第 3 章

CHAPTER 3

类 型 进 阶

在掌握了 TypeScript 的基础类型系统后，本章将深入探讨类型缩窄、联合类型、交叉类型等高级的类型特性。

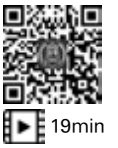
3.1 变化的类型

开启"noImplicitAny"配置（详见 8.4.1 节）后，赋值有类型缩窄的效果，代码如下：

```
//chapter3/vary.ts
//初始 a 的类型为 any
let a
//a 的类型被推断为 number
a = 1
a.toFixed()
//a 的类型被推断为 string
a = 's'
a.at(1)
```

如果变量的初始类型是有范围的，则后续的类型只能在这个范围内推断，代码如下：

```
//chapter3/vary.ts
  let b: number | string
  b = 1
  //类型被推断为 number
  b.toFixed()
  b = 'abc'
  //类型被推断为 string
  b.charAt(0)
  b = true
//～～
//不能将类型 boolean 分配给类型 string | number
```



3.2 类型缩窄

类型缩窄是 TypeScript 中的一种机制，通过控制流分析来确定一个变量的具体类型。根据不同的条件和上下文，TypeScript 能够“缩窄”变量的类型，在使用这些变量时能够获得更严格的类型检查。

3.2.1 相等判断

“==”或“===”可以用于类型缩窄。将变量的类型从宽泛的类型（例如 number）缩窄为具体的字面量类型（例如 1），代码如下：

```
//chapter3/narrow/equal.ts
function f(x: number) {
    //全等判断把 x 类型缩窄为字面量 1
    if (x === 1) {
       console.log(x === 3)
       //          ~~~~
       //此比较似乎是无意的，因为类型 1 和 3 没有重叠
    }
}
```

“===”可以用于联合类型缩窄，代码如下：

```
//chapter3/narrow/equal.ts

function f2(x: number | string) {
   if (x === '17') {
      x.charAt(0)                          //x 的类型被缩窄为 '17'
   } else {
      console.log(x)                       //x 的类型仍然是 number | string
   }
   if (x === 17) {
      x.toFixed()                          //x 的类型被缩窄为 17
   }
   else {
      console.log(x)                       //x 的类型仍然是 number | string
   }
}
```

“==”也可以用于类型缩窄，但与“===”有区别，代码如下：

```
function f3(x: string | null | undefined) {
   if (x == null) {
      console.log(x)                       //x 的类型被缩窄为 null | undefined
   } else {
      console.log(x)                       //x 的类型被缩窄为 string
```

```
    }
}
function f4(x: string | null | undefined) {
    if (x === null) {
        console.log(x);                    //x 的类型被缩窄为 null
    } else {
        console.log(x);                    //x 的类型被缩窄为 string | undefined
    }
}
```

x==null 会将 x 的类型从 string|null|undefined 缩窄为 null|undefined，因为 null==undefined 为 true。

x===null 会将 x 的类型从 string|null|undefined 缩窄为 null，因为 null===undefined 为 false。

建议使用“===”而不是“==”进行类型缩窄。

3.2.2 真假判断

判断为假值的有 false、0、NaN、""、0n、null、undefined，示例代码如下：

```
//chapter3/narrow/boolean.ts
function f1(ok: 0 | 1) {
    if (ok) {
        ok === 0
//~~~~~~~~~
//此比较似乎是无意的，因为类型 1 和 0 没有重叠
    }
}
```

经过 if 判断，TypeScript 会排除假值。由于 ok 的类型在 if 分支是 1，ok 的值不可能为 0，所以 TypeScript 提示错误。

对象和对象属性也可以进行真假判断，代码如下：

```
//chapter3/narrow/boolean.ts
interface Options {
    location?: {
        x?: number,
        y?: number
    }
}
function f2(options?: Options) {
    if (options && options.location && options.location.x) {
          //x 的类型是 number
        const x = options.location.x
    }
}
```

通过 if 判断排除 options、options.location、options.location.x 为空的情况，x 的类型必

然是 number。

因为 0 和""在 if 语句中判断为假，所以很容易产生逻辑陷阱，代码如下：

```
//chapter3/narrow/boolean.ts
function f3(a: number | undefined) {
    if (a) {
        return a + 1
    }
}
function f4(a: string | undefined) {
    if (a) {
        return a.charAt(0)
    }
}
```

本意是想先用 if(a)排除 undefined，然后对 number 和 string 类型进行处理，但由于 0 和""为假，所以 if(a)把 0 和""也给排除了。

注意 需要开启 strictNullChecks 配置，否则 undefined 类型被忽略，即使不用 if 判断，TypeScript 也不会提示错误，详见 8.4.1 节。

解决的办法是对这两个特殊值进行单独判断，代码如下：

```
//chapter3/narrow/boolean.ts
function f5(a: number | undefined) {
    if (a || a === 0) {                    //a!== undefined
        return a + 1
    }
}
function f6(a: string | undefined) {
    if (a || a === '') {                   //a!== undefined
        return a.charAt(0)
    }
}
```

NaN 比较特殊，在 TypeScript 中 NaN 是 number 类型，无法在类型上进行缩窄。如果要排除 NaN，则可以用 Number.isNaN，代码如下：

```
//chapter3/narrow/boolean.ts
function f7(a: number) {
    if (!Number.isNaN(a)) {
        //a 不是 NaN
    }
}
```

3.2.3 typeof

在 JavaScript 中，typeof 仅用于运行时类型检查。在 TypeScript 中，typeof 不仅可以用

于运行时检查,还可以用于类型推断和类型缩窄,帮助开发者在编译时确保类型安全。

typeof 可以判断 number、bigint、string、function、boolean、undefined 、object 和函数类型,代码如下:

```
//chapter3/narrow/typeof.ts
function f(value: unknown) {
    if (typeof value == 'string') {
        //value 的类型是 string
    }
    if (typeof value === 'number') {
        //value 的类型是 string
    }
    if (typeof value === 'boolean') {
        //value 的类型是 boolean
    }
    if (typeof value === 'bigint') {
        //value 的类型是 bigint
    }
    if (typeof value === 'symbol') {
        //value 的类型是 symbol
    }
    if (typeof value === 'undefined') {
        //value 的类型是 undefined
    }
    if (typeof value == 'object') {
        //value 的类型是 object 或 null
    }
    if (typeof value == 'function') {
        //value 的类型是 Function
    }
}
```

由于 typeof null 也会返回'object',因此在处理对象时需要额外判断,代码如下:

```
//chapter3/narrow/typeof.ts
function f2(value:number|object | null) {
    if (typeof value === 'object') {
        //这里 value 可能是 object 或 null
        if (value === null) {
            console.log('value is null')
        } else {
            //在这个块中,value 的类型被缩窄为 object
            console.log('object value:', value)
        }
    } else {
        //在这个块中,value 的类型被缩窄为 number
        console.log('number value:', value.toFixed())
    }
}
```

3.2.4 in

in 用于检查对象中是否存在某种属性，并且可以在 TypeScript 中帮助缩窄类型，代码如下：

```
//chapter3/narrow/in.ts
type Fish = { swim: () => void }
type Bird = { fly: () => void }

function f(animal: Fish | Bird) {
    if ('swim' in animal) {
        //animal 是 Fish
        animal.swim()
    }
    else {
        //animal 是 Bird
        animal.fly()
    }
}
```

3.2.5 instanceof

instanceof 是一个用来检查对象的原型链中是否存在某个构造函数的运算符。在 TypeScript 中，它不仅用于运行时检查对象是不是某个类的实例，还可以帮助编译器在代码块中缩窄变量的类型，增强类型推断的精确性，代码如下：

```
//chapter3/narrow/instanceof.ts
class Animal { eat() { } }
function f(value: Animal | number) {
    if (value instanceof Animal) {
        //value 的类型是 Animal
        value.eat()
    }
    else {
        //value 的类型是 number
        value.toFixed()
    }
}
```

instanceof 会检查对象的原型链。如果对象是目标类的子类的实例，则 instanceof 也会返回值 true，代码如下：

```
class Plant { }
class Tree extends Plant { }

const tree = new Tree();
```

```
console.log(tree instanceof Tree)                    //true
console.log(tree instanceof Plant)                   //true
```

instanceof 只能用于检查对象是不是某个类或构造函数的实例,不能用于检查接口或类型别名。

3.2.6 isArray

isArray 是判断数组的最佳选择,代码如下:

```
//chapter3/narrow/isArray.ts
function f(value: number | number[]) {
    if (Array.isArray(value)) {
        //value 的类型是 number[]
        value.push(1)
    }
    else {
        //value 的类型是 number
        value.toFixed(1)
    }
}
```

3.2.7 标记联合类型

对于任意两个对象类型,既可以用 in 判断对象中有没有特定的属性进行类型缩窄,也可以让一组对象类型都拥有一种同类型的属性,通过判断此属性进行类型缩窄,代码如下:

```
//chapter3/narrow/union.ts
type Fish = { name: 'fish', swim: () => {} }
type Bird = { name: 'bird', fly: () => {} }
type Dog = { name: 'dog', bark: () => {} }

function f1(animal: Fish | Bird | Dog) {
    switch (animal.name) {
        case 'fish':
            animal.swim()
            break
        case 'bird':
            animal.fly()
            break
        case 'dog':
            animal.bark()
            break

    }
}
```

用 if 也可以,代码如下:

```
//chapter3/narrow/union.ts
function f2(animal: Fish | Bird | Dog) {
    if (animal.name === 'fish') {
        animal.swim()
    }
    else if (animal.name === 'bird') {
        animal.fly
    }
    else {
        animal.bark()
    }
}
```

Fish、Bird、Dog 用 name 作标记。通过 name 可以很方便地区分它们。作标记的属性的类型可以是 string、number、boolean、symbol。

3.2.8 自定义类型缩窄函数

如果前面几种方案都不适合，则可以考虑自定义类型缩窄函数，代码如下：

```
//chapter3/narrow/custom/index1.ts
interface Cat {
    climb: () => void,
    eat: () => {}
}
interface Dog {
    eat: () => {}
}
function isCat(animal: Cat | Dog): animal is Cat {
    return 'climb' in animal
}
function f(animal: Cat | Dog) {
    if (isCat(animal)) {
        animal.climb()
    }
}
```

animal is Cat 的意思是：如果函数的返回值为 true，则在这个函数之后，TypeScript 将认为 animal 的类型是 Cat。isCat 函数用于告诉 TypeScript 编译器，在满足某些条件时，animal 是 Cat 类型。

自定义类型缩窄函数的返回值必须写成的形式为 parameterName is Type：，其中 parameterName 必须是当前函数签名中参数的名称，例如本例中，parameterName 是参数 animal，Type 是 Cat。

自定义类型缩窄函数的强大之处在于，无论初始类型是什么都能准确地把类型缩窄到指定的类型。

自定义类型缩窄函数是终级法宝，但是自定义类型缩窄函数最终会被编译成 JavaScript，

不会像类型那样编译后就被擦除,它会被保留在最终的代码里。如果开发者对最终代码的多少比较敏感,则需要考虑到这个问题。

当类型改变时,自定义类型缩窄函数可能会判断失败,需要记得同步修改。

TypeScript 5.5 对类型推断的能力有所提高,在满足一定条件下,可以让普通函数起到类型守卫的作用。

例如上面的 isCat 方法,可以直接用普通的函数,示例代码如下:

```
//chapter3/narrow/custom/index2.ts
interface Cat {
   climb: () => void,
   eat: () => {}
}
interface Dog {
   eat: () => {}
}
function isCat(animal: Cat | Dog) {
   return 'climb' in animal
}
function f(animal: Cat | Dog) {
   if (isCat(animal)) {
      animal.climb()                              //animal 的类型是 Cat
   }
}
```

并不是随便一个函数就能有这样的效果,还需要满足一些条件,这些条件有些复杂,不必去了解,当需要类型守卫时,直接把它写成类型守卫就可以了,这是最稳妥的方式。那么 TypeScript 5.5 的这种能力有什么用处呢? 它的意义在于之前 TypeScript 无法自动推断类型的情况现在可以自动推断了,示例代码如下:

```
//chapter3/narrow/custom/index3.ts
const list: (number | string)[] = [1, 'a']
const numberList = list.filter((item) => typeof item === 'number')
```

TypeScript 5.5 中 numberList 的类型是 number[],在 5.5 之前的版本,numberList 的类型是(number|string)[]。

3.2.9 switch(true)

TypeScript 5.3 支持 switch(true)写法,代码如下:

```
//chapter3/narrow/switch.ts
interface Animal {
   eat: () => void
}
interface Fish extends Animal {
   swim: () => void
}
```

```
interface Plant {
   hasLeaves: boolean
}
function isFish(value: object): value is Fish {
   return 'swim' in value
}
function isAnimal(value: object): value is Animal {
   return 'eat' in value
}
function isPlant(value: object): value is Plant {
   return 'hasLeaves' in value
}
function f1(x: Animal | Plant) {
   switch (true) {
      case isPlant(x):
         //x 的类型是 Plant
         console.log(x.hasLeaves)
         break
      case isFish(x):
         //x 的类型是 Fish
         x.swim()
      //注意,这里没有 break,后面的分支也会执行
      case isAnimal(x):
         //x 的类型是 Animal
         x.eat()
         break
   }
}
```

在 JavaScript 中,switch (true) 是一种不太常见但非常灵活地使用 switch 语句的方式。这种用法的关键点是 switch 的表达式是 true,而 case 子句中的表达式会与 true 进行比较。

函数 f1 通过 switch (true) 来判断传入对象 x 的类型:

(1) 如果 isPlant(x)的返回值为 true,则只执行 console.log(x.hasLeaves)。

(2) 如果 isFish(x)的返回值为 true,则执行 x.swim()并且无论如何都会继续执行 x.eat()。

(3) 如果 isAnimal 的返回值为 true 而 isB(x)的返回值为 false,则仅执行 x.to()。

如果改用 if,则可能会写成这样,代码如下:

```
//chapter3/narrow/switch.ts
function f2(x: Animal | Plant) {
   if (isPlant(x)) {
      //x 的类型是 Plant
      console.log(x.hasLeaves)
   } else if (isFish(x)) {
      //x 的类型是 Fish
      x.swim()
      //x 的类型也是 Animal
```

```
        x.eat()
    } else {
        //x 的类型是 Animal
        x.eat()
    }
}
```

这个示例并不是很复杂,看起来两种写法的差别不大。通常优先考虑 if else 写法,如果 if else 的写法很复杂,则可以考虑用 switch(true)。

3.2.10 小结

类型缩窄的本质是将一个宽泛的类型(父类型)精确为一个更具体的类型(子类型),例如:

(1) 将 string|number 缩窄为 string。

(2) 将 Animal 缩窄为 Dog(假设 Dog 是 Animal 的子类型)。

(3) 将 unknown 缩窄为具体的类型(例如 string、number 等)。

类型越精确,TypeScript 的提示也会更准确。

虽然有这么多类型缩窄的手段,但最佳的做法是在程序的设计上避免多类型的问题。

3.3 详尽检查

详尽检查是指在处理联合类型时,TypeScript 会检查是否所有可能的类型分支都被覆盖。如果没有覆盖所有分支,则 TypeScript 会发出警告或错误。

首先看一个不使用详尽检查的示例,代码如下:

```
//chapter3/check.ts
type Digit = 1 | 2
function f1(a: Digit) {
    switch (a) {
        case 1:
            break
        case 2:
            break
        default:
            break
    }
}
```

如果把 Digit 的类型修改为 1,则 case 2 语句会提示错误:类型 2 不可与类型 1 进行比较。如果把 Digit 的类型修改为 1 | 2 | 3 呢? 没有任何提示。在 JavaScript 中,可以在 default 分支抛出异常来警示类型 3 没有被处理,但异常只有在代码运行阶段才能抛出。如果代码没有进行全路径测试,则这个问题还是发现不了。

详尽检查是类型缩窄的一个应用。详尽检查检查的是类型，当类型发生改变时给出错误提示，只需在最后加一个 never 类型守卫就可以了。number 字面量详尽检查的代码如下：

```
//chapter3/check.ts
type Digit = 1 | 2
function f2(a: Digit) {
   switch (a) {
      case 1:
         break
      case 2:
         break
      default:
         const guard: never = a
   }
}
```

Digit 类型只有两个值，即 1 和 2。由于 switch 语句已经穷举了这两个值，所以代码不可能运行到 default 分支，这时，a 的类型被缩窄到 never。never 是所有类型的底类型，这已经是类型缩窄的极限了。

当 Digit 类型变为 1 | 2 | 3 后，default 分支的 a 的类型缩窄为 3，TypeScript 提示错误：不能将类型 number 分配给类型 never。

枚举也可以进行详尽性检查，代码如下：

```
   //chapter3/check.ts
const enum Status {
   run,
   stop
}
function f3(status: Status) {
   switch (status) {
      case Status.run:
         break
      case Status.stop:
         break
      default:
         const guard: never = status
   }
}
```

详尽检查相当于在编译阶段对所有可能的类型进行全路径测试，相比于运行时的全路径测试，成本要低得多。

3.4 断言

类型断言是一种编译时语法，也是一种为编译器提供如何分析代码的方法。类型断言不会影响代码的运行时行为，它仅仅是在编译阶段为编译器提供额外的类型信息。

3.4.1 as 断言

as 是类型断言,相当于告诉编译器:你不用管了,我确定这是 xx 类型,示例代码如下:

```
let a: unknown = 1
let b = a as number
```

类型转换还有一种写法,代码如下:

```
let b = <number>a
```

a as number 与<number>a 的语法等价,但是<>这种写法会与 JSX 的语法存在歧义,建议在任何地方都用 as 的语法。

1. 双重 as 断言

只有当两种类型存在父子关系时,才能用 as 断言,否则会报错,代码如下:

```
let a = 1
let b = a as string
//      ~~~~~~
//类型 number 到类型 string 的转换可能是错误的,因为两种类型不能充分重叠
```

可以用 unknown(或 any)作为类型转换的跳板,代码如下:

```
let a = 1
let b = a as unknown as string
```

先断言成 unknown,再断言成想要的类型,这种方式可以把一种类型转换成任何类型,但是如果代码中真出现这种双重断言,则很可能是代码有问题。

2. as const

as const 是一种类型断言语法,它的主要作用是让 TypeScript 将值推断为只读的精确的字面量类型,而不是更宽泛的类型,代码如下:

```
let a = 1 as const                    //a 的类型为 1,不会扩大为 number
```

当使用 as const 声明一个对象时,TypeScript 会将其中的元素的类型推断为常量类型,这意味着它们不能被修改,代码如下:

```
let fish = {
  name: 'fish',
  skill: {
    swim: () => { }
  }
} as const
```

对象的属性和子属性都会获得只读属性。as const 的作用相当于如下代码:

```
let fish: {
   readonly name: 'fish',
```

```
    readonly skill: {
        readonly swim: () => void
    }
} = {
    name: 'fish',
    skill: {
        swim: () => { }
    }
}
```

当使用 as const 声明一个数组时，TypeScript 会将其推断为只读元组，代码如下：

```
const arr = [1, 2] as const                    //arr 的类型为 readonly [1, 2]
```

如果没有 as const，[1, 2]会被扩大为 number[]，const arr 中的 const 只能确保 arr 变量本身不能被重新赋值（不能将 arr 指向另一个数组），但它不会阻止修改数组的内容（例如 arr[0]=3 是允许的）。

as const 会将数组推断为只读的，并且会尽可能地缩小类型范围。对于[1, 2] as const，TypeScript 将其推断为 readonly [1, 2]，这意味着数组的类型是一个只读的元组，并且元素的值也被固定为 1 和 2。

const 用于限制变量重新赋值，而 as const 用于限制值的修改和类型推断。

3.4.2 赋值断言

在 TypeScript 中，一个变量必须初始化才能使用，代码如下：

```
    let a: number
    a.toFixed()
//～～
//在赋值前使用了变量 a
```

有时值已经初始化，但 TypeScript 判断不出来，这时需要用“!”告诉 TypeScript 编译器，这个变量在使用前一定会被赋值，代码如下：

```
//chapter3/const.ts
let a!: number
fun(1)
a.toFixed()
function fun(value: number) {
    a = value
}
```

当开启 strictPropertyInitialization（详见 8.4.1 节）配置后，类属性必须初始化，代码如下：

```
//chapter3/const.ts
class User {
    name!: string
    constructor(name: string) {
```

```
        this.init(name)
    }
    init(name: string) {
        this.name = name
    }
}
```

TypeScript 只能检测到 constructor 中进行的初始化，这种在 init 方法中对 name 进行了初始化的情况，需要手动为 name 加上赋值断言，告诉 TypeScript 编译器：name 一定会被初始化。

3.4.3 非空断言

如果一个值的类型包含 undefined、null 类型，则可以用类型缩窄的方法排除 undefined、null 类型，代码如下：

```
let a: number | undefined
if (a!== undefined) {
  a.toFixed()
}
```

如果确定变量的值一定不为空，则也可以用“!”断言，代码如下：

```
let a: number | undefined
a!.toFixed()
```

在这个例子中，TypeScript 不再有错误提示，但运行时会报错，所以如果断言是非空断言，则此断言是有风险的。比较安全的方式是“?.”，代码如下：

```
let a: number | undefined
a?.toFixed()
```

“?.”是 JavaScript 的语法，保证运行阶段 a?.toFixed()不会报错。a?.toFixed()的逻辑相当于如下代码：

```
(a === null || a === undefined)? undefined : a.toFixed()
```

断言只在编译阶段有效，在运行的时候，如果类型不对，则会报错，所以要谨慎地使用断言。

用“!”非空断言时，优先考虑“?.”，但是，如果确定值不为空，则建议使用“!”，因为“!”会简化代码逻辑，也容易让错误早些暴露出来。

3.5 联合类型与交叉类型

在 TypeScript 中，联合类型和交叉类型是两种非常重要的类型组合方式。在第 2 章中，已经对联合类型与交叉类型进行了初步介绍，本节详细讲解联合类型与交叉类型。

3.5.1 联合类型

联合类型允许一个值是多种类型中的一种。使用竖线“|”联合不同的类型，例如 string | number 是类型 string 与类型 number 的联合，1 | 2 是类型 1 与类型 2 的联合。

1. 字面量联合类型

不同的字面量可以联合起来组成字面量联合类型，代码如下：

```
type Number1 = 1 | 2 | 3
```

字面量 1、2、3 通过“ | ”联合组成字面量联合类型 1 | 2 | 3。

字面量类型与字面量联合类型联合，字面量类型可能会被吸收（吸收只是一种形象化的说法），代码如下：

```
type Number1 = 1
type Number2 = 1 | 2
type Number3 = Number1 | Number2
```

Number3 的类型是 1 | 2，1 被 1 | 2 吸收了。再如，number 字面量和 number 联合会被 number 吸收，代码如下：

```
type Number4 = 1 | 2 | 3 | number
```

Number4 的类型为 number，字面量 1、2、3 都被 number 吸收了。

2. 仅联合类型中的共有成员可直接访问

联合类型表示一个值可以是多种类型中的任意一种。联合类型的成员之间是“或”的关系。联合类型 A | B 表示一个值既可以是 A 类型，也可以是 B 类型，但不能同时是 A 和 B 类型。在联合类型中，只能访问所有成员类型共有的成员。如果一个成员不是所有类型共有的，则在没有类型收窄的情况下不能直接访问它，代码如下：

```
//chapter3/union.ts
function f1(a: number | string) {
   a.toString()
   a.toFixed()
//～～～～～～
//类型 string|number 上不存在属性 toFixed
    a.charAt(0)
//～～～～～～
//类型 string | number 上不存在属性 charAt
}
```

在没有确定 a 是 number 还是 string 之前，唯一能确定的是 a 拥有 number 和 string 共同的方法和属性，例如 toString 方法。

由于既无法确定 a 一定有 toFixed 方法，也无法确定 a 一定有 charAt 方法，所以 TypeScript 会提示错误。

3. 在联合类型中,相同属性的类型会被推断为这些类型的联合类型

示例代码如下:

```
//chapter3/union.ts
interface Animal {
   name: string,
   swim?: () => void
}
interface Fish {
   name: 'fish',
   swim: () => void
}
function f2(animal: Animal | Fish) {
   //name 的类型为 string
   console.log(animal.name)
   animal.swim()
//~~~~~~
//不能调用可能是未定义的对象
```

Animal 和 Fish 都有 name 属性,类型为 string 和 fish,组成联合类型是 string | 'fish','fish'被 string 吸收,最终 Animal | Fish 中 name 的类型为 string。

Animal 和 Fish 都有 swim 属性,一个可选,另一个必选,组成联合类型后,swim 属性既可能存在,也可能不存在。为了安全地调用 swim 方法,需要通过类型收窄来确保 swim 方法存在。

4. any、unknown、never 的联合规则

(1) 任何类型与 any 联合都是 any。any 会吞噬其他类型,any 表示任意类型,完全绕过类型检查。

(2) 除 any 外,unknown 与任何类型联合都是 unknown。unknown 是 TypeScript 中的"安全版 any",它表示任何可能的类型。当 unknown 与具体类型联合时,由于 TypeScript 无法确定最终类型的具体范围,因此结果仍然是 unknown。

(3) never 与任何类型联合都会被吸收,例如 string | never 的结果是 string。never 是 TypeScript 中的"底部类型",表示不可能存在的值。在联合类型中,由于 never 表示"没有值",因此它不会对联合类型的结果产生任何影响。

(4) 在开启 strictNullChecks(详见 8.4.1 节)配置后,unknown 相当于{} | null | undefined 的联合类型,代码如下:

```
//chapter3/union.ts
function f3(x: unknown, y: {} | null | undefined) {
    x = y                                    //正确
    y = x                                    //正确
}
```

5. 使用联合类型

只使用联合类型共有属性的代码如下:

```
//chapter3/union.ts
function f4(a:number[] | string){
   console.log(a.length)
}
```

由于 f4()只需参数 a 的 length 属性，并且 number[]和 string 都有 length 属性，所以 length 属性可以直接使用。

把参数 a 的类型换成{length:number}也是可以的，{length:number}比 number[] | string 在类型表达上更加明确：只要有 length 属性，其他的属性都不关心。

number[] | string 相比于{length:number}还有一定限制：必须是 number[]或 string。

对于只使用联合类型共有属性的情形，建议优先考虑使用表达更加明确的类型，在 f4 中 a 的类型用{length:number}更合适。

对于联合类型，通常会先用各种类型缩窄(详见 3.1 节)的手段将类型缩窄至特定的类型，再进行处理，代码如下：

```
//chapter3/union.ts
function f5(a:number | string){
   if(typeof a === 'number'){
      a.toFixed()
   }
   else{
      a.charAt(0)
   }
}
```

3.5.2 交叉类型

在 TypeScript 中，交叉类型是将多种类型合并成一种新类型的机制。交叉类型允许将多种类型的属性结合在一起，形成一种同时拥有所有属性的新类型。A & B 表示 A 类型与 B 类型的交叉。type C=A&B 表示 C 必须是 A 类型也必须是 B 类型。

number、string、bigint、boolean、symbol、null、undefined 之间相互交叉的结果是 never，它们的字面量类型交叉的结果也是 never，例如 1&2 的结果是 never。

子类型与父类型交叉的结果为子类型，代码如下：

```
type A = 17 & number                          //17
type B = (string | number | null) & string    //string
```

1. 对象类型的交叉

简单来讲，两个对象类型进行交叉相当于是对两种类型进行合并，例如{a: number}&{b: string}交叉后的类型为{ a: number, b: string}。如果两个对象有相同的属性，但类型不同，则会尝试进行合并，如果合并不成功，则为 never，代码如下：

```
//chapter3/intersection/index1.ts
type A = { a: string }
type B = { a: number }
type C = A & B
```

C 的属性 a 的类型为 never。

如果两个对象有相同属性,一个可选,另一个必选,则结果为必选。举个可选属性合并的例子,代码如下:

```
//chapter3/intersection/index1.ts
interface D { age: number }
interface E { age?: number }
type F = D & E                          //{age:number}
```

2. 对象类型与原始类型的交叉

对象类型与原始类型的交叉虽然在语法上是允许的,但一般没有实际的意义,因为无法有实际的值相对应,除非是为了区分相同的类型,代码如下:

```
//chapter3/intersection/index2.ts
type A = { n: number } & 'web'
type B = { n: number } & 'native'
```

{n: number}与 web、native 交叉后就得到了两种不同的类型 A、B,非常容易区分。

{n: number}与 number、symbol、boolean 交叉的效果是一样的。具体用哪一个区别不大,可以优先用可读性好的那个。

A、B 在使用时需要用 as 进行强转,代码如下:

```
//chapter3/intersection/index2.ts
function f(a: A) {
   console.log(a)
}
f({ n: 1 } as A)
```

通常区分两种具有相同属性的对象类型,可以增加一个字段,示例代码如下:

```
//chapter3/intersection/index2.ts
type C = { n: number, type: 'web' }
type D = { n: number, type: 'native' }
```

3. 特殊类型的交叉规则

(1) 任何类型与 never 交叉都是 never。

(2) 除 never 外,任何类型与 any 交叉都是 any。

(3) {}与联合类型 A 交叉,可以删除 A 中的 null、undefined 类型(需要开启 strictNullChecks 配置,详见 8.4.1 节)。

(4) 除 never、any,unknown 与任何类型 A 交叉结果都是 A。

4. 类型交叉实现继承

类型通过 extends 实现继承，代码如下：

```
//chapter3/intersection/index3.ts
interface Animal{
   name:string
}
interface Dog extends Animal{
   bark:()=>{}
}
```

用类型交叉也可以实现相同的效果，代码如下：

```
//chapter3/intersection/index4.ts
type Animal = {
   name: string
}
type Bark = {
   bark: () => {}
}
type Dog = Animal & Bark
```

这两种方式实现的 Dog 类型是一样的，都有一个 name 属性和 bark 方法，但是类型 Bark 还可以和其他类型进行联合或交叉，这是 extends 与交叉实现方式的一个不同点。

3.5.3 小结

联合类型表示一种类型可以是多种类型中的任意一个。交叉类型表示一种类型同时满足多种类型的条件，要求同时具备所有指定类型的属性和方法，实际上是将它们结合在一起。联合与交叉可以通过现有的类型生成新类型。其优点是可以复用现有的类型；缺点是会增加复杂性，在实际使用时，要在灵活性和复杂性之间做好权衡。

strictNullChecks 配置将影响 undefined、null 和其他类型联合的结果。如果 strictNullChecks 为 false，则 undefined、null 会被联合类型“吸收”。

3.6 satisfies

在网页中，可以用 px、%、vw 等表示宽度。定义 Box 接口，width 为一个元素的宽度，可以写成 100、100px、100%、100vw，代码如下：

```
//chapter3/satisfy/index.ts
interface Box {
   width: number | string
}
let box1: Box = {
   width: 100
```

```
}
let box2: Box = {
   width: 300
}
const width = box1.width + box2.width
//              ~~~~~~~~~~~~
//运算符"+"不能应用于类型 string|number 和 string|number
```

因为 width 的类型可能是 number 或 string,所以 TypeScript 不允许应用"+"运算符,但是 box1、box2 的 width 的类型是 number,是可以相加的,有没有什么办法解决这个问题呢?

一种解决的办法是把类型标注去掉,让 TypeScript 自己推断,代码如下:

```
//chapter3/satisfy/plan1.ts
let box1 = {
   width: 100
}
let box2 = {
   width: 300
}
box1.width + box2.width
```

虽然不报错了,但也失去了类型保护,box1、box2 中可以随便添加其他属性,并且不会提示错误。

另一种解决的办法是用 as 强转,代码如下:

```
//chapter3/satisfy/plan2.ts
const width = (box1.width as number) + (box2.width as number)
```

强转相当于无视类型,就算 width 不是 number 类型,TypeScript 也不会提示错误。

一种可行的解决办法是类型缩窄(详见 3.1 节),代码如下:

```
//chapter3/satisfy/plan3.ts
if (typeof box1.width === 'number' && typeof box2.width === 'number') {
   const width = box1.width + box2.width
}
```

这样是可以的,但是如果类型比较多,则写起来有些麻烦。能不能既要 Box 类型保护,又让 TypeScript 标注准确的类型呢?这就需要 satisfies 出场了,代码如下:

```
//chapter3/satisfy/plan4.ts
interface Box {
   width: number | string
}
let box1 = {
   width: 100
} satisfies Box
let box2 = {
   width: 200
```

```
} satisfies Box

const width = box1.width + box2.width
```

satisfies Box 让 box1、box2 都得到了类型 Box 的保护，例如当尝试在 box1 中添加一个字段时会立即得到错误提示，代码如下：

```
//chapter3/satisfy/test.ts
let box1 = {
   width: 100,
   abc:1
//～～～
//对象字面量只能指定已知属性，并且 abc 不在类型 Box 中
} satisfies Box
```

width 的类型是 number，不再需要类型缩窄了，satisfies 完美地解决了问题。

x satisfies T 的作用是检查 x 是否符合类型 T 的要求，同时保持 x 的更具体的类型信息，而不是直接将 x 的类型变为 T。相比直接的类型断言（as），satisfies 提供了更严格的类型检查。

3.7 重载签名

在 JavaScript 中没有重载函数的概念，但是函数参数没有类型和个数的限制，可以很容易实现重载函数的效果，示例代码如下：

```
//chapter3/overload/index1.js
function hello(name, age) {
   if (age !== undefined) {
      return `你好 ${name}，年龄 ${age}岁`
   }
   return `你好 ${name}`
}

hello('小明')                    //你好小明
hello('小红', 16)                //你好小红，年龄 16 岁
```

1. 不使用重载签名

用 TypeScript 实现 hello 函数的代码如下：

```
//chapter3/overload/index1.ts
function hello(name: string, age?: number): string {
   if (age !== undefined) {
      return `你好 ${name}，年龄 ${age}岁`
   }
   return `你好 ${name}`
}
```

在调用 hello('小明')和 hello('小红', 16)时,把鼠标的光标放在 hello 上,函数签名都是 function hello(name: string, age?: number): string,如图 3-1 所示。

图 3-1 hello 函数签名

2. 使用重载签名

在 TypeScript 中,重载签名允许为同一个函数定义多个不同的调用签名,首先定义 hello()的重载签名,代码如下:

```
function hello(name: string): string
function hello(name: string, age: number): string
```

这两个函数签名声明了 hello 函数有两种不同的调用方式。第 1 种是接收 1 个字符串并返回 1 个字符串,第 2 种是接收 1 个字符串和 1 个数字并返回 1 个字符串。

把函数签名和实现函数放在一起,代码如下:

```
//chapter3/overload/index2.ts
function hello(name: string): string
function hello(name: string, age: number): string
function hello(name: string, age?: number) :string {
   if (age !== undefined) {
      return `你好${name},年龄${age}岁`
   }
   return `你好${name}`
}
hello('小明')
hello('小红', 16)
```

把鼠标的光标放在 hello 上,如图 3-2 所示。

图 3-2 hello 函数重载签名(1)

hello('小明')匹配了第 1 个重载签名。+1 overload 的意思是还有一个重载签名。把鼠标的光标放在 hello('小红', 16)上,如图 3-3 所示。

图 3-3 hello 函数重载签名(2)

hello('小红', 16)匹配了第 2 个重载签名。函数签名重载让函数签名更加明确,不再有可选类型。当不同的输入类型会有不同的返回类型时,函数重载签名可以提供更准确的类

型检查，代码如下：

```
//chapter3/overload/index3.js
function hello(name: string): string
function hello(age: number): number
function hello(nameOrAge: string | number): string | number {
    if (typeof nameOrAge === "string") {
        return `你好 ${nameOrAge}`
    }
    else {
        //距离成年还有多少年
        return 18 - nameOrAge
    }
}
//匹配 function hello(name: string): string
hello('小明')
//匹配 function hello(age: number): number
hello(17)
```

两个函数签名声明了 hello 函数有两种不同的调用方式。第 1 种是接收 1 个字符串并返回 1 个字符串，第 2 种是接收 1 个 number 并返回 1 个 number。

如果没有重载签名，则 hello 的返回值无论参数是什么类型，返回值类型都是 string | number。

以上两个示例只是为了演示重载签名的作用。实际上，第 1 个示例用可选参数是可以的，第 2 个示例把 hello 定义为泛型函数（详见第 6 章）更适合些。

有些函数的参数比较复杂，适合用重载签名辅助，示例代码如下：

```
//chapter3/overload/index4.ts
function formatDate(date: Date, format: string): string;
function formatDate(month:number, day: number): string;
function formatDate(dateOrMonth: Date | number, formatOrDate: string | number): string {
    if (typeof dateOrMonth === 'number') {
        const date = new Date()
        date.setMonth(dateOrMonth)
        if (typeof formatOrDate == 'number') {
            date.setDate(formatOrDate)
            return date.toLocaleDateString()
        }
        else {
            throw new Error('当 dateOrMonth 为 number 时，formatOrDate 必须为 number')
        }
    }
    else {
        //根据 dateOrMonth 和 formatOrDate 的格式信息，例如 YYYY-MM-DD，返回形如 2025-03-18
        //的字符串
        if (typeof formatOrDate === 'string') {
            //省略实现过程
            return '2025-03-18'
```

```
        }
        else {
            throw new Error('当 dateOrMonth 为 Date 时,formatOrDate 必须为 string')
        }
    }
}
//匹配 function formatDate(date: Date, format: string): string
formatDate(new Date(),'YYYY-MM-DD')
//匹配 function formatDate(month: number, day: number): string
formatDate(10,1)
formatDate(new Date(),1)
//~~~~
//没有与此调用匹配的重载
formatDate(10,'YYYY-MM-DD')
//~~~~
//没有与此调用匹配的重载
```

formatDate()接收两个参数,如果第 1 个参数是 Date,则第 2 个参数必须是 string;如果第 1 个参数是 number,则第 2 个参数必须是 number。如果参数类型不对,则会得到错误提示。虽然 formatDate()的返回值类型都是 string,但参数比较复杂,如果没有重载签名,签名提示并不明确,而且输入的类型不符合要求,则不会有错误提示。没有使用重载函数的代码如下:

```
//chapter3/overload/index5.ts

function formatDate(dateOrMonth: Date | number, formatOrDate: string | number): string {
    if (typeof dateOrMonth === 'number') {
        const date = new Date()
        date.setMonth(dateOrMonth)
        if (typeof formatOrDate == 'number') {
            date.setDate(formatOrDate)
            return date.toLocaleDateString()
        }
        else {
            throw new Error('当 dateOrMonth 为 number 时,formatOrDate 必须为 number')
        }
    }
    else {
        //根据 dateOrMonth 和 formatOrDate 的格式信息,例如 YYYY-MM-DD,返回形如 2025-03-18
        //的字符串
        if (typeof formatOrDate === 'string') {
            //省略实现过程
            return '2025-03-18'
        }
        else {
            throw new Error('当 dateOrMonth 为 Date 时,formatOrDate 必须为 string')
        }
    }
```

```
}
//formatDate(dateOrMonth: Date | number, formatOrDate: string | number): string
formatDate(new Date(), 'YYYY-MM-DD')
//formatDate(dateOrMonth: Date | number, formatOrDate: string | number): string
formatDate(10, 1)
formatDate(new Date(), 1)                          //没有错误提示
formatDate(10, 'YYYY-MM-DD')                       //没有错误提示
```

没有重载签名，formatDate 的第 1 个参数的类型提示为 Date | number，第 2 个参数的类型提示为 string | number。

3. 小结

重载签名主要用于处理不同类型的输入，返回不同类型的输出这种情况，让函数能够根据传入参数的不同而执行不同的逻辑。编译器会尝试从第 1 个重载签名开始，按照签名的顺序逐一匹配参数类型和参数个数。一旦找到匹配的重载声明，就会调用对应的函数实现。如果没有找到匹配的声明，就会产生编译错误。

重载签名可以让函数的用途更明确，使代码更具可读性。不同的参数组合可以明确表示不同的操作。有些函数的返回值虽然相同，但参数比较复杂，可能也需要重载签名。

有些重载签名的作用用类型编程（详见第 7 章）的方式也可以达到同样的效果，重载签名很清晰，类型编程可能比较简洁。

重载签名的作用如果可以用泛型函数（详见第 6 章）实现，则优先使用泛型函数。

如果是返回值类型相同但参数类型不同的情况，则通常可以用联合类型和可选参数解决，不需要使用重载签名。

通常，让 1 个函数只实现 1 个逻辑是优先要考虑的问题，让代码保持简单，简单的代码易于维护。只有在经过权衡得知收益比损失更大的情况下，才会考虑把多个逻辑用 1 个函数实现。

【练习 3-1】 写出函数 getNames 的函数签名。getNames 用于通过学号获取学生的姓名。getNames 既可以接收 1 个 number 类型参数（学号）并返回一名学生的姓名，也可以接收多个 number 类型参数（多个学号）并返回对应的多名学生的姓名。答案详见//chapter3/answer1.ts。

【练习 3-2】 动物园有老虎、狐狸和猴子 3 种动物，每种动物对应 1 个接口，写出函数 getAnimal 的签名，根据不同的动物名称返回对应的动物的接口。答案详见//chapter3//answer2.ts。

3.8 索引访问类型

在 TypeScript 中，可以通过对象的属性名来获取该属性的类型。语法是 T[K]，其中 T 是一种对象类型，K 是 T 中的一个属性名（键）。

1. 对象类型

示例代码如下：

```
//chapter3/indexAccess/object.ts
interface Person {
   name: string,
   age: number
}
//string
type Name = Person['name']
//number
type Age = Person['age']
//string | number
type NameAndAge = Person['name' | 'age']
```

Person['name']从 Person 类型中提取 name 属性的类型，结果是 string。

Person['age']从 Person 类型中提取 age 属性的类型，结果是 number。

Person['name' | 'age']就是从 Person 类型中提取 name 和 age 属性的类型，结果是 string 或 number(string | number)。

简单来讲，T[K]就是通过属性名 K 来获取对象类型 T 中对应属性的类型。

注意索引访问类型 Person['name']、Person['age']中的 name、age 都是类型，不是值，代码如下：

```
//chapter3/indexAccess/object.ts
const name = 'name'
type Name1 = Person[name]
//                  ~~
//name 表示值,但在此处用作类型
type nameAlias = 'name'
type Name2 = Person[nameAlias]
```

Person[name]中的 name 是值，TypeScript 会提示错误。Person[nameAlias]中的 nameAlias 是类型，正确。

索引访问类型必须用“[]”不能用“.”，代码如下：

```
//chapter3/indexAccess/object.ts
type Name3 = Person.name
//           ~~~~~~~
//无法访问 Person.name
```

对由所有的键组成的类型可以用 keyof(详细用法详见 7.4.1 节)得到，代码如下：

```
//chapter3/indexAccess/object.ts
type AllPropertyType = keyof Person
```

2. 数组类型

在 TypeScript 中，数组的本质是一个对象，可以通过索引访问类型来获取数组元素的类型，示例代码如下：

```
//chapter3/indexAccess/array.ts
type Ages1 = [
   { age: number },
   { age: number }
]
type Test1 = Ages1[0]
type Test2 = Ages1['0']
type Test3 = Ages1[number]
```

访问数组元素的类型有 3 种写法，在这个示例中，3 种写法的结果是一样的，即都是{age:number}。访问数组元素的类型通常不会使用第 2 种写法，第 1 种和第 3 种写法是有区别的，代码如下：

```
//chapter3/indexAccess/array.ts
type Ages2 = [
   { age: 17 },
   { age: 16 }
]
//{age:17}
type Test4 = Ages2[0]
//{age:16}|{age:17}
type Test5 = Ages2[number]
```

使用 0 获得 Ages 的第 1 个元素的类型，使用 number 获得 Ages 所有元素的类型组成的联合类型。

3. 小结

当有一种复杂的对象类型且想要获取其中某个特定属性的类型时，索引访问类型就非常方便，代码如下：

```
//chapter3/indexAccess/summary.ts
interface Config {
   dialog: {
      theme: 'light' | 'dark',
      fontSize: number
   }
}
type FontSize = Config['dialog']['fontSize']
```

索引访问类型可以灵活地提取对象类型中某个属性的类型，增强了类型的灵活性和可重用性。如果过多使用索引访问类型，则可能会降低代码的可读性，这种情况可以用类型组合的方式，先定义属性的类型，再组合成对象的类型，代码如下：

```
//chapter3/indexAccess/summary.ts
type Theme = 'light' | 'dark'
interface Dialog {
   theme: Theme,
   fontSize: number
}
```

3.9 索引签名

如果一个对象有多种属性,无法具体写出每种属性,则要怎么描述这种类型呢?索引签名就是用来解决这个问题的。索引签名用于定义一个对象的任意属性,允许在不知道具体属性名称的情况下使用这些属性。它通常用于动态属性的情况,代码如下:

```
//chapter3/indexSigniture/index.ts
interface Obj{
   [key:string]:number
}
const obj:Obj = {
   a:1,
   b:2
   //... 任意多个
}
```

key 可以写成其他的字符串,例如写成 k 也是可以的,代码如下:

```
interface Obj{
   [k:string]:number
}
```

JavaScript 中的对象属性键会自动地将 number 转换为 string,因此 obj[1]等价于 obj['1']。TypeScript 的索引签名[key: string]已经涵盖了这种行为,代码如下:

```
//chapter3/indexSigniture/string.ts
interface Obj {
   [key: string]: number
}
const obj: Obj = {
   1: 1
}
const value1 = obj[1]                          //正确
const value2 = obj['1']                        //正确
```

键的类型可以为 number,代码如下:

```
//chapter3/indexSigniture/number.ts
interface Obj {
    [key: number]: number
}
const obj: Obj = {
    1: 1, //ok
    abc: 2
//~~~
//对象字面量只能指定已知属性,并且 abc 不在类型 Obj 中
}
```

```
const value1 = obj[1]                                    //正确
const value2 = obj['1']                                  //正确
```

当键的类型为 number 时，字符串类型不能作为对象的键。在访问属性时，string、number 类型都可以。

键的类型也可以是 symbol 类型，但并不常用，了解一下即可，代码如下：

```
//chapter3/indexSigniture/symbol.ts
interface Obj {
    [key: symbol]: number
}
const obj = {
    [Symbol()]: 1
}
```

如果有的属性是必须有的，则可以明确写出来，但是类型要和索引签名的类型一致，代码如下：

```
//chapter3/indexSigniture/required.ts
interface Obj {
    height: number,                                      //必须是 number 类型
    [key: string]: number
}
const obj1: Obj = {}
//～～～～
//类型{}中缺少属性 height,但类型 Obj 中需要该属性
const obj2: Obj = {
    height: 1,                                           //只有 height 是必选的
    width: 2
}
```

明确写出来的属性是必选属性，不能缺少。

索引签名可以在不指定对象键的情况下定义键和值的类型，虽然很方便，但也会降低安全性，只在无法确定键的情况下才考虑索引签名。