重学TypeScript

> 链接：https://juejin.cn/post/7003171767560716302#heading-70

为什么要有 TypeScript

`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。

`TypeScript` 诞生的根本原因是 `JavaScript` 是弱类型语言（可以隐性的进行语言类型转变），无法做到在编译阶段进行类型检查，提早发现错误。

`TypeScript` 的初衷就是为了做类型检查，提早发现错误，所以「类型」是其最核心的特性。当然它只是给出你代码可能不会按预期执行的警告，比如你未按照声明的类型传参，你的代码还是可以运行的。这一点与强类型语言还是有本质的区别，强类型语言会直接导致编译不通过，因为 `TypeScript` 只是转译。

> 跟 `JavaScript` 不同，`TypeScript` 文件后缀使用 `.ts` 扩展名。浏览器是不识别 `.ts` 文件，所以使用时必须提前把 `TS` 代码转换成 `JavaScript` 代码。这个转换过程被称为 `转译`，`编译` 和 `转译` 的微小差别在于：
>
> - 编译是把源码转变成另一种语言
> - 转译是把源码转变另一个相同抽象层级的语言

我是不喜欢 `TypeScript` 的，因为在我看来它导致了这么几个问题：

1. 学习成本增加；
2. 代码量增加；
3. 代码复杂度增加

当然 `TypeScript` 带来的收益是可观的，静态检查使得提前发现错误，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。

我又是喜欢 `TypeScript` 的，因为它是先进的 `JavaScript`：

> `TypeScript` 提供最新的和不断发展的 `JavaScript` 特性，包括那些来自 `2015` 年的 `ECMAScript` 和未来的提案中的特性，比如异步功能和 `Decorators`，以帮助建立健壮的组件。

老实说，两年前我就看过这玩意😂，之前总是零零散散的看，对于高级部分总是掌握不了，这次一定且必须得沉淀下来。

~ 唉，大势所趋，这玩意不会不行了，不然别人的代码都看不懂了。

~后面学完了高级部分，打心底里感觉 `TS` 既高级又复杂，默默流下了菜的泪水。

正文

基础类型

JS的八种内置类型

- 字符串（string）
- 数字（number）
- 布尔值（boolean）
- 未定义（undefined）
- 空值（null）
- 对象（object）
- 大整数（bigInt，ES6 新增）
- 符号（symbol，ES6 新增）

`TS` 一一对应的 `example`（冒号后面有无空格都可以）：

```
let name: string = "bob";
let age: number = 37;
let isDone: boolean = false;
let u: undefined = undefined;
let n: null = null;
let obj: object = {x: 1};
let bigLiteral: bigint = 100n;
let sym: symbol = Symbol("me");
复制代码
```

Array

对数组类型的定义有两种方式：

```
// 元素类型[]
let list: number[] = [1, 2, 3];
// Array<元素类型>
let list: Array<number> = [1, 2, 3];
复制代码
```

定义指定对象成员的数组：

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

let arr: MyObject[] = [{name: "兔兔", age: 18}] // OK
复制代码
```

Tuple

上面定义数组类型的方式，只能定义出内部全为某种类型的数组。对于内部不同类型的数组可以使用元组类型来定义：

```
let x: [string, number];

x = ['hello', 10]; // OK
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。例如，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[]`。

undefined和null

注意这俩比较特殊

> 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`赋值给任何类型的变量。

eg:

```
let str: string = 'hello';
str = null; // OK
str = undefined; // OK

let a: null = undefined; // OK
let b: undefined = null; // OK
```
复制代码

当然也可以通过指定`--strictNullChecks`标记，开启严格模式检查。这种情况下，`null`和`undefined`和其他类型是平等关系，只能赋值给`any`和它们各自的类型，有一个例外是`undefined`还可以赋值给`void`类型（想想你为一个函数声明返回类型为`viod`时，但函数在未显式`return`的情况下，默认返回的就是`undefined`，此时就是这个例外的表现）。

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`（只在`--strictNullChecks`未指定时）和`undefined`。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

any和unknown

`any`会跳过类型检查器对值的检查，任何值都可以赋值给`any`类型，它通常被称为`top type`，所以会有`any`大法好的说法。

```
let notSure: any = 4;
notSure = "maybe a string instead"; // OK
notSure = false; // OK
复制代码
```

`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：

> `unknown`是`top type`(任何类型都是它的`subtype`)，而`any`既是`top type`，又是`bottom type`(它是任何类型的`subtype`)，这导致`any`基本上就是放弃了任何类型检查。

因为`any`既是`top type`，又是`bottom type`，所以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。但`unknown`只是`top type`，任何类型的值都可以赋值给它，但它只能赋值给`unknown`和`any`，因为只有它俩是`top type`。

```
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

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：

```
let err: never;
let num: number = 4;

num = err; // OK
```
复制代码

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
```

重点：`never`与其他类型的联合后，是没有`never`的

```
// type Eg2 = string | number
type Eg2 = string | number | never
复制代码
```

类型断言

类型断言好比其它语言里的类型转换，类型转换通常发生在你比`TS`更了解某个值的详细信息的时候。

两种方式实现：

```
// 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
```

类型推论

如果没有明确的指定类型，那么`TypeScript`会依照类型推论的规则推断出一个类型。

如下：

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

为什么是`Error`，因为事实上，它等价于：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7; // Error
复制代码
```

`TypeScript`会在没有明确的指定类型的时候推测出一个类型，这就是类型推论。

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
```

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
```

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。

```
interface A {
  name: string,
  age: number
}
interface B {
  name: string,
  gender: string
}

let a: A & B = { // OK
    name: "兔兔",
    age: 18,
    gender: "男"
};
复制代码
```

`a` 既是 `A` 类型，同时也是 `B` 类型。

注意点：交叉类型取的多个类型的并集，但是如果 `key` 相同但是类型不同，则该 `key` 为 `never` 类型。

```
type A = string & number // A 为 never 类型

let a: A = (() => {throw new Error()})(); // OK
复制代码
```

接口

首先，通俗的理解下此处接口的概念：一般后台定义接口就是前端调用的接口，定义一些参数等，`TS`里的接口类似，也可以理解为定义一些参数，规定变量里面有什么参数，参数是什么类型，使用时就必须有这些对应类型的参数，少或者多参数、参数类型不对都会报错。更简单的，你可以理解为这就是在定义一个较为详细的对象类型。

第一个示例：

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

你懒的写`interface`，可以这么写。这种写法，较为宽松，只会检查那些必需的属性是否存在。

第二个示例：

```
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
```

这种写法也是宽松的，同上也是因为发生了赋值。

注意：**在 type、interface 中可以使用逗号、分号，class 中不能用逗号。不过三者都支持行结尾不要符号。**

为什么赋值就使得类型检测变得宽松了

细品这句话：

> TypeScript 的核心原则之一是对值所具有的结构进行类型检查。 它有时被称做**鸭式辨型法**或**结构性子类型化。**

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(当然在`TS`里面不这样说)。

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。

```
interface LabeledValue {
  label: string;
}
function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

printLabel({ size: 10, label: "Size 10 Object" }); // Error
复制代码
```

可选属性

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

可选属性就是在可选属性名字定义的后面加一个`?`符号，来证明该属性是可有可无的。

只读属性

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

let p: Point = { x: 10, y: 20 };
p.x = 5; // Error
复制代码
```

在属性名前用`readonly`关键字来指定只读属性，该对象属性只能在对象刚刚创建的时候修改其值，与`const`类似，但`const`只能防止修改基础类型，对于引用类型只是防止修改引用地址，内部属性是可以变的，防止修改引用类型的内部属性，应该使用`readonly`。

ReadonlyArray

对于数组，`TS`还有`ReadonlyArray<T>`类型，此类型将数组的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
```

最后一行，可以看到就算把整个 `ReadonlyArray` 赋值到一个普通数组也是不可以的，此时可以使用类型断言：

```
a = ro as number[];
复制代码
```

注意：`readonly` 声明的只读数组类型与 `ReadonlyArray` 声明的只读数组类型，二者等价。

证明：

```
let arr1: readonly number[] = [1, 2];
let arr2: ReadonlyArray<number> = [1, 2, 3];

arr1[0] = 0; // Error
arr2[0] = 0; // Error
arr1.push(3); //Errpr
arr2.push(4); //Error
arr1 = arr2; // OK
复制代码
```

绕开额外属性检查的方式

1. 类型兼容

   一开始的例子就已经阐述的很彻底了，利用赋值操作，不再赘述。

2. 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
   ```

3. 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
```

任意属性

`TypeScript` 支持两种索引签名：字符串和数字。

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，因为确定属性与可选属性也算任意属性中的一种：

```
interface Person {
    name: boolean; // Error
    age?: number; // Error
    sex: string; // OK
    girl: undefined; // OK
    [propName: string]: string;
}
复制代码
```

可以同时使用两种类型的索引，但是数字索引的返回值必须是字符串索引返回值类型的子类型。 这是因为当使用 `number` 来索引时，`JavaScript` 会将它转换成 `string` 然后再去索引对象。

```
class Animal {
  name: string;
}
class Dog extends Animal {
  breed: string;
}

interface NotOkay {
  [x: number]: Animal; // Error
  [x: string]: Dog;
}

interface Okay {
  [x: number]: Dog; // OK
  [x: string]: Animal;
}
复制代码
```

还有一点我们需要注意，当任意属性使用联合类型且属性中存在可选属性时，需要联合 `undefined` 类型，否则编译报错，原因显而易见，因为可选属性可有可无：

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

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

额外的，你也可以将任意属性设置为只读，防止给属性赋值。

接口继承接口

接口继承接口使用关键字`extends`，继承的本质是复制，抽出共同的代码，所以子接口拥有父接口的类型定义：

```
interface Shape {
  color: string;
}
interface Square extends Shape {
  sideLength: number;
}

let square: Square = { sideLength: 1 }; // Error
let square1: Square = { sideLength: 1, color: 'red' }; // OK
复制代码
```

`TS`中与众不同的一点：接口可以多继承。

```
interface Shape {
  color: string;
}
interface PenStroke {
  penWidth: number;
}
interface Square extends Shape, PenStroke {
  sideLength: number;
}

let square: Square = { sideLength: 1 } // Error
let square1: Square = { sideLength: 1, color: 'red' } // Error
let square2: Square = { sideLength: 1, color: 'red', penWidth: 2 } // OK
复制代码
```

这里需要注意的一点是，大部分语言是不支持多继承的，原因显而易见，多继承会引发混乱：

1. 若子类继承的父类中拥有相同的成员变量，子类在引用该变量时将无法判别使用哪个父类的成员变量；

2. 若一个子类继承的多个父类拥有相同方法，同时子类并未覆盖该方法（若覆盖，则直接使用子类中该方法），那么调用该方法时将无法确定调用哪个父类的方法。

`python`支持多继承，所谓的多继承，本质就是`mixin`，`JS`也可以利用`mixin`实现多继承。

在`TS`中，若多继承的两个或多个父接口有相同属性，但定义的类型不同，`TS`会直接报错，并未采取`mixin`策略。对于这点，我想反问一句：既然是抽出共同的代码，那你又何必把共性代码到处放呢？`TS`就是这样想的，这应该能解决掉你的困惑。所以在使用多继承时，先确保父接口没有共有属性，或共有属性定义的类型都相同。

```
interface Shape {
  name: string;
  color: string;
}
interface PenStroke {
  name: number;
  penWidth: number;
}
interface Square extends Shape, PenStroke { // Error
  sideLength: number;
}
复制代码
interface Shape {
  name: string;
  color: string;
}
interface PenStroke {
  name: string;
  penWidth: number;
}
interface Square extends Shape, PenStroke { // OK
  sideLength: number;
}
let square: Square = { // OK
    sideLength: 1,
    color: 'red',
    penWidth: 12,
    name: '兔神'
}
复制代码
```

接口中的`new`

在`TS`的官网示例中，有看到

```
interface ClockConstructor {
  new (hour: number, minute: number): any;
}
复制代码
```

这样的写法，让人很是疑惑。官网对于接口中使用`new`也没有详细的说明，只有例子，这一点很差劲。

我的理解是：`new`后面跟构造函数，是用来创建实例的。而接口是用来描述对象类型的，那么包含构造函数的对象类型是什么？答案是`类 class`。

```
// 例1
interface ClockConstructor {
  new (hour: number, minute: number): any;
}

let C:ClockConstructor = class {} // OK

// 例2
interface CPerson {
  new(name: string): Date;
}

let p: CPerson = class People extends Date {} // OK
```
复制代码

我们没有显示声明`constructor`，所以这里会有一个空的`constructor`函数顶上来。在这里我发现检测比较奇怪：

```
interface ClockConstructor {
  new (hour: number, minute: number): any;
}

let C:ClockConstructor = class { // OK
    constructor() {}
}

let C1:ClockConstructor = class { // OK
    constructor(h: number) {}
}

let C2:ClockConstructor = class { // OK
    constructor(h: number, m: number) {}
}

let C3:ClockConstructor = class { // Error
    constructor(h: string, m: number) {}
}

let C4:ClockConstructor = class { // Error
    constructor(h: number, m: number, b: number) {}
}
```
复制代码

这里的检测机制我们看到的是：参数少的，兼容参数多的，并不严格。我很疑惑，当学习到后面的时候，才发现原来这里是有个`双向协变`的概念：

> `Ts`在函数参数的比较中实际上默认采取的策略是双向协变：只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功。

函数类型

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

在 `TypeScript` 的类型定义中，`=>` 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。切忌与 `ES6` 的箭头函数混淆了。

用接口定义函数类型

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

let mySearch: SearchFunc = function(source: string, subString: string) { // OK
  let result = source.search(subString);
  return result >-1;
};
```
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。

函数中的 `this` 声明

`TypeScript` 会通过代码流分析来推断出 `this` 在函数中应该是什么，我们也可以明确指定函数中的 `this` 应是何种类型。示例如下：

```
interface Obj {
    fn: (this: Obj, name: string) => void;
}

let obj: Obj = {
    fn(name: string) {}
}

obj.fn("兔兔"); // OK
```
复制代码

因为 `JavaScript` 规范规定你不能有一个名为 `this` 的参数，所以 `TypeScript` 使用这个语法空间来让你在函数体中声明 `this` 的类型。

注意：这个 `this` 类型声明必须放在参数的首位：

```
interface Obj {
    // Error: A 'this' parameter must be the first parameter
    fn: (name: string, this: Obj) => void;
}
复制代码
```

再来一个更好的例子，感受一下：

```
interface Obj {
    fn: (this: Obj, name: string) => void;
}

let obj: Obj = {
    fn(name: string) {}
}

let rab: Obj ={
    fn(name: string) {}
}

obj.fn("兔兔"); // OK
obj.fn.call(rab, "兔兔"); // OK
obj.fn.call(window, "兔兔"); // Error: this 应该为 Obj 类型
复制代码
```

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
```

重载

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理。

`重载`的概念在学`JAVA`（`JAVA中的重载`）的时候接触到的，`JS`是没有这个概念的，`TS`的重载个人感觉更应该称之为`函数签名重载`。因为最后函数实现的内部还是依赖判断类型来处理，前面的函数定义只是为了精确表达输入类型对应的输出类型。

```
function reverse(x: number): number;
function reverse(x: string): string;
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('');
    }
}
复制代码
```

内置对象

`ECMAScript`标准提供的内置对象有：

`String`、`Number`、`Boolean`、`Error`、`Date`、`RegExp`等。

```
let s: String = new String('兔神');
let n: Number = new Number(123);
let b: Boolean = new Boolean(1);
let e: Error = new Error('Error occurred');
let d: Date = new Date();
let r: RegExp = /[a-z]/;
```
复制代码

`DOM` 和 `BOM` 的内置对象有: `Document`、`HTMLElement`、`Event`、`NodeList` 等。

```
let body: HTMLElement = document.body;
let allDiv: NodeList = document.querySelectorAll('div');
document.addEventListener('click', function(e: MouseEvent) {
  // Do something
});
```
复制代码

类数组对象 `IArguments`:

```
function sum() {
    let args: IArguments = arguments;
}
```
复制代码

`IArguments` 实际上就是:

```
interface IArguments {
    [index: number]: any;
    length: number;
    callee: Function;
}
```
复制代码

当然还有很多，在这里可以看到 TypeScript 核心库的定义文件

类型别名

类型别名就是给一种类型起个别的名字，之后只要使用这个类型的地方，都可以用这个名字作为类型代替。它只是起了一个名字，并不是创建了一个新类型。使用 `type` 关键字来定义:

```
type StringType = string;
let str: StringType;
str = 'hello';
str = 123 // Error
```
复制代码

注意: 类型别名不能被 `extends` 和 `implements`，且不能出现在声明右侧的任何地方。

`type` 实现继承，则可以使用交叉类型 `type A = B & C & D`。

字符串字面量类型

字符串字面量类型用来约束取值只能是某几个字符串中的一个。

```
type Name = 'ALisa' | 'Bob' | 'Cola'

let name: Name = 'Alisa'; // Error ①
let name1: Name = 'ALisa'; // OK
let name2: Name = 'Bob'; // OK
let name3: Name = 'Cola'; // OK
let name4: Name = '兔兔'; // Error
复制代码
```

上面的报错①原因在于：

在默认状态下，`TS` 将 `DOM typings` 作为全局的运行环境，所以当我们声明 `name` 时， 与 `DOM` 中的全局 `window` 对象下的 `name` 属性出现了重名。因此，报了 `Cannot redeclare block-scoped variable 'name'` 错误。

Enum 枚举类型

> 枚举是一个被命名的整型常数的集合，枚举在日常生活中很常见，例如表示星期的`SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY`就是一个枚举。

枚举是一种数据结构，使用枚举我们可以定义一些带名字的常量，清晰地表达意图或创建一组有区别的用例。 `TS` 支持数字的和基于字符串的枚举。

数字枚举

使用枚举来定义一周的7天：

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
复制代码
```

枚举成员会被赋值为从 `0` 开始递增的数字，我们可以像访问对象属性一样访问枚举成员：

```
console.log(Days.Sun) // 0
console.log(Days.Mon) // 1
......
console.log(Days.Sat) // 6
复制代码
```

我们还可以初始化枚举成员，那么该初始化成员后面的成员会在它的基础上自动增长 `1`：

```
enum Days {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};

console.log(Days.Sun) // 1
console.log(Days.Mon) // 2
......
console.log(Days.Sat) // 7
```
复制代码

字符串枚举

字符串枚举很简单，直接赋给每个成员字符串字面量：

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```
复制代码

异构枚举

异构枚举也就是说，枚举可以混合字符串和数字成员：

```
enum Direction {
    name = '兔兔',
    age = 18
}
```
复制代码

可以计算的成员

可以计算的成员意思就是，初始化枚举成员时，可以使用表达式、函数等方式动态求值，还可以是对之前定义的常量枚举成员的引用。

需高度重视的一点：使用可计算的成员前提是，当前枚举必须为数字枚举，即所有成员都必须为`number`类型。

```
enum Direction {
    None,
    Read = 1 << 1,
    Write = 1 << 2,
    ReadWrite = Read | Write,
    G = "123".length,
    Age = 18,
    Sex = getSex()
}

function getSex() {
    return 12
}
```
复制代码

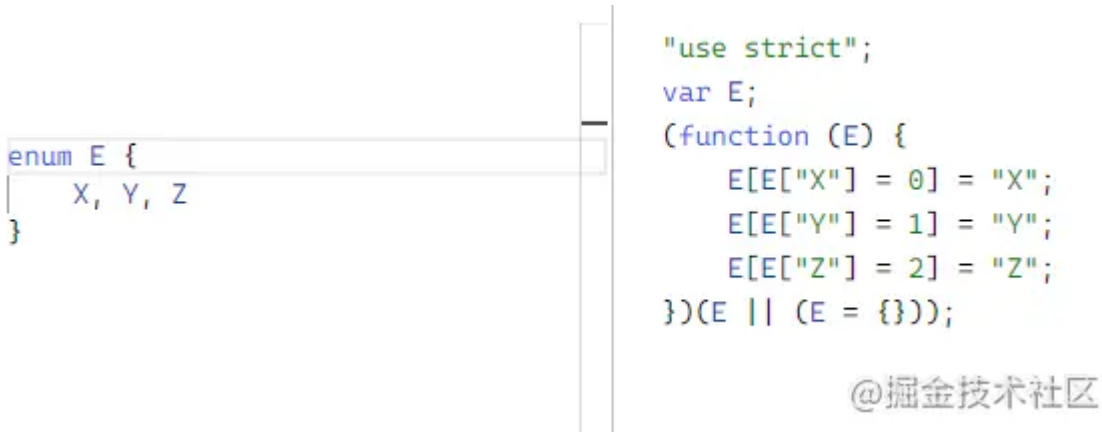
若常量枚举表达式求值后为 `NaN` 或 `Infinity`，则会在编译阶段报错。 官网原话：

> It is a compile time error for constant enum expressions to be evaluated to `NaN` or `Infinity`.

但是亲测并不会报错。

运行时的枚举

枚举是在运行时真正存在的对象，看看枚举编译后的样子：



在这里又学到了，这里 `LIFE` 函数内的写法很巧妙，将 `E["X"] = 0; E[0] = 'X';` 直接简化为 `E[E["X"] = 0] = "X";`。这也是枚举具有反向映射的原因所在，需要注意的是，字符串枚举成员不具有反向映射。

因为枚举是在运行时真正存在的对象，所以我们可以将它可以传递给函数：

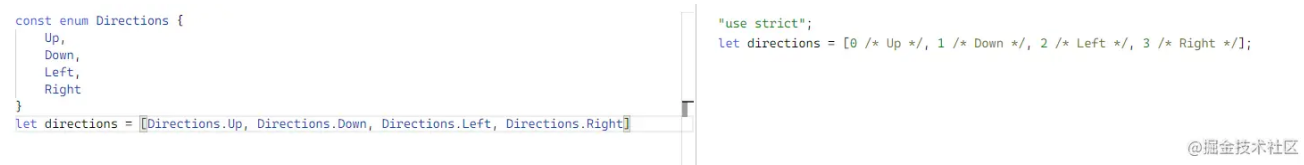
```
enum E {
    X, Y, Z
}
function f(obj: { X: number }) {
    return obj.X;
}

f(E); // OK
```

复制代码

const 枚举

通过在枚举上使用 `const` 修饰符来定义，可以避免在额外生成的代码上的开销和额外的非直接的对枚举成员的访问。

这句话可能不好理解，直接看图说话：



我们可以看到 `const` 枚举编译后，并没有像上面一样，编译成一个 `LIFE` 函数，而是直接被删除掉了，反之在应用到的地方直接填充常量上去。这样做的目的就是为了节省性能。

类

public 修饰符

属性修饰符默认为`public`公共的，即类的属性、方法可以在外部访问，你可以根据个人喜好决定是否显式声明。

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
复制代码
```

private 修饰符

`private`与`public`相对，私有修饰符，即类的属性、方法不可以在外部访问。

```
class Animal {
    public static age: number = 18;
    private static title: string = '兔兔';
}

Animal.age; // OK
Animal.title; // Error
复制代码
```

protected 修饰符

`protected`修饰符与`private`修饰符的行为很相似，但有一点不同，`protected`成员在**派生类**中仍然可以访问。注意，这里是**派生类中**，而不是**实例、子类实例。**

例 1：

```
class Animal {
    private age: number = 18;
    protected title: string = '兔兔';
}

class Dog extends Animal {
    getAge() {
        console.log(this.age) // Error
    }

    getTitle() {
        console.log(this.title) // OK
    }
}
复制代码
```

例 2：

```
class Animal {
    private static age: number = 18;
    protected static  title: string = '兔兔';
}

class Dog extends Animal {
    getAge() {
        console.log(Dog.age) // Error
    }

    getTitle() {
        console.log(Dog.title) // OK
    }
}
复制代码
```

参数属性

我们也可以在类的内部方法上对参数使用`public、private、protected`修饰符，它的作用是可以更方便地让我们在一个地方定义并初始化一个成员。

```
class Animal {
    constructor(public name: string, private age: number, protected sex: string) {}
}
复制代码
```

等同于：

```
class Animal {
    public name: string;
    private age: number;
    protected sex: string;
    constructor(name: string, age: number, sex: string) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }
}
复制代码
```

证明：

```
class Animal {
    constructor(public name: string) {}
    getName() {
        console.log(this.name)
    }
}

let animal = new Animal('兔兔');
animal.getName(); // "兔兔"
```
复制代码

抽象类

抽象类做为其它派生类的基类使用, 不允许被实例化。 不同于接口，抽象类可以包含成员的实现细节。

`abstract`关键字是用于定义抽象类和在抽象类内部定义抽象方法。

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}

let animal = new Animal(); // Error: 抽象类不允许被实例化
```
复制代码

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。 抽象方法的语法与接口方法相似。 两者都是定义方法签名但不包含方法体。 然而，抽象方法必须包含`abstract`关键字并且可以包含访问修饰符。

```
abstract class Department {
    constructor(public name: string) {
    }
    printName(): void {
        console.log('Department name: ' + this.name);
    }
    abstract printMeeting(): void; // 必须在派生类中实现
}
class AccountingDepartment extends Department {
    constructor() {
        super('Accounting and Auditing'); // 在派生类的构造函数中必须调用 super()
    }
    printMeeting(): void {
        console.log('The Accounting Department meets each Monday at 10am.');
    }
    generateReports(): void {
        console.log('Generating accounting reports...');
    }
}
let department: Department; // OK: 允许创建一个对抽象类型的引用
department = new Department(); // Error: 不能创建一个抽象类的实例
department = new AccountingDepartment(); // OK: 允许对一个抽象子类进行实例化和赋值
```

```
department.printName(); // OK
department.printMeeting(); // OK
department.generateReports(); // Error: 方法在声明的抽象类中不存在
复制代码
```

类实现接口

> 与 `C#` 或 `Java` 里接口的基本作用一样，`TypeScript` 也能够用接口来明确的强制一个类去符合某种契约。

意思也就是，我们也可以用类去实现接口，这里使用关键字 `implements`：

```
interface Title{
  title: string;
}
class title implements Title{
  title: string = '兔兔';
  age: number = 18; // 在实现接口的基础上，也可以添加其他的属性和方法
}
复制代码
```

一个类可以实现多个接口：

```
interface Age {
  age: number;
}

interface Title{
  title: string;
}
class title implements Title, Age{
  title: string = '兔兔';
  age: number = 18;
}
复制代码
```

抽象类与接口的区别

类可以继承抽象类，类也可以实现接口，这两种情境下，我总觉得抽象类与接口极其相似，所以辨析一下。

> 抽象类是用来捕捉子类的通用特性的，而接口则是抽象方法的集合；抽象类不能被实例化，只能被用作子类的超类，是被用来创建继承层级里子类的模板，而接口只是一种形式，接口自身不能做任何事情。
>
> 其次，抽象类可以有默认的方法实现，子类使用extends关键字来继承抽象类，如果子类不是抽象类的话，它需要提供抽象类中所有声明方法的实现。而接口完全是抽象的，它根本不存在方法的实现，子类使用关键字implements来实现接口，它需要提供接口中所有声明方法的实现。

静态部分与实例部分

首先看一个示例：用构造器签名定义一个接口，并试图实现这个接口：

```
interface Person {
  new(name: string)
}
class People implements Person {
  constructor(name: string) {
    // ...
  }
}
// 报错: no match for the signature 'new (name: string): any'.
```
复制代码

这是因为：**当类实现一个接口时，只对实例部分进行类型检查**，而`constructor`存在于静态部分，所以不在检查的范围内。（这里静态部分指构造函数，原因是：静态属性或静态方法都直接挂在构造函数上）

所以做法如下：

```
// 针对类构造函数的接口
interface CPerson {
  new(name: string);
}
// 针对类的接口
interface IPerson {
  name: string;
  age: number;
}
function create(c: CPerson, name: string): IPerson {
  return new c(name);
}
class People implements IPerson {
  name: string;
  age: number;
  // 这里未声明 构造函数，根据 ES6 规定会有默认的顶上来
}

let p = create(People, 'funlee'); // 可以
```
复制代码

这里的做法是再定义一个新的不带构造器签名的接口出来，让类实现这个接口，这么绕了一圈，目的就是为了利用类型的兼容性原则，做到构造函数的类型检查，所以我们也可以利用函数表达式来书写：

```
interface CPerson {
  new(name: string):any;
}

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

let p: CPerson = class People implements IPerson {
    name: string;
    age: number;
```

```
    constructor(name: string) {}
}
```
复制代码

接口继承类

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

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```
复制代码

什么？接口竟然还可以继承类，这是什么骚操作。目前为止，我们知道接口只可以继承接口，因为它们是同一类别的，对于接口继承类，官方的解释是：**在TS中声明一个类的时候，同时也声明了一个类的实例的类型。**

所以，我们可以声明一个变量为`Greeter`类型：`let greeter: Greeter = new Greeter("world");`，这里冒号后面的`Greeter`此时就是作为类的实例类型而存在的，`new`后面的`Greeter`作为构造函数存在。

更进一步，我们知道`class`本质是`function`的语法糖:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```
复制代码

转译为`ES5`:

```
"use strict";
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
```
复制代码

这个类的实例类型`Greeter`就对应转译的`ES5`中构造函数`Greeter`的实例类型，既然指实例类型，所以在接口继承类的时候，构造函数、静态属性、静态方法是不被包含的（实例的类型当然不应该包括构造函数、静态属性或静态方法）：

```
class Point {
    /** 静态属性，坐标系原点 */
    static origin = new Point(0, 0);
    /** 静态方法，计算与原点距离 */
    static distanceToOrigin(p: Point) {
        return Math.sqrt(p.x * p.x + p.y * p.y);
    }
    /** 实例属性，x 轴的值 */
    x: number;
    /** 实例属性，y 轴的值 */
    y: number;
    /** 构造函数 */
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
    /** 实例方法，打印此点 */
    printPoint() {
        console.log(this.x, this.y);
    }
}
```
复制代码

同时声明的类型等同于：

```
interface PointInstanceType {
    x: number;
    y: number;
    printPoint(): void;
}

let p1: Point;
let p2: PointInstanceType; // p1 的类型与 p2 等价
```
复制代码

我相信**接口继承类**，我以后都不会用到，没必要搞的乱七八糟的。

泛型

泛型是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

泛型定义

泛型使用`<类型变量>`定义：

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

也可以一次定义多个类型参数：

```
function swap<T, U>(tuple: [T, U]): [U, T] {
    return [tuple[1], tuple[0]];
}

swap([7, 'seven']); // ['seven', 7]
复制代码
```

使用泛型变量

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

// 使用
identity<number>(1); // OK：明确的指定`T`是`number`类型
identity(1); // OK：让编译器自己推断类型
复制代码
```

泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法。

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

上例中，泛型`T`不一定包含属性`length`，所以编译的时候报错了。

解决方法：

```
// 1
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // OK
    return arg;
}

// 2
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // OK
    return arg;
}
```
复制代码

上述两种写法本质上是表明了参数是`Array`类型，所以可以使用`length`属性。

泛型类型是不允许使用类型断言的，因为泛型表示任意或所有类型，例如`string`类型无法断言为`object`类型。所以为了给泛型加上约束，我们需要使用继承接口来实现：

```
interface Lengthwise {
    length: number;
}
function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // OK
    return arg;
}

loggingIdentity({length: 10, value: 3}); // OK
loggingIdentity([1,2]); // OK
```
复制代码

泛型接口

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity: GenericIdentityFn = identity;
```
复制代码

我们可以把泛型参数提前到接口名上：

```
interface Person<T> {
    name: T;
    getAge(arg: T): T;
}

let myIdentity: Person<string> = {
    name: "兔兔",
    getAge(name) {
        return name
    }
};
```
复制代码

泛型类

与泛型接口类似，泛型也可以用于类的类型定义中：

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
```
复制代码

需要注意一点：泛型类指的是实例部分的类型，所以类的静态属性不能使用泛型类型。

```
class GenericNumber<T> {
    name: T;
    static zeroValue: T; // Error
    add: (x: T, y: T) => T;
    constructor(name: T) {
        this.name = name;
    }
}
```
复制代码

泛型参数的默认类型

在 `TypeScript 2.3` 以后，我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，这个默认类型就会起作用。

```
function createArray<T = string>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}
```
复制代码

类型守卫

联合类型适合于那些值可以为不同类型的情况。但当我们想确切地了解参数为某种类型时，怎么办?

```
interface A {
    name: string;
    age: number;
}
interface B {
    sex: string;
    home: string;
}

function doSomething(person: A | B): void {
    if(person.name) { // Error
        // ...
    }
}
```
复制代码

上面的写法导致编译错误，因为并不能确定在运行时 `person` 的类型是 `A` 还是 `B`，你可能会认为即使类型为 `B` 时，`name` 属性不存在，也会返回 `undefined` 从而不进入当前判断，这很正确，但这里是 `TS`，对于类型中未定义的属性，访问就会报错，这才能称得上是**类型检查**。当然 `A` 和 `B` 的共有属性可以正常访问，因为大家都有。

为了告诉程序，我们知道自己在做什么，我们使用断言来使其工作:

```
function doSomething(person: A | B): void {
    if((person as A).name) { // OK
        // ...
    }
}
```
复制代码

这种方式的弊端很明显，就是我们在需要访问类型属性的地方不得不多次使用类型断言。**类型守卫机制**帮我们解决这类问题，类型守卫就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。这概念听起来竟有些类似 `JS` 中的 `with` 语法，只不过这里限定的是作用域中的类型。

用户自定义的类型守卫

使用类型判定

要定义一个类型守卫，我们只要简单地定义一个函数，它的返回值是一个**类型谓词**:

```
function isA(person: A | B): person is A{
    return(person as A).name !== undefined;
}

// 使用
function doSomething(person: A | B): void {
    if(isA(person)) { // OK
        // ...
    }
}
复制代码
```

所谓的**类型谓词**，就是指`parameterName is Type`，并且`parameterName`必须是来自于当前函数签名里的一个参数名。

上述代码这么理解：返回值为`true`时，`person is A`成立。

这种写法，太痛苦了，因为必须额外定义函数判别是否为指定类型，如果有多个类型需要判别，那就得额外定义相应数量的函数，那这种写法的好处呢？上面说了好处就是可以限定作用域中的类型。

举个简单的例子来说明：

```
function isString1(test: any): test is string{
    return typeof test === "string";
}
复制代码
function isString2(test: any): boolean{
    return typeof test === "string";
}
复制代码
```

这里的两个例子，一个返回值是类型谓词，一个返回值是布尔值，使用：

```
function doSomething(param: any): void {
    if(isString1(param)) {
        console.log(param.toFixed(2)); // 编译时 Error
    }
    if(isString2(param)) {
        console.log(param.toFixed(2)); // 编译时 OK，但运行时 Error
    }
}

doSomething("兔兔");
复制代码
```

第一个`if`条件判断使用了类型谓词，所以在这个`if`作用域中的`param`类型被限定为`string`，`string`类型是没有`toFixed()`方法的，所以编译错误；

第二个`if`条件判断使用布尔类型返回值，`param`的类型未被限定为`string`，此时`param`的类型为`any`，而`any`会跳过类型检查，所以编译能通过。

使用in操作符

`in` 操作符是用来查找对象属性的，注意会查找原型链上的属性。我们平时用 `in` 最多的地方应该就是遍历对象时的 `for...in...` 循环了。

这里我们使用 `in` 来进行判断属性存在与否，也不失为良策：

```
function doSomething(person: A | B): void {
    if("name" in person) { // OK
        // ...
    }
}
复制代码
```

`typeof` 类型守卫

`in` 都可以，那 `typeof` 自然也可以。其实前面函数重载，就是使用的 `typeof` 类型守卫。

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") { // OK
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") { // OK
        return padding + value;
    }
    throw new Error(`Expected string or number, got '${padding}'.`);
}
复制代码
```

注意点：`typeof` 类型守卫只有两种形式能被识别：`typeof v === "typename"` 和 `typeof v !== "typename"`，`"typename"` 必须是 `"number"`，`"string"`，`"boolean"` 或 `"symbol"`。但是 `TypeScript` 并不会阻止你与其它字符串比较，语言不会把那些表达式识别为类型守卫。

`instanceof` 类型守卫

`typeof` 都出来了，那 `instanceof` 你应该也不会感到意外，我们同样可以用 `instanceof` 来细化类型。与 `typeof` 类似，这里就不举例说明了。

`keyof` 索引类型查询操作符

对应任何类型 `T`，`keyof T` 的结果为该类型上所有公共属性名的联合：

```
interface Eg1 {
  name: string,
  readonly age: number,
}
// T1的类型实则是 "name" | "age"
type T1 = keyof Eg1

class Eg2 {
  private name: string;
  public readonly age: number;
  protected home: string;
```

```
}
// T2实则被约束为 "age"
// 因为name和home不是公有属性，所以不能被keyof获取到
type T2 = keyof Eg2
```
复制代码

再来个例子：

```
interface Eg1 {
  name: string,
  readonly age: number,
}

interface Eg2 {
  sex: string
}
// T1的类型实则是 "name" | "age" | { sex: string }
type T1 = keyof Eg1 | Eg2

let a: T1 = "name"; // OK
let b: T1 = "age"; // OK
let c: T1 = { // OK
  sex: "男"
}
```
复制代码

注意：`keyof any`的结果为`string | number | symbol`，原因想想也很简单，这不就是我们常见的三种键值类型嘛。

TypeScript 2.8 作用于交叉类型的`keyof`被转换成作用于交叉成员的`keyof`的联合。 换句话说，`keyof (A & B)`会被转换成`keyof A | keyof B`。 这个改动应该能够解决`keyof`表达式推断不一致的问题。

```
type A = { a: string };
type B = { b: string };
type T1 = keyof (A & B);  // "a" | "b"
type T2<T> = keyof (T & B);  // keyof T | "b"
type T3<U> = keyof (A & U);  // "a" | keyof U
type T4<T, U> = keyof (T & U);  // keyof T | keyof U
type T5 = T2<A>;  // "a" | "b"
type T6 = T3<B>;  // "a" | "b"
type T7 = T4<A, B>;  // "a" | "b"
```
复制代码

`T[K]`索引访问操作符

```
interface Eg1 {
  name: string,
  readonly age: number,
}
// string
type V1 = Eg1['name']
// string | number
type V2 = Eg1['name' | 'age']
// any
type V3 = Eg1['name' | 'age2222'] // Error
// string | number
type V4 = Eg1[keyof Eg1]
```
复制代码

`T[keyof T]` 的方式，可以获取到 `T` 所有 `key` 的类型组成的联合类型； `T[keyof K]` 的方式，获取到的是 `T` 中的 `key` 且同时存在于 `K` 时的类型组成的联合类型。

注意：如果 `[]` 中的 `key` 有不存在 `T` 中的，则是 `any`；因为 `TS` 也不知道该 `key` 最终是什么类型，所以是 `any`；且也会报错。

映射类型

`TypeScript` 提供了从旧类型中创建新类型的一种方式 — **映射类型**。 在映射类型里，新类型以相同的形式去转换旧类型里每个属性。

需要使用关键字 `in`：

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

type ReadonlyRabbit<T> = { // 映射类型
    readonly [K in keyof T]: T[K];
}

// 使用
let rabbit: ReadonlyRabbit<Rabbit> = {
    name: "兔兔",
    age: 18
}

rabbit.name = "蛋黄" // Error: readonly
```
复制代码

问题：为什么 `type` 可以用来做类型映射，而 `interface` 不行?

答：因为叫 `typescript` 不叫 `interfacescript`，哈哈哈，开玩笑。原因是 `type` 支持计算属性，`interface` 不支持，`typescript` 的设定，所以不要再问为什么了。

下面来看看最简单的映射类型和它的组成部分：

```
type Keys = 'option1' | 'option2';
type Flags = { [K in Keys]: boolean };
复制代码
```

它的语法与索引签名的语法类型，内部使用了`for .. in`。 具有三个部分：

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. 属性的结果类型。

有条件类型

有条件的类型会以一个条件表达式进行类型关系检测，从而在两种类型中选择其一，使用关键字`extends`配合三目运算符：

```
T extends U ? X : Y
复制代码
```

上面的类型意思是，若`T`能够赋值给`U`，那么类型是`X`，否则为`Y`。

```
type TypeName<T> =
    T extends string ? "string" :
    T extends number ? "number" :
    T extends boolean ? "boolean" :
    T extends undefined ? "undefined" :
    T extends Function ? "function" :
    "object";
type T0 = TypeName<string>;  // "string"
type T1 = TypeName<"a">;  // "string"
type T2 = TypeName<true>;  // "boolean"
type T3 = TypeName<() => void>;  // "function"
type T4 = TypeName<string[]>;  // "object"
复制代码
```

分布式有条件类型

分布式有条件类型就是`extends`前面的参数为联合类型时则会分解（依次遍历所有的子类型进行条件判断）联合类型进行判断。然后将最终的结果组成新的联合类型。

意思即`T extends U ? X : Y`，若`T`的类型为`A | B | C`，则会被解析为`(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`。

```
// type A1 = 1
type A1 = 'x' extends 'x' ? 1 : 2;

// type A2 = 2
type A2 = 'x' | 'y' extends 'x' ? 1 : 2;

// type A3 = 1 | 2
type P<T> = T extends 'x' ? 1 : 2;
type A3 = P<'x' | 'y'>
复制代码
```

看例子说话，所以上面说的 **extends 前面的参数为联合类型时则会分解**应该再精确为 **extends 前面的类型是泛型，且泛型传入的是联合类型时**才会分解。

- 阻止extends关键词对于联合类型的分发特性

如果不想被分解（分发），做法也很简单，可以通过简单的元组类型包裹以下：

```
type P<T> = [T] extends ['x'] ? 1 : 2;
// type A4 = 2
type A4 = P<'x' | 'y'>
复制代码
```

`infer` 操作符

> 在有条件类型的`extends`子语句中，允许出现`infer`声明，它会引入一个待推断的类型变量。

我们获取到的信息是：

1. `infer`操作符只允许出现在`extends`子语句中；
2. 它是用来推断类型变量的。

一个例子：

```
// ReturnType 为内置工具类型，作用：由函数类型 T 的返回值类型构造一个类型。
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;

type func = () => number;
type variable = () => string;
type funcReturnType = ReturnType<func>; // funcReturnType 类型为 number
type varReturnType = ReturnType<variable>; // varReturnType 类型为 string
复制代码
```

`infer`还可以用于解包，下面是一个解包数组里的元素类型的例子：

```
type Ids = number[];
type Names = string[];
type Unpacked<T> = T extends (infer R)[] ? R : T;

type idType = Unpacked<Ids>; // idType 类型为 number
type nameType = Unpacked<Names>; // nameType 类型为string
复制代码
```

`infer` 还有个相当重要的特性：

- `infer` 推导的名称相同并且都处于**协变**的位置，则推导的结果将会是**联合类型**；
- `infer` 推导的名称相同并且都处于**逆变**的位置，则推导的结果将会是**交叉类型**。

协变与**逆变**的解释可以看这里Ts高手篇

协变的例子：

```
// 例1
type Foo<T> = T extends { a: infer U; b: infer U } ? U : never;

type T10 = Foo<{ a: string; b: string }>; // T10类型为 string
type T11 = Foo<{ a: string; b: number }>; // T11类型为 string | number

// 例2
type ElementOf<T> = T extends (infer R)[] ? R : never;

type Tuple = [string, number];
type Union = ElementOf<Tuple>; // Union 类型为 string | number
复制代码
```

逆变的例子：

```
type Bar<T> = T extends {
  a: (x: infer U) => void;
  b: (x: infer U) => void;
} ? U : never;

// type T1 = string
type T1 = Bar<{ a: (x: string) => void; b: (x: string) => void }>;

// type T2 = never
type T2 = Bar<{ a: (x: string) => void; b: (x: number) => void }>;
复制代码
```

字符串模板类型

模板字符串能够对文本进行一定程度上的约束，与 `ES6` 类似，只不过 `${}` 内放的为类型且不支持计算：

```
type HTTP = `http://${string}`
type HTTPS = `https://${string}`

let a: HTTP = 'http://small-rabbit.top'; // OK
let b: HTTPS = 'https://small-rabbit.top'; // OK
复制代码
```

结合泛型的使用：

```
type EventName<T extends string> = `${T}Changed`;
type T0 = EventName<'foo'>;  // 'fooChanged'
type T1 = EventName<'foo' | 'bar' | 'baz'>;  // 'fooChanged' | 'barChanged' |
'bazChanged'

let t: T0 = 'fooChanged'; // OK
let a: T1 = 'fooChanged'; // OK
let b: T1 = 'barChanged'; // OK
let c: T1 = 'bazChanged'; // OK
复制代码
type Concat<S1 extends string, S2 extends string> = `${S1}${S2}`;
type T = Concat<'Hello', 'World'>;  // 'HelloWorld'

let t: T = 'HelloWorld'; // OK
复制代码
```

字符串模板中的联合类型会被展开后排列组合：

```
type T1 = "top" | "bottom";
type T2 = "left" | "right";
type T3 = `${T1}-${T2}`; // 'top-left' | 'top-right' | 'bottom-left' | 'bottom-right'

let t1: T3 = 'top-left'; // OK
let t2: T3 = 'top-right'; // OK
let t3: T3 = 'bottom-left'; // OK
let t4: T3 = 'bottom-right'; // OK
复制代码
```

为了帮助修改字符串文字类型，还添加了一些新的实用程序类型别名来修改字母中的大小写。新的类型别名是 `Uppercase`，`Lowercase`，`Capitalize` 和 `Uncapitalize`。前两个转换字符串中的每个字符，后两个只转换字符串中的第一个字符。

```
type Cases<T extends string> = `${Uppercase<T>} ${Lowercase<T>} ${Capitalize<T>}
${Uncapitalize<T>}`;
type T = Cases<'bar'>;  // 'BAR bar Bar bar'

let t: T = 'BAR bar Bar bar'; // OK
复制代码
```

参考

TypeScript 入门教程

TypeScript 4.0 使用手册

Ts高手篇：22个示例深入讲解Ts最晦涩难懂的高级类型工具