【TypeScript 4.5】005-第 5 章 函数

【TypeScript 4.5】005-第 5 章 函数
一、函数类型表达式
1、概述
函数
函数类型表达式
2、代码演示
代码示例及解释
执行结果
使用类型别名
二、调用签名
1、概述
说明
代码示例
2、代码演示
代码示例及解释
执行结果
三、构造签名
1、概述
说明
代码示例
2、代码演示
代码示例及解释
执行结果
3、同时使用调用签名和构造签名
四、泛型函数
1、概述
说明
代码分析
泛型写法
调用分析
2、代码示例
正确写法
错误写法
另一种错误写法
还有一种错误写法
3、多个泛型
代码示例及解释
执行结果
五、泛型函数-限制条件
1、概述
说明
代码示例
2、代码示例与解释
六、泛型函数-使用受限值
1、代码分析
2、执行分析
七、泛型函数-指定类型参数
1、概述
说明
代码示例
2、代码演示
八、泛型函数-编写有些通用函数的准则

1、三个准则
2、代码演示
代码示例及解释（1）
代码示例及解释（2）
代码示例及解释（3）
九、函数的可选参数
1、概述
2、代码分析
十、回调中的可选参数
1、概述
2、代码演示
代码示例及解释
执行结果
十一、函数重载-基本语法
1、概述
说明
代码示例
2、代码示例及解释
十二、函数重载-重载签名和实现签名
1、三个问题
2、代码演示
参数不正确
参数类型不正确*
返回类型不正确*
十三、函数重载-编写好的重载
1、准则
2、代码演示
出现问题
解决问题
十四、函数内的 this 声明
1、概述
说明
代码示例
2、代码演示
老师的写法
我的写法*
十五、需要了解的其他类型
1、void
概述
代码示例
2、object
概述
3、unknown
概述
代码示例
4、never
概述
代码示例
联合类型无参数的时候
5、Function
概述
代码示例
十六、参数展开运算符-形参展开
1、概述
2、代码示例
十七、参数展开运算符-实参展开
1、概述
说明

代码示例
2、代码演示
代码示例
执行结果
3、一个问题
代码分析
解决问题
十八、参数解构
1、概述
2、代码示例
十九、返回 void 类型
1、概述*
2、代码演示

一、函数类型表达式

1、概述

函数

函数是任何应用程序的基本构件

无论是本地函数

还是从模块中导入的函数

或者类上的函数等等

向其他很多值一样

这些函数也是一个值

TypeScript 有很多方法来描述如何调用函数

函数类型表达式

```
fn: (a: string) => void
```

2、代码演示

代码示例及解释

```ts
// 参数是一个函数
function greeter(fn: (a: string) => void) {
    fn("hello world!")
}
// 此函数符合 greeter 参数的标准
function printSth(s: string) {
    console.log(s)
}
// 调用 greeter 传入 printSth 函数
greeter(printSth)
```

执行结果

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\01-function.js
hello world!
```

使用类型别名

> 两种写法等效，这种写法似乎更优雅一点点！

```ts
// 定义一个类型别名
type greeterFuntion = (a: string) => void
// 参数是一个函数
function greeter(fn: greeterFuntion) {
    fn("hello world!")
}
// 此函数符合 greeter 参数的标准
function printSth(s: string) {
    console.log(s)
}
// 调用 greeter 传入 printSth 函数
greeter(printSth)
```

二、调用签名

1、概述

说明

在 JavaScript 除了可调用之外

函数也可以有**属性**

然而**函数类型表达式**的语法**不允许**声明属性

如果我们想用属性来描述可调用的东西

可以在一个**对象类型**中写一个调用签名

> 2022.02.08 01:02:20 增补内容：

> 那么对象类型就像一个函数了！可以像调用函数一样调用对象！
>
> 比如：`let message: string = "hello world" message()`，其中的 `message()` 是会报错的！因为 message 没有调用签名！
>
> 参考下面的代码示例，DescribableFunction 里面有调用签名，调用签名就类似“函数部分”，有了它就可以像调用函数一样调用对象类型！可以使用诸如 `message()` 的形式进行调用，与调用函数一样！

代码示例

> 2022.02.08 09:19:02 增补内容：
>
> 调用签名实现了什么？实现了使得一个对象像一个函数一样调用！即带上括号，甚至有参数和返回值！

```
type DescribableFunction = {
    description: string, // 属性
    // 下面这行代码叫做【调用签名】
    (someArg: number): boolean // 函数类型表达式，注意这里使用的是冒号（:），而不是箭头（=>）！
}
```

2、代码演示

代码示例及解释

```
type DescribableFunction = {
    description: string, // 属性
    (someArg: number): boolean // 函数类型表达式
}
function doSth(fn: DescribableFunction){
    console.log(fn.description + "returned" + fn(6))
}
function fn1(num: number) {
    console.log(num)
    return true
}
// 必须给函数绑定属性，否则下一行会报错！
fn1.description = "hello world"
doSth(fn1)
```

执行结果

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\02-sign.js
6
hello worldreturnedtrue
```

三、构造签名

1、概述

说明

JavaScript 的函数可以用 new 操作符来调用

TypeScript 将这些称之为**构造函数**

因为它们通常会创建一个新的对象

我们可以通过在**调用签名**之前加一个 new 关键字来写一个**构造签名**

代码示例

> `(s: string): Ctor` 这些内容叫做【调用签名！】

```
// 多一个 new
type SomeConstructor = {
    new (s: string): Ctor
}
```

2、代码演示

代码示例及解释

> 2022.02.08 09:38:09 增补内容：
>
> 这里构造签名的参数与类的构造函数对应，具体参考第 8 章：类！

```
class Ctor {
    s: string
    constructor(s: string) {
        this.s = s
    }
}
type SomeConstructor = {
    new (s: string): Ctor
}
// SomeConstructor 可以将其理解为一个构造函数
function fn(ctor: SomeConstructor) {
    return new Ctor("难以理解的代码！")
}
const f = fn(Ctor)
console.log(f.s)
```

执行结果

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\03-constructor.js
难以理解的代码!
```

3、同时使用调用签名和构造签名

```ts
type CallOrConstructor = {
    new (s: string): Date
    (n?: number): number
}
function fn(date: CallOrConstructor) {
    let a = new date("2022-2-7")
    let b = date(100)
}
```

四、泛型函数

1、概述

说明

在写函数的时候输入的类型与输出的类型常常存在一定关系

我们会使用发泛型来解决

代码分析

> 此处，函数输入与返回都是any，我们希望函数返回值类型就是数组元素的类型，这就用到了泛型!

```ts
function firstElement(arr any[]) {
    return arr[0]
}
```

泛型写法

> 字母 T 是随便写的！T 也就是 Type 的缩写!

```ts
function firstElement<T>(arr: T[]): T | undefined {
    return arr[0]
}
```

调用分析

```
const a = firstElement(["a", "b", "c"]) // a 是 string 类型
const b = firstElement([1, 2, 3]) // b 是 number 类型
const c = firstElement([]) // c 是 undefined 类型
```

2、代码示例

正确写法

```
function firstElement<T>(arr: T[]): T | undefined {
    return arr[0]
}
firstElement(["a", "b", "c"])
```

错误写法

> 表示我们实现了输入类型与返回类型的一致！

```
function firstElement<T>(arr: T[]): T | undefined {
    return 100 // 报错：不能将类型“100”分配给类型“T | undefined”。ts(2322)
}
firstElement(["a", "b", "c"])
```

另一种错误写法

> 报错原因：因为 T 类型并不是固定的是 number 类型！

```
function firstElement<T>(arr: T[]): T | undefined {
    return 100 // 报错：不能将类型“100”分配给类型“T | undefined”。
}
firstElement([1, 2, 3])
```

还有一种错误写法

> 这里的 可以不写，但是如果写了后面的数组元素类型就必须与其一致！
>
> 一般不写，TS 自动推断即可！

```
function firstElement<T>(arr: T[]): T | undefined {
    return arr[0]
}
firstElement<string>([1, 2, 3]) // 报错：不能将类型“number”分配给类型“string”。
```

3、多个泛型

代码示例及解释

> 说明：
>
> - 两个泛型参数 T、Q
> - 函数的第一个参数是 T 数组，第二个参数是返回 Q 的函数
> - 函数的返回值是 Q 数组

```
function map<T, Q>(arr: T[], fn: (arg: T) => Q): Q[] {
    // 此处调用的是数组的 map 方法，里面需要传入一个函数
    return arr.map(fn)
}
const parsed = map(['1', '2', '3'], n => parseInt(n))
console.log(parsed)
```

执行结果

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\04-type.js
[ 1, 2, 3 ]
```

五、泛型函数-限制条件

1、概述

说明

有时候我们希望一个泛型是某个类型的子集

此时我们就需要对泛型的取值范围进行限制了

这和 Java 不一样

代码示例

> `T extends { length: number }` 这使得 T 一定具有属性 length，且是 number 类型！

```
function longest<T extends { length: number }>(a: T, b: T) {
    if (a.length >= b.length) {
        return a
    } else {
        return b
    }
}
```

2、代码示例与解释

```
// 定义函数，要求类型 T 必须具有 length 属性，且是 number 类型
function longest<T extends { length: number }>(a: T, b: T) {
    if (a.length >= b.length) {
        return a
    } else {
        return b
    }
}
// 调用测试
const a = longest([1, 2, 3], [4, 5])
const b = longest("hello", "hi")
// const c = longest(1000, 200) // 报错：类型“number”的参数不能赋给类型“{ length: number; }”的参数。
```

六、泛型函数-使用受限值

1、代码分析

> 使用通用约束条件时的常见错误!

```
function minimumLength<T extends { length: number }> (obj: T, minimum: number): T {
    if (obj.length > minimum) {
        return obj
    } else {
        // 报错：不能将类型“{ length: number; }”分配给类型“T”。
        // "{ length: number; }" 可赋给 "T" 类型的约束，但可以使用约束 "{ length: number; }" 的其他子类型实例化 "T"。
        return { length: minimum }
    }
}
```

2、执行分析

```
function minimumLength<T extends { length: number }> (obj: T, minimum: number):
T {
    if (obj.length > minimum) {
        return obj
    } else {
        // 报错：不能将类型“{ length: number; }”分配给类型“T”。
        // "{ length: number; }" 可赋给 "T" 类型的约束，但可以使用约束 "{ length:
number; }" 的其他子类型实例化 "T"。
        return { length: minimum }
    }
}
// 测试
const arr = minimumLength([1, 2, 3], 5)
// 执行时报错：arr.slice is not a function
console.log(arr.slice(0))
```

七、泛型函数-指定类型参数

1、概述

说明

TypeScript 通常可以推断出通用的函数调用中预期的类型参数

但并非总是如此

此时我们需要在调用类型参数的时候指定类型参数

代码示例

```
const arr = combine<string | number>([1, 2, 3], ["hello"])
```

2、代码演示

```
function combine<T>(arr1: T[], arr2: T[]): T[] {
    return arr1.concat(arr2)
}
const arr1 = combine([1, 2, 3], [4, 5, 6])
// 报错：不能将类型“string”分配给类型“number”。
// 因为此时 ts 将类型推断为数值类型
const arr2 = combine([1, 2, 3], ["hello"])
// 我们可以主动告诉数组的类型为联合类型 string | number
const arr3 = combine<string | number>([1, 2, 3], ["hello"])
```

八、泛型函数-编写有些通用函数的准则

1、三个准则

- 可能的情况下，使用参数本身，而不是对其进行约束；
- 总是尽可能少得使用类型参数；
- 如果一个类型的参数只出现在一个地方，请重新考虑你是否真的需要它。

2、代码演示

代码示例及解释（1）

> 可能的情况下，使用参数本身，而不是对其进行约束

```
// 此写法更规范，推断返回类型为 T
function firstElement1<T>(arr: T[]) {
    return arr[0]
}
// 推断返回类型为 any
function firstElement2<T extends any[]>(arr: T) {
    return arr[0]
}
// 调用
const x = firstElement1([1, 2, 3])
const y = firstElement2([1, 2, 3])
// 结论：可能的情况下，使用参数本身，而不是对其进行约束
```

代码示例及解释（2）

> 总是尽可能少得使用类型参数

```
// 写法1
function filter1<T>(arr: T[], func: (arg: T) => boolean) {
    return arr.filter(func)
}
// 写法2
function filter2<T, Func extends (arg: T) => boolean>(arr: T[], func: Func) {
    return arr.filter(func)
}
// 评论：第一种写法更好，第二种写法的 Func 除了使得函数更复杂之外，什么也没做！
```

代码示例及解释（3）

> 如果一个类型的参数只出现在一个地方，请重新考虑你是否真的需要它

```ts
// 完全没必要使用泛型
function greet<Str extends string>(s: Str) {
    console.log("hello " + s)
}
// 更简洁
function greet1(s: string) {
    console.log("hello " + s)
}
```

九、函数的可选参数

1、概述

JavaScript 中的一些函数经常需要一个**可变数量**的参数

2、代码分析

```ts
function f(n: number) {
    console.log(n.toFixed()) // 0个参数
    console.log(n.toFixed(3)) // 1个参数
}
// 我们可以在参数后面加一个问号（?）表示该参数是可选的
function fn(n?: number) {} // 此时可传入参数 n ，也可以不传！
// 扩展：默认参数，如果想不传参数也可以设置默认参数，如下：
function fn(n: number = 100) {} // 不传参数的时候 n 的值为 100
```

十、回调中的可选参数

1、概述

当为回调写一个函数类型时

永远不要写一个可选参数

除非你打算在不传递该参数的情况下调用函数

> 因为在编写调用回调的函数时容易出错！

2、代码演示

代码示例及解释

```ts
// 定义函数，这里的 index 是可选参数！
function myForEach(arr: any[], callback: (arg: any, index?: number) => void){
    for (let i = 0; i < arr.length; i++) {
        callback(arr[i], i)
    }
```

```
}
// 调用测试
myForEach([1, 2, 3], a => console.log(a))
myForEach([10, 20, 30], (a, i) => console.log(a, i))
// index 是可选参数，当不传入时
function myForEach1(arr: any[], callback: (arg: any, index?: number) => void){
    for (let i = 0; i < arr.length; i++) {
        callback(arr[i])
    }
}
myForEach1([1, 2, 3], a => console.log(a))
myForEach1([10, 20, 30], (a, i) => console.log(a, i))
```

执行结果

> 当回调函数的可选参数未传入时，为 undefined ，可能会导致很多错误!

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\10-callbacks.js
1
2
3
10 0
20 1
30 2
1
2
3
10 undefined
20 undefined
30 undefined
```

十一、函数重载-基本语法

1、概述

说明

JavaScript 函数可以在不同参数数量和类型的情况下被调用

在 TypeScript 中我们可以通过编写重载签名

来指定一个可以以不同方式调用的函数

为此，我们通常要写一定数量的函数签名（两个或更多）

代码示例

```
// 前两个称之为 重载签名
function makeDate(timestamp: number): Date
function makeDate(m: number, d: number, y: number): Date
// 后面一个称之为 实现签名
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
    // ...
}
```

2、代码示例及解释

```
function makeDate(timestamp: number): Date
function makeDate(m: number, d: number, y: number): Date
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
    if (d !== undefined && y !== undefined) {
        return new Date(y, mOrTimestamp, d)
    } else {
        return new Date(mOrTimestamp)
    }
}

const d1 = makeDate(12345678)
const d2 = makeDate(2, 9, 2022)
// 报错：没有需要 2 参数的重载，但存在需要 1 或 3 参数的重载。
// 参数是由重载签名决定的，实现签名只是为了适配重载签名，且对外部是隐藏的！
// 所以这里没有能够传两个参数的重载！
const d3 = makeDate(2, 9)
```

十二、函数重载-重载签名和实现签名

1、三个问题

- 参数不正确
- 参数类型不正确
- 返回类型不正确

2、代码演示

参数不正确

```
function fn(x: string): void // 重载签名带参数
function fn(){} // 实现签名无参数
// 此处传入参数并不报错，也就再次说明实现签名是隐藏的！
fn("hello")
```

参数类型不正确*

> 此处存疑！与老师讲的不同！

```
function fn(x: boolean): void
function fn(x: string): void
function fn(x: boolean | string) { // 此处只写一个 boolean 或者 string ，老师那里报错，学生们这里正常运行！
    console.log(typeof x)
    console.log(x)
}
// 测试
fn("hello")
fn(true)
```

返回类型不正确*

> 此处存疑！与老师讲的不同！
>
> 返回什么类型的值都可以，我觉得是不合理的，应该手动添加多个重载签名返回值类型的联合类型！

```
// 老师的这里会报错！学生们的不会！此处是按照学生的写的！
function fn(x: boolean): string
function fn(x: string): boolean
function fn(x: boolean) {
    if(Math.random() > 0.5){
        return 1
    } else {
        return "111"
    }
}
console.log(typeof fn("111")) // number （这不是固定的）
console.log(typeof fn(true)) // string （这不是固定的）
```

十三、函数重载-编写好的重载

1、准则

在可能的情况下

总是倾向于使用联合类型的参数

而不是重载函数

2、代码演示

出现问题

```
function len(x: string): number
function len(arr: any[]): number
function len(a: any) {
    return a.length
}

len("hello")
len([1, 2, 3])
// 报错了，我们不能传入一个可能是"hello"也可能是数组的参数，这个类型是联合类型 "hello" | number[]
len(Math.random() > 0.5 ? "hello" : [100, 200 ,300])
```

解决问题

```
// 解决方法：将函数重载换成联合类型
function len(x: string | any[]) {
    return a.length
}
len("hello")
len([1, 2, 3])
len(Math.random() > 0.5 ? "hello" : [100, 200 ,300])
```

十四、函数内的 this 声明

1、概述

说明

TypeScript 会通过代码流分析来推断函数中的 this 应该是什么

代码示例

```
const user = {
    id: 123,
    admin: false,
    becomeAdmin: function () {
        // 此处的 this.admin 即指代的是上面的 admin
        this.admin = true
    }
}
// this 作为函数的参数
interface DB {
    filterUsers(filter: (this: User) => boolean): User[]
}
```

2、代码演示

老师的写法

> 此处的 this 完全体现不出来作用！换成其他的也会得到同样的结果！

```
interface User {
    admin: boolean
}
// this 作为函数的参数
interface DB {
    filterUsers(filter: (this: User) => boolean): User[]
}
const db: DB = {
    filterUsers: (filter: (this: User) => boolean) => {
        let user1: User = { admin: true }
        let user2 : User = { admin: true }
        let user3: User = { admin: false }
        return [user1, user2, user3]
    }
}
const admins = db.filterUsers(function(this: User) {
    return this.admin
})
console.log(admins) // [ { admin: true }, { admin: true }, { admin: false } ]
```

我的写法*

> 我没有使用 this，到底 this 怎么使用，还需要进一步探索！

```
interface User {
    admin: boolean,
    age: number
}
interface DB {
    filterUsers(filter: (user: User) => boolean): User[]
}
const db: DB = {
    filterUsers: (filter: (user: User) => boolean) => {
        let user1: User = { admin: true, age: 25 }
        let user2 : User = { admin: true, age: 18 }
        let user3: User = { admin: false, age: 22 }
        let users: User[] = [user1, user2, user3]
        let returnUsers: User[] = [];
        for (let i = 0; i < users.length; i++) {
            if(filter(users[i])) {
                returnUsers.push(users[i])
            }
        }
        return returnUsers
    }
}
const admins = db.filterUsers(function(user: User) {
    return user.admin && user.age > 20
```

```
})

console.log(admins) // [ { admin: true, age: 25 } ]
```

十五、需要了解的其他类型

1、void

概述

表示没有任何返回值的函数的返回值；

代码示例

> 自动推断返回值为 void

```
function go () { return } // 有没有 return 无所谓
```

2、object

概述

object 指的是任何不是基元的值：string、number、bigint、boolean、symbol、null、undefined

不同与 {} 、也不同于 Object

在 JavaScript 中函数值是对象

它们**有属性**

比如在原型链里有 Object.prototype

是 Object 的一个实例

可以调用 Object.key 等等

TypeScript 也有对应，也就是小写的 object

注意！小写的 object 不是 大写的 Object

在 TS 中始终使用小写的 object

3、unknown

概述

unknown 类型代表任何值

这**与 any 类型类似**

但**更安全**

因为对未知 unknown 值做任何使其都是不合法的!

代码示例

```
function fn1(a: any) {
    a.x // 正常
}
function fn1(a: unknown) {
    a.x // 报错：对象的类型为 "unknown"。
}
```

4、never

概述

never 表示永远不会被观察到的值

代码示例

> 返回值为 never 的情况：抛出异常；终止执行；死循环。

```
function fail(msg: string): never {
    throw new Error(msg)
}
```

联合类型无参数的时候

```
function fail(x: string | number) {
    if (typeof x === "string") {
        // do sth
    } else if (typeof x === "number") {
        // do sth
    } else {
        // 此时为 never 类型
    }
}
```

5、Function

概述

全局性的 Function 类型描述了诸如 bind、call、apply 和其他存在于 JavaScript 中所有函数值的属性

他还有一个特殊的属性

即 Function 类型的值总是可以被调用

这些调用返回 any

代码示例

> 返回类型是 any，一般情况下我们避免这么做，因为 any 作为返回类型是不安全的!

```
function doSth(f: Function) {
    return f(1, 2, 3)
}
```

> 如果我们需要接受一个任意的函数，而不打算调用它，可以使用箭头函数的方式定义类型

```
() => void
```

十六、参数展开运算符-形参展开

1、概述

可选参数和函数重载可以让函数接收各种固定数量的参数

现在我们使用展开运算符来定义函数

可以接收无限数量的参数

2、代码示例

> 此处的 m 相当于一个数组，所有参数自第二个起被放进这个数组里面!

```
function multiply(n: number, ...m: number[]) {
    return m.map((x) => n * x)
}
const arr = multiply(10, 1, 2, 3, 4, 5)
```

十七、参数展开运算符-实参展开

1、概述

说明

上一节，我们可以把真实的参数**合并**带形参里

通过展开运算符放到一个变量里

相反，我们也可以将一个数组**展开**到一个调用函数里

代码示例

```
const arr100 = [1, 2, 3]
const arr200 = [4, 5, 6]
arr100.push(...arr200)
```

2、代码演示

代码示例

```
const arr100 = [1, 2, 3]
const arr200 = [4, 5, 6]
arr100.push(...arr200) // 与下效果相同
arr100.push(7, 8, 9, 10) // 与上效果相同
console.log(arr100)
```

执行结果

```
PS D:\MyFile\VSCodeProjects\study-ts\第 5 章 函数\dist> node .\17-spread.js
[
  1, 2, 3, 4,  5,
  6, 7, 8, 9, 10
]
```

3、一个问题

代码分析

```
const arr100 = [8, 9]
const x = Math.atan2(8, 9) // 没问题
const y = Math.atan2(...arr100) // 报错：扩张参数必须具有元组类型或传递给 rest 参数。
// 此处是因为 Math.atan2() 的参数只能是两个，而 arr100 的元素个数是不确定的
```

解决问题

> 关于 `as const` 参考文章：https://blog.csdn.net/weixin_43263355/article/details/120943605

```
const arr100 = [8, 9] as const // 将 8 个 9 断言为常量（文字类型）
const x = Math.atan2(...arr100)
```

十八、参数解构

1、概述

我们可以使用**参数重构**的方法

将对象实参**解压**到函数主体的一个或多个局部变量中

2、代码示例

```
function sum({a, b, c}: {a: number, b: number, c: number}) {
    console.log(a + b + c)
}
sum({ a: 10, b: 20, c: 30 })
```

十九、返回 void 类型

1、概述*

函数的 void 返回类型可以产生一些不寻常的行为

这种行为我们看上去确实预期的

返回类型为 void 上下文类型

并不强迫函数不返回任何东西

一个具有 void 返回类型的上下文函数类型（type vf = () => void）

在实现时，**可以返回**任何其他值，**但它会被忽略 问题：但是实际上可以打印出来返回值！**

当一个字面的函数定义有一个 void 返回类型时

该函数必须不返回任何东西

2、代码演示

> 此时出现的必须注意的问题，我的 f1() f2() f3() 打印出来是有结果的！并非被忽略了！

```
// 情况一
type voidFunc = () => void
// 函数实现写法一
```

```
const f1: voidFunc = () => {
    return "hello"
}
// 函数实现写法二
const f2: voidFunc = () => true
// 函数实现写法三
const f3: voidFunc = function() {
    return 100
}
// 调用：老师讲的返回值会被忽略，但在我这里却可以正常打印！
console.log(f1()) // hello
console.log(f2()) // true
console.log(f3()) // 100
// 情况二
function go(): void {
    return "hello" // 报错：不能将类型“string”分配给类型“void”。
}
```