匹配签名

匹配顺序：调用重载函数时，会使用实参的类型，按照重载签名的顺序从上到下依次匹配。因此，尽量将匹配率高的重载签名放在前面。

另外，当多个重载签名中的参数个数不一致时，要注意在实现签名中将多余的参数设为可选参数或默认参数。

```
// 重载签名：参数的数量不一致
function fn(x: string): string;
function fn(x: number, y: number): string;
function fn(x: number, y: number, z: number): string;

// 实现签名：将多余的参数设为可选或默认
function fn(x: string | number, y?: number, z: number = 10): string | void {
    y = y || 0;
    if (typeof x === 'number') {
        return x + y + z + '';
    }
    return x;
}

console.log(fn('1')); // '1'
console.log(fn(1, 2)); // '13'
console.log(fn(1, 2, 3)); // '6'
```

当然，重载签名也可以设定可选参数和联合类型，但要注意不能超出实现签名提供的范围。

```
// 重载签名
function fn(x: string): string;
function fn(x: number, y?: number): number;
function fn(x: string | number, y?: number, z?: number): string | number;

// 实现签名
function fn(x: string | number, y?: number, z?: number): string | number | void
{}
```

对象

对象类型

限制对象类型，其实就是在指定对象的属性及其值的类型。一方面规定了对象允许的属性，另一方面规定了各属性值的类型。

语法：在变量后紧跟一个 `:` 以及一个 `{}`，在 `{}` 中以 `;` 分隔每条属性及其值类型，每条属性与值类型之间以 `:` 连接。

对象属性

```
// 规定了对象允许的属性及其值的类型
let person: {name: string; age: number} = {
    name: 'wanzi',
    age: 14
};
```

注释：

- 在对象的类型注解中，低版本使用分号 `;` 来分隔各属性，高版本中可使用逗号 `,`。为了兼容建议继续使用分号。
- 给变量进行对象的类型注解后，必须给它赋值一个合规的对象。属性不能多，也不能少，且值必须是规定的类型。

对象方法

```
let person: {fn1(): void; fn2(a:string, b: number): void} = {
    fn1() {},
    fn2(a, b) {}
};
```

规定：

- 添加对象方法的类型注解时，必须指定返回值的类型。
- 当指定返回值的类型为：`{}`、`void` 或 `any` 时，相当于没有限定类型，`TS` 会对其进行类型推论，但不会将其固定为某一种类型。

```
// 当返回值的注解类型为：{}、void或any时，允许方法返回任何类型的值，且值的类型不固定。
let person: { fn1(): void; fn2(a: string, b: number): void } = {
    fn1() {
        return 1;
    },
    fn2(a, b) {
        return a ? a : b;
     }
};

console.log(person.fn1());          // 1
console.log(person.fn2('a', 2));    // 'a'
console.log(person.fn2('', 2));     // 2
```

为了加强对对象方法返回值的类型限定，可以在右边（即：直接在方法上）限定其返回值的类型。

```
// 此时，会对fn2的返回值应用右边的类型限定
let person: { fn1(): void; fn2(a: string, b: number): void } = {
    fn1() {
        return 1;
    },
    fn2(a, b): number | string {
        return a ? a : b;
    }
};
```

因为箭头函数其实是一种表达式。因此，当需要定义箭头函数时，要采用定义属性的方式，将其设置为对象的属性。

```
let person: {af: () => void} = {
     // 箭头函数
    af() {
        // 因为被设置为对象属性，所以内部的this指向该对象。
    }
};
```

注解格式

当注解对象时，可以在每一个分号后换行以美化格式。当每一行只定义和注解一条属性时，分号也可以省略。

```
let person: {
    fn1(): void;
    fn2(a: string, b: number): void // 省略分号
    fn3: () => void
} = {
    fn1() {},
    fn2(a, b) {},
    fn3() {}
};
```

类型别名

对象是在编程中需要大量使用的数据结构。因此，单一地使用对象注解不仅相当繁琐，而且代码将会非常臃肿。此时，可以使用类型别名来预定义对象的类型，以提高复用性。

```
// 定义对象的类型别名
type CustomPerson = {
    name: string;
    age: number;
    sayName(): void
};

let person: CustomPerson = {
    name: 'wanzi',
    age: 14,
    sayName() {
        console.log(this.name);
```

```
    }
};

person.sayName(); // 'wanzi'
```

可选属性

与函数的可选参数一样，注解对象的属性时，可以在属性名之后，冒号之前使用一个问号来将该属性设定为可选。但是对象的属性不同于函数的参数，它没有严格的顺序规定。因此，对象的可选属性可以出现在对象中的任何位置，它主要靠属性名来匹配和辨认。

```
let person: {
    name: string;
    age?: number; // 可选属性
    sayName(): void
} = {
    name: 'wanzi',
    sayName() {
        console.log(this.name);
    }
};

console.log(person); // { name: 'wanzi', sayName: [Function: sayName] }
```

只读属性

在注解对象的属性时，可以在属性名前应用一个 `readonly` 关键字，来将该属性设定为只读属性。

```
let person: {
    readonly name: string; // 只读属性
    age?: number; // 可选属性
    sayName(): void
} = {
    name: 'wanzi',
    sayName() {
        console.log(this.name);
    }
};

person.name = 'cherry'; // Cannot assign to 'name' because it is a read-only
property.
```

注意：只读属性与使用 `const` 声明的常量一样，只是不允许修改它的栈值（如：通过等号重新赋值）。但如果是通过替换其父对象引用的方式，则可以指定一次只读属性的值，因为这相当于是在给它进行初始化。不过，通常也不会替换掉整个父对象的引用。

```
type CustomPerson = {
    readonly name: string;
    age: number;
    sayName(): void
};

let person: CustomPerson;
```

```
person = {
    name: 'wanzi', // 初始化只读属性
    age: 16,
    sayName() {}
}

console.log(person); // { name: 'wanzi', age: 16, sayName: [Function: sayName] }

person = {
    name: 'cherry', // 初始化只读属性
    age: 19,
    sayName() {}
}

console.log(person); // { name: 'cherry', age: 19, sayName: [Function: sayName]
}
```

另外，通过 `TS` 的类型兼容机制，可以绕过对只读属性的限制而修改它们的值。

```
// 类型兼容：一个对象的属性及其类型，被另一个对象的属性及其类型完全兼容的情况。
interface IPerson1 {
    name: string;
    age: number
}

// IPerson2的属性类型完全兼容IPerson1
interface IPerson2 {
    readonly name: string;
    readonly age: number | string
}

let person1: IPerson1 = {
    name: 'wanzi',
    age: 14
};

// 当类型完全兼容时，可以直接将被兼容的类型（或子类型）赋值给兼容的类型（或父类型）。
let person2: IPerson2 = person1; // 两者引用同一个堆，但它们对属性的要求不同。

// person2的两个属性都是只读的，但可以通过修改person1，来间接地修改person2中的属性值。因为两
者维护的是同一个内存空间。
person1.name = 'cherry' // 直接修改堆中的内容
person1.age = 20;       // 直接修改堆中的内容

console.log(person1); // { name: 'cherry', age: 20 }
console.log(person2); // { name: 'cherry', age: 20 }
```


只读数组

定义数组时，若使用 `ReadonlyArray` 泛型注解，则可以将该数组定义成一个只读数组。即：其根元素的栈值不可被修改。

注释：所有会影响数组自身的 `API` 都被只读数组剔除了。如果要修改只读数组，就只能整个替换它的引用。

```
let arr: ReadonlyArray<number> = [1, 2, 3];

arr[0] = 10; // Index signature in type 'readonly number[]' only permits
reading.
```

只读数组也可以在对象中使用。

```
type CustomPerson = {
    readonly name: string;
    age: number;
    arr: ReadonlyArray<number>
};
```

动态属性

为了增加对象的灵活性，`TS` 允许通过索引签名的方式，使用中括号语法，为对象设置动态属性。

设置了动态属性后，就允许动态地向对象中额外添加任意多个属性了，这些属性及其值只要符合类型要求即可。

```
type CustomPerson = {
    readonly name: string;
    age: number;
    [propName: string]: any // 对象属性默认为string型，其值此处被指定为any型。
};

// 该对象除了要指定必须的属性外，还可以额外地添加任意数量的属性。
let person: CustomPerson = {
    name: 'wanzi',
    age: 16,
    // 上面是对象的必选属性，下面是动态添加的对象属性。
    sex: '女',
    id: 1
};

console.log(person); // { name: 'wanzi', age: 16, sex: '女', id: 1 }
```

注意：

- 中括号中的属性名是随意的，但必须指定其类型，可以是：`string`、`number` 或 `symbol`。
- 动态属性的值类型也必须指定，并且此后动态添加的属性都必须符合这个值类型要求。
- 一旦明确了动态属性的值类型，则其他的属性值都必须符合该动态属性的值类型要求。
- 索引签名可以指定多个，且可以写在对象中的任意位置，但索引签名的类型不能重复。

```
// 每种类型的索引签名只能定义一个，不能重复定义。
type CustomPerson = {
    name: string;
    [a: string]: any; // string型的索引签名，值类型为any
    age: number;
    [b: number]: any; // number型的索引签名，值类型为any
    [c: symbol]: any; // symbol型的索引签名，值类型为any
};
```

注释：中括号中的属性名十分随意，可以完全相同，也可以是已知的属性名。它只是一种临时占位符，没有任何额外的实际意义。

```
// 一旦明确了某个索引签名的值类型，则其余的值类型都必须符合它的要求
type CustomPerson = {
    name: string;
    [a: string]: string; // string型的索引签名，值类型为string
    age: any;
    [b: number]: string; // number型的索引签名，值类型为string
    [c: symbol]: string; // symbol型的索引签名，值类型为string
};
```

上述代码，因为将索引签名的值类型指定为了 `string` 型，所以其他的值类型也都必须是 `string` 型或者 `any` 型（兼容任意类型）。

因此，在设置动态属性时，为了获得更大的兼容性，动态属性的值类型要能够兼容其他属性的值类型，通常会设其为 `any` 型。这样，其他属性的值才能够指定合理的类型。

```
type CustomPerson = {
    name: string;
    [a: string]: string | number; // 兼容其他字符串属性的值类型（如果有undefined、null或void值类型，需要在此指定）
    age: number;
    [b: number]: number; // 在动态属性中：数字属性的值类型不能超过字符串属性的值类型
    [c: symbol]: boolean; // 兼容其他符号属性的值类型（如果有undefined、null或void值类型，需要在此指定）
};
```

字符串属性和符号属性是两种属性，因此动态字符串属性的值类型，只能限制字符串属性的值类型，并不会限制符号属性的值类型。

利用数字型动态属性（即：数字索引签名），也可以实现定义数组的类型，因为数组就是典型的以数字为属性的对象。

```
// 定义一维数组的类型
type MyArr = {
    [i: number]: any // 元素为任意类型
}

let arr: MyArr = [1, '2', true, undefined, null, []];

// 定义多维数组的类型
type MyArr = {
    [i: number]: Array<number | string> | boolean
}

let arr: MyArr = [[1, 2], ['3', 4, 5], true];
```

接口类型

当对象的数据类型比较复杂或者想要对对象进行更复杂的扩展时，则可以使用接口来定义对象类型。

接口主要针对函数、对象和类，使用一个 `interface` 关键字来定义，目的主要是为了提高复用性。

定义对象

```
// 接口名通常使用一个大写的字母“I”开头，然后紧跟一个花括号即可。
// 接口定义对象类型
interface IPerson {
    name: string;
    age: number;
    sayName(): void
}

// 使用接口时，也只能且必须定义已知的属性和方法。
let person: IPerson = {
    name: 'wanzi',
    age: 14,
    sayName() {
        console.log(this.name);
    }
};

person.sayName(); // 'wanzi'
```

所有适用于对象类型的语法，也都基本适用于接口类型。

定义函数

使用接口定义函数，同样是限制其输入和输出。其语法类似定义对象类型与函数类型的结合。

```
// 接口定义函数类型
interface IFn {
    (x: number, y: number): number
}

let fn1: IFn = (x, y) => x + y;
```

如果使用类型别名来定义函数类型，则要像下面这样改写。

```
type Fn = (x: number, y: number) => number; // 没有花括号，且使用箭头指定返回值的类型
```

同名接口

可以定义多个同名的接口，当使用的接口是用于定义对象类型时，这些同名的接口最终会合并为一个接口。

```
// 使用多个同名的接口定义函数类型
// 多个同名的接口最终会合并为一个，重复的部分必须类型相同。
interface IPoint {
    x: number
}
interface IPoint {
```

```
    x: number; // 重复的部分类型必须一致，不能多也不能少，否则合并会失败
    y: number
}
interface IPoint {
    z: number
}

/* 最后合并为一个对象类型：
interface IPoint {
    x: number;
    y: number;
    z: number
}*/

let point: IPoint = {
    x: 1,
    y: 2,
    z: 3
};
```

当使用的接口是用于定义函数类型时，这些同名的接口最终会合并为该函数类型的重载签名。

注意：

- 合并后的重载签名，必须完全被实现签名所兼容。
- 实现合并的重载签名时，可简单地将输入的类型进行联合，但不可以简单地将输出的类型也进行联合（建议给它一个 `any` 类型）。

```
// 使用多个同名的接口定义函数类型
interface IFn {
    (x: number, y: number): number | void
}
interface IFn {
    (x: string, y: string): string | void // 后定义的重载签名优先级更高
}

/* 最后合并为该函数类型的重载签名，且“后来者居上”
interface IFn {
    (x: string, y: string): string | void
    (x: number, y: number): number | void
}
*/

// 实现签名（简单地联合输出的类型可能会导致该函数的类型不符合接口的类型要求）
let fn: IFn = (x: number | string, y: number | string): any => {
    // 在TS中，不能将运算符应用于类型不明确的操作数
    if (typeof x === 'number' && typeof y === 'number') {
        return x + y;
    } else if (typeof x === 'string' && typeof y === 'string'){
        return x + y;
    }
}

// 使用函数
console.log(fn(1, 2));      // 3
```

```
console.log(fn('1', '2'));  // '12'
```

如果有可选的参数，那么要在重载签名和实现签名中都将该参数设为可选。

```
// 使用同名接口定义重载签名
interface IFn {
    (x: number, y: number): number | void
}
interface IFn {
    (x: string, y: string): string | void
}
interface IFn {
    (x: string, y: string, z?: string): string | void // 高版本中，这里不用设置可选。但低版本中需要。
}

// 实现签名
let fn: IFn = (x: string | number, y: string | number, z?: string): any => {}

fn(1, 2);
fn('1', '2');
fn('1', '2', '3');
```

接口继承

接口可以像类一样，通过 `extends` 关键字来实现单次继承，目的是为了在已有接口的基础上进行扩展【接口类型一次只可扩展一个】。

```
interface IBase {
    a: number
}

// 扩展已有的接口类型：花括号中为扩展的部分，扩展后的接口类型赋值给Sub。
interface ISub extends IBase {
    // 扩展的部分
    b: number
}

let obj: ISub = {
    a: 1,
    b: 2
};
```

扩展类型别名：在定义新的类型别名时，在右边可以使用一个 `&` 链接已有类型别名和扩展的部分【类型别名可以一次性扩展多个】。

```
type BaseObject = {
    a: number
};

// 扩展已有的类型别名
type SubObject = BaseObject & {
    // 扩展部分1
    b: number
```

```
} & {
    // 扩展部分2
    c: number
}

let obj: SubObject = {
    a: 1,
    b: 2,
    c: 3
};
```

其实，接口类型与类型别名是可以相互扩展的。

```
type BaseObject = {
    a: number
};

// 接口类型扩展类型别名
interface ISub extends BaseObject {
    b: number
}

// 类型别名扩展接口类型
type SubObject = ISub & {
    c: number
}
```

接口类型与类型别名的主要区别：

1. 首先，使用接口只能定义对象和函数（包括类）的类型；而类型别名几乎可以定义所有的类型。
2. 其次，使用接口定义对象和函数类型的语法跟使用类型别名定义它们的语法很明显是不一样的。
3. 再者，接口允许定义多个同名的接口（同名接口最终会合并为一个）；而类型别名不允许重名。
4. 最后，接口类型使用 `extends` 关键字且只能实现单次扩展；而类型别名使用 `&` 且可多次扩展。

当然，只定义简单的类型约束时，使用类型别名就足够了。但如果要定义复杂的类型约束，可考虑使用接口类型。

元组类型

`TS` 在限制数组类型时，只限制了其中元素的类型，但这样是不够严谨的。因为数组的长度和数组槽位的类型都还没有得到限制。因此，要将对数组的类型进行彻底地限制，就可以使用元组类型。

元组本身也是一种数组，只不过它更加严格，它额外规定了数组槽位的数量以及各个槽位的类型。

```
// 数组将类型约束到外面（即：整体），元组将类型约束到内部（即：局部）。
// 定义一个长度为3，各槽位的类型依次如下的元组。
let tuple: [number, string, boolean] = [1, '2', true]; // 元组的长度和槽位类型必须都符合要求
```

伸缩元组

元组的长度已经得到了限制，因此不可以使用越界的索引去扩展元组。

```
tuple[3] = 3; // Tuple type '[number, string, boolean]' of length '3' has no
element at index '3'.
```

但是因为元组具有 `push()` 等方法，因此，元组可以通过 `push` 等 `API` 进行伸缩。但扩展的槽位类型只能是已知槽位的类型之一。

```
// 1、扩展元组
// 可以通过push或unshift，向元组中推入number、string或boolean类型的元素。
tuple.push(3);
tuple.push('4');
tuple.push(false);
tuple.unshift(); // 空插：什么也不插入

// 推入其他类型的元素，则会导致报错。
tuple.push(undefined);
// Argument of type 'undefined' is not assignable to parameter of type 'string |
number | boolean'.
-------------------------------------------------------------------------------
---------------------------

// 2、压缩元组
tuple.pop();
tuple.shift();
```

可选元素

元组允许设置可选的元素。注解时，在槽位的类型后面紧跟一个 `?`，表示将该槽位设为可选。

```
let tup: [number, number?, number?];

tup = [1];
tup = [1, 2];
tup = [1, 2, 3];
```

注意：元组元素与函数的参数列表（类数组）一样，具有严格的顺序要求。因此，可选元素必须在必选元素之后。

剩余元素

同样，函数的参数列表可以通过扩展运算符设定剩余参数，那么元组也可以使用扩展运算符来设定剩余元素。但可放在元组中任意位置。

```
// 剩余元素可以设置在元组的任何位置，但只能设置一个。另外，元组的剩余元素必须是数组类型，且可以指定它的类型。

// 设置在元组首部
let tup: [...string[], number, number] = ['1', '2', 3, 4];

// 设置在元组中间
let tup: [number, ...string[], number] = [1, '2', '3', 4];

// 设置在元组尾部
let tup: [number, number, ...string[]] = [1, 2, '3', '4'];
```

注释：因为 `rest` 元素不能跟在另一个 `rest` 元素之后，因此，在元组中也只能设置一个剩余元素。

只读元组

使用 `readonly` 关键字，可以将一个元组设为只读。与只读数组一样，所有会影响元组自身的 `API` 都被只读元组剔除了。如果要修改只读元组，就只能整个替换它的引用。

```
let tup: readonly[number, number, ...string[]]; // 只读元组没有push、pop等可以改变元组自身的方法。

tup = [1, 2, '3', '4'];
```

解构赋值

`TS` 允许对数组、元组和对象进行解构赋值。解构时不允许再设置类型约束，且只能解构已知的属性，不允许溢出解构。

```
// 解构元组
let tup: [number, string] = [1, '2'];

// 完全解构
let [num, str] = tup;
console.log(num); // 1
console.log(str); // '2'

// 部分解构
let [, str] = tup;
console.log(str); // '2'

// 溢出解构
let [, , more] = tup; // Tuple type '[number, string]' of length '2' has no element at index '2'.
```

```
// 解构对象
interface IObj {
    name: string,
    age: number
}
```

```
let obj: IObj = {
    name: 'wanzi',
    age: 16
};

// 解构并重命名
let {name: a, age: b} = obj;

console.log(a); // 'wanzi'
console.log(b); // 16
```

类型断言

`never` 类型

`never` 类型的值，表示："永远不存在的值"，或"永远获取不到的值"。

常发生在死循环和无限递归等场景中，它们大多都是因为永远获取不到一个临界值（终止值或开关），而永远得不到终止。

另外，由于 `JS` 是单线程执行的。因此一旦同步执行发生错误，就会立即终止程序，那么后面的代码也会永远得不到结果。

```
// 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) {}
}
```

`never` 类型常用于排除错误，它是所有类型的子类型（即：可以赋值给任何类型）。但是，`never` 类型的变量只接受 `never` 类型的值，即使是 `any` 类型也不可以。

```
// 注意：never类型没有字面量，并且在异常后程序通常会终止，因此会很少使用它。
let ne: never;

function error(message: string): never {
    throw new Error(message);
}

ne = error('终止程序，以获得一个never类型的值'); // 赋给一个never类型的值
```

`unknown` 类型

`TypeScript 3.0` 引入了一个顶级的 `unknown` 类型。 对照于 `any`，`unknown` 是类型安全的。 任何值都可以赋给 `unknown`，但是当没有类型断言或基于控制流的类型细化时 `unknown` 不可以赋值给其它类型，除了它自己和 `any` 外。 同样地，在 `unknown` 没有被断言或细化到一个确切类型之前，是不允许在其上进行任何操作的。

`any` 的问题

由于 `any` 类型能够兼容所有的类型。因此 `any` 类型的变量可以存储所有类型的值。而变量一旦被定义成了 `any` 类型，那么它就会失去类型检查机制（编译或赋值时都不会检查其值类型）。因此 `any` 型变量又可以赋给几乎所有类型的变量，除了 `never` 等特殊的类型外。

```
// any类型的变量可以存储所有类型的值
let an: any = 1;
an = '丸子';
an = true;

// any型变量也可以赋给除never外的其他类型的变量
let num: number = an;
let str: string = an;
let obj: object = an;
```

无论是在交换两个变量的值，还是在引用另一个变量的值时，`TS` 的类型检查机制都会首先确定："要赋的值是否符合变量的类型要求"，只有当类型符合时，才会将该值赋给指定的变量。

```
let a: number | string;

a = 1;

let b: number | boolean = a; // 值为number型，符合变量的类型要求，允许赋值。

a = 'a';

let c: string | array = a;   // 值为string型，符合变量的类型要求，允许赋值。
```

而这对于已经失去了类型检查机制的 `any` 型变量来说，在赋值时不会检查它的值类型，所以它的值可以轻松地被几乎所有类型的变量所引用（`never` 等特殊类型除外）。

这样就会带来一种奇特的怪象（"原先看似不合理的操作，实际上都得到了合理的执行"），它使得 `any` 类型的变量几乎完全如同 `JS` 的变量一样随心所欲，肆意破坏类型保护机制。

```
let an: any = 1; // 注意：此时，an的值为number型，但an变量始终为any型。变量的类型不会随着值类型的变化而变化。

// any型变量可以赋给其他类型的变量，尽管它存储的值类型不合规定。
let num: number = an;
let str: string = an;
let obj: object = an;

console.log(typeof num); // 'number'
console.log(typeof str); // 'number'
console.log(typeof obj); // 'number'
```

而 `unknown` 类型（不定型，未知型），正是为了弥补 `any` 类型的这些缺陷所设计出来的，虽然它也不尽完美。

`unknown` 型

首先，`unknown` 型数据也可以赋值给 `any` 型变量。除此之外，它只能赋值给 `unknown` 型变量。不过，能够赋值给 `any` 型变量的数据也都可以赋值给 `unknown` 型变量。

```
let un: unknown;

// 能够赋值给any型变量的数据也都可以赋值给unknown型变量
un = 1;
un = 'a';
un = [];

// unknown型变量只能赋值给同类型或any的型变量（亮点所在）
let an: any = un;
```

`unknown` 型变量，因为其类型不明确或未知，所以它们只能赋值给同类型或 `any` 的型变量。这正是它较之 `any` 型变量的亮点所在。

```
// 将unknown型变量赋值给其他类型，会导致报错。进一步杜绝了变量保存不合变量类型要求的值。
let un: unknown = 1;

// 尽管已知值类型符合变量的类型要求，也不允许将unknown型变量赋值给number型变量。
let num: number = un; // Type 'unknown' is not assignable to type 'number'.
```

注意：在严格模式下，大多数类型的变量在使用前都必须要先进行赋值。但这些类型是例外：`undefined`、`void`、`any` 和 `unknown`。

类型断言

类型断言：就是将对某个值的类型检查任务交给程序员，由程序员推断并告诉 `TS` 该值的类型，进而阻止 `TS` 去猜测这个值的类型。因为，有时候你会比 `TypeScript` 更加了解某个值的详细信息。通过类型断言这种方式可以告诉编译器：“相信我，我知道自己在干什么”。

类型断言没有运行时的影响，只是在编译阶段起作用。 `TS` 会假设程序员，已经对数据进行了必要的检查，并相信程序员的推断结果。

更重要的一点是，`TS` 只负责编译，不负责执行。`TS` 只将 `ts` 文件编译成 `js` 文件，而执行 `js` 文件的是机器。因此，`TS` 所有的类型约束和类型检查都只发生在编译期，它没有执行期（实际上，`TS` 目前也只有编译器，而没有执行器。它自身都是使用 `JS` 开发的）。

因此，如果某些表达式的结果在编译期无法确定，只能等待它被执行后才能得到结果的话，`TS` 就会预先提醒潜在的类型安全危险。

```
// 因此，TS只能做静态的类型检查。
let arr: number[] = [1, 2, 3, 4];

// 找出arr中第一个大于2的元素，并将其赋值给num
let num: number = arr.find(i => i > 2); // Type 'number | undefined' is not
assignable to type 'number'.
```

上面这段代码看起来没有问题，但实际上它在书写完就会报出错误。因为，`num` 变量的值是一个表达式，而这个表达式只能在执行后才知道其结果。`TS` 获取不到表达式的结果，就将其推论为 `undefined`。而这样，便会因为值类型与变量类型不匹配而提示类型错误。

要解决这个问题，只需告诉 `TS` 的类型检查机制，这个表达式的值类型即可。然后在 `TS` 编译时，它就会使用这个类型去匹配变量类型。

在需要自己断言的值后面，使用一个 `as` 关键字加上断言的类型，来告诉 `TS` 你对该值类型的断言结果。

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

// 断言arr.find(i => i > 2)表达式的值类型为number类型
let num: number = arr.find(i => i > 2) as number;

// 类型断言，还有一种不推荐的写法：在值前面使用<类型>
let num: number = <number>arr.find(i => i > 2);
```

注意：类型断言只能用在 `TS` 的类型推论机制无法明确一个值的类型的情况。当 `TS` 已经明确知道该值的类型时，还试图使用类型断言去蒙骗它，会被认为是“有意的行为”，这会引发编译错误。因此，不要滥用类型断言。

```
// 禁止的行为
let num: number = '1' as number; // Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.

// 允许的行为
let num: number = 100 as number;
```

实际上，`TS` 对大多数的表达式都会提前获取其结果以推断值类型。但对于调用 `API` 或函数获取结果的表达式，可能无法获取到结果。

```
// 编译报错
let num: number;

// 调用函数或方法
function fn() {
    num = 30;
};
fn();

console.log(num); // Variable 'num' is used before being assigned.
```

当值是通过调用函数或方法 `API` 的方式获取而来时，`TS` 会因为没有执行能力而获取不到结果，然后选择忽略一切方法和函数的调用。

```
// 但TS能够获取自执行函数表达式的结果，因为内部的函数已经被转换成了普通的表达式。
let num: number;

// 自执行的函数表达式
(function () {
    num = 30;
})();

console.log(num); // 30
```

`JS` 迁移

当想要将使用 `JS` 代码开发的项目升级或迁移到 `TS` 时，要特别留意类型方面的注意事项。比如下面的 `JS` 代码。

```
let person = {};

person.name = 'wanzi';
person.age = 16;
```

上面代码在 `JS` 中没有任何问题，但在 `TS` 中，类型推论机制不会将 `{}` 推论为 `object` 类型，而是直接将其视作一种 `{}` 类型。然后该类型会因为没有 `name` 和 `age` 等用户自定义的属性而报错。此时，就可以给它应用类型断言，将其断言为对象类型。

```
// 初始化时给一个空对象，后续再向其中添加属性。
let person = {} as {name: string, age: number};

console.log(person); // {}

// 后续只能添加已知的属性，但它们都已经不是必须的了。
person.name = 'wanzi';

console.log(person); // {name: 'wanzi'}
```

如果直接对变量进行类型注解，那么每个指定的属性都是必须设置的。而这也正是类型断言比类型注解更灵活的地方。

```
// 注解的每个属性都是必须的，且初始化值必须符合要求
let person: {name: string, age: number} = {
    name: 'wanzi',
    age: 16
};
```

关于迁移，更多参考：`JavaScript` 迁移

非空断言

`ES8` 支持在调用 `API` 时，在 `.` 的前面使用 `?` 来判断调用者是否有意义（即：不为 `undefined` 或 `null`）。只有当意义时，才会调用相应的 `API`；即使没有意义，也只会返回 `undefined`，从而避免了无意义的报错。

```
let arr; // 值为undefined

// 如果直接对其调用pop，则会发生类型错误。
arr.pop(); // Uncaught TypeError: Cannot read properties of undefined (reading
'push')

// 而对其使用可选链运算符(?)就能避免报错。
arr?.pop(); // undefined
```

TS 则通过一个 ! 来支持非空断言，即：断言某个值一定不是空类型（此处指 undefined 型或 null 型），以打消 TS 的类型顾虑。

```
// 值域中包含undefined和null类型，则使用前必须保证它一定有意义。
let str: string | undefined | null;

// 因为str还没有指定具体的值，所以其值类型是不明确的。
// 则TS会因为该值的类型可能为空值类型，而提示语法错误。
str.toString(); // 'str' is possibly 'null' or 'undefined'.
```

此时，可以使用非空断言来将值类型中的空类型排除在外，使 TS 编译器相信该值一定有意义（即使它实际上没有意义）。

```
let str: string | undefined | null;

// 告诉TS该值的类型一定非空，应允许其通过编译。
// 但如果值确实是空值，那么一定会在执行期报错。
str!.toString();
```

虽然使用非空断言使得 str!.toString() 通过了编译期，但由于 str 的值确实为 undefined，所以会在执行期抛出类型错误。

另外，由于非空断言的存在，所以允许变量在严格模式下即使没赋任何值也可以被正常使用。而这，可能会导致意外的情况发生。

```
let str: string;

// 断言该值非空，那么就可以正常使用它
console.log(str!); // undefined

console.log(str!.length); // Cannot read properties of undefined (reading
'length')
```

更多参考：TypeScript 非空断言

赋值断言

严格模式下，TS 的变量在使用前必须要赋值。然而使用非空断言可以打破这个限制，如前所述。

其实，在非空断言的基础上，还有一种叫赋值断言的语法，即：断言赋给该变量的值一定有意义。

```ts
// 当TS知道变量的值有意义之后，就不会再严格要求它使用前一定赋值了。
// 这样，对于TS无法调用方法API或函数获取结果的问题便可以得到解决。
let num!: number; // 断言该变量的值一定有意义。

// 调用函数或方法
function fn() {
    num = 30;
};
fn();

console.log(num * 2); // 60
```

当然，使用非空断言也可以解决这个问题。非空断言与赋值断言无异，只是非空断言需要在每次使用时断言，而赋值断言只需断言一次。

```ts
let num: number;

// 调用函数或方法
function fn() {
    num = 30;
};
fn();

// 在每次使用变量时，都需要进行断言
console.log(num!);      // 30
console.log(num! * 2);  // 60
```

尤其是在访问或重写可选属性时，非空断言可以解决其类型纠纷的问题，因为可选属性实际上就是在原有类型上联合一个 `undefined`。

```ts
interface IT {
    name: string;
    age: number;
    sex?: string;
    info?: {
        a?: number;
        b?: number;
    };
}

let t: IT = {
    name: '',
    age: 0,
    info: {
        a: 1
    }
}

// 没有问题，因为在联合类型中选择了一种
t.sex = '';
// 提示报错，因为上一级info可能没有意义
t.info.b = 0; // 't.info' is possibly 'undefined'.

// 此时，就可以对info进行非空断言
```

```
t.info!.b = 0;

// 为了更方便赋值，也可以取出info
let t_info = t.info!;
t_info.b = 0;
```

联合类型

联合类型也会引发潜在的类型危险，因为联合类型有时会让 `TS` 在类型推论中裁决不定。特别是当使用某个类型特有的属性或方法时，会因为其他类型没有这样的属性和方法而报错。

```
interface ITest1 {
    name: string;
    getName(): void
}
interface ITest2 {
    name: string;
    sayName(): void
}

function fn(arg: ITest1 | ITest2): boolean {
    if (typeof arg.getName === 'function') {
        return true;
    }
    return false;
}
```

上面的代码，会因为 `ITest2` 接口类型没有 `getName()` 方法，提示语法错误。要解决这个问题，只需要在使用接口的特有属性或者方法时，明确指定接口类型即可。此时，就可以使用类型断言（另外，这里使用 `?.` 运算符是没有用的）。

```
// 类型断言
function fn(arg: ITest1 | ITest2): boolean {
    // 明确指定使用哪个接口类型，避免TS类型检查犯难
    if (typeof (arg as ITest1).getName === 'function') {
        return true;
    }
    return false;
}
```

任意类型

当在某个类型上使用该类型本身不具有的属性或方法时，会导致编译报错。如前所述，如下所示。

```
let num: number = 1;

console.log(num.length); // Property 'length' does not exist on type 'number'.
```

而此时，还可以将其值断言为 `any` 类型。因为 `any` 型数据不会被类型检查，所以就能阻止在编译时报错，使它顺利度过编译期。

```
let num: number = 1;

// any型变量可以访问任何类型的属性和方法，并且不会报错。
console.log((num as any).length); // undefined
console.log((num as any).concat); // undefined

// 但如果调用其他类型的方法，则会因不存在而在执行期报错。
console.log((num as any).push()); // num.pop is not a function
```

尤其是，`TS` 不允许在 `window` 对象上自定义属性，因为这些属性是 `window` 对象原先不具有的。

```
window.a = 1; // Property 'a' does not exist on type 'Window & typeof
globalThis'.
```

但这在 `JS` 中是完全支持的行为，如果要让 `TS` 也允许这种行为，可以将 `window` 对象断言为 `any` 型来对其移除类型检查机制。

```
(window as any).a = 1;
```

另外，也可以将 `any` 类型断言为一个具体的类型。常用在获取 `any` 的型函数返回值的场景中，明知返回值的类型时。

```
function getData(a: number): any {
    return a < 5 ? '' : [];
}

let str: string = getData(4); // 存入any型数据

// 因为返回值的类型为any型，所以编译器不会觉察到错误。
str.push(); // 但在执行时，会因为没有该方法而报错。
```

此时，为了避免在执行期报错，可以将 `any` 型数据断言为一个确切的类型。这样错误就会提前在编译时显现，也就能提前纠正它。

```
let str: string = getData(4) as string; // 断言结果为string类型

// 此时，编译器就能够觉察到错误，并提前预报了。
str.push(); // Property 'push' does not exist on type 'string'.
```

然而，编译器完全信任程序的断言能力。如果程序员断言失误或者将其恶意断言为其他类型，编译器也不会觉察到任何问题。

```
let str: string = getData(7) as string; // 断言结果为string类型，实际存入的是一个数组。

console.log(str); // []
```

由于，编译器会对你的断言深信不疑。所以，即使实际存入的是数组，也不能调用数组的 `API`，因为你已经告诉编译器它是字符类型。

```
let str: string = getData(7) as string;

// 编译器会将str视作字符串类型，即使你知道存入的其实是数组。
str.push(); // Property 'push' does not exist on type 'string'.
```

泛型

字面量类型

在 `JS` 中，字面量只作为数据使用，提供一种直接书写各类型数据的简便方式。而在 `TS` 中，字面量不仅可以作为数据使用，还可以作为数据类型来使用。字面量类型用于指定一个变量的值必须是指定的字面量。

```
let str: "abc" = "abc";

// 如果仅仅是这样，那么使用const应该更加明智。
const str = 'abc';
```

字面量类型最大的用途是，当明确结果的范围或期待获得某些结果时，用来限制值域。比如：结合联合类型限定函数的输入域或输出域。

```
type ArgsString = 'JM' | 'JJ' | 'JS';        // 限定输入域
type resString = '剑魔' | '剑姬' | '剑圣';    // 限定输出域

function fn(code: ArgsString): resString {
    return code === 'JM' ? '剑魔' : code === 'JJ' ? '剑姬' : '剑圣';
}

console.log(fn('JM')); // '剑魔'
console.log(fn('JJ')); // '剑姬'
console.log(fn('JS')); // '剑圣'

// 输入的值超出输入域
console.log(fn('TS')); // Argument of type '"TS"' is not assignable to parameter
of type 'ArgsString'.
```


条件类型

更多参考：中文参考、官网原文

泛型

之前使用的函数重载、类型别名、接口类型等都是在预先定义函数的输入或输出类型。而泛型则是在调用时指定值的类型。

语法：声明函数时，在函数名后面使用一对尖括号包裹一个泛型变量。该变量用来接收调用函数时以同样方式指定的类型。

```
// 声明函数时，声明一个泛型变量
function fn<T> (arg: string) {}

// 调用函数时，指定输入参数类型
fn<string>('wanzi');
```

此时，变量 `T` 中保存的就是 `string` 类型。因为 `T` 是一个变量，所以在后续的声明中或函数内部当然可以使用它，尤其是输入和输出。

```
// 指定输入和输出的类型均为泛型变量T，而T的实际类型只有在调用时才能确定。
function fn<T> (arg: T): T {
    // 函数内部也可以使用泛型变量
    let res: T = arg;
    return res;
}

fn<string>('wanzi');
fn<number>(123.456);
```

这样一来，参数的类型就不必在声明时具体指定，而是可以通过在调用时按照实际需求指定了，大大增加了参数类型的灵活性。

泛型变量

泛型变量，首先是一种变量，毫无疑问它可以接收任何类型，包括单一类型、联合类型、类型别名或接口类型等。

```
// 类型别名
type MyType_NSB = number | string | boolean;
// 接口类型
interface IMyType_NSB {
    (a: MyType_NSB): MyType_NSB
}

function fn<T>(arg: T): T {
    return arg;
}

fn<number | string | boolean>(false);
fn<MyType_NSB>(123);
fn<IMyType_NSB>((a: MyType_NSB): MyType_NSB => a);
```

类型参数

泛型变量，其实是一种类型参数。因此，泛型变量也可以像函数参数一样指定多个，且传入与接收的位置始终一一对应。

```
function fn<T, U> (a: T, b: U): T {
    return a;
}

fn<string, number>('丸子', 14);
fn<string, number>('樱桃', 15);
```

类型推论

由于类型推论机制的存在，调用时其实可以不指定类型。类型推论机制会自动推论各参数的类型，并将其传给相应位置上的类型参数。

```
function fn<T, U> (a: T, b: U): T {
    return a;
}

// 应用类型推论
fn('丸子', 14);
fn('樱桃', 15);
```

泛型约束

泛型变量的类型只有在调用时才能确定，并且永远不会被固定为某一种确切的类型，它相当于是一种动态的类型。它的类型在声明时有无限的可能性，是无法确定的。因此，在声明函数，尤其是在实现函数的逻辑时，必须要考虑到所有类型的情况。

比如，下面的函数实现就没有考虑到泛型变量的类型拥有无限可能的特点，导致报错。因为当它为其他类型时，未必会拥有 `name` 属性。

```
function fn<T>(a: T): T {
    return a.name; // Property 'name' does not exist on type 'T'.
}

fn({name: 'wanzi'});
```

如果要保持泛型始终能够代表所有的数据类型，那么泛型自身便不会拥有特定于任一类型的属性和方法（包括原生的和自定义的）。

解决上例的问题，只需帮助类型检查机制将泛型数据（即：`a` 的值）确定为特定的类型即可。可以在操作特定类型时，使用类型断言。

```
interface ITest {
    name: string
}

function fn<T>(a: T) {
    return (a as ITest).name; // 使用类型断言，但如果传入参数不合规范，那么执行后将会得到undefined。
}

console.log(fn({ name: 'wanzi' })); // 'wanzi'
console.log(fn(1)); // undefined
```

然而，如果要使得泛型能够拥有某特定类型的属性或方法的话，则可以使用泛型约束。泛型约束，就是让泛型继承特定的类型，如下。

```ts
interface ITest {
    name: string
}

// 使泛型变量T继承ITest接口，从而拥有其特性
function fn<T extends ITest>(a: T) {
    return a.name;
}

// 调用时，传入参数必须兼容接口的类型标准
console.log(fn({ name: 'wanzi' })); // 'wanzi'
```

一旦对泛型变量进行了类型约束，那么该泛型变量便被固定成了确切的类型，不能再代表其他类型了。然后在调用函数时，只能传入兼容特定类型标准的数据，即：可以多，但不能少。

```ts
// 刚好兼容接口类型
fn({name: 'wanzi'}); // 允许

// 溢出兼容接口类型
fn({name: 'wanzi', age: 14}); // 允许

// 没有兼容接口类型
fn({a: 1, b: 2}); // 报错
fn([1, 2, 3, 4]); // 报错
fn(1234);         // 报错
```

泛型约束如同直接将函数参数固定成了指定的类型。但它要比后者更加灵活，因为后者，要求实参必须完全符合类型要求，而不是兼容。

```ts
interface ITest {
    name: string
}

// 直接对参数进行类型注解
function fn(a: ITest) {
    return a.name;
}

// 实参必须完全符合接口类型的要求
fn({name: 'wanzi'});

// 不能对实参进行任何扩展，只能且必须添加接口中已知的属性
fn({name: 'wanzi', age: 14});
// Object literal may only specify known properties, and 'age' does not exist in
type 'ITest'.
```

相互制约

前面提到过，泛型变量其实是一种类型参数。那么，泛型变量便可以向在 `JS` 中使用参数那样，借用其他参数的值。只不过，泛型变量借用的是类型，而且前面的泛型变量可以借用后面泛型变量的类型。

由此，便可以使用泛型变量去约束（即：继承）其他的泛型变量了。而继承者必须兼容被继承者，如前所述。

```
// 使T继承U，那么传参时T必须兼容U
function fn<T extends U, U>(a: T, b: U) {}

// T刚好兼容U
fn({name: 'wanzi'}, {name: ''}); // 允许
// T溢出兼容U
fn({name: 'wanzi', age: 14}, {name: ''}); // 允许

// T没有兼容U
fn({age: 14}, {name: ''}); // 报错
```

然而，相互制约制约的是类型，而不是值。那么，只要被制约者的类型满足制约者类型要求即可。于是，便出现了如下的奇特场景。

```
// 允许的行为
fn(1, 2);
fn('abc', 'xyz');
fn(true, false);

// 错误的行为
fn(undefined, null);
```

工具类型

`TS` 提供了很多实用的工具类型（`Utility Types`，实用程序类型），这些工具类型通常会使用到 `typeof` 和 `keyof` 这两个操作符。

更多工具类型参考：`Utility Types`

`typeof`

在 `JS` 中，`typeof` 操作符用于获取某个值的类型，并以字符串的形式给出检测到的类型。由于 `TS` 的类型系统限制的是变量的类型，所以 `TS` 的 `typeof` 用于获取某个变量的类型，并且以数据类型的形式给出检测到的类型，这意味着其检测结果还可以继续作为类型使用。

```
let an: any = 1;

type T = typeof an; // 由于检测的是变量的类型，所以T为any类型

let t: T = '';
```

当然，也可以利用类型推论机制，首先推论变量的类型。然后，再使用 `typeof` 去获取推论的类型，尤其是用在对象或函数的类型中。

```
let userInfo = {
    name: '丸子',
    age: 14,
    sex: '女',
    like: ['电影', '唱歌', '游戏']
};
```

```
type Info = typeof userInfo;

// 最终，Info会接收到一个如下的类型：
/*
type Info = {
    name: string;
    age: number;
    sex: string;
    like: string[];
}
*/
```

```
function fn(x: number, y: number): number[] {
    return [x, y];
}

type Fn = typeof fn; // 语法提示结果：type Fn = (x: number, y: number) => number[]
```

`keyof`

`keyof` 用于获取指定类型中的属性，并将其字面量类型联合成一个类型。尤其用在对象类型和数组类型中，联合其属性的字面量类型。

```
interface IPerson {
    name: string;
    age: number;
    1: number;
}

type Test = keyof IPerson; // 相当于：type Test = 'name' | 'age' | 1;

let test: Test;
// 只接受类型中的已知属性名
test = 'name';
test = 'age';
test = 1;
```

`typeof` 只能用来检测变量的类型，不能用来检测类型本身（包括字面量类型和其他类型）。而 `keyof` 虽然不可以检测变量的类型（那是 `typeof` 应该做的事），但它可以用于检测指定类型（包括字面量类型和其他类型）的属性字面量类型。

```
// 用于检测字面量类型
type T = keyof '';
// type T = number | typeof Symbol.iterator | "toString" | "charAt" |
"charCodeAt" | "concat" | "indexOf" | "lastIndexOf" | "localeCompare" | "match" |
"replace" | "search" | "slice" | ... 28 more ... | "sup"

type T = keyof 0;
// type T = "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf"
| "toLocaleString"

type T = keyof true; // type T = "valueOf"
```

```ts
type T = keyof [];
// type T = number | 'concat' | 'copyWithin' | 'entries' | 'every' | ... |
'values'

type T = keyof [1, 2];
// type T = number | '0' | '1' | 'concat' | 'copyWithin' | 'entries' | 'every' |
... | 'values'

type T = keyof {}; // type T = never

type T = keyof {name: 'wanzi', age: 14};// type T = "name" | "age"

type T = keyof undefined; // type T = never

type T = keyof null; // type T = never
```

```ts
// 用于检测其它的类型
type T = keyof string;
// type T = number | typeof Symbol.iterator | "toString" | "charAt" |
"charCodeAt" | "concat" | "indexOf" | "lastIndexOf" | "localeCompare" | "match" |
"replace" | "search" | "slice" | ... 28 more ... | "sup"

type T = keyof number;
// type T = "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf"
| "toLocaleString"

type T = keyof boolean; // type T = "valueOf"

type T = keyof Array<any>;
// type T = number | 'concat' | 'copyWithin' | 'entries' | 'every' | ... |
'values'

type T = keyof Function;
// type T = 'apply' | 'arguments' | 'bind' | 'call' | 'caller' | 'length' |
'name' | 'prototype' | 'toString'

type T = keyof object; // type T = never

type T = keyof void; // type T = never

type T = keyof any; // type T = string | number | symbol

type T = keyof never; // type T = string | number | symbol
```

但在动态属性（即：索引签名）中，`keyof` 会检测到动态属性的类型，而不是该类型中某个具体的字面量类型。

```ts
interface IT {
    name: string;
    [a: number]: number;
    [b: symbol]: symbol;
}

type T = keyof IT; // type T = 'name' | number | symbol
```

如果索引签名是 `string` 类型，则 `keyof` 的检测结果中还会额外包含 `number` 类型。因为访问对象的数字属性，要使用中括号语法，而中括号中允许存在 `number` 型和 `string` 型。并且在中括号中访问数字属性时，既可以使用数字本身，也可以使用字符串型数字。

```
interface IT {
    name: string;
    [prop: string]: string;
}

type T = keyof IT; // type T = string | number
```

如前所述，`keyof` 返回的是联合类型。因此，在声明变量时，也可以直接使用其检测到的联合类型，以限定变量的类型范围。

```
let a: keyof number;
// let a: "toString" | "toFixed" | "toExponential" | "toPrecision" | "valueOf" |
"toLocaleString"

a = "toString";
a = "valueOf";
```

映射类型

`TS` 中的映射，类似于数学中映射的概念。映射类型，就是将一个类型映射成另一个结构基本相同的类型，常用在复杂的类型中。但是它可以对映射出来的类型进行一些个性化的设计，比如将其中的某些部分设为可选或只读等。

使用内置工具类型 `Partial` 可以将指定类型中的所有属性设为可选。在 `Partial` 后紧跟一个被尖括号包裹的类型以获得其映射类型。

```
type Person = {
    name: string;
    age: number;
    sex: string
};

// 在Person类型上映射出一个新的类型，并在新类型中将原先所有的必选属性转为可选属性。
type T = Partial<Person>;
/*
type T = {
    name?: string | undefined;
    age?: number | undefined;
    sex?: string | undefined;
}
*/
```

下面来看看最简单的映射类型和它的组成部分：

```ts
type Keys = 'option1' | 'option2';

// {[属性名 in 属性名集合]}: 属性值的类型
type Flags = {
    [K in Keys]: boolean
};
```

它的语法与索引签名的语法类似，内部使用了 `for-in`（`TS` 的 `for-in` 可以遍历联合类型）。 具有以下三个部分：

1. 类型变量 `K`，它会依次绑定到每个属性。
2. 字符串字面量联合的 `Keys`，它包含了要迭代的属性名的集合。
3. 属性值的类型。

在这个简单的例子里， `Keys` 是硬编码的的属性名列表并且属性值的类型永远是 `boolean`，因此这个映射类型等同于：

```ts
type Flags = {
    option1: boolean;
    option2: boolean;
}
```

在真正的应用里，可能不同于上面情况的 `Readonly` 或 `Partial` 工具类型会基于一些已存在的类型，且按照一定的方式转换字段。 这就是 `keyof` 和索引访问类型要做的事情：

```ts
type NullablePerson = { [P in keyof Person]: Person[P] | null } // 可空类型
type PartialPerson = { [P in keyof Person]?: Person[P] }        // 可选类型
```

但它更有用的地方是可以有一些通用版本。那就是像定义函数的泛型变量一样，在工具类型名之后紧跟一个被尖括号包裹的类型变量。然后，在使用到该工具类型时，也可以像调用函数那样传入指定的类型，并最终得到一个新的类型。

可以将工具类型视作一种函数，因为它们也是为了完成某些任务而设计的，或者说它们具有特定的功能性（这正是函数所独有的特点）。

```ts
// 定义工具类型
type Nullable<T> = { [P in keyof T]: T[P] | null }  // 可空类型
type Partialable<T> = { [P in keyof T]?: T[P] }     // 可选类型

// 使用工具类型
type MyNullable = Nullable<{name: string; age: number; sex: string}>
type MyPartialable = Partialable<{name: string; age: number; sex: string}>
```

```
// 结果如下：
type MyNullable = {
    name: string | null;
    age: number | null;
    sex: string | null;
}

type MyPartialable = {
    name?: string | undefined;
    age?: number | undefined;
    sex?: string | undefined;
}
```

在这些例子里，属性列表是 `keyof T` 且属性值类型是 `T[P]` 的变体。 这是使用通用映射类型的一个好模版。 因为这类转换是同态的，映射只作用于 `T` 的属性而没有其它的。 编译器知道在添加任何新属性之前可以拷贝所有存在的属性修饰符。 例如，假设 `Person.name` 是只读的，那么 `Partial<Person>.name` 也将是只读的且为可选的。

深度映射

默认情况下，工具类型只会映射到根属性，不会映射二级及以下的属性。

```
interface IUserInfo {
    id: number;
    name: string;
    info: {
        a: number;
        b: string
    }
}

type T = Partial<IUserInfo>
/*
type T = {
    id?: number | undefined;
    name?: string | undefined;
    info?: {
        a: number;
        b: string;
    } | undefined;
}
*/
```


而要实现深度映射，使映射效果作用到更深的层次，则可以使用递归的方式进行深度遍历和分析。解决这个问题的关键在于，如何确定是否还具有二级属性，即：属性值本身是否是对象类型。要判断一个类型是否满足指定类型的标准，可以使用 `extends` 关键字。因为只有当继承者满足被继承者的类型要求后，`extends` 关键字才会允许继承发生。

当使用 `extends` 关键字来判断两个类型之间的是否具有继承关系且前者是否能够继承后者时，它要求在获得判断结果后必须返回类型。

```
// 格式：A类型 extends B类型 ? 条件成立时返回的类型 : 条件失败时返回的类型;
type T = string extends object ? boolean : number; // type T = number
```

```ts
type T = string[] extends object ? boolean : number; // type T = boolean

type T = object extends object ? boolean : number; // type T = boolean

type T = {
    name: string
} extends {
    age: number
} ? boolean : number; // type T = number

type T = {
    name: string;
    age: number
} extends {
    age: number
} ? boolean : number; // type T = boolean
```

利用这一点，便可以判断在 `A extends object` 中，`A` 类型是否是 `object` 类型或其子类型。

```ts
type DeepPartial<T> = {
    [P in keyof T]?: T[P] extends object ? DeepPartial<T[P]> : T[P];
}
```

现在已经实现了深度映射，但是 `TS` 的类型检查机制并不能检测到下级属性的修饰符。只有当写到下级属性时，才能看到它们的修饰符。

```ts
interface IUserInfo {
    id: number;
    name: string;
    info: {
        a: number;
        b: string
    }
}

type T = DeepPartial<IUserInfo>

let t: T = {
    id: 1,
    info: {
        a: 1 // 写到这里，才能知道a是可选属性
    }
}
```

但上面 `DeepPartial` 类型并不是完美的，当 `info` 本身就具有 `?` 修饰符时，会导致深度映射失败。因为此时 `info` 的类型不单单是一个 `object` 类型，而是一个包含 `undefined` 的联合类型。所以在 `T[P] extends object` 的判断中，会得到条件不成立的结果。

```ts
// 对DeepPartial进一步的完善
type DeepPartial<T> = {
    [P in keyof T]?: T[P] extends object | undefined ? DeepPartial<T[P]> : T[P];
}
```

逆向映射

逆向映射，就是将属性的某个修饰符移除。在要移除的属性修饰符前面，使用一个 `-` 移除符。例如：`-?` 表示将可选属性转为必选属性。

```
// 如下是必选工具类型的内部实现
type Required<T> = {
    [P in keyof T]-?: T[P];
}
```

同样的，`Required` 工具类型也只能映射到一级属性，对于下级属性是影响不到的。只能通过递归等方式自定义可深度映射的工具类型。

```
interface IUserInfo {
    id: number;
    name: string;
    age?: number;
    info?: {
        a?: number;
        b?: string;
    };
    tup?: [number?, string?];
}

// 对一级可选属性的兼容
type DeepRequired<T> = {
    [P in keyof T]-?: T[P] extends object | undefined ? DeepRequired<T[P]> :
T[P];
}

type T = DeepRequired<IUserInfo>;
/*
type T = {
    id: number;
    name: string;
    age: number;
    info: {
        a: number;
        b: string;
    }
    tup: [number, string];
}
*/
```

只读属性

使用 `Readonly` 工具类型可以将属性设为只读，其原理就是在一级属性前加上一个 `readonly` 关键字，如下所示。

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

```
// 深度映射
type DeepReadonly<T> = {
    readonly [P in keyof T]: T[P] extends object | undefined ?
DeepReadonly<T[P]> : T[P];
}
```

而想要将只读属性恢复成普通的属性，只需将前缀的 `readonly` 关键字移除即可。

```
type ResetReadonly<T> = {
    -readonly [P in keyof T]: T[P];
}

// 深度移除
type DeepResetReadonly<T> = {
    -readonly [P in keyof T]: T[P] extends object | undefined ?
DeepResetReadonly<T[P]> : T[P];
}
```

挑选属性

`Pick` 工具类型用于从指定的类型中挑出所需的属性，并将其组成一个新的类型返回。`Pick` 接收两个参数，首参指定源类型，次参以联合类型的方式指定所需的属性。

```
interface IUserInfo {
    name: string;
    age: number;
    sex?: string;
    info?: {
        a?: number;
        b?: string;
    };
    tup?: [number, string];
}

type T = Pick<IUserInfo, "name" | "age">;
/*
type T = {
    name: string;
    age?: number | undefined;
}
*/
```

其原理如下，其中 `K extends keyof T` 表示参数 `K` 是一个继承自 `keyof T` 的类型，通常为其传入硬编码的属性字面量联合类型。

```
// 类型K虽然继承自keyof T，但不能超过它的范围，因为只允许挑选已知的属性。
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
}
```

```
// K extends keyof T
// 当使用extends关键字继承一个联合类型时，意味着将继承者的类型限制在被继承者的联合类型范围之内。
type T = "a" extends "a" | "b" ? true : false; // type T = true
type T = "b" extends "a" | "b" ? true : false; // type T = true
type T = "c" extends "a" | "b" ? true : false; // type T = false

type T = "a" | "b" extends "a" | "b" ? true : false;        // type T = true
type T = "a" | "b" | "c" extends "a" | "b" ? true : false;  // type T = false
```

记录类型

`Record` 类型用于构造一个新的对象类型，并将第一个参数作为其属性名的类型，第二个参数作为其属性值的类型。

```
// 联合类型
type T = Record<"a" | "b" | "c", boolean>;
/*
type T = {
    a: boolean;
    b: boolean;
    c: boolean;
}
*/

// 单一类型
type T = Record<number, boolean>; // type T = {[x: number]: boolean;}
```

其原理如下：

```
type Record<K extends string | number | symbol, T> = { [P in K]: T; }
```

返回类型

`ReturnType` 工具类型用于获取函数类型中返回值的类型，它接收一个函数类型的参数。

```
function fn(a: number): any {
    return "";
}

// 当typeof函数时，会获得其函数类型
type T = typeof fn; // type T = (a: number) => any

type U = ReturnType<T>; // type U = any
```

其原理如下，其中使用了 `infer` 关键字来推论返回值的类型。它引入了一个类型变量 `R` 来保存其推论的返回值类型，且这个类型变量 `R` 只能用在条件类型的真分支语句中。【`infer`：推论】

```
// 推论返回值的类型，并且当传入的类型T满足条件（即：为函数类型）时，返回该推论结果；不满足则返回any型。
type ReturnType<T extends (...args: any) => any> = T extends (...args: any) => infer R ? R : any
```

提取类型

`Extract` 工具类型用于提取两个类型中的交叉类型，接收两个参数。【`Extract`：提取，摘录】

```
type A = Extract<"a" | "b", "a" | "b" | "c">; // type A = "a" | "b"
```

其原理如下：

```
type Extract<T, U> = T extends U ? T : never
```

由其原理可知，当两者类型中有重叠部分时，返回 `T` 与 `U` 的交叉类型；无重叠部分时，返回 `never` 类型。

```
type T = Extract<"a" | "b", "a" | "c" | "d">; // type T = "a"

type T = Extract<{a: string; b: string} | {c: string;}, {a: string;}>; // type T
= {a: string; b: string;}
```

排除类型

`Exclude` 工具类型用于从源类型中排除出指定的类型，接收两个参数（源类型，要排除的类型）。【`exclude`：排除】

```
type Union = "a" | "b" | "c";

// 从Union类型中，排除出"b" | "c"类型
type T = Exclude<Union, "b" | "c">; // type T = "a"
```

其原理如下：

```
type Exclude<T, U> = T extends U ? never : T
```

由其原理可知，当两者类型中有重叠部分时，返回 `never` 类型；无重叠时，表示从 `T` 中剔除与 `U` 的交叉类型，返回 `T` 的剩余部分。

```
// 有交叉，去交叉。
type T = Exclude<"a" | "b", "b" | "c">; // type T = "a"

// 无交叉，无剔除。
type T = Exclude<"a" | "b", "c">;       // type T = "a" | "b"

// 条件满足
type T = Exclude< "a", "a" | "b">;          // type T = never
type T = Exclude< "a" | "b", "a" | "b">;    // type T = never
```

由其原理可知，`Exclude` 工具类型最适合于联合类型。不能用于 `T` 继承自 `U` 的情况，否则会得到 `never` 类型，如下所示。

```
// type T = never的情况
type T = Exclude<{a: string;}, {a: string;}>;

type T = Exclude<{a: string; b?: number;}, {a: string;}>;

type T = Exclude<{a: string; b?: number;}, {}>;

type T = Exclude<{a: string; b?: number; } | {c: string}, {}>;

// 推荐用法
type T = Exclude<{a: string;} | {c: string}, {a: string}>; // type T = {c:
string;}

type T = Exclude<{a: string; b?: number;} | {c: string}, {a: string}>; // type T
= {c: string;}
```

删除属性

`Omit` 类型用于在源类型上构造一个新的对象类型，并将源类型中的指定属性删除。【`omit`：省略，删除】

```
type UserInfo = {
    id: number;
    name: string;
    age?: number;
    sex?: string;
};

// Omit<源类型，要删除的属性类型>
type T = Omit<UserInfo, "age" | "sex">; // type T = {id: number; name: string;}
```

其原理如下：

```
type Omit<T, K extends string | number | symbol> = {
    [P in Exclude<keyof T, K>]: T[P];
}
```

由其原理可知，`Omit` 首先利用 `Exclude` 从源类型的属性中删除两者重叠的属性，然后将源类型中剩余的属性及其值置入新类型中。

非空类型

`NonNullable` 类型用于从源类型中祛除空类型，即：`undefined` 和 `null`，不包括 `void` 类型。

```
type T = NonNullable<number | string | undefined | null | void>; // type T =
string | number | (void & {})
```

其原理如下：

```
// 旧版写法：
type NonNullable<T> = T extends null | undefined ? never : T

// 新版写法：
type NonNullable<T> = T & {}
```

两版由于原理不同，所以对于 `void` 类型的处理也不同。

```
type N1<T> = T extends null | undefined ? never : T;
type N2<T> = T & {};

type T1 = N1<void>; // type T1 = void
type T2 = N2<void>; // type T2 = void & {}
```

参数类型

`Parameters` 类型用于获取指定函数类型中参数的类型，并将获得的类型放入一个元组类型中。

```
function fn(a: string, b?: number, c = false): void {}

type T = Parameters<typeof fn>; // type T = [a: string, b?: number | undefined,
c?: boolean | undefined]
```

其原理如下，其中使用 `infer` 推断参数的类型，并将其推论结果置于条件类型的真分支语句中。

```
type Parameters<T extends (...args: any) => any> = T extends (...args: infer P)
=> any ? P : never
```

由其原理可知，当传入的是函数类型时，返回其参数的类型；如不是函数类型，则返回 `never` 类型。

```
type T = Parameters<(a: string, b: number) => void>; // type T = [a: string, b:
number]
```

面向对象

`TS` 主要是在 `JS` 的基础上对类型进行了较为严格的要求。因此，在面向对象编程中也会在 `JS` 的写法有所增益。

声明

预先声明

在 `TS` 的类中通过 `this` 读写指定的属性或方法时，必须在类中的顶部位置预先声明属性或方法，否则将不能被 `this` 调用。

```
// 在JS中
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}

console.log(new Point(0, 0)); // Point { x: 0, y: 0 }
```

以上代码在 `JS` 中运行没有任何问题。但当将它迁移到 `TS` 时，便会因为没有预先 `x` 和 `y` 声明而报错，因为类型 `Point` 上没有它们。

```
// 在TS中
class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

console.log(new Point(0, 0)); // Point { x: 0, y: 0 }
```

`TS` 严格限制类型，禁止未经声明地在类型上自定义属性。因此对于 `TS` 类来说，只有预先声明的属性和方法，才会被添加到其类型上。

参数声明

`TS` 会在公共的属性和方法前，默认加上一个 `public` 修饰符，通常可以省略。上面的代码，实际上应该是这样的。

```
class Point {
    public x: number;
    public y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

console.log(new Point(0, 0)); // Point { x: 0, y: 0 }
```

实际上，`public` 不仅可以用于声明属性和方法，还可以用于声明参数，并且 `public` 参数会成为类类型上的属性。了解到这一点之后，便可以解决将所有属性声明都堆叠到类顶部的问题了。

```
class Point {
    constructor(public x: number, public y: number) {
        this.x = x;
        this.y = y;
    }
}

console.log(new Point(0, 0)); // Point { x: 0, y: 0 }
```

类修饰符

为标注类字段的使用权限，`TS` 为类类型提供了三个修饰符：`private`（私有的）、`protected`（受保护的）和 `public`（公共的）。

其中，私有字段只能在类自身的内部使用，受保护字段只能在类及其子类的内部使用，公共字段则可以在类及其子类的内外部使用。

```
class Base {
    public a: string;
    protected b: number;
    private c: boolean;
    constructor(a: string, b: number, c: boolean) {
        this.a = a;
        this.b = b;
        this.c = c;
        console.log(this.a, this.b, this.c); // 's' 0 true
    }
}

class Sub extends Base {
    constructor(a: string, b: number, c: boolean) {
        super(a, b, c);
        console.log(this.a); // 's'
        console.log(this.b); // 0
        console.log(this.c); // Property 'c' is private and only accessible
within class 'Base'.
    }
}

let sub = new Sub('s', 0, true);
```

只读字段

在类中也可以使用 `readonly` 修饰符来将字段设为只读。更重要的是，它可以结合类修饰符一起使用，但必须放在它们后面。

初始化：只读字段只能在 `constructor()` 中进行赋值，被称为初始化。然后它们在其他任何地方都只能访问，不能再修改了。

```
class Base {
    public readonly a: string;
    protected readonly b: number;
    private readonly c: boolean;
    constructor(a: string, b: number, c: boolean) {
        // 初始化只读字段
        this.a = a;
        this.b = b;
        this.c = c;
        console.log(this.a, this.b, this.c); // 'b' 0 true
    }
    fn() {
        // 只能访问，不能修改
        this.a = '10'; // Cannot assign to 'a' because it is a read-only
property.
    }
}

let base = new Base('b', 0, true);
```


在 Vue 中使用

创建 `vue` 项目时，可以勾选 `TypeScript` 来将项目中原先支持 `JS` 语法的地方替换成支持 `TS` 的语法。

变化：

- 项目中所有的 `js` 文件都会被升级为 `ts` 文件。
- 项目中所有支持 `JS` 语法的地方，也都支持 `TS` 的语法，并被 `TS` 的语法所替换。

注意：要清楚项目中 `Vue` 和 `JS` 各自的环境。`Vue` 具有自己的语法，不要在该使用 `Vue` 语法的地方使用 `JS` 或 `TS` 语法。

```
<template>
    <div class="home">
        {{a}}
    </div>
</template>

<script lang="ts" setup>
    let a: string = "Home";
</script>
```

配置

配置文件

通过 `tsc --init` 指令，或者直接在项目根目录下创建，都会将隐藏在项目根目录下的 `tsconfig.json` 配置文件生成出来。

更多参考：`tsconfig`

基本配置

更多参考：基本配置

```
{
    "extends": "@tsconfig/node12/tsconfig.json",
    "compilerOptions": {
        "preserveConstEnums": true
    },
    "include": [], // 指定需要被编译的文件
    "exclude": ["node_modules", "**/*.spec.ts"] // 指定不要被编译的文件
}
```

`extends`

配置是可以继承的，通过 `extends` 就可以使该配置文件继承另一个配置文件。其值是要继承的另一个配置文件的路径字符串。

更多参考：`extends` 配置

`files`

`files` 用于指定需要被编译的文件（如：`.ts`、`.tsx`、`.d`、`.js` 和 `.jsx`）。如果找不到，就会报错。

编译文件：在 `files` 中指定要被编译的 `ts` 文件后，直接执行 `tsc` 指令，便可以将指定的 `ts` 文件编译成相应的 `js` 文件了。

```
"files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "tsc.ts"
]
```

在 `files` 中必须指定确切的文件路径（相对或绝对），其灵活性不高。当需要指定很多文件或一个目录时，则应配置 `include` 选项。

更多参考：`files` 配置

`include`

`include` 用于指定需要被编译的文件或目录（包括其中的文件和所有后代文件）。

由于要编译的文件可能很多，因此 `"include"` 允许在其数组中使用一些正则表达式（支持使用通配符来创建全局匹配模式）。

- `*`：匹配零个或多个字符（不包括目录分隔符）
- `?`：匹配任意一个字符（不包括目录分隔符）
- `**/`：匹配任何嵌套到任何级别的目录

如果在全局匹配模式中未指定文件扩展名，则只能包含受支持的扩展名的文件（例如：默认的 `.ts`、`.tsx` 和 `.d`。如果 `allowJs` 设置为 `true`，则默认为 `.js` 和 `.jsx`）。

```
// 默认匹配.ts、.tsx和.d文件
"include": [
    "./src",         // 匹配src目录及其后代目录下的所有文件
    "./src/*",       // 匹配src目录下的所有文件，不含其后代
    "./src/**/*"     // 与"./src"一样
]
```

更多参考：include 配置

`exclude`

`exclude` 用于指定不需要被编译的文件或目录（包括其中的文件和所有后代文件）。

它的语法与 `include` 相似。通常用来从 `include` 中排除不需要被编译的文件或目录。

更多参考：exclude 配置

初始配置

执行 `tsc --init` 指令，会在控制台提示 `TS` 的初始化配置，并将这些初始化配置加到生成的 `tsconfig.json` 配置文件中。

```
Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true


You can learn more at https://aka.ms/tsconfig
```

编译配置

在初始化的 `tsconfig.json` 中，默认只有一个根节点 `"compilerOptions"`，即：`TS` 的编译器配置。

```
{
    "compilerOptions": {...}
}
```

以下列出部分的编译器配置，所有配置请参考：编译配置、MSBuild 中的编译配置 【MSBuild：Microsoft Build Engine】

目标版本

在编译器的配置中，有一个 "target" 根节点。它用于配置将 ts 文件编译成指定 ES 版本的 js 文件，值为 ES 版本。

```
"target": "es2016" // 编译为：ES5版本的js文件，并使它尽量向下（向后）兼容之前的版本。
```

```
let a: number = 0;      // 编译结果：var a = 0;

const b: string = '';   // 编译结果：var b = '';
```

模块系统

module 用于为程序指定模块系统。在 node 项目中，很可能通常需要的是 "CommonJS"。

```
"module": "commonjs" // 采用commonjs标准
```

更多参考：module 配置

严格模式

```
"strict": true // 启用所有严格的类型检查设置
```

模块交互

```
"esModuleInterop": true // ES模块之间的交互操作
```

开启 "esModuleInterop" 配置，将允许模块之间进行交互，如：在模块中导入另一个模块。它会生成额外的 JavaScript 以简化对导入 CommonJS 模块的支持，使得 'allowSyntheticDefaultImports' 类型兼容。

声明检查

```
"skipLibCheck": true // 跳过对Lib（即：编译过程中需要引入的库文件的列表）的检查
```

开启它，表示 —— 跳过对所有的声明文件（*.d.ts）的类型检查。

这可以在编译期间节省时间，但代价是类型系统的准确性。例如，两个库可以以不一致的方式定义同一类型的两个副本。TypeScript 不会对所有 d.ts 文件进行全面检查，而是会对你在应用源代码中特别引用的代码进行类型检查。

在以下情况中，你应该开启它：

- 在 node_modules 中有两个库类型的副本时，您可能会考虑使用 skipLibCheck。在这些情况下，您应该考虑使用像 yarn 的解析这样的特性来确保您的树中只有一个依赖项副本，或者研究如

何通过理解依赖项解析来确保只有一个副本，从而在不使用其他工具的情况下解决问题。
- 当你在 `TypeScript` 版本之间迁移时，这些更改会导致 `node_modules` 和 `JS` 标准库的破坏，而你不想在 `TypeScript` 更新期间处理这些破坏。

更多参考：`Skip Lib Check`

编译引入

`lib` 用于指定在编译过程中需要引入的库文件。【`lib`：`library`】

当在项目引入一些第三方库时，可能需要将其中一些库文件引入到编译过程中使用。

`TypeScript` 为内置的 `JS APIs`（如：`Math`）提供了一个默认的类型定义集，也为出现在浏览器环境中的东西（如：`document`）提供了类型定义。`TypeScript` 还为匹配你指定的目标版本中较新的 `JS` 特性提供了 `APIs`，例如：如果目标版本是 `ES6` 或更新的，则 `Map` 的定义是可用的。

然而，出于以下原因，你可能会想要改变这些。

- 你的程序不是在浏览器中运行的，所以你不需要 `"dom"` 的类型定义
- 运行时平台提供了某些 `JavaScript API` 对象（可能是通过 `polyfills`），但还不支持给定 `ECMAScript` 版本的完整语法
- 对于更高级别的 `ECMAScript` 版本，您有一些（但不是全部）`polyfills` 或原生实现

在 `TypeScript 4.5` 中，`lib` 文件会被 `npm` 模块覆盖，更多参考：`Supporting lib from node_modules`。

```
"lib": [
    "esnext",
    "dom",
    "dom.iterable",
    "scripthost"
]
```

更多参考：`lib` 配置

导入 `JS`

`allowJs` 用于指定是否允许在项目中导入 `js` 文件。

默认情况下，在 `ts` 文件中导入 `js` 文件会导致报错。开启该配置，将会允许 `.ts` 和 `.tsx` 文件与现有的 `JavaScript` 文件共存。

```
"allowJs": true
```

更多参考：`allowJs` 配置

编译输出

`outDir` 用于指定将编译文件输出到哪个目录。其值为输出目录的路径字符串。

默认情况下，编译后的 `js` 文件将会被发送到在生成它的 `ts` 文件的同一目录。

```
"outDir": "dist" // 将js文件输出到dist目录中
```

更多参考：`outDir` 配置