2023 typescript史上最强学习入门文章(2w字)

原文地址：https://juejin.cn/post/7018805943710253086

前言

这篇文章前前后后写了几个月,之前总是零零散散的学习,然后断断续续的写文章(懒),自己参考了很多文章以及看了一些ts视频,然后把基础的知识点全部总结了一下.自我感觉比掘金上的所有`typescript入门`的热门文章都要详细 哈哈,因为那些热门文章我全部都参考了,内容基本都包含了.这一次一定得沉淀下来。好好的把这篇文章给写完.

本来自己以前是不喜欢ts的,因为它有一定的学习成本,代码量增加,代码复杂度增加等.后来慢慢觉得，ts的静态检查使得开发者提前发现错误，在前端工程化开发的今天确实有必要，因为团队成员技术水平参差不齐，`TypeScript`可以帮助避免很多错误的发生，当然如果你是`any大法`的信仰者，我劝你善良。不要为了用`TypeScript`而用`TypeScript`，用它的前提一定要是它能帮你解决特定的问题。

> 忠告:
> 不要学习TypeScript, 因为它的学习成本很低
> 不要学习TypeScript, 因为它能减少团队无效沟通
> 不要学习TypeScript, 因为它能让你的代码更健壮
> 不要学习TypeScript, 因为它能帮助你快速掌握其它后端语言
> 不要学习TypeScript, 因为你会迷恋它

~ 唉，大势所趋，这玩意现在是一定要学的了！

typescript介绍

什么是typescript?

> TypeScript简称TS
> TS和JS之间的关系其实就是Less/Sass和CSS之间的关系
> 就像Less/Sass是对CSS进行扩展一样, TS也是对JS进行扩展
> 就像Less/Sass最终会转换成CSS一样, 我们编写好的TS代码最终也会换成JS
> TypeScript是JavaScript的超集，因为它扩展了JavaScript，有JavaScript没有的东西。
> 硬要以父子类关系来说的话，TypeScript是JavaScript子类，继承的基础上去扩展。

为什么需要TypeScript?

> 简单来说就是因为JavaScript是弱类型, 很多错误只有在运行时才会被发现
> 而TypeScript提供了一套静态检测机制, 可以帮助我们在编译时就发现错误

TypeScript特点

> 支持最新的JavaScript新特特性
> 支持代码静态检查
> 支持诸如C,C++,Java,Go等后端语言中的特性 (枚举、泛型、类型转换、命名空间、声明文件、类、接口等)

搭建typescript学习环境

安装最新版typescript

```
npm i -g typescript
复制代码
```

安装ts-node

```
npm i -g ts-node
复制代码
```

创建一个 tsconfig.json 文件

```
tsc --init
复制代码
```

然后新建index.ts,输入相关练习代码，然后执行 ts-node index.ts

官方playground

官方也提供了一个在线开发 TypeScript 的云环境——Playground。

基于它，我们无须在本地安装环境，只需要一个浏览器即可随时学习和编写 TypeScript，同时还可以方便地选择 TypeScript 版本、配置 tsconfig，并对 TypeScript 实时静态类型检测、转译输出 JavaScript 和在线执行。

而且在体验上，它也一点儿不逊色于任何本地的 IDE，对于刚刚学习 TypeScript 的我们来说，算是一个不错的选择。

基础数据类型

JS的八种内置类型

```
let str: string = "jimmy";
let num: number = 24;
let bool: boolean = false;
let u: undefined = undefined;
let n: null = null;
let obj: object = {x: 1};
let big: bigint = 100n;
let sym: symbol = Symbol("me");
复制代码
```

注意点

null和undefined

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` 赋值给其他类型。

```
// null和undefined赋值给string
let str:string = "666";
str = null
str= undefined

// null和undefined赋值给number
let num:number = 666;
num = null
num= undefined

// null和undefined赋值给object
let obj:object ={};
obj = null
obj= undefined

// null和undefined赋值给Symbol
let sym: symbol = Symbol("me");
sym = null
sym= undefined

// null和undefined赋值给boolean
let isDone: boolean = false;
isDone = null
isDone= undefined

// null和undefined赋值给bigint
let big: bigint =  100n;
big = null
big= undefined
复制代码
```

如果你在tsconfig.json指定了`"strictNullChecks":true`， `null` 和 `undefined` 只能赋值给 `void` 和它们各自的类型。

number和bigint

虽然`number`和`bigint`都表示数字，但是这两个类型不兼容。

```
let big: bigint =  100n;
let num: number = 6;
big = num;
num = big;
复制代码
```

会抛出一个类型不兼容的 ts(2322) 错误。

其他类型

Array

对数组类型的定义有两种方式：

```
let arr:string[] = ["1","2"];
let arr2:Array<string> = ["1","2"];
复制代码
```

定义联合类型数组

```
let arr:(number | string)[];
// 表示定义了一个名称叫做arr的数组，
// 这个数组中将来既可以存储数值类型的数据，也可以存储字符串类型的数据
arr3 = [1, 'b', 2, 'c'];
复制代码
```

定义指定对象成员的数组：

```
// interface是接口,后面会讲到
interface Arrobj{
    name:string,
    age:number
}
let arr3:Arrobj[]=[{name:'jimmy',age:22}]
复制代码
```

函数

函数声明

```
function sum(x: number, y: number): number {
    return x + y;
}
复制代码
```

函数表达式

```
let mySum: (x: number, y: number) => number = function (x: number, y: number):
number {
    return x + y;
};
复制代码
```

用接口定义函数类型

```
interface SearchFunc{
  (source: string, subString: string): boolean;
}
复制代码
```

采用函数表达式接口定义函数的方式时，对等号左侧进行类型限制，可以保证以后对函数名赋值时保证参数个数、参数类型、返回值类型不变。

可选参数

```
function buildName(firstName: string, lastName?: string) {
    if (lastName) {
        return firstName + ' ' + lastName;
    } else {
        return firstName;
    }
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
复制代码
```

注意点：可选参数后面不允许再出现必需参数

参数默认值

```
function buildName(firstName: string, lastName: string = 'Cat') {
    return firstName + ' ' + lastName;
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
复制代码
```

剩余参数

```
function push(array: any[], ...items: any[]) {
    items.forEach(function(item) {
        array.push(item);
    });
}
let a = [];
push(a, 1, 2, 3);
复制代码
```

函数重载

由于 JavaScript 是一个动态语言，我们通常会使用不同类型的参数来调用同一个函数，该函数会根据不同的参数而返回不同的类型的调用结果：

```
function add(x, y) {
 return x + y;
}
add(1, 2); // 3
add("1", "2"); //"12"
复制代码
```

由于 TypeScript 是 JavaScript 的超集，因此以上的代码可以直接在 TypeScript 中使用，但当 TypeScript 编译器开启 `noImplicitAny` 的配置项时，以上代码会提示以下错误信息：

```
Parameter 'x' implicitly has an 'any' type.
Parameter 'y' implicitly has an 'any' type.
复制代码
```

该信息告诉我们参数 x 和参数 y 隐式具有 `any` 类型。为了解决这个问题，我们可以为参数设置一个类型。因为我们希望 `add` 函数同时支持 string 和 number 类型，因此我们可以定义一个 `string | number` 联合类型，同时我们为该联合类型取个别名：

```
type Combinable = string | number;
```
复制代码

在定义完 Combinable 联合类型后，我们来更新一下 `add` 函数：

```
function add(a: Combinable, b: Combinable) {
    if (typeof a === 'string' || typeof b === 'string') {
     return a.toString() + b.toString();
    }
    return a + b;
}
```
复制代码

为 `add` 函数的参数显式设置类型之后，之前错误的提示消息就消失了。那么此时的 `add` 函数就完美了么，我们来实际测试一下：

```
const result = add('Semlinker', ' Kakuqo');
result.split(' ');
```
复制代码

在上面代码中，我们分别使用 `'Semlinker'` 和 `' Kakuqo'` 这两个字符串作为参数调用 add 函数，并把调用结果保存到一个名为 `result` 的变量上，这时候我们想当然的认为此时 result 的变量的类型为 string，所以我们就可以正常调用字符串对象上的 `split` 方法。但这时 TypeScript 编译器又出现以下错误信息了：

```
Property 'split' does not exist on type 'number'.
```
复制代码

很明显 `number` 类型的对象上并不存在 `split` 属性。问题又来了，那如何解决呢？这时我们就可以利用 TypeScript 提供的函数重载特性。

> **函数重载或方法重载是使用相同名称和不同参数数量或类型创建多个方法的一种能力。** 要解决前面遇到的问题，方法就是为同一个函数提供多个函数类型定义来进行函数重载，编译器会根据这个列表去处理函数的调用。

```
type Types = number | string
function add(a:number,b:number):number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;
function add(a:Types, b:Types) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}
const result = add('Semlinker', ' Kakuqo');
result.split(' ');
```
复制代码

在以上代码中，我们为 add 函数提供了多个函数类型定义，从而实现函数的重载。之后，可恶的错误消息又消失了，因为这时 result 变量的类型是 `string` 类型。

Tuple(元组)

元祖定义

众所周知，数组一般由同种类型的值组成，但有时我们需要在单个变量中存储不同类型的值，这时候我们就可以使用元组。在 JavaScript 中是没有元组的，元组是 TypeScript 中特有的类型，其工作方式类似于数组。

元组最重要的特性是可以限制`数组元素的个数和类型`，它特别适合用来实现多值返回。

元祖用于保存定长定数据类型的数据

```
let x: [string, number];
// 类型必须匹配且个数必须为2

x = ['hello', 10]; // OK
x = ['hello', 10,10]; // Error
x = [10, 'hello']; // Error
复制代码
```

注意，元组类型只能表示一个已知元素数量和类型的数组，长度已指定，越界访问会提示错误。如果一个数组中可能有多种类型，数量和类型都不确定，那就直接`any[]`

元祖类型的解构赋值

我们可以通过下标的方式来访问元组中的元素，当元组中的元素较多时，这种方式并不是那么便捷。其实元组也是支持解构赋值的：

```
let employee: [number, string] = [1, "Semlinker"];
let [id, username] = employee;
console.log(`id: ${id}`);
console.log(`username: ${username}`);
复制代码
```

以上代码成功运行后，控制台会输出以下消息：

```
id: 1
username: Semlinker
复制代码
```

这里需要注意的是，在解构赋值时，如果解构数组元素的个数是不能超过元组中元素的个数，否则也会出现错误，比如：

```
let employee: [number, string] = [1, "Semlinker"];\
let [id, username, age] = employee;
复制代码
```

在以上代码中，我们新增了一个 age 变量，但此时 TypeScript 编译器会提示以下错误信息：

```
Tuple type '[number, string]' of length '2' has no element at index '2'.
复制代码
```

很明显元组类型 `[number, string]` 的长度是 `2`，在位置索引 `2` 处不存在任何元素。

元组类型的可选元素

与函数签名类型，在定义元组类型时，我们也可以通过 `?` 号来声明元组类型的可选元素，具体的示例如下：

```
let optionalTuple: [string, boolean?];
optionalTuple = ["Semlinker", true];
console.log(`optionalTuple : ${optionalTuple}`);
optionalTuple = ["Kakuqo"];
console.log(`optionalTuple : ${optionalTuple}`);
复制代码
```

在上面代码中，我们定义了一个名为 `optionalTuple` 的变量，该变量的类型要求包含一个必须的字符串属性和一个可选布尔属性，该代码正常运行后，控制台会输出以下内容：

```
optionalTuple : Semlinker,true
optionalTuple : Kakuqo
复制代码
```

那么在实际工作中，声明可选的元组元素有什么作用？这里我们来举一个例子，在三维坐标轴中，一个坐标点可以使用 `(x, y, z)` 的形式来表示，对于二维坐标轴来说，坐标点可以使用 `(x, y)` 的形式来表示，而对于一维坐标轴来说，只要使用 `(x)` 的形式来表示即可。针对这种情形，在 TypeScript 中就可以利用元组类型可选元素的特性来定义一个元组类型的坐标点，具体实现如下：

```
type Point = [number, number?, number?];

const x: Point = [10]; // 一维坐标点
const xy: Point = [10, 20]; // 二维坐标点
const xyz: Point = [10, 20, 10]; // 三维坐标点

console.log(x.length); // 1
console.log(xy.length); // 2
console.log(xyz.length); // 3
复制代码
```

元组类型的剩余元素

元组类型里最后一个元素可以是剩余元素，形式为 `...X`，这里 `X` 是数组类型。**剩余元素代表元组类型是开放的，可以有零个或多个额外的元素。** 例如，`[number, ...string[]]` 表示带有一个 `number` 元素和任意数量 `string` 类型元素的元组类型。为了能更好的理解，我们来举个具体的例子：

```
type RestTupleType = [number, ...string[]];
let restTuple: RestTupleType = [666, "Semlinker", "Kakuqo", "Lolo"];
console.log(restTuple[0]);
console.log(restTuple[1]);
复制代码
```

只读的元组类型

TypeScript 3.4 还引入了对只读元组的新支持。我们可以为任何元组类型加上 `readonly` 关键字前缀，以使其成为只读元组。具体的示例如下：

```
const point: readonly [number, number] = [10, 20];
复制代码
```

在使用 `readonly` 关键字修饰元组类型之后，任何企图修改元组中元素的操作都会抛出异常：

```
// Cannot assign to '0' because it is a read-only property.
point[0] = 1;
// Property 'push' does not exist on type 'readonly [number, number]'.
point.push(0);
// Property 'pop' does not exist on type 'readonly [number, number]'.
point.pop();
// Property 'splice' does not exist on type 'readonly [number, number]'.
point.splice(1, 1);
复制代码
```

void

`void` 表示没有任何类型，和其他类型是平等关系，不能直接赋值:

```
let a: void;
let b: number = a; // Error
复制代码
```

你只能为它赋予 `null` 和 `undefined` （在 `strictNullChecks` 未指定为true时）。声明一个 `void` 类型的变量没有什么大用，我们一般也只有在函数没有返回值时去声明。

值得注意的是，方法没有返回值将得到 `undefined`，但是我们需要定义成 `void` 类型，而不是 `undefined` 类型。否则将报错:

```
function fun(): undefined {
  console.log("this is TypeScript");
};
fun(); // Error
复制代码
```

never

`never` 类型表示的是那些永不存在的值的类型。

值会永不存在的两种情况：

1. 如果一个函数执行时抛出了**异常**，那么这个函数永远不存在返回值（因为抛出异常会直接中断程序运行，这使得程序运行不到返回值那一步，即具有不可达的终点，也就永不存在返回了）；
2. 函数中执行无限循环的代码（**死循环**），使得程序永远无法运行到函数返回值那一步，永不存在返回。

```
// 异常
function err(msg: string): never { // OK
  throw new Error(msg);
}

// 死循环
function loopForever(): never { // OK
  while (true) {};
}

复制代码
```

`never`类型同`null`和`undefined`一样，也是任何类型的子类型，也可以赋值给任何类型。

但是没有类型是`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`

```
let ne: never;
let nev: never;
let an: any;

ne = 123; // Error
ne = nev; // OK
ne = an; // Error
ne = (() => { throw new Error("异常"); })(); // OK
ne = (() => { while(true) {} })(); // OK
复制代码
```

在 TypeScript 中，可以利用 never 类型的特性来实现全面性检查，具体示例如下：

```
type Foo = string | number;

function controlFlowAnalysisWithNever(foo: Foo) {
  if (typeof foo === "string") {
    // 这里 foo 被收窄为 string 类型
  } else if (typeof foo === "number") {
    // 这里 foo 被收窄为 number 类型
  } else {
    // foo 在这里是 never
    const check: never = foo;
  }
}
复制代码
```

注意在 else 分支里面，我们把收窄为 never 的 foo 赋值给一个显示声明的 never 变量。如果一切逻辑正确，那么这里应该能够编译通过。但是假如后来有一天你的同事修改了 Foo 的类型：

```
type Foo = string | number | boolean;
复制代码
```

然而他忘记同时修改`controlFlowAnalysisWithNever`方法中的控制流程，这时候 else 分支的 foo 类型会被收窄为`boolean`类型，导致无法赋值给 never 类型，这时就会产生一个编译错误。通过这个方式，我们可以确保`controlFlowAnalysisWithNever`方法总是穷尽了 Foo 的所有可能类型。 通过这个示例，我们可以得出一个结论：**使用 never 避免出现新增了联合类型没有对应的实现，目的就是写出类**

型绝对安全的代码。

any

在 TypeScript 中，任何类型都可以被归为 any 类型。这让 any 类型成为了类型系统的顶级类型.

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

```
let a: string = 'seven';
a = 7;
// TS2322: Type 'number' is not assignable to type 'string'.
复制代码
```

但如果是 `any` 类型，则允许被赋值为任意类型。

```
let a: any = 666;
a = "Semlinker";
a = false;
a = 66
a = undefined
a = null
a = []
a = {}
复制代码
```

在any上访问任何属性都是允许的,也允许调用任何方法.

```
let anyThing: any = 'hello';
console.log(anyThing.myName);
console.log(anyThing.myName.firstName);
let anyThing: any = 'Tom';
anyThing.setName('Jerry');
anyThing.setName('Jerry').sayHello();
anyThing.myName.setFirstName('Cat');
复制代码
```

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型：

```
let something;
something = 'seven';
something = 7;
something.setName('Tom');
复制代码
```

等价于

```
let something: any;
something = 'seven';
something = 7;
something.setName('Tom');
复制代码
```

在许多场景下，这太宽松了。使用 `any` 类型，可以很容易地编写类型正确但在运行时有问题的代码。如果我们使用 `any` 类型，就无法使用 TypeScript 提供的大量的保护机制。请记住，`any 是魔鬼！`尽量不要用any。

为了解决 `any` 带来的问题，TypeScript 3.0 引入了 `unknown` 类型。

unknown

`unknown`与`any`一样，所有类型都可以分配给`unknown`：

```
let notSure: unknown = 4;
notSure = "maybe a string instead"; // OK
notSure = false; // OK
复制代码
unknown`与`any`的最大区别是： 任何类型的值可以赋值给`any`，同时`any`类型的值也可以赋值给任何类型。`unknown` 任何类型的值都可以赋值给它，但它只能赋值给`unknown`和`any
let notSure: unknown = 4;
let uncertain: any = notSure; // OK

let notSure: any = 4;
let uncertain: unknown = notSure; // OK

let notSure: unknown = 4;
let uncertain: number = notSure; // Error

复制代码
```

如果不缩小类型，就无法对`unknown`类型执行任何操作：

```
function getDog() {
 return '123'
}

const dog: unknown = {hello: getDog};
dog.hello(); // Error
复制代码
```

这种机制起到了很强的预防性，更安全，这就要求我们必须缩小类型，我们可以使用`typeof`、`类型断言`等方式来缩小未知范围：

```
function getDogName() {
 let x: unknown;
 return x;
};
const dogName = getDogName();
// 直接使用
const upName = dogName.toLowerCase(); // Error
// typeof
if (typeof dogName === 'string') {
  const upName = dogName.toLowerCase(); // OK
}
// 类型断言
const upName = (dogName as string).toLowerCase(); // OK
复制代码
```

Number、String、Boolean、Symbol

首先，我们来回顾一下初学 TypeScript 时，很容易和原始类型 number、string、boolean、symbol 混淆的首字母大写的 Number、String、Boolean、Symbol 类型，后者是相应原始类型的`包装对象`，姑且把它们称之为对象类型。

从类型兼容性上看，原始类型兼容对应的对象类型，反过来对象类型不兼容对应的原始类型。

下面我们看一个具体的示例：

```
let num: number;
let Num: Number;
Num = num; // ok
num = Num; // ts(2322)报错
复制代码
```

在示例中的第 3 行，我们可以把 number 赋给类型 Number，但在第 4 行把 Number 赋给 number 就会提示 ts(2322) 错误。

因此，我们需要铭记不要使用对象类型来注解值的类型，因为这没有任何意义。

object、Object 和 {}

另外，object（首字母小写，以下称“小 object”）、Object（首字母大写，以下称“大 Object”）和 {}（以下称“空对象”）

小 object 代表的是所有非原始类型，也就是说我们不能把 number、string、boolean、symbol等 原始类型赋值给 object。在严格模式下，`null` 和 `undefined` 类型也不能赋给 object。

> JavaScript 中以下类型被视为原始类型：`string`、`boolean`、`number`、`bigint`、`symbol`、`null` 和 `undefined`。

下面我们看一个具体示例：

```
let lowerCaseObject: object;
lowerCaseObject = 1; // ts(2322)
lowerCaseObject = 'a'; // ts(2322)
lowerCaseObject = true; // ts(2322)
lowerCaseObject = null; // ts(2322)
lowerCaseObject = undefined; // ts(2322)
lowerCaseObject = {}; // ok
复制代码
```

在示例中的第 2~6 行都会提示 ts(2322) 错误，但是我们在第 7 行把一个空对象赋值给 object 后，则可以通过静态类型检测。

大Object 代表所有拥有 toString、hasOwnProperty 方法的类型，所以所有原始类型、非原始类型都可以赋给 Object。同样，在严格模式下，null 和 undefined 类型也不能赋给 Object。

下面我们也看一个具体的示例：

```
let upperCaseObject: Object;
upperCaseObject = 1; // ok
upperCaseObject = 'a'; // ok
upperCaseObject = true; // ok
upperCaseObject = null; // ts(2322)
upperCaseObject = undefined; // ts(2322)
upperCaseObject = {}; // ok
```
复制代码

在示例中的第 2到4 行、第 7 行都可以通过静态类型检测，而第 5~6 行则会提示 ts(2322) 错误。

从上面示例可以看到，大 Object 包含原始类型，小 object 仅包含非原始类型，所以大 Object 似乎是小 object 的父类型。实际上，大 Object 不仅是小 object 的父类型，同时也是小 object 的子类型。

下面我们还是通过一个具体的示例进行说明。

```
type isLowerCaseObjectExtendsUpperCaseObject = object extends Object ? true :
false; // true
type isUpperCaseObjectExtendsLowerCaseObject = Object extends object ? true :
false; // true
upperCaseObject = lowerCaseObject; // ok
lowerCaseObject = upperCaseObject; // ok

```
复制代码

在示例中的第 1 行和第 2 行返回的类型都是 true，第3 行和第 4 行的 upperCaseObject 与 lowerCaseObject 可以互相赋值。

> **注意：尽管官方文档说可以使用小 object 代替大 Object，但是我们仍要明白大 Object 并不完全等价于小 object。**

{}空对象类型和大 Object 一样，也是表示原始类型和非原始类型的集合，并且在严格模式下，null 和 undefined 也不能赋给 {}，如下示例：

```
let ObjectLiteral: {};
ObjectLiteral = 1; // ok
ObjectLiteral = 'a'; // ok
ObjectLiteral = true; // ok
ObjectLiteral = null; // ts(2322)
ObjectLiteral = undefined; // ts(2322)
ObjectLiteral = {}; // ok
type isLiteralCaseObjectExtendsUpperCaseObject = {} extends Object ? true :
false; // true
type isUpperCaseObjectExtendsLiteralCaseObject = Object extends {} ? true :
false; // true
upperCaseObject = ObjectLiteral;
ObjectLiteral = upperCaseObject;
```
复制代码

在示例中的第 8 行和第 9 行返回的类型都是 true，第10 行和第 11 行的 ObjectLiteral 与 upperCaseObject 可以互相赋值，第2~4 行、第 7 行的赋值操作都符合静态类型检测；而第5 行、第 6 行则会提示 ts(2322) 错误。

综上结论：{}、大 Object 是比小 object 更宽泛的类型（least specific），{} 和大 Object 可以互相代替，用来表示原始类型（null、undefined 除外）和非原始类型；而小 object 则表示非原始类型。

类型推断

```
{
  let str: string = 'this is string';
  let num: number = 1;
  let bool: boolean = true;
}
{
  const str: string = 'this is string';
  const num: number = 1;
  const bool: boolean = true;
}
复制代码
```

看着上面的示例，可能你已经在嘀咕了：定义基础类型的变量都需要写明类型注解，TypeScript 太麻烦了吧？在示例中，使用 let 定义变量时，我们写明类型注解也就罢了，毕竟值可能会被改变。可是，使用 `const` 常量时还需要写明类型注解，那可真的很麻烦。

实际上，TypeScript 早就考虑到了这么简单而明显的问题。

在很多情况下，TypeScript 会根据上下文环境自动推断出变量的类型，无须我们再写明类型注解。因此，上面的示例可以简化为如下所示内容：

```
{
  let str = 'this is string'; // 等价 let str: string = 'this is string'; 下面类似
  let num = 1; // 等价
  let bool = true; // 等价
}
{
  const str = 'this is string'; // 不等价
  const num = 1; // 不等价
  const bool = true; // 不等价
}
复制代码
```

我们把 TypeScript 这种基于赋值表达式推断类型的能力称之为`类型推断`。

在 TypeScript 中，具有初始化值的变量、有默认值的函数参数、函数返回的类型都可以根据上下文推断出来。比如我们能根据 return 语句推断函数返回的类型，如下代码所示：

```
{
  /** 根据参数的类型，推断出返回值的类型也是 number */
  function add1(a: number, b: number) {
    return a + b;
  }
  const x1= add1(1, 1); // 推断出 x1 的类型也是 number

  /** 推断参数 b 的类型是数字或者 undefined，返回值的类型也是数字 */
  function add2(a: number, b = 1) {
    return a + b;
  }
  const x2 = add2(1);
  const x3 = add2(1, '1'); // ts(2345) Argument of type "1" is not assignable to
parameter of type 'number | undefined
}
```

复制代码

如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 `any` 类型而完全不被类型检查：

```
let myFavoriteNumber;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```
复制代码

类型断言

有时候你会遇到这样的情况，你会比 TypeScript 更了解某个值的详细信息。通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

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”。类型断言好比其他语言里的类型转换，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 类型检测无法做到绝对智能，毕竟程序不能像人一样思考。有时会碰到我们比 TypeScript 更清楚实际类型的情况，比如下面的例子：

```
const arrayNumber: number[] = [1, 2, 3, 4];
const greaterThan2: number = arrayNumber.find(num => num > 2); // 提示 ts(2322)
```
复制代码

其中，greaterThan2 一定是一个数字（确切地讲是 3），因为 arrayNumber 中明显有大于 2 的成员，但静态类型对运行时的逻辑无能为力。

在 TypeScript 看来，greaterThan2 的类型既可能是数字，也可能是 undefined，所以上面的示例中提示了一个 ts(2322) 错误，此时我们不能把类型 undefined 分配给类型 number。

不过，我们可以使用一种笃定的方式——**类型断言**（类似仅作用在类型层面的强制类型转换）告诉 TypeScript 按照我们的方式做类型检查。

比如，我们可以使用 as 语法做类型断言，如下代码所示：

```
const arrayNumber: number[] = [1, 2, 3, 4];
const greaterThan2: number = arrayNumber.find(num => num > 2) as number;
```
复制代码

语法

```
// 尖括号 语法
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

// as 语法
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```
复制代码

以上两种方式虽然没有任何区别，但是尖括号格式会与react中JSX产生语法冲突，因此我们更推荐使用 as 语法。

非空断言

在上下文中当类型检查器无法断定类型时，一个新的后缀表达式操作符 `!` 可以用于断言操作对象是非 null 和非 undefined 类型。**具体而言，x! 将从 x 值域中排除 null 和 undefined 。**

具体看以下示例：

```
let mayNullOrUndefinedOrString: null | undefined | string;
mayNullOrUndefinedOrString!.toString(); // ok
mayNullOrUndefinedOrString.toString(); // ts(2531)
复制代码
type NumGenerator = () => number;

function myFunc(numGenerator: NumGenerator | undefined) {
  // Object is possibly 'undefined'.(2532)
  // Cannot invoke an object which is possibly 'undefined'.(2722)
  const num1 = numGenerator(); // Error
  const num2 = numGenerator!(); //OK
}
复制代码
```

确定赋值断言

允许在实例属性和变量声明后面放置一个 `!` 号，从而告诉 TypeScript 该属性会被明确地赋值。为了更好地理解它的作用，我们来看个具体的例子：

```
let x: number;
initialize();

// Variable 'x' is used before being assigned.(2454)
console.log(2 * x); // Error
function initialize() {
  x = 10;
}

复制代码
```

很明显该异常信息是说变量 x 在赋值前被使用了，要解决该问题，我们可以使用确定赋值断言：

```
let x!: number;
initialize();
console.log(2 * x); // Ok

function initialize() {
  x = 10;
}
复制代码
```

通过 `let x!: number;` 确定赋值断言，TypeScript 编译器就会知道该属性会被明确地赋值。

字面量类型

在 TypeScript 中，字面量不仅可以表示值，还可以表示类型，即所谓的字面量类型。

目前，TypeScript 支持 3 种字面量类型：字符串字面量类型、数字字面量类型、布尔字面量类型，对应的字符串字面量、数字字面量、布尔字面量分别拥有与其值一样的字面量类型，具体示例如下：

```
{
  let specifiedStr: 'this is string' = 'this is string';
  let specifiedNum: 1 = 1;
  let specifiedBoolean: true = true;
}
```
复制代码

比如 'this is string' （这里表示一个字符串字面量类型）类型是 string 类型（确切地说是 string 类型的子类型），而 string 类型不一定是 'this is string'（这里表示一个字符串字面量类型）类型，如下具体示例：

```
{
  let specifiedStr: 'this is string' = 'this is string';
  let str: string = 'any string';
  specifiedStr = str; // ts(2322) 类型 '"string"' 不能赋值给类型 'this is string'
  str = specifiedStr; // ok
}
```
复制代码

比如说我们用“马”比喻 string 类型，即“黑马”代指 'this is string' 类型，“黑马”肯定是“马”，但“马”不一定是“黑马”，它可能还是“白马”“灰马”。因此，'this is string' 字面量类型可以给 string 类型赋值，但是 string 类型不能给 'this is string' 字面量类型赋值，这个比喻同样适合于形容数字、布尔等其他字面量和它们父类的关系。

字符串字面量类型

一般来说，我们可以使用一个字符串字面量类型作为变量的类型，如下代码所示：

```
let hello: 'hello' = 'hello';
hello = 'hi'; // ts(2322) Type '"hi"' is not assignable to type '"hello"'
```
复制代码

实际上，定义单个的字面量类型并没有太大的用处，它真正的应用场景是可以把多个字面量类型组合成一个联合类型（后面会讲解），用来描述拥有明确成员的实用的集合。

如下代码所示，我们使用字面量联合类型描述了一个明确、可 'up' 可 'down' 的集合，这样就能清楚地知道需要的数据结构了。

```
type Direction = 'up' | 'down';

function move(dir: Direction) {
  // ...
}
move('up'); // ok
move('right'); // ts(2345) Argument of type '"right"' is not assignable to
parameter of type 'Direction'

```
复制代码

通过使用字面量类型组合的联合类型，我们可以限制函数的参数为指定的字面量类型集合，然后编译器会检查参数是否是指定的字面量类型集合里的成员。

因此，相较于使用 string 类型，使用字面量类型（组合的联合类型）可以将函数的参数限定为更具体的类型。这不仅提升了程序的可读性，还保证了函数的参数类型，可谓一举两得。

数字字面量类型及布尔字面量类型

数字字面量类型和布尔字面量类型的使用与字符串字面量类型的使用类似，我们可以使用字面量组合的联合类型将函数的参数限定为更具体的类型，比如声明如下所示的一个类型 Config：

```
interface Config {
    size: 'small' | 'big';
    isEnable:  true | false;
    margin: 0 | 2 | 4;
}
复制代码
```

在上述代码中，我们限定了 size 属性为字符串字面量类型 'small' | 'big'，isEnable 属性为布尔字面量类型 true | false（布尔字面量只包含 true 和 false，true | false 的组合跟直接使用 boolean 没有区别），margin 属性为数字字面量类型 0 | 2 | 4。

let和const分析

我们先来看一个 const 示例，如下代码所示：

```
{
  const str = 'this is string'; // str: 'this is string'
  const num = 1; // num: 1
  const bool = true; // bool: true
}
复制代码
```

在上述代码中，我们将 const 定义为一个不可变更的常量，在缺省类型注解的情况下，TypeScript 推断出它的类型直接由赋值字面量的类型决定，这也是一种比较合理的设计。

接下来我们看看如下所示的 let 示例：

```
{

  let str = 'this is string'; // str: string
  let num = 1; // num: number
  let bool = true; // bool: boolean
}
复制代码
```

在上述代码中，缺省显式类型注解的可变更的变量的类型转换为了赋值字面量类型的父类型，比如 str 的类型是 'this is string' 类型（这里表示一个字符串字面量类型）的父类型 string，num 的类型是 1 类型的父类型 number。

这种设计符合编程预期，意味着我们可以分别赋予 str 和 num 任意值（只要类型是 string 和 number 的子集的变量）：

```
  str = 'any string';
  num = 2;
  bool = false;
复制代码
```

我们将 TypeScript 的字面量子类型转换为父类型的这种设计称之为 "literal widening"，也就是字面量类型的拓宽，比如上面示例中提到的字符串字面量类型转换成 string 类型，下面我们着重介绍一下。

类型拓宽(Type Widening)

所有通过 let 或 var 定义的变量、函数的形参、对象的非只读属性，如果满足指定了初始值且未显式添加类型注解的条件，那么它们推断出来的类型就是指定的初始值字面量类型拓宽后的类型，这就是字面量类型拓宽。

下面我们通过字符串字面量的示例来理解一下字面量类型拓宽：

```
  let str = 'this is string'; // 类型是 string
  let strFun = (str = 'this is string') => str; // 类型是 (str?: string) =>
string;
  const specifiedStr = 'this is string'; // 类型是 'this is string'
  let str2 = specifiedStr; // 类型是 'string'
  let strFun2 = (str = specifiedStr) => str; // 类型是 (str?: string) => string;
```

复制代码

因为第 1~2 行满足了 let、形参且未显式声明类型注解的条件，所以变量、形参的类型拓宽为 string（形参类型确切地讲是 string | undefined）。

因为第 3 行的常量不可变更，类型没有拓宽，所以 specifiedStr 的类型是 'this is string' 字面量类型。

第 4~5 行，因为赋予的值 specifiedStr 的类型是字面量类型，且没有显式类型注解，所以变量、形参的类型也被拓宽了。其实，这样的设计符合实际编程诉求。我们设想一下，如果 str2 的类型被推断为 'this is string'，它将不可变更，因为赋予任何其他的字符串类型的值都会提示类型错误。

基于字面量类型拓宽的条件，我们可以通过如下所示代码添加显示类型注解控制类型拓宽行为。

```
{
  const specifiedStr: 'this is string' = 'this is string'; // 类型是 '"this is
string"'
  let str2 = specifiedStr; // 即便使用 let 定义，类型是 'this is string'
}
```

复制代码

实际上，除了字面量类型拓宽之外，TypeScript 对某些特定类型值也有类似 "Type Widening" （类型拓宽）的设计，下面我们具体来了解一下。

比如对 null 和 undefined 的类型进行拓宽，通过 let、var 定义的变量如果满足未显式声明类型注解且被赋予了 null 或 undefined 值，则推断出这些变量的类型是 any：

```
{
  let x = null; // 类型拓宽成 any
  let y = undefined; // 类型拓宽成 any

  /** -----分界线------- */
  const z = null; // 类型是 null

  /** -----分界线------- */
  let anyFun = (param = null) => param; // 形参类型是 null
  let z2 = z; // 类型是 null
  let x2 = x; // 类型是 null
  let y2 = y; // 类型是 undefined
}

```

复制代码

注意：在严格模式下，一些比较老的版本中（2.0）null 和 undefined 并不会被拓宽成“any”。

为了更方便的理解类型拓宽,下面我们举个例子,更加深入的分析一下

假设你正在编写一个向量库，你首先定义了一个 Vector3 接口，然后定义了 getComponent 函数用于获取指定坐标轴的值：

```
interface Vector3 {
  x: number;
  y: number;
  z: number;
}

function getComponent(vector: Vector3, axis: "x" | "y" | "z") {
  return vector[axis];
}
```
复制代码

但是，当你尝试使用 getComponent 函数时，TypeScript 会提示以下错误信息：

```
let x = "x";
let vec = { x: 10, y: 20, z: 30 };
// 类型“string”的参数不能赋给类型“"x" | "y" | "z"”的参数。
getComponent(vec, x); // Error
```
复制代码

为什么会出现上述错误呢？通过 TypeScript 的错误提示消息，我们知道是因为变量 x 的类型被推断为 string 类型，而 getComponent 函数期望它的第二个参数有一个更具体的类型。这在实际场合中被拓宽了，所以导致了一个错误。

这个过程是复杂的，因为对于任何给定的值都有许多可能的类型。例如：

```
const arr = ['x', 1];
```
复制代码

上述 arr 变量的类型应该是什么？这里有一些可能性：

- ('x' | 1)[]
- ['x', 1]
- [string, number]
- readonly [string, number]
- (string | number)[]
- readonly (string|number)[]
- [any, any]
- any[]

没有更多的上下文，TypeScript 无法知道哪种类型是 “正确的”，它必须猜测你的意图。尽管 TypeScript 很聪明，但它无法读懂你的心思。它不能保证 100% 正确，正如我们刚才看到的那样的疏忽性错误。

在下面的例子中，变量 x 的类型被推断为字符串，因为 TypeScript 允许这样的代码：

```
let x = 'semlinker';
x = 'kakuqo';
x = 'lolo';
```
复制代码

对于 JavaScript 来说，以下代码也是合法的：

```
let x = 'x';
x = /x|y|z/;
x = ['x', 'y', 'z'];
复制代码
```

在推断 x 的类型为字符串时，TypeScript 试图在特殊性和灵活性之间取得平衡。一般规则是，变量的类型在声明之后不应该改变，因此 string 比 string|RegExp 或 string|string[] 或任何字符串更有意义。

TypeScript 提供了一些控制拓宽过程的方法。其中一种方法是使用 `const`。如果用 const 而不是 let 声明一个变量，那么它的类型会更窄。事实上，使用 const 可以帮助我们修复前面例子中的错误：

```
const x = "x"; // type is "x"
let vec = { x: 10, y: 20, z: 30 };
getComponent(vec, x); // OK
复制代码
```

因为 x 不能重新赋值，所以 TypeScript 可以推断更窄的类型，就不会在后续赋值中出现错误。因为字符串字面量型 “x” 可以赋值给 "x"|"y"|"z"，所以代码会通过类型检查器的检查。

然而，`const 并不是万灵药。对于对象和数组，仍然会存在问题`。

以下这段代码在 JavaScript 中是没有问题的：

```
const obj = {
  x: 1,
};

obj.x = 6;
obj.x = '6';

obj.y = 8;
obj.name = 'semlinker';
复制代码
```

而在 TypeScript 中，对于 obj 的类型来说，它可以是 `{readonly x: 1}` 类型，或者是更通用的 `{x: number}` 类型。当然也可能是 `{[key: string]: number}` 或 object 类型。对于对象，TypeScript 的拓宽算法会将其内部属性视为将其赋值给 let 关键字声明的变量，进而来推断其属性的类型。因此 obj 的类型为 `{x: number}`。这使得你可以将 obj.x 赋值给其他 number 类型的变量，而不是 string 类型的变量，并且它还会阻止你添加其他属性。

因此最后三行的语句会出现错误：

```
const obj = {
  x: 1,
};

obj.x = 6; // OK


// Type '"6"' is not assignable to type 'number'.
obj.x = '6'; // Error

// Property 'y' does not exist on type '{ x: number; }'.
```

```
obj.y = 8; // Error

// Property 'name' does not exist on type '{ x: number; }'.
obj.name = 'semlinker'; // Error
```
复制代码

TypeScript 试图在具体性和灵活性之间取得平衡。它需要推断一个足够具体的类型来捕获错误，但又不能推断出错误的类型。它通过属性的初始化值来推断属性的类型，当然有几种方法可以覆盖 TypeScript 的默认行为。一种是提供显式类型注释：

```
// Type is { x: 1 | 3 | 5; }
const obj: { x: 1 | 3 | 5 } = {
  x: 1
};
```
复制代码

另一种方法是使用 const 断言。不要将其与 let 和 const 混淆，后者在值空间中引入符号。这是一个纯粹的类型级构造。让我们来看看以下变量的不同推断类型：

```
// Type is { x: number; y: number; }
const obj1 = {
  x: 1,
  y: 2
};

// Type is { x: 1; y: number; }
const obj2 = {
  x: 1 as const,
  y: 2,
};

// Type is { readonly x: 1; readonly y: 2; }
const obj3 = {
  x: 1,
  y: 2
} as const;
```
复制代码

当你在一个值之后使用 const 断言时，TypeScript 将为它推断出最窄的类型，没有拓宽。对于真正的常量，这通常是你想要的。当然你也可以对数组使用 const 断言：

```
// Type is number[]
const arr1 = [1, 2, 3];

// Type is readonly [1, 2, 3]
const arr2 = [1, 2, 3] as const;
```
复制代码

既然有类型拓宽，自然也会有类型缩小，下面我们简单介绍一下 Type Narrowing。

类型缩小(Type Narrowing)

在 TypeScript 中，我们可以通过某些操作将变量的类型由一个较为宽泛的集合缩小到相对较小、较明确的集合，这就是 "Type Narrowing"。

比如，我们可以使用类型守卫（后面会讲到）将函数参数的类型从 any 缩小到明确的类型，具体示例如下：

```
{
  let func = (anything: any) => {
    if (typeof anything === 'string') {
      return anything; // 类型是 string
    } else if (typeof anything === 'number') {
      return anything; // 类型是 number
    }
    return null;
  };
}
复制代码
```

在 VS Code 中 hover 到第 4 行的 anything 变量提示类型是 string，到第 6 行则提示类型是 number。

同样，我们可以使用类型守卫将联合类型缩小到明确的子类型，具体示例如下：

```
{
  let func = (anything: string | number) => {
    if (typeof anything === 'string') {
      return anything; // 类型是 string
    } else {
      return anything; // 类型是 number
    }
  };
}
复制代码
```

当然，我们也可以通过字面量类型等值判断（===）或其他控制流语句（包括但不限于 if、三目运算符、switch 分支）将联合类型收敛为更具体的类型，如下代码所示：

```
{
  type Goods = 'pen' | 'pencil' |'ruler';
  const getPenCost = (item: 'pen') => 2;
  const getPencilCost = (item: 'pencil') => 4;
  const getRulerCost = (item: 'ruler') => 6;
  const getCost = (item: Goods) =>  {
    if (item === 'pen') {
      return getPenCost(item); // item => 'pen'
    } else if (item === 'pencil') {
      return getPencilCost(item); // item => 'pencil'
    } else {
      return getRulerCost(item); // item => 'ruler'
    }
  }
}
复制代码
```

在上述 getCost 函数中，接受的参数类型是字面量类型的联合类型，函数内包含了 `if` 语句的 3 个流程分支，其中每个流程分支调用的函数的参数都是具体独立的字面量类型。

那为什么类型由多个字面量组成的变量 item 可以传值给仅接收单一特定字面量类型的函数 `getPenCost、getPencilCost、getRulerCost` 呢？这是因为在每个流程分支中，编译器知道流程分支中的 item 类型是什么。比如 item === 'pencil' 的分支，item 的类型就被收缩为“pencil”。

事实上，如果我们将上面的示例去掉中间的流程分支，编译器也可以推断出收敛后的类型，如下代码所示：

```
  const getCost = (item: Goods) =>  {
    if (item === 'pen') {
      item; // item => 'pen'
    } else {
      item; // => 'pencil' | 'ruler'
    }
  }
复制代码
```

一般来说 `TypeScript` 非常擅长通过条件来判别类型，但在处理一些特殊值时要特别注意 —— 它可能包含你不想要的东西！例如，以下从联合类型中排除 null 的方法是错误的：

```
const el = document.getElementById("foo"); // Type is HTMLElement | null
if (typeof el === "object") {
  el; // Type is HTMLElement | null
}
复制代码
```

因为在 JavaScript 中 `typeof null` 的结果是 "object"，所以你实际上并没有通过这种检查排除 `null` 值。除此之外，falsy 的原始值也会产生类似的问题：

```
function foo(x?: number | string | null) {
  if (!x) {
    x; // Type is string | number | null | undefined\
  }
}
复制代码
```

因为空字符串和 0 都属于 falsy 值，所以在分支中 x 的类型可能是 string 或 number 类型。帮助类型检查器缩小类型的另一种常见方法是在它们上放置一个明确的 “标签”：

```
interface UploadEvent {
  type: "upload";
  filename: string;
  contents: string;
}

interface DownloadEvent {
  type: "download";
  filename: string;
}

type AppEvent = UploadEvent | DownloadEvent;

function handleEvent(e: AppEvent) {
  switch (e.type) {
    case "download":
      e; // Type is DownloadEvent
```

```
      break;
    case "upload":
      e; // Type is UploadEvent
      break;
  }
}
复制代码
```

这种模式也被称为 "标签联合" 或 "可辨识联合"，它在 TypeScript 中的应用范围非常广。

联合类型

联合类型表示取值可以为多种类型中的一种，使用 `|` 分隔每个类型。

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven'; // OK
myFavoriteNumber = 7; // OK
复制代码
```

联合类型通常与 `null` 或 `undefined` 一起使用：

```
const sayHello = (name: string | undefined) => {
  /* ... */
};
复制代码
```

例如，这里 `name` 的类型是 `string | undefined` 意味着可以将 `string` 或 `undefined` 的值传递给 `sayHello` 函数。

```
sayHello("semlinker");
sayHello(undefined);
复制代码
```

通过这个示例，你可以凭直觉知道类型 A 和类型 B 联合后的类型是同时接受 A 和 B 值的类型。此外，对于联合类型来说，你可能会遇到以下的用法：

```
let num: 1 | 2 = 1;
type EventNames = 'click' | 'scroll' | 'mousemove';
复制代码
```

以上示例中的 `1`、`2` 或 `'click'` 被称为字面量类型，用来约束取值只能是某几个值中的一个。

类型别名

类型别名用来给一个类型起个新名字。类型别名常用于联合类型。

```
type Message = string | string[];
let greet = (message: Message) => {
  // ...
};
复制代码
```

注意：类型别名，诚如其名，即我们仅仅是给类型取了一个新的名字，并不是创建了一个新的类型。

交叉类型

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，使用`&`定义交叉类型。

```
{
  type Useless = string & number;
}
```
复制代码

很显然，如果我们仅仅把原始类型、字面量类型、函数类型等原子类型合并成交叉类型，是没有任何用处的，因为任何类型都不能满足同时属于多种原子类型，比如既是 string 类型又是 number 类型。因此，在上述的代码中，类型别名 Useless 的类型就是个 never。

交叉类型真正的用武之地就是将多个接口类型合并成一个类型，从而实现等同接口继承的效果，也就是所谓的合并接口类型，如下代码所示：

```
  type IntersectionType = { id: number; name: string; } & { age: number };
  const mixed: IntersectionType = {
    id: 1,
    name: 'name',
    age: 18
  }
```
复制代码

在上述示例中，我们通过交叉类型，使得 IntersectionType 同时拥有了 id、name、age 所有属性，这里我们可以试着将合并接口类型理解为求并集。

思考

这里，我们来发散思考一下：如果合并的多个接口类型存在同名属性会是什么效果呢?

如果同名属性的类型不兼容，比如上面示例中两个接口类型同名的 name 属性类型一个是 number，另一个是 string，合并后，name 属性的类型就是 number 和 string 两个原子类型的交叉类型，即 never，如下代码所示：

```
  type IntersectionTypeConfict = { id: number; name: string; }
  & { age: number; name: number; };
  const mixedConflict: IntersectionTypeConfict = {
    id: 1,
    name: 2, // ts(2322) 错误，'number' 类型不能赋给 'never' 类型
    age: 2
  };
```
复制代码

此时，我们赋予 mixedConflict 任意类型的 name 属性值都会提示类型错误。而如果我们不设置 name 属性，又会提示一个缺少必选的 name 属性的错误。在这种情况下，就意味着上述代码中交叉出来的 IntersectionTypeConfict 类型是一个无用类型。

如果同名属性的类型兼容，比如一个是 number，另一个是 number 的子类型、数字字面量类型，合并后 name 属性的类型就是两者中的子类型。

如下所示示例中 name 属性的类型就是数字字面量类型 2，因此，我们不能把任何非 2 之外的值赋予 name 属性。

```
  type IntersectionTypeConfict = { id: number; name: 2; }
  & { age: number; name: number; };

  let mixedConflict: IntersectionTypeConfict = {
    id: 1,
    name: 2, // ok
    age: 2
  };
  mixedConflict = {
    id: 1,
    name: 22, // '22' 类型不能赋给 '2' 类型
    age: 2
  };
```
复制代码

那么如果同名属性是非基本数据类型的话，又会是什么情形。我们来看个具体的例子：

```
interface A {
  x:{d:true},
}
interface B {
  x:{e:string},
}
interface C {
  x:{f:number},
}
type ABC = A & B & C
let abc:ABC = {
  x:{
    d:true,
    e:'',
    f:666
  }
}
```
复制代码

以上代码成功运行后，会输出以下结果：

由上图可知，在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，那么是可以成功合并。

接口（Interfaces）

在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

什么是接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类（classes）去实现（implement）。

TypeScript 中的接口是一个非常灵活的概念，除了可用于[对类的一部分行为进行抽象]以外，也常用于对「对象的形状（Shape）」进行描述。

简单的例子

```
interface Person {
    name: string;
    age: number;
}
let tom: Person = {
    name: 'Tom',
    age: 25
};
```
复制代码

上面的例子中，我们定义了一个接口 `Person`，接着定义了一个变量 `tom`，它的类型是 `Person`。这样，我们就约束了 `tom` 的形状必须和接口 `Person` 一致。

接口一般首字母大写。

定义的变量比接口少了一些属性是不允许的：

```
interface Person {
    name: string;
    age: number;
}
let tom: Person = {
    name: 'Tom'
};

// index.ts(6,5): error TS2322: Type '{ name: string; }' is not assignable to
type 'Person'.
//   Property 'age' is missing in type '{ name: string; }'.
```
复制代码

多一些属性也是不允许的：

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// index.ts(9,5): error TS2322: Type '{ name: string; age: number; gender:
string; }' is not assignable to type 'Person'.
//   Object literal may only specify known properties, and 'gender' does not
exist in type 'Person'.
```
复制代码

可见，**赋值的时候，变量的形状必须和接口的形状保持一致。**

可选 | 只读属性

```
interface Person {
  readonly name: string;
  age?: number;
}
复制代码
```

只读属性用于限制只能在对象刚刚创建的时候修改其值。此外 TypeScript 还提供了 `ReadonlyArray<T>` 类型，它与 `Array<T>` 相似，只是把所有可变方法去掉了，因此可以确保数组创建后再也不能被修改。

```
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!
复制代码
```

任意属性

有时候我们希望一个接口中除了包含必选和可选属性之外，还允许有其他的任意属性，这时我们可以使用 **索引签名** 的形式来满足上述要求。

```
interface Person {
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    name: 'Tom',
    gender: 'male'
};
复制代码
```

需要注意的是，**一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集**

```
interface Person {
    name: string;
    age?: number;
    [propName: string]: string;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// index.ts(3,5): error TS2411: Property 'age' of type 'number' is not assignable
to string index type 'string'.
```

```
// index.ts(7,5): error TS2322: Type '{ [x: string]: string | number; name:
string; age: number; gender: string; }' is not assignable to type 'Person'.
//   Index signatures are incompatible.
//     Type 'string | number' is not assignable to type 'string'.
//       Type 'number' is not assignable to type 'string'.
```
复制代码

上例中，任意属性的值允许是 `string`，但是可选属性 `age` 的值却是 `number`，`number` 不是 `string` 的子属性，所以报错了。

另外，在报错信息中可以看出，此时 `{ name: 'Tom', age: 25, gender: 'male' }` 的类型被推断成了 `{ [x: string]: string | number; name: string; age: number; gender: string; }`，这是联合类型和接口的结合。

一个接口中只能定义一个任意属性。如果接口中有多个类型的属性，则可以在任意属性中使用联合类型：

```
interface Person {
    name: string;
    age?: number; // 这里真实的类型应该为：number | undefined
    [propName: string]: string | number | undefined;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};
```
复制代码

鸭式辨型法

所谓的**鸭式辨型法**就是 `像鸭子一样走路并且嘎嘎叫的就叫鸭子`，即具有鸭子特征的认为它就是鸭子，也就是通过制定规则来判定对象是否实现这个接口。

例子

```
interface LabeledValue {
  label: string;
}
function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}
let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj); // OK
```
复制代码
```
interface LabeledValue {
  label: string;
}
function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}
printLabel({ size: 10, label: "Size 10 Object" }); // Error
```
复制代码

上面代码，在参数里写对象就相当于是直接给`labeledObj`赋值，这个对象有严格的类型定义，所以不能多参或少参。而当你在外面将该对象用另一个变量`myObj`接收，`myObj`不会经过额外属性检查，但会根据类型推论为`let myObj: { size: number; label: string } = { size: 10, label: "Size 10 Object" };`，然后将这个`myObj`再赋值给`labeledObj`，此时根据类型的兼容性，两种类型对象，参照**鸭式辨型法**，因为都具有`label`属性，所以被认定为两个相同，故而可以用此法来绕开多余的类型检查。

绕开额外属性检查的方式

鸭式辨型法

如上例子所示

类型断言

类型断言的意义就等同于你在告诉程序，你很清楚自己在做什么，此时程序自然就不会再进行额外的属性检查了。

```
interface Props {
  name: string;
  age: number;
  money?: number;
}

let p: Props = {
  name: "兔神",
  age: 25,
  money: -100000,
  girl: false
} as Props; // OK
复制代码
```

索引签名

```
interface Props {
  name: string;
  age: number;
  money?: number;
  [key: string]: any;
}

let p: Props = {
  name: "兔神",
  age: 25,
  money: -100000,
  girl: false
}; // OK
复制代码
```

接口与类型别名的区别

实际上，在大多数的情况下使用接口类型和类型别名的效果等价，但是在某些特定的场景下这两者还是存在很大区别。

> TypeScript 的核心原则之一是对值所具有的结构进行类型检查。 而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数据模型。

> type(类型别名)会给一个类型起个新名字。 type 有时和 interface 很像，但是可以作用于原始值（基本类型），联合类型，元组以及其它任何你需要手写的类型。起别名不会新建一个类型 - 它创建了一个新 名字来引用那个类型。给基本类型起别名通常没什么用，尽管可以做为文档的一种形式使用。

Objects / Functions

两者都可以用来描述对象或函数的类型，但是语法不同。

Interface

```
interface Point {
  x: number;
  y: number;
}

interface SetPoint {
  (x: number, y: number): void;
}

复制代码
```

Type alias

```
type Point = {
  x: number;
  y: number;
};

type SetPoint = (x: number, y: number) => void;

复制代码
```

Other Types

与接口不同，类型别名还可以用于其他类型，如基本类型（原始值）、联合类型、元组。

```
// primitive
type Name = string;

// object
type PartialPointX = { x: number; };
type PartialPointY = { y: number; };

// union
type PartialPoint = PartialPointX | PartialPointY;

// tuple
type Data = [number, string];

// dom
let div = document.createElement('div');
type B = typeof div;
```

复制代码

接口可以定义多次,类型别名不可以

与类型别名不同，接口可以定义多次，会被自动合并为单个接口。

```
interface Point { x: number; }
interface Point { y: number; }
const point: Point = { x: 1, y: 2 };
```
复制代码

扩展

两者的扩展方式不同，但并不互斥。接口可以扩展类型别名，同理，类型别名也可以扩展接口。

接口的扩展就是继承，通过 `extends` 来实现。类型别名的扩展就是交叉类型，通过 `&` 来实现。

接口扩展接口

```
interface PointX {
    x: number
}

interface Point extends PointX {
    y: number
}
```
复制代码

类型别名扩展类型别名

```
type PointX = {
    x: number
}

type Point = PointX & {
    y: number
}
```
复制代码

接口扩展类型别名

```
type PointX = {
    x: number
}
interface Point extends PointX {
    y: number
}
```
复制代码

类型别名扩展接口

```
interface PointX {
    x: number
}
type Point = PointX & {
    y: number
}
复制代码
```

泛型

泛型介绍

假如让你实现一个函数 `identity`，函数的参数可以是任何值，返回值就是将参数原样返回，并且其只能接受一个参数，你会怎么做?

你会觉得这很简单，顺手就写出这样的代码:

```
const identity = (arg) => arg;
复制代码
```

由于其可以接受任意值，也就是说你的函数的入参和返回值都应该可以是任意类型。 现在让我们给代码增加类型声明:

```
type idBoolean = (arg: boolean) => boolean;
type idNumber = (arg: number) => number;
type idString = (arg: string) => string;
...
复制代码
```

一个笨的方法就像上面那样，也就是说 JS 提供多少种类型，就需要复制多少份代码，然后改下类型签名。这对程序员来说是致命的。这种复制粘贴增加了出错的概率，使得代码难以维护，牵一发而动全身。并且将来 JS 新增新的类型，你仍然需要修改代码，也就是说你的代码**对修改开放**，这样不好。还有一种方式是使用 any 这种"万能语法"。缺点是什么呢? 我举个例子:

```
identity("string").length; // ok
identity("string").toFixed(2); // ok
identity(null).toString(); // ok
...
复制代码
```

如果你使用 any 的话，怎么写都是 ok 的， 这就丧失了类型检查的效果。实际上我知道我传给你的是 string，返回来的也一定是 string，而 string 上没有 toFixed 方法，因此需要报错才是我想要的。也就是说我真正想要的效果是: `当我用到id的时候，你根据我传给你的类型进行推导`。比如我传入的是 string，但是使用了 number 上的方法，你就应该报错。

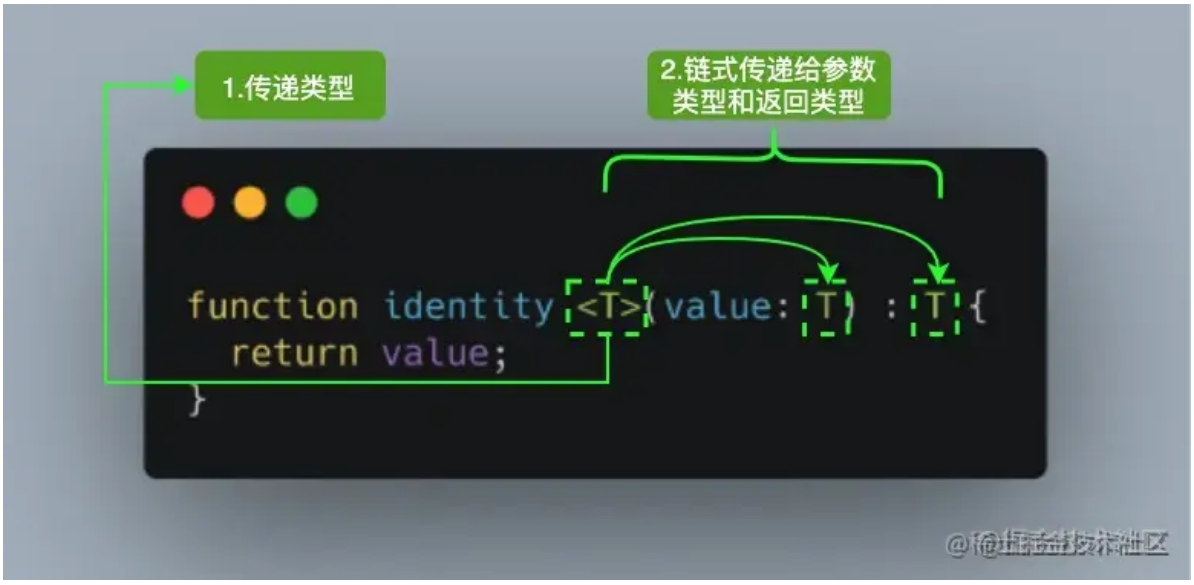
为了解决上面的这些问题，我们**使用泛型对上面的代码进行重构**。和我们的定义不同，这里用了一个 类型 T，这个 **T 是一个抽象类型，只有在调用的时候才确定它的值**，这就不用我们复制粘贴无数份代码了。

```
function identity<T>(arg: T): T {
  return arg;
}
```
复制代码

其中 `T` 代表 **Type**，在定义泛型时通常用作第一个类型变量名称。但实际上 `T` 可以用任何有效名称代替。除了 `T` 之外，以下是常见泛型变量代表的意思：

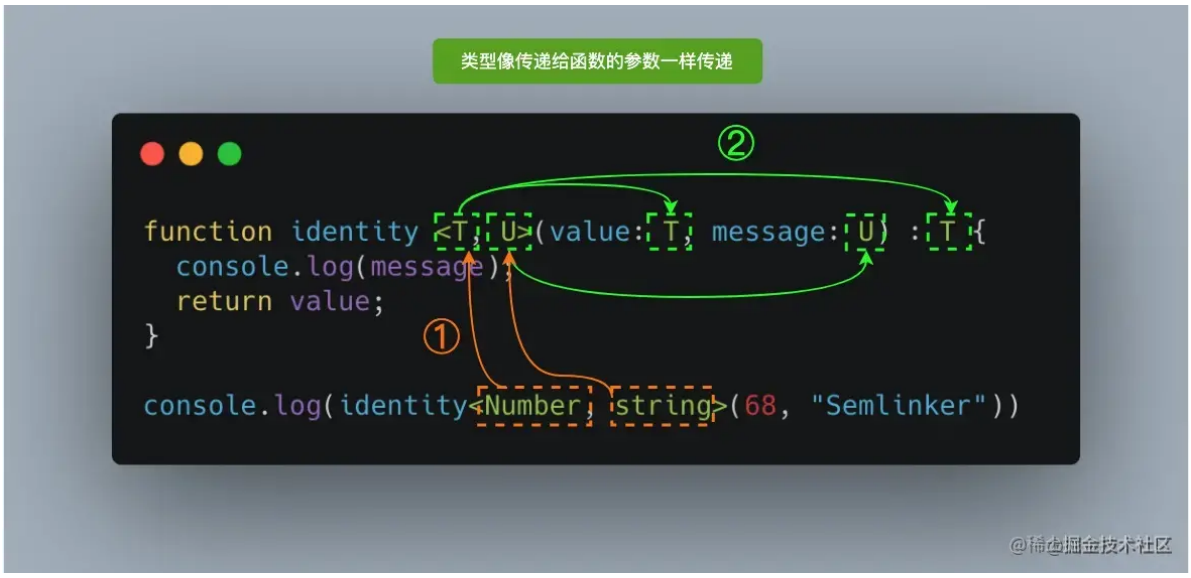
- K（Key）：表示对象中的键类型；
- V（Value）：表示对象中的值类型；
- E（Element）：表示元素类型。

来张图片帮助你理解



其实并不是只能定义一个类型变量，我们可以引入希望定义的任何数量的类型变量。比如我们引入一个新的类型变量 `U`，用于扩展我们定义的 `identity` 函数：

```
function identity <T, U>(value: T, message: U) : T {
  console.log(message);
  return value;
}
console.log(identity<Number, string>(68, "Semlinker"));
```
复制代码



除了为类型变量显式设定值之外，一种更常见的做法是使编译器自动选择这些类型，从而使代码更简洁。我们可以完全省略尖括号，比如：

```
function identity <T, U>(value: T, message: U) : T {
  console.log(message);
  return value;
}
console.log(identity(68, "Semlinker"));
```
复制代码

对于上述代码，编译器足够聪明，能够知道我们的参数类型，并将它们赋值给 T 和 U，而不需要开发人员显式指定它们。

泛型约束

假如我想打印出参数的 size 属性呢？如果完全不进行约束 TS 是会报错的：

```
function trace<T>(arg: T): T {
  console.log(arg.size); // Error: Property 'size doesn't exist on type 'T'
  return arg;
}
```
复制代码

报错的原因在于 T 理论上是可以是任何类型的，不同于 any，你不管使用它的什么属性或者方法都会报错（除非这个属性和方法是所有集合共有的）。那么直观的想法是限定传给 trace 函数的**参数类型**应该有 size 类型，这样就不会报错了。如何去表达这个**类型约束**的点呢？实现这个需求的关键在于使用类型约束。 使用 extends 关键字可以做到这一点。简单来说就是你定义一个类型，然后让 T 实现这个接口即可。

```
interface Sizeable {
  size: number;
}
function trace<T extends Sizeable>(arg: T): T {
  console.log(arg.size);
  return arg;
}
```
复制代码

有的人可能说我直接将 Trace 的参数限定为 Sizeable 类型可以么？如果你这么做，会有类型丢失的风险，详情可以参考这篇文章A use case for TypeScript Generics。

泛型工具类型

为了方便开发者 TypeScript 内置了一些常用的工具类型，比如 Partial、Required、Readonly、Record 和 ReturnType 等。不过在具体介绍之前，我们得先介绍一些相关的基础知识，方便读者可以更好的学习其它的工具类型。

1.typeof

typeof 的主要用途是在类型上下文中获取变量或者属性的类型，下面我们通过一个具体示例来理解一下。

```
interface Person {
  name: string;
  age: number;
}
const sem: Person = { name: "semlinker", age: 30 };
type Sem = typeof sem; // type Sem = Person
```
复制代码

在上面代码中，我们通过 `typeof` 操作符获取 sem 变量的类型并赋值给 Sem 类型变量，之后我们就可以使用 Sem 类型：

```
const lolo: Sem = { name: "lolo", age: 5 }
```
复制代码

你也可以对嵌套对象执行相同的操作：

```
const Message = {
    name: "jimmy",
    age: 18,
    address: {
      province: '四川',
      city: '成都'
    }
}
type message = typeof Message;
/*
 type message = {
    name: string;
    age: number;
    address: {
        province: string;
        city: string;
    };
}
*/
```
复制代码

此外，`typeof` 操作符除了可以获取对象的结构类型之外，它也可以用来获取函数对象的类型，比如：

```
function toArray(x: number): Array<number> {
  return [x];
}
type Func = typeof toArray; // -> (x: number) => number[]
```
复制代码

2.keyof

`keyof` 操作符是在 TypeScript 2.1 版本引入的，该操作符可以用于获取某种类型的所有键，其返回类型是联合类型。

```
interface Person {
  name: string;
  age: number;
}

type K1 = keyof Person; // "name" | "age"
type K2 = keyof Person[]; // "length" | "toString" | "pop" | "push" | "concat" | "join"
type K3 = keyof { [x: string]: Person };  // string | number
```
复制代码

在 TypeScript 中支持两种索引签名，数字索引和字符串索引：

```
interface StringArray {
  // 字符串索引 -> keyof StringArray => string | number
  [index: string]: string;
}

interface StringArray1 {
  // 数字索引 -> keyof StringArray1 => number
  [index: number]: string;
}
```
复制代码

为了同时支持两种索引类型，就得要求数字索引的返回值必须是字符串索引返回值的子类。**其中的原因就是当使用数值索引时，JavaScript 在执行索引操作时，会先把数值索引先转换为字符串索引。**所以 `keyof { [x: string]: Person }` 的结果会返回 `string | number`。

keyof也支持基本数据类型：

```
let K1: keyof boolean; // let K1: "valueOf"
let K2: keyof number; // let K2: "toString" | "toFixed" | "toExponential" | ...
let K3: keyof symbol; // let K1: "valueOf"
```
复制代码

keyof 的作用

JavaScript 是一种高度动态的语言。有时在静态类型系统中捕获某些操作的语义可能会很棘手。以一个简单的 `prop` 函数为例：

```
function prop(obj, key) {
  return obj[key];
}
```
复制代码

该函数接收 obj 和 key 两个参数，并返回对应属性的值。对象上的不同属性，可以具有完全不同的类型，我们甚至不知道 obj 对象长什么样。

那么在 TypeScript 中如何定义上面的 `prop` 函数呢？我们来尝试一下：

```
function prop(obj: object, key: string) {
  return obj[key];
}
```
复制代码

在上面代码中，为了避免调用 prop 函数时传入错误的参数类型，我们为 obj 和 key 参数设置了类型，分别为 `{}` 和 `string` 类型。然而，事情并没有那么简单。针对上述的代码，TypeScript 编译器会输出以下错误信息：

```
Element implicitly has an 'any' type because expression of type 'string' can't be used to index type '{}'.
复制代码
```

元素隐式地拥有 `any` 类型，因为 `string` 类型不能被用于索引 `{}` 类型。要解决这个问题，你可以使用以下非常暴力的方案：

```
function prop(obj: object, key: string) {
  return (obj as any)[key];
}
复制代码
```

很明显该方案并不是一个好的方案，我们来回顾一下 `prop` 函数的作用，该函数用于获取某个对象中指定属性的属性值。因此我们期望用户输入的属性是对象上已存在的属性，那么如何限制属性名的范围呢？这时我们可以利用本文的主角 `keyof` 操作符：

```
function prop<T extends object, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}
复制代码
```

在以上代码中，我们使用了 TypeScript 的泛型和泛型约束。**首先定义了 T 类型并使用 `extends` 关键字约束该类型必须是 object 类型的子类型，然后使用 `keyof` 操作符获取 T 类型的所有键，其返回类型是联合类型，最后利用 `extends` 关键字约束 K 类型必须为 `keyof T` 联合类型的子类型。** 是骡子是马拉出来遛遛就知道了，我们来实际测试一下：

```
type Todo = {
  id: number;
  text: string;
  done: boolean;
}

const todo: Todo = {
  id: 1,
  text: "Learn TypeScript keyof",
  done: false
}

function prop<T extends object, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}

const id = prop(todo, "id"); // const id: number
const text = prop(todo, "text"); // const text: string
const done = prop(todo, "done"); // const done: boolean
复制代码
```

很明显使用泛型，重新定义后的 `prop<T extends object, K extends keyof T>(obj: T, key: K)` 函数，已经可以正确地推导出指定键对应的类型。那么当访问 todo 对象上不存在的属性时，会出现什么情况？比如：

```
const date = prop(todo, "date");
复制代码
```

对于上述代码，TypeScript 编译器会提示以下错误：

```
Argument of type '"date"' is not assignable to parameter of type '"id" | "text"
| "done"'.
复制代码
```

这就阻止我们尝试读取不存在的属性。

3.in

`in` 用来遍历枚举类型：

```
type Keys = "a" | "b" | "c"

type Obj =  {
  [p in Keys]: any
} // -> { a: any, b: any, c: any }
复制代码
```

4.infer

在条件类型语句中，可以用 `infer` 声明一个类型变量并且对它进行使用。

```
type ReturnType<T> = T extends (
  ...args: any[]
) => infer R ? R : any;
复制代码
```

以上代码中 `infer R` 就是声明一个变量来承载传入函数签名的返回值类型，简单说就是用它取到函数返回值的类型方便之后使用。

5.extends

有时候我们定义的泛型不想过于灵活或者说想继承某些类等，可以通过 extends 关键字添加泛型约束。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
复制代码
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3);  // Error, number doesn't have a .length property
```
复制代码

这时我们需要传入符合约束类型的值，必须包含length属性：

```
loggingIdentity({length: 10, value: 3});
```
复制代码

索引类型

在实际开发中，我们经常能遇到这样的场景，在对象中获取一些属性的值，然后建立对应的集合。

```
let person = {
    name: 'musion',
    age: 35
}

function getValues(person: any, keys: string[]) {
    return keys.map(key => person[key])
}

console.log(getValues(person, ['name', 'age'])) // ['musion', 35]
console.log(getValues(person, ['gender'])) // [undefined]
```
复制代码

在上述例子中，可以看到getValues(persion, ['gender'])打印出来的是[undefined]，但是ts编译器并没有给出报错信息，那么如何使用ts对这种模式进行类型约束呢？这里就要用到了索引类型,改造一下getValues函数，通过 **索引类型查询**和 **索引访问** 操作符：

```
function getValues<T, K extends keyof T>(person: T, keys: K[]): T[K][] {
  return keys.map(key => person[key]);
}

interface Person {
    name: string;
    age: number;
}

const person: Person = {
    name: 'musion',
    age: 35
}

getValues(person, ['name']) // ['musion']
getValues(person, ['gender']) // 报错：
// Argument of Type '"gender"[]' is not assignable to parameter of type '("name"
| "age")[]'.
// Type "gender" is not assignable to type "name" | "age".

```
复制代码

编译器会检查传入的值是否是Person的一部分。通过下面的概念来理解上面的代码：

```
T[K]表示对象T的属性K所表示的类型，在上述例子中，T[K][] 表示变量T取属性K的值的数组

// 通过[]索引类型访问操作符，我们就能得到某个索引的类型
class Person {
    name:string;
    age:number;
 }
 type MyType = Person['name'];  //Person中name的类型为string type MyType = string
```

复制代码

介绍完概念之后，应该就可以理解上面的代码了。首先看泛型，这里有T和K两种类型，根据类型推断，第一个参数person就是person，类型会被推断为Person。而第二个数组参数的类型推断（K extends keyof T），keyof关键字可以获取T，也就是Person的所有属性名，即['name', 'age']。而extends关键字让泛型K继承了Person的所有属性名，即['name', 'age']。这三个特性组合保证了代码的动态性和准确性，也让代码提示变得更加丰富了

```
getValues(person, ['gender']) // 报错:
// Argument of Type '"gender"[]' is not assignable to parameter of type '("name"
| "age")[]'.
// Type "gender" is not assignable to type "name" | "age".
```

复制代码

映射类型

> 根据旧的类型创建出新的类型, 我们称之为映射类型

比如我们定义一个接口

```
interface TestInterface{
    name:string,
    age:number
}
```

复制代码

我们把上面定义的接口里面的属性全部变成可选

```
// 我们可以通过+/-来指定添加还是删除

type OptionalTestInterface<T> = {
  [p in keyof T]+?:T[p]
}

type newTestInterface = OptionalTestInterface<TestInterface>
// type newTestInterface = {
//    name?:string,
//    age?:number
// }
```

复制代码

比如我们再加上只读

```
type OptionalTestInterface<T> = {
 +readonly [p in keyof T]+?:T[p]
}

type newTestInterface = OptionalTestInterface<TestInterface>
// type newTestInterface = {
//   readonly name?:string,
//   readonly age?:number
// }
```
复制代码

由于生成只读属性和可选属性比较常用, 所以TS内部已经给我们提供了现成的实现 Readonly / Partial,会面内置的工具类型会介绍.

内置的工具类型

Partial

> `Partial<T>` 将类型的属性变成可选

定义

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```
复制代码

在以上代码中，首先通过 `keyof T` 拿到 `T` 的所有属性名，然后使用 `in` 进行遍历，将值赋给 `P`，最后通过 `T[P]` 取得相应的属性值的类。中间的 `?` 号，用于将所有属性变为可选。

举例说明

```
interface UserInfo {
    id: string;
    name: string;
}
// error: Property 'id' is missing in type '{ name: string; }' but required in
type 'UserInfo'
const xiaoming: UserInfo = {
    name: 'xiaoming'
}
```
复制代码

使用 `Partial<T>`

```
type NewUserInfo = Partial<UserInfo>;
const xiaoming: NewUserInfo = {
    name: 'xiaoming'
}
```
复制代码

这个 NewUserInfo 就相当于

```
interface NewUserInfo {
    id?: string;
    name?: string;
}
复制代码
```

但是 `Partial<T>` 有个局限性，就是只支持处理第一层的属性，如果我的接口定义是这样的

```
interface UserInfo {
    id: string;
    name: string;
    fruits: {
        appleNumber: number;
        orangeNumber: number;
    }
}

type NewUserInfo = Partial<UserInfo>;

// Property 'appleNumber' is missing in type '{ orangeNumber: number; }' but
required in type '{ appleNumber: number; orangeNumber: number; }'.
const xiaoming: NewUserInfo = {
    name: 'xiaoming',
    fruits: {
        orangeNumber: 1,
    }
}
复制代码
```

可以看到，第二层以后就不会处理了，如果要处理多层，就可以自己实现

DeepPartial

```
type DeepPartial<T> = {
     // 如果是 object，则递归类型
    [U in keyof T]?: T[U] extends object
      ? DeepPartial<T[U]>
      : T[U]
};

type PartialedWindow = DeepPartial<T>; // 现在T上所有属性都变成了可选啦
复制代码
```

Required

> Required将类型的属性变成必选

定义

```
type Required<T> = {
    [P in keyof T]-?: T[P]
};
复制代码
```

其中 `-?` 是代表移除 `?` 这个 modifier 的标识。再拓展一下，除了可以应用于 `?` 这个 modifiers，还有应用在 `readonly`，比如 `Readonly<T>` 这个类型

```
type Readonly<T> = {
    readonly [p in keyof T]: T[p];
}
```
复制代码

Readonly

> `Readonly<T>` 的作用是将某个类型所有属性变为只读属性，也就意味着这些属性不能被重新赋值。

定义

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

举例说明

```
interface Todo {
 title: string;
}

const todo: Readonly<Todo> = {
 title: "Delete inactive users"
};

todo.title = "Hello"; // Error: cannot reassign a readonly property
```
复制代码

Pick

> Pick 从某个类型中挑出一些属性出来

定义

```
type Pick<T, K extends keyof T> = {
    [P in K]: T[P];
};
```
复制代码

举例说明

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
复制代码
```

Record

> `Record<K extends keyof any, T>` 的作用是将 `K` 中所有的属性的值转化为 `T` 类型。

定义

```
type Record<K extends keyof any, T> = {
    [P in K]: T;
};
复制代码
```

举例说明

```
interface PageInfo {
  title: string;
}

type Page = "home" | "about" | "contact";

const x: Record<Page, PageInfo> = {
  about: { title: "about" },
  contact: { title: "contact" },
  home: { title: "home" },
};
复制代码
```

ReturnType

> 用来得到一个函数的返回值类型

定义

```
type ReturnType<T extends (...args: any[]) => any> = T extends (
  ...args: any[]
) => infer R
  ? R
  : any;
复制代码
```

`infer` 在这里用于提取函数类型的返回值类型。`ReturnType<T>` 只是将 infer R 从参数位置移动到返回值位置，因此此时 R 即是表示待推断的返回值类型。

举例说明

```
type Func = (value: number) => string;
const foo: ReturnType<Func> = "1";
复制代码
```

`ReturnType` 获取到 `Func` 的返回值类型为 `string`，所以，`foo` 也就只能被赋值为字符串了。

Exclude

> `Exclude<T, U>` 的作用是将某个类型中属于另一个的类型移除掉。

定义

```
type Exclude<T, U> = T extends U ? never : T;
复制代码
```

如果 `T` 能赋值给 `U` 类型的话，那么就会返回 `never` 类型，否则返回 `T` 类型。最终实现的效果就是将 `T` 中某些属于 `U` 的类型移除掉。

举例说明

```
type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number
复制代码
```

Extract

> `Extract<T, U>` 的作用是从 `T` 中提取出 `U`。

定义

```
type Extract<T, U> = T extends U ? T : never;
复制代码
```

举例说明

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"
type T1 = Extract<string | number | (() => void), Function>; // () =>void
复制代码
```

Omit

> `Omit<T, K extends keyof any>` 的作用是使用 `T` 类型中除了 `K` 类型的所有属性，来构造一个新的类型。

定义

```
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;
复制代码
```

举例说明

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
复制代码
```

NonNullable

> `NonNullable<T>` 的作用是用来过滤类型中的 `null` 及 `undefined` 类型。

定义

```
type NonNullable<T> = T extendsnull | undefined ? never : T;
复制代码
```

举例说明

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
复制代码
```

Parameters

> `Parameters<T>` 的作用是用于获得函数的参数类型组成的元组类型。

定义

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

举例说明

```
type A = Parameters<() =>void>; // []
type B = Parameters<typeofArray.isArray>; // [any]
type C = Parameters<typeofparseInt>; // [string, (number | undefined)?]
type D = Parameters<typeofMath.max>; // number[]
```
复制代码

tsconfig.json

tsconfig.json介绍

tsconfig.json 是 TypeScript 项目的配置文件。如果一个目录下存在一个 tsconfig.json 文件，那么往往意味着这个目录就是 TypeScript 项目的根目录。

tsconfig.json 包含 TypeScript 编译的相关配置，通过更改编译配置项，我们可以让 TypeScript 编译出 ES6、ES5、node 的代码。

tsconfig.json 重要字段

- files - 设置要编译的文件的名称；
- include - 设置需要进行编译的文件，支持路径模式匹配；
- exclude - 设置无需进行编译的文件，支持路径模式匹配；
- compilerOptions - 设置与编译流程相关的选项。

compilerOptions 选项

```
{
  "compilerOptions": {

    /* 基本选项 */
    "target": "es5",                       // 指定 ECMAScript 目标版本: 'ES3'
(default), 'ES5', 'ES6'/'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
    "module": "commonjs",                  // 指定使用模块: 'commonjs', 'amd',
'system', 'umd' or 'es2015'
    "lib": [],                             // 指定要包含在编译中的库文件
    "allowJs": true,                       // 允许编译 javascript 文件
    "checkJs": true,                       // 报告 javascript 文件中的错误
    "jsx": "preserve",                     // 指定 jsx 代码的生成: 'preserve',
'react-native', or 'react'
    "declaration": true,                   // 生成相应的 '.d.ts' 文件
    "sourceMap": true,                     // 生成相应的 '.map' 文件
    "outFile": "./",                       // 将输出文件合并为一个文件
    "outDir": "./",                        // 指定输出目录
    "rootDir": "./",                       // 用来控制输出目录结构 --outDir.
    "removeComments": true,                // 删除编译后的所有的注释
    "noEmit": true,                        // 不生成输出文件
    "importHelpers": true,                 // 从 tslib 导入辅助工具函数
    "isolatedModules": true,               // 将每个文件做为单独的模块 （与
'ts.transpileModule' 类似）.

    /* 严格的类型检查选项 */
    "strict": true,                        // 启用所有严格类型检查选项
    "noImplicitAny": true,                 // 在表达式和声明上有隐含的 any类型时报错
    "strictNullChecks": true,              // 启用严格的 null 检查
    "noImplicitThis": true,                // 当 this 表达式值为 any 类型的时候，生成
一个错误
```

```
    "alwaysStrict": true,                     // 以严格模式检查每个模块，并在每个文件里加入
'use strict'

    /* 额外的检查 */
    "noUnusedLocals": true,                   // 有未使用的变量时，抛出错误
    "noUnusedParameters": true,               // 有未使用的参数时，抛出错误
    "noImplicitReturns": true,                // 并不是所有函数里的代码都有返回值时，抛出错
误
    "noFallthroughCasesInSwitch": true,       // 报告 switch 语句的 fallthrough 错误。
（即，不允许 switch 的 case 语句贯穿）

    /* 模块解析选项 */
    "moduleResolution": "node",               // 选择模块解析策略： 'node' (Node.js) or
'classic' (TypeScript pre-1.6)
    "baseUrl": "./",                          // 用于解析非相对模块名称的基目录
    "paths": {},                              // 模块名到基于 baseUrl 的路径映射的列表
    "rootDirs": [],                           // 根文件夹列表，其组合内容表示项目运行时的结
构内容
    "typeRoots": [],                          // 包含类型声明的文件列表
    "types": [],                              // 需要包含的类型声明文件名列表
    "allowSyntheticDefaultImports": true,     // 允许从没有设置默认导出的模块中默认导入。

    /* Source Map Options */
    "sourceRoot": "./",                       // 指定调试器应该找到 TypeScript 文件而不
是源文件的位置
    "mapRoot": "./",                          // 指定调试器应该找到映射文件而不是生成文件的
位置
    "inlineSourceMap": true,                  // 生成单个 soucemaps 文件，而不是将
sourcemaps 生成不同的文件
    "inlineSources": true,                    // 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

    /* 其他选项 */
    "experimentalDecorators": true,           // 启用装饰器
    "emitDecoratorMetadata": true             // 为装饰器提供元数据的支持
  }
}
```

复制代码

编写高效 TS 代码的一些建议

尽量减少重复代码

对于刚接触 TypeScript 的小伙伴来说，在定义接口时，可能一不小心会出现以下类似的重复代码。比如：

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate {
  firstName: string;
  lastName: string;
  birth: Date;
}
复制代码
```

很明显，相对于 `Person` 接口来说，`PersonWithBirthDate` 接口只是多了一个 `birth` 属性，其他的属性跟 `Person` 接口是一样的。那么如何避免出现例子中的重复代码呢？要解决这个问题，可以利用 `extends` 关键字：

```
interface Person {
  firstName: string;
  lastName: string;
}

interface PersonWithBirthDate extends Person {
  birth: Date;
}
复制代码
```

当然除了使用 `extends` 关键字之外，也可以使用交叉运算符（&）：

```
type PersonWithBirthDate = Person & { birth: Date };
复制代码
```

另外，有时候你可能还会发现自己想要定义一个类型来匹配一个初始配置对象的「形状」，比如：

```
const INIT_OPTIONS = {
  width: 640,
  height: 480,
  color: "#00FF00",
  label: "VGA",
};

interface Options {
  width: number;
  height: number;
  color: string;
  label: string;
}
复制代码
```

其实，对于 Options 接口来说，你也可以使用 typeof 操作符来快速获取配置对象的「形状」：

```
type Options = typeof INIT_OPTIONS;
复制代码
```

在实际的开发过程中，重复的类型并不总是那么容易被发现。有时它们会被语法所掩盖。比如有多个函数拥有相同的类型签名：

```
function get(url: string, opts: Options): Promise<Response> { /* ... */ }
function post(url: string, opts: Options): Promise<Response> { /* ... */ }
复制代码
```

对于上面的 get 和 post 方法，为了避免重复的代码，你可以提取统一的类型签名：

```
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;
const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
复制代码
```

使用更精确的类型替代字符串类型

假设你正在构建一个音乐集，并希望为专辑定义一个类型。这时你可以使用 `interface` 关键字来定义一个 `Album` 类型：

```
interface Album {
  artist: string; // 艺术家
  title: string; // 专辑标题
  releaseDate: string; // 发行日期：YYYY-MM-DD
  recordingType: string; // 录制类型："live" 或 "studio"
}
复制代码
```

对于 `Album` 类型，你希望 `releaseDate` 属性值的格式为 `YYYY-MM-DD`，而 `recordingType` 属性值的范围为 `live` 或 `studio`。但因为接口中 `releaseDate` 和 `recordingType` 属性的类型都是字符串，所以在使用 `Album` 接口时，可能会出现以下问题：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  releaseDate: "November 31, 1991", // 与预期格式不匹配
  recordingType: "Studio", // 与预期格式不匹配
};
复制代码
```

虽然 `releaseDate` 和 `recordingType` 的值与预期的格式不匹配，但此时 TypeScript 编译器并不能发现该问题。为了解决这个问题，你应该为 `releaseDate` 和 `recordingType` 属性定义更精确的类型，比如这样：

```
interface Album {\
  artist: string; // 艺术家
  title: string; // 专辑标题
  releaseDate: Date; // 发行日期：YYYY-MM-DD
  recordingType: "studio" | "live"; // 录制类型："live" 或 "studio"
}
复制代码
```

重新定义 `Album` 接口之后，对于前面的赋值语句，TypeScript 编译器就会提示以下异常信息：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  // 不能将类型“string”分配给类型“Date”。ts(2322)
  releaseDate: "November 31, 1991", // Error
  // 不能将类型“"Studio"”分配给类型“"studio" | "live"”。ts(2322)\
  recordingType: "Studio", // Error
};
复制代码
```

为了解决上面的问题，你需要为 `releaseDate` 和 `recordingType` 属性设置正确的类型，比如这样：

```
const dangerous: Album = {
  artist: "Michael Jackson",
  title: "Dangerous",
  releaseDate: new Date("1991-11-31"),
  recordingType: "studio",
};
复制代码
```

定义的类型总是表示有效的状态

假设你正在构建一个允许用户指定页码，然后加载并显示该页面对应内容的 Web 应用程序。首先，你可能会先定义 `State` 对象：

```
interface State {
  pageContent: string;
  isLoading: boolean;
  errorMsg?: string;
}
复制代码
```

接着你会定义一个 `renderPage` 函数，用来渲染页面：

```
function renderPage(state: State) {
  if (state.errorMsg) {
    return `呜呜呜，加载页面出现异常了...${state.errorMsg}`;
  } else if (state.isLoading) {
    return `页面加载中~~~`;
  }
  return `<div>${state.pageContent}</div>`;
}

// 输出结果：页面加载中~~~
console.log(renderPage({isLoading: true, pageContent: ""}));
// 输出结果：<div>大家好</div>
console.log(renderPage({isLoading: false, pageContent: "大家好呀"}));
复制代码
```

创建好 `renderPage` 函数，你可以继续定义一个 `changePage` 函数，用于根据页码获取对应的页面数据：

```
async function changePage(state: State, newPage: string) {
  state.isLoading = true;
```

```
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const text = await response.text();
    state.isLoading = false;
    state.pageContent = text;
  } catch (e) {
    state.errorMsg = "" + e;
  }
}
```
复制代码

对于以上的 `changePage` 函数，它存在以下问题：

- 在 catch 语句中，未把 `state.isLoading` 的状态设置为 `false`；
- 未及时清理 `state.errorMsg` 的值，因此如果之前的请求失败，那么你将继续看到错误消息，而不是加载消息。

出现上述问题的原因是，前面定义的 `State` 类型允许同时设置 `isLoading` 和 `errorMsg` 的值，尽管这是一种无效的状态。针对这个问题，你可以考虑引入可辨识联合类型来定义不同的页面请求状态：

```
interface RequestPending {
  state: "pending";
}

interface RequestError {
  state: "error";
  errorMsg: string;
}

interface RequestSuccess {
  state: "ok";
  pageContent: string;
}

type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: { [page: string]: RequestState };
}
```
复制代码

在以上代码中，通过使用可辨识联合类型分别定义了 3 种不同的请求状态，这样就可以很容易的区分出不同的请求状态，从而让业务逻辑处理更加清晰。接下来，需要基于更新后的 `State` 类型，来分别更新一下前面创建的 `renderPage` 和 `changePage` 函数：

更新后的 renderPage 函数

```
function renderPage(state: State) {
  const { currentPage } = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case "pending":
      return `页面加载中~~~`;
    case "error":
      return `呜呜呜，加载第${currentPage}页出现异常了...${requestState.errorMsg}`;
    case "ok":
      `<div>第${currentPage}页的内容：${requestState.pageContent}</div>`;
  }
}
```
复制代码

更新后的 changePage 函数

```
async function changePage(state: State, newPage: string) {
  state.requests[newPage] = { state: "pending" };
  state.currentPage = newPage;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`无法正常加载页面 ${newPage}: ${response.statusText}`);
    }
    const pageContent = await response.text();
    state.requests[newPage] = { state: "ok", pageContent };
  } catch (e) {
    state.requests[newPage] = { state: "error", errorMsg: "" + e };
  }
}
```
复制代码

在 `changePage` 函数中，会根据不同的情形设置不同的请求状态，而不同的请求状态会包含不同的信息。这样 `renderPage` 函数就可以根据统一的 `state` 属性值来进行相应的处理。因此，通过使用可辨识联合类型，让请求的每种状态都是有效的状态，不会出现无效状态的问题。

更多

光说不练假把式,点击下面的链接去练练手吧

typescript练习题

如果文章有什么错误,欢迎在评论区指出,如果觉得对你有帮助的话,欢迎 点赞收藏 哦,你的点赞就是我继续写作更新的动力.

最后

如果你想学习es7- es12新出的特性，请参考 快2022年了，这些ES7-ES12的知识点你都掌握了嘛？