* 文档

## **TypeScript常用类型**

## **3.1 类型注解**

## **示例代码：**

## **let age : number =18**

## **说明：代码中的 : number 就时类型注解。**

## **作用：为变量添加类型约束。比如：上述代码中，约定变量age的类型为number（数值类型）。**

## **解释：约定了什么类型，就只能给变量赋值该类型的值；否则，就会报错。**



## **3.2 常用基础类型概述**

## **可以将TS中常用基础类型细分为两类：1 JS已有类型 2 TS新增类型**

## **1.JS已有类型**

## **原始类型：number/string/boolean/null/undefinesd/symbol。**

## **对象类型：object(包括，数组、对象、函数等对象)。**

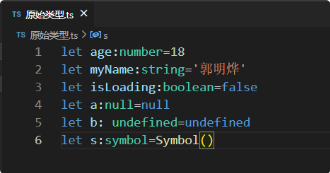
## **2.TS新增类型**

## **联合类型、自定义类型（类型别名）、接口、元组、字面量类型、枚举、void、any等。**

## **3.3 原始类型**

## **1.原始类型：number/string/boolean/null/undefinesd/symbol**

## **特点：简单。这些类型。完全按照JS中类型的名称来书写。**

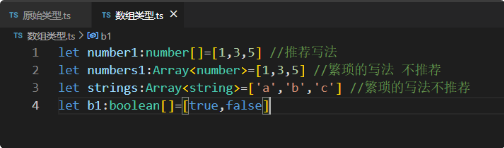


## **3.4 数组类型**

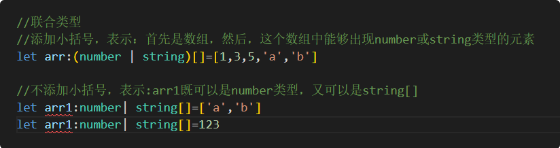
## **2.对象类型：object（包括，数组、对象、函数等对象）。**

## **特点：对象类型，在TS中更加细化，每个具体的对象都有自己的类型语法。**

## **数组类型的两种写法：（推荐使用number[]写法）**



## **需求：数组中既有number类型，又有string类型，这个数组的类型应该怎么写？**



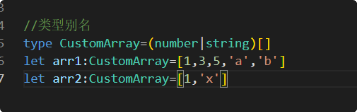
## **解释：|（竖线）在TS中叫做联合类型（由两个或多个其他类型组成的类型，表示可以是这些类型中的任意一种）。**

## **注意：这是TS中联合类型的语法，只有一根竖线，不要和JS中的或（||）混淆了。**

## **3.25类型别名**

## **类型别名（自定义类型） ：为任意类型起别名。**

## **使用场景：当同一类型（复杂）被多次使用时，可以通过类型别名，简化该类型的使用。**



## **解释：**

### **使用type关键字来创建类型别名。**

### **类型别名（比如：此处的CustomArray），可以是任意合法的变量名称。**

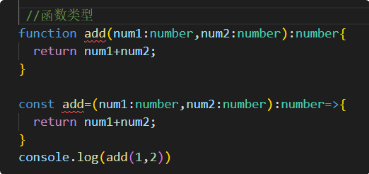
### **创建类型别名后，直接使用该类型别名作为变量的类型注解即可。**

# **3.6 函数类型**

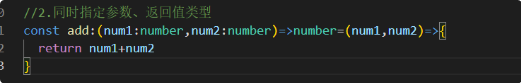
## **函数的类型实际上指的是：函数参数和返回值的类型。**

## **为函数指定类型的两种方式：1 单独指定参数、返回值的类型 2 同时指定参数、返回值的类型。**

## **1.单独指定参数、返回值的类型：**



# **2.同时指定参数、返回值类型：**



## **解释：当函数作为表达式时，可以通过类似箭头函数形式的语法来为函数添加类型**

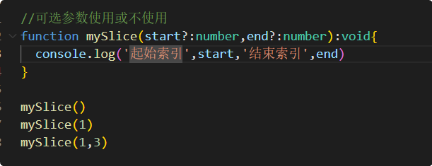
## **注意：这种形式只适用于函数表达式。**

## **如果函数没有返回值，那么，函数返回值类型为：void。**



## **使用函数实现某些功能时，参数可以传也可以不传。这种情况下，再给函数参数指定类型时，就用到可选参数了。**

## **比如，数组的slice方法，可以slice（）也可以slice（1）还可以slice（1，3）。**



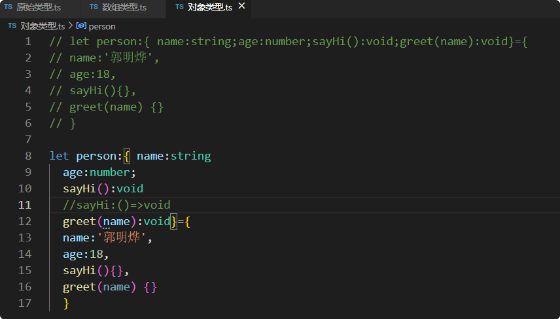
## **可选参数：在可传可不传的参数名称后面添加？（问号）。**

## **注意：可选参数只能出现在参数列表的最后，也就是说可选参数后面不能再出现必选参数。**

# **3.7 对象类型**

## **JS中的对象是由属性和方法构成的，而TS中对象的类型就是在描述对象的结构（有什么类型的属性和方法）。**

## **对象类型的写法：**



## **解释：**

### **直接使用{}来描述对象结构。属性采用属性名：类型的形式；方法采用方法名（）：返回值类型。**

### **如果方法有参数，就在方法名后面的小括号中指定参数类型（比如：greet（name:string）:void）。**

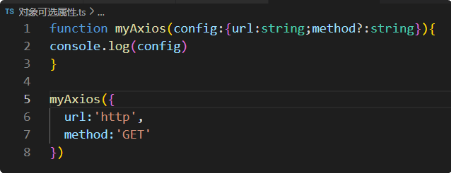
### **在一行代码中只指定一个属性类型（通过换行来分隔多个属性类型），可以去掉；（分号）。**

### **方法的类型也可以使用箭头函数形式（比如：{sayHi:()=>void}。**

## **对象可选属性：**

## **对象的属性和方法，也可以是可选的，此时就用到可选属性了。**

## **比如，我们在使用axios({....))时，如果发送GET请求，method属性就可以省略。**



## **可选属性的语法和函数可选参数的语法一致，都使用?（问号）来表示。**

# **3.8 接口**

## **当一个对象类型被多次使用时，一般会使用接口（interface）来描述对象的类型，达到复用的目的。**

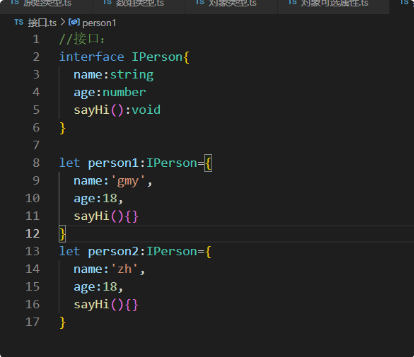
## **解释：**

### **使用interface关键字来声明接口。**

### **接口名称（比如，此处的iPerson），可以是任意合法的变量名称。**

### **声明接口后，直接使用接口名称作为变量的类型。**

### **因为每一行只有一个属性类型，因此，属性类型没有；（分号）。**



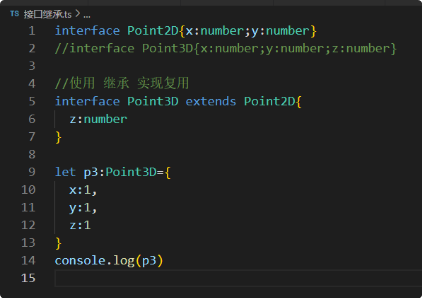
## **interface（接口）和type（类型别名）的对比：**

* **相同点：都可以给对象指定类型。**
* **不同点：1.接口，只能为对象指定类型**

## **2.类型别名，不仅可以为对象指定类型，实际上可以为任意类型指定别名。**

## **如果两个接口之间有相同的属性和方法，可以将公共的属性或方法抽离出来，通过继承来实现服用。**

## **比如，这两个接口都有x,y两个属性，重复写两次，可以，但很繁琐。**



## **解释：**

## **使用extends（继承）关键字实现了接口Point3D继承Point2D。**

## **继承后，Point3D就有了Point2D的所有属性和方法（此时，Point3D同时有x,y,z三个属性）。**

## **3.9 元组**

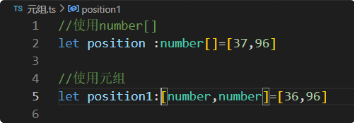
## **场景：在地图中，使用经纬网坐标标记位置信息。**

## **可以用数组来记录坐标，那么，该数组只有两个元素，且这两个元素都是数值类型。**

## **使用numberp[]的缺点：不严谨，因为该类型的数组中可以出现任意多个数字。**

## **更好的方式：元组（tuple）。**

## **元组类型是另一种类型的数组，他确切的知道包含多少个元素，以及特定索引对应的类型。**



## **解释：**

## **元组类型可以确切的标记出有多少元素，以及每个元素的类型。**

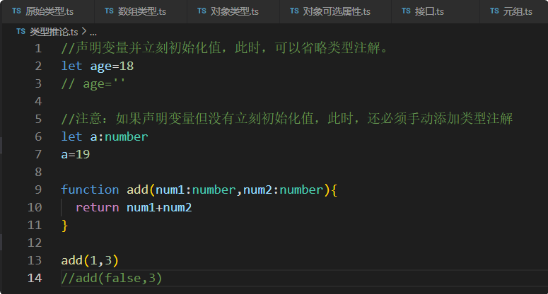
## **该示例中，元素有两个元素，每个元素的类型都是number（有需要可以更改类型，只要输入对应即可）。**

## **3.10 类型推论**

## **在TS中，某些没有明确指出类型的地方，TS的类型推论机制会帮助提供类型。**

## **换句话说：由于有类型推论的存在，这些地方，类型注解可以省略不写！**

## **发生类型推论的两种常见场景：1.声明变量并初始化时 2.决定函数返回值时。**



## **注意：这两种情况下，类型注解可以省略不写！**

## **推荐：能省略类型注解的地方就省略（偷懒，充分利用TS类型推论的能力，提升开发效率）。**

## **技巧：如果不知道类型，可以通过鼠标放在变量名称上，利用VSCODE的提示来查看类型。**

## **3.11 类型断言**

## **使用类型断言：**



## **解释：**

## **使用as关键字实现类型断言。**

## **关键字as后面的类型是一个更加集体的类型（HTMLAnchorElement是HTMLElement的子类型）。**

## **通过类型断言，aLInk的类型变得更加具体，这样就可以访问a标签特有的属性或方法了。**

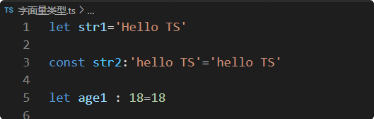
## **另一种语法，使用<>语法，这种语法形式不常用 知道即可：**

### **const aLink =<HTMLAnchorElement>document.getElementById('link')**

## **技巧：在浏览器控制台，通过console.dir()打印DOM元素，在属性列表的最后面，即可看到该元素的类型。**

## **3.12 字面量类型**

## **思考以下代码，两个变量的类型分别是什么？**



## **通过TS类型推论机制，可以得到答案：**

## **变量str1的类型为：string**

## **变量str2的类型为：'hello TS'**

## **解释：**

## **str1是一个变量（let）,它的值可以是任意字符串，所以类型为：strng。**

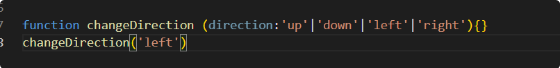
## **str2是一个常量（const），它的值不能变化只能是'hello TS'，所以，他的类型为'hello TS'。**

## **注意：此处的'hello TS',就是一个字面量类型，也就是说某个特定的字符串也可以作为TS中的类型。除字符串以外，任意的JS字面量（比如，对象、数字等）都可以作为类型使用。**

## **使用模式：字面量类型配合联合类型一起使用。**

## **使用场景：用来表示一组明确的可选值列表。**

## **比如：在贪吃蛇游戏中，游戏的方向的可选值只能是上、下、左、右中的任意一个。**



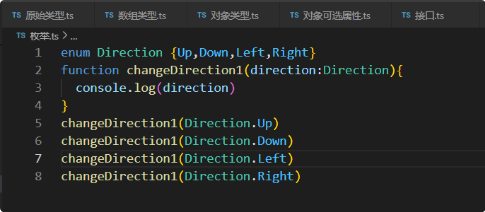
## **解释：参数direction的之恩那个是up/down/left/right中的任意一个。**

## **优势：相比于string类型，使用字面量类型更加精确、严谨。**

## **3.13 枚举**

## **枚举的功能类似于字面量类型+联合类型组合的功能，也可以表示一组明确的可选值。**

## **枚举：定义一组命名常量，他描述一个值，该值可以是这些命名常量中的一个。**



## **解释：**

## **使用enum关键字定义枚举。**

## **约定枚举名称，枚举中的值以大写字母开头。**

## **没居中呃呃多个值之间通过（逗号）分隔。**

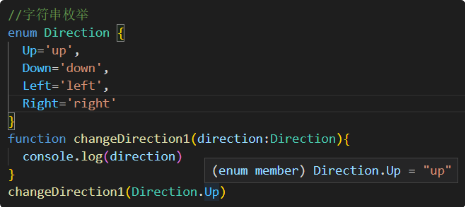
## **定义好枚举后，直接使用枚举名称作为类型注解。**

## **注意：形参direction的类型为枚举Direction，那么，实参的值就应该是枚举DIrection成员中的任意一个。**

## **访问枚举成员：changeDirection(DIrection.Up)**

## **解释：类似于JS中的对象，直接通过点(.)语法访问枚举的成员。**

## **字符串枚举：枚举成员的值是字符串。**

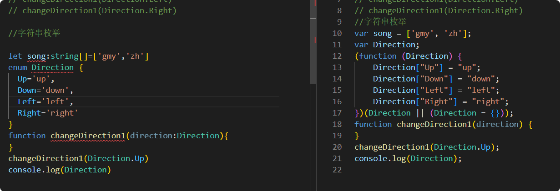


## **注意：字符串枚举没有自增长行为，因此，字符串枚举的每个成员必须有初始值。**

## **枚举是TS为数不多的非Javascript类型级拓展（不仅仅是类型）的特性之一。**

## **因为：其他类型仅仅被当作类型，而枚举不仅用作类型，还提供值（枚举成员都是有值的）。**

## **也就是说，其他的类型会在编译为JS代码时自动移除。但是，枚举类型会被编译为JS代码！**



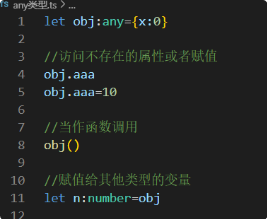
## **说明：枚举和前面讲的字面量类型+联合类型组合的功能类似，都用来表示一组明确的可选值列表。**

## **一般情况下，推荐使用字面量类型+联合类型组合的方式，因为相比枚举，这种方式更加直观、简洁、高效。**

## **3.14 any类型**

## **原则：不推荐使用any! 这会让TypeScript变成“AnyScript”（失去TS类型保护的优势）。**

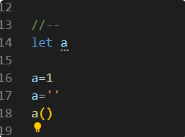
## **因为当值的类型为any时，可以对该值进行任意操作，并且不会有代码提示。**

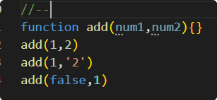


## **解释：以上操作都不会有任何类型错误提示，即使可能存在错误！**

## **尽可能的避免使用any类型，除非临时使用any来‘避免’书写很长、很复杂的类型！**

## **其他隐式具有any类型的情况：1 声明变量不提供类型也不提供默认值 2 函数参数不加类型。**





## **注意：因为不推荐使用any，所以，这两种情况下都应该提供类型！**

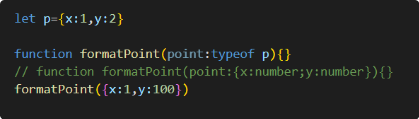
## **3.15 typeof**

## **众所周知，JS中提供了typeof操作符，用来在JS中获取数据的类型。**



## **实际上，TS也提供了typeof操作符：可以在类型上下文中引用变量或属性的类型（类型查询）。**

## **使用场景：根据已有变量的值，获取该值的类型，来简化类型书写。**

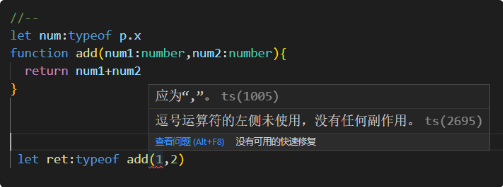


## **解释：**

## **使用typeof操作符来获取变量p的类型，结果与第一种（对象字面量形式的类型）相同。**

## **typeof出现在类型注解的位置（参数名称的冒号后面）所处的环境就在类型上下文（区别于JS代码）。**

## **注意：typeof只能用来查询变量或属性的类型，无法查询其他形式的类型（比如，函数调用的类型）**



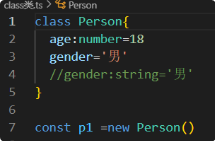
* 文档

**4.TypeScript高级类型**

## **4.1 class类**

## **TypeScript全面支持ES6中引入的class关键字，并为其添加了类型注解和其他语法（比如，可见性修饰符等）。**

## **class基本使用，如下：**



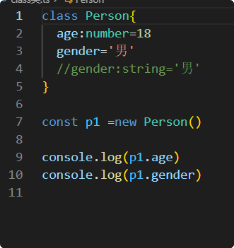


## **解释：**

## **根据TS中的类型推论，可以知道Person类的实例对象p1的类型是Person。**

## **TS中的chass，不仅提供了class的语法功能，也作为一种类型存在。**

## **实例属性初始化：**

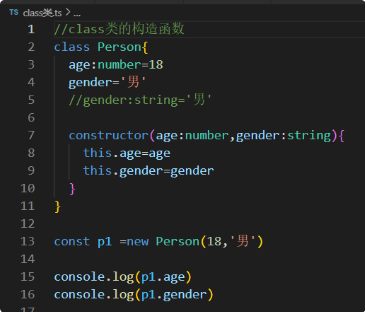


## **解释：**

## **声明成员age，类型为number（初始值为18）。**

## **声明成员gender，并设置初始值，此时，可省略类型注解（TS类型推论为string类型）。**

## **构造函数：**

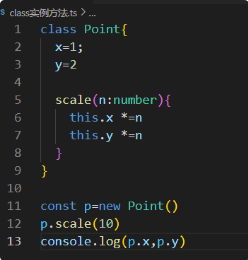


## **解释：**

## **成员初始化（比如，age:number）后，才可以通过this.age来访问实例成员。**

## **需要为构造函数指定类型注解，否则会被隐式推断为any；构造函数不需要返回值类型**

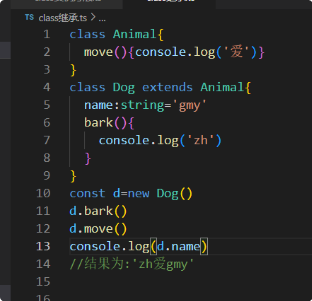
## **实例方法：**



## **解释：方法的类型注解（参数和返回值）与函数用法相同。**

## **类继承的两种方式：1 extends（继承父类） 2 implements（实现接口）。**

## **说明：JS中只有extends，而implements是TS提供的。**

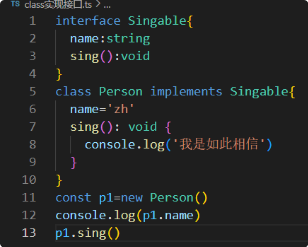


## **解释：**

## **通过extends关键字实现继承。**

## **子类Dog继承父类Animal，则Dog的实例对象d就同时具有了父类animal和子类Dog的所有属性和方法。**

## **implements实现接口：**



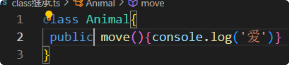
## **解释：**

## **通过implements关键字让class实现接口。**

## **Person类实现接口Singable意味着，Person类中必须提供Singable接口中指定的所有方法和属性。**

## **类成员可见性：可以使用TS来控制class的方法或属性对于class外的代码是否可见。**

## **可见性修饰符包括：1 public（公有的）2 protected（受保护的）3 private（私有的）。**



## **解释：**

## **在类属性或方法前面添加public关键字，来修饰该属性或方法是共有的。**

## **因为public是默认可见性，所以，可以直接省略。**

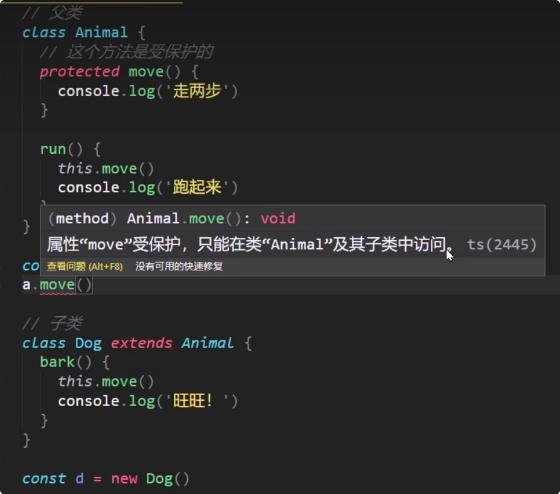
## **2.protected：表示受保护的，仅对其声明所在类和子类中（非实例对象）可见。**



## **解释：**

## **在类属性或方法前面添加protected关键字，来修饰该属性或方法是受保护的。**

## **在子类的方法内部可以通过this来访问父类中受保护的成员，但是，对实例不可见。**



## **3.private：表示私有的，只在当前类中可见，对实例对象以及子类也是不可见的。**



## **解释：**

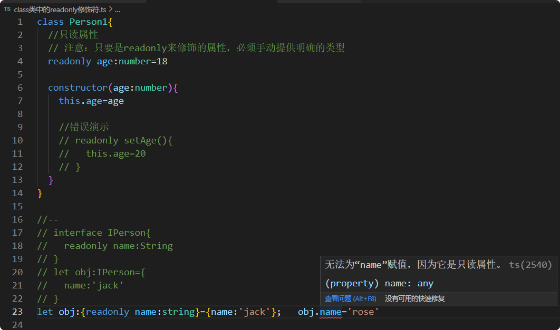
## **在类属性或方法前面添加private关键字，来修饰该属性或方法是私有的。**

## **私有的属性或方法只在当前类中可见，对子类的实例对象也都是不可见的！**

## **class中的readonly修饰符**

## **除了可见性修饰符之外，还有一个常见修饰符就是：readonly（只读修饰符）。**

## **readonly：表示只读，用来防止在构造函数之外对属性进行赋值。**



## **解释：**

## **使用readonly关键字修饰该属性是只读的，注意只能修饰属性不能修饰方法。**

## **注意：属性age后面的类型注解（比如，此处的number）如果不加，则age的类型为18（字面量类型）。**

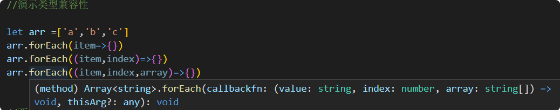
## **接口或者{}表示的对象类型，也可以使用readonly。**

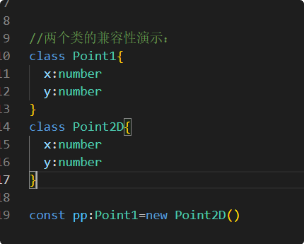
## **4.2 类型兼容性**

## **两种类型系统：1 Structural Type System（结构化类型系统） 2 Nominal Type System（标明类型系统）。**

## **TS采用的是结构化类型系统，也叫duck typing（鸭子类型），类型检查关注的是值所具有的形状。**

## **也就是说，在结构类型系统中，如果两个对象具有相同的形状，则认为他们属于同一类型。**





## **解释：**

## **Point1和Point2D是两个名称不同的类。**

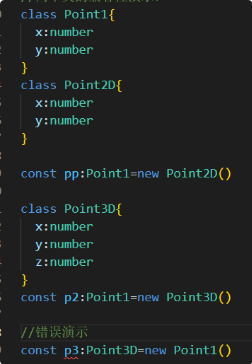
## **变量pp的类型被显示标注为Point1类型，但是，它的值却是Point2D的实例，并且没有类型错误。**

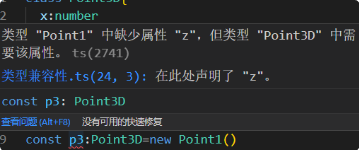
## **因为TS是结构化类型系统，只检查Point1和Point2D的结构是否相同（相同，都具有x和y两个属性，属性类型也相同。）**

## **但是，如果在Nominal Type System（比如，C#、Java等）他们是不同的类，类型无法兼容。**

## **注意：在结构化类型系统中，如果两个对象具有相同的形状，则认为他们属于同一类型，这种说法并不准确。**

## **更准确的说法：对于对象类型来说，y的成员至少与x相同，则x兼容y（成员多的可以赋值给少的）。**





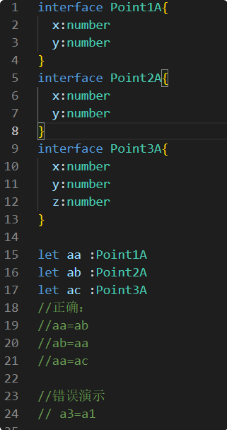
## **解释：**

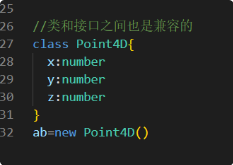
## **Point3D的成员至少与Point1相同，则Point1兼容Point3D.**

## **所以，成员多的Point3D可以赋值给成员少的Point1。**

## **除了class之外，TS中的其他类型也存在相互兼容的情况，包括：1 接口兼容性 2 函数兼容性等。**

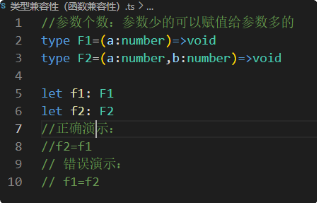
* **接口之间的兼容性，类似于class，并且，class和interface之间也可以兼容。**

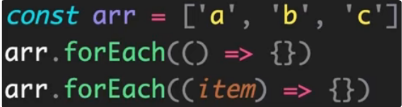




* **函数之间兼容性比较复杂，需要考虑：1 参数个数 2 参数类型 3 返回值类型。**

## **参数个数，参数多的兼容参数少的（或者说，参数少的可以赋值给多的）。**





## **解释：**

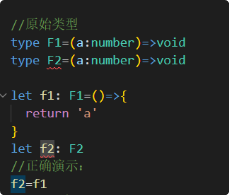
## **参数少的可以赋值给参数多的，所以f1可以赋值给f2。**

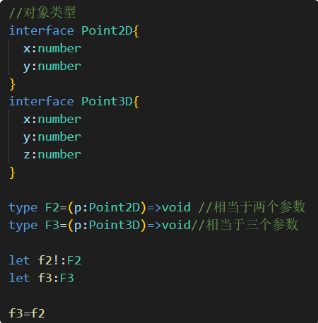
## **数组forEach方法的第一个参数就是回调函数，该示例中类型为：(value:string,index:number,array:string[])=>void。**

## **在JS中省略用不到的函数参数实际上是最常见的，这样的使用方式，促成了TS中函数类型之间的兼容性。**

## **并且因为回调函数是有类型的，所以，TS会自动推导出参数item、index、array的类型。**

## **参数类型，相同位置的参数类型要相同或兼容。**



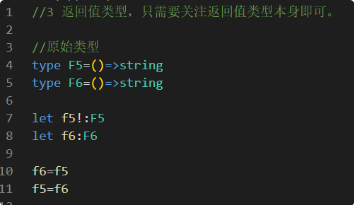


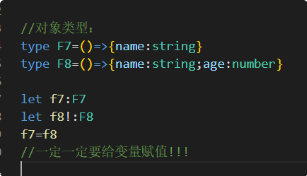
## **解释：**

## **注意，此处与前面讲到的接口兼容性冲突。**

## **技巧：将对象拆开，把每个属性看作一个个参数，则，参数少的（f2）可以赋值给参数多的（f3）。**

## **3.返回值类型，只关注返回值类型即可：**





## **解释：**

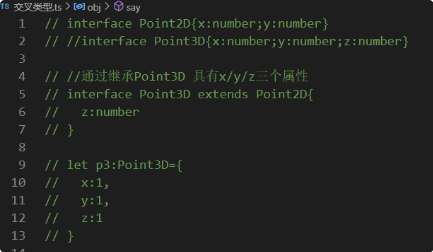
## **如果返回值类型是原始类型，此时两个类型要相同，比如，F5和F6。**

## **如果返回值类型是对象类型，此时成员多的可以赋值给成员少的，比如F7和F8.**

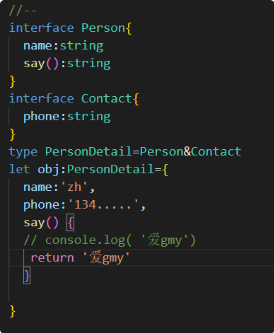
## **4.3 交叉类型**

## **交叉类型（&)：功能类似于接口继承（extends），用于组合多个类型为一个类型（常用于对象类型）。**

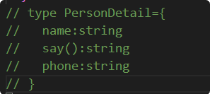
## **接口继承：**



## **比如：**



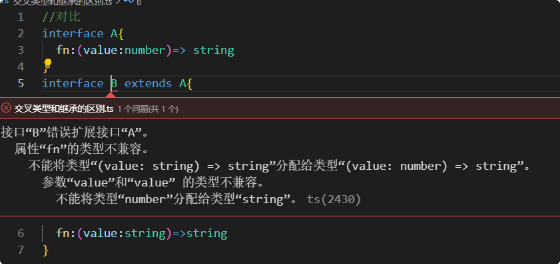
## **解释：使用交叉类型后，新的类型PersonDetail就同时具备了Person和Contact的所有类型属性。相当于：**



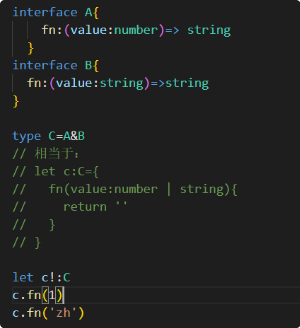
## **交叉类型（&）和接口类型（extends）的对比：**

* **相同点：都可以实现对象类型的组合。**
* **不同点：两种方式实现类型组合时，对于同名属性之间，处理类型冲突的方式不同。**

## **接口类型：**



## **交叉类型：**



## **说明：以上代码，接口继承会报错（类型不兼容）；交叉类型没有错误，可以简单理解为：**



## **4.4 泛型**

## **泛型是可以在保证类型安全的前提下，让函数等与多种类型一起工作，从而实现复用，常用于：函数、接口、class中，**

## **需求：创建一个id函数，传入什么数据就返回该数据本身（也就是说，参数和返回值类型相同）。**

function id(value:number):number{return value}

## **比如，id(10)调用以上函数就会直接返回10本身，但是，该函数只接收数值类型，无法用于其他类型。**

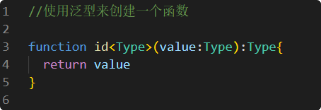
## **为了能让函数接收任意类型，可以将参数类型修改为any，但是，这样就失去了TS的类型保护，类型不安全。**

function　ｉｄ（ｖａｌｕｅ：ａｎｙ）：ａｎｙ｛ｒｅｔｕｒｎ　ｖａｌｕｅ｝

## **泛型在保证类型安全（不丢失类型信息）的同时，可以让函数等与多种不同的类型一起工作，灵活可复用。**

## **实际上，在c#和Java等编程语言中，泛型都是用来实现可复用型组件功能的主要工具之一。**

## **创建泛型函数：**



## **解释：**

## **语法：在函数名称的后面添加<>（尖括号)，尖括号中添加类型变量，比如此处的Type。**

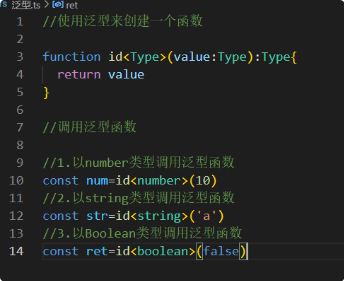
## **类型变量Type，是一种特殊类型的变量，他处理类型而不是值。**

## **该类型变量相当于一个类型容器，能够捕获到用户提供的类型（具体是什么类型由用户调用该函数时指定）。**

## **因为Type是类型，因此可以将其作为函数参数和返回值的类型，表示参数和返回值具有相同的类型。**

## **类型变量Type，可以是任意合法的变量名称。**

## **调用泛型函数：**



## **解释：**

## **语法：在函数名称的后面添加<>（尖括号），尖括号中指定具体的类型，比如：此处的number。**

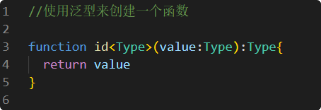
## **当传入属性number后，这个类型就会被函数声明时指定的类型变量Type捕获到。**

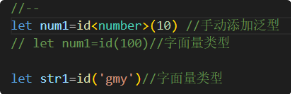
## **此时，Type的类型是number，所以，函数id参数和返回值的类型也都是number。**

## **同时，如果传入类型string，函数id参数和返回值类型就是string。**

## **这样，通过泛型据做到了让id函数与多种不同的类型一起工作，实现了复用的同时保证了类型安全。**

## **简化调用泛型函数：**





## **解释：**

## **在调用泛型函数时，可以省略<类型>来简化泛型函数的调用。**

## **此时，TS内部会采用一种叫做类型参数推断的机制，来根据传入的实参自动推断出类型变量Type的类型。**

## **比如，传入实参10，TS会自动推断出变量number。并作为Type的类型。**

## **推荐：使用这种简化的方式调用泛型函数，使代码更短，更易于阅读。**

## **说明：当编译器无法推断类型或者推断的类型不准确时，就需要显式地传入类型参数。**

## **泛型约束：默认情况下，泛型函数的类型变量Type可以代表多个类型，这导致无法访问任何属性。**

## **比如，id（'a'）调用函数时获取参数的长度：**

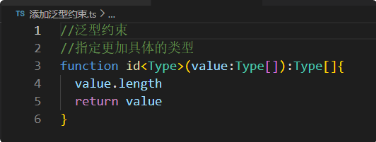
function id<Type>(value:Type):Type{console.log(value.length) ;return value}

## **解释：Type可以代表任意类型，无法保证一定存在length属性，比如number类型就没有length。**

## **此时，就需要为泛型添加约束来收缩类型(缩窄类型取值范围)**

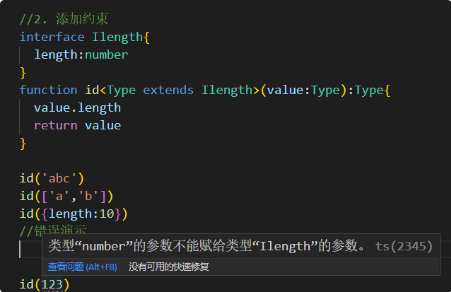
## **添加泛型约束收缩类型，主要有以下两种方式：1 制定更加具体的类型 2 添加约束**

## **指定更加具体的类型：**



## **比如，将约束修改为Type[]（Type类型的数组），因为只要是数组就一定存在length属性，因此就可以访问了。**

## **2.添加约束：**



## **解释：**

## **创建描述约束的接口Ilength,该接口要求提供length属性。**

## **通过extends关键字使用该接口，为泛型（类型变量）添加约束。**

## **该约束表示：传入的类型必须具有length属性。**

## **注意：传入的实参（比如，数组）只要有length属性即可，这也符合前面讲到的接口的类型兼容性。**

## **泛型的类型变量可以有多个，并且类型变量之间还可以约束（比如，第二个类型变量受第一个类型变量约束）。**

## **比如，创建一个函数来获取对象中属性的值：**



## **解释：**

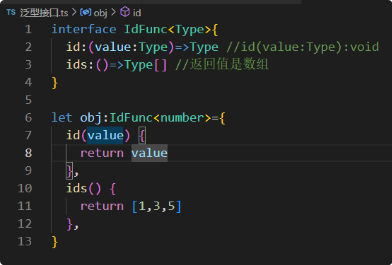
## **添加了第二个类型变量Key，两个类型变量之间使用,(逗号)分隔。**

## **keyof关键字接收一个对象类型，生成其键名称（可能是字符串或数字）的联合类型。**

## **本示例中keyof Type实际上获取的是person对象所有键的联合类型，也就是：'name' | 'age'。**

## **类型变量Key受Type约束，可以理解为：Key之恩那个是Type所邮件中的任意一个，或者说只能访问对象中存在的属性。**

## **泛型接口：接口也可以配合泛型使用，以增加其灵活性，增强其复用性。**



## **解释：**

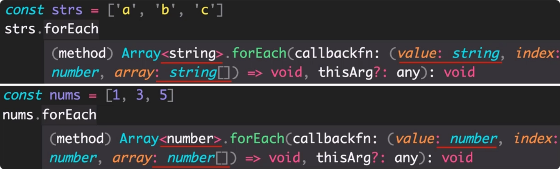
## **在接口名称的后面添加<类型变量>，那么，这个接口就变成了泛型接口。**

## **接口的类型变量，对接口中所有其他成员可见，也就是接口中所有成员都可以使用类型变量。**

## **使用泛型接口时，需要显式指定具体的类型（比如，此处的IdFunc<number>)。**

## **此时，id方法的参数和返回值类型都是number；ids方法的返回值类型是number[]。**

## **实际上，JS中的数组在TS中就是一个泛型接口。**

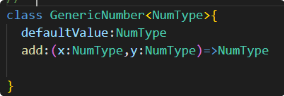


## **解释：当我们在使用数组时，TS会根据数组的不同类型，来自动将类型变量设置为相应的类型。**

## **技巧：可通过Ctrl+鼠标左键来查看具体信息。**



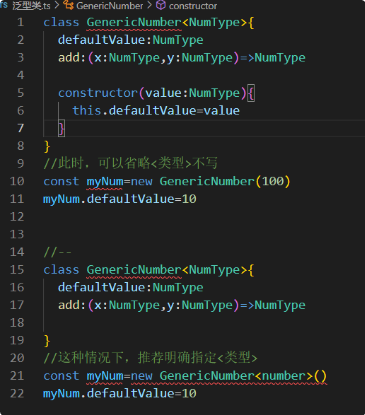
## **创建泛型类：**



## **解释：**

## **类似于泛型接口，在class名称后面添加<类型变量>，这个类就变成了泛型类。**

## **此处的add方法，采用的是箭头函数形式的类型书写方法。**



## **类似于泛型接口，在创建class实例时，在类名后面通过<类型>来指定明确的类型。**

## **泛型工具类型：TS内置了一些常用的工具类型，来简化TS中的一些常见操作。**

## **说明：他们都是基于泛型实现的（泛型适用于多种类型，更加通用），并且是内置的，可以直接在代码中使用。**

## **这些工具类型有很多，主要学习以下几个：**

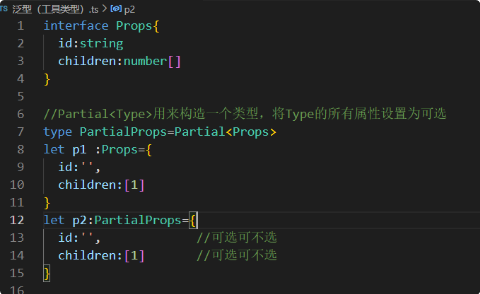
## **Partial<Type>**

## **Readonly<Type>**

## **Pick<Type,Keys>**

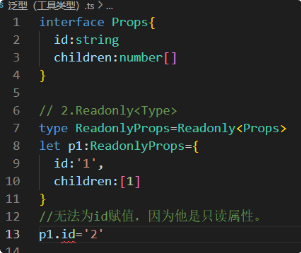
## **Record<Keys,Type>**

## **1.泛型工具类型：Partial<Type>用来构造（创建）一个类型，将Type的所有属性设置为可选。**



## **解释：构造出来的新类型PartialProps结构和Props相同，但所有类型都变为可选的。**

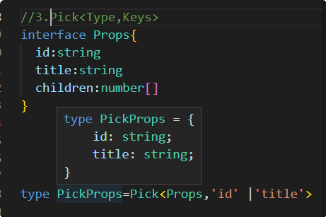
## **2.泛型工具类型：Readonly<Type>用来构造一个类型，将Type的所有属性都设置为readonly（只读）。**



## **解释：构造出来的新类型ReadonlyProps结构和Props相同，但所有属性都变为只读的**

## **当我们想重新给id属性赋值时，就会报错：无法为id赋值，因为他是只读属性。**

## **3.泛型工具类型：Pick<Type,Keys>从Type中选择一组属性来构造新类型。**



## **解释：**

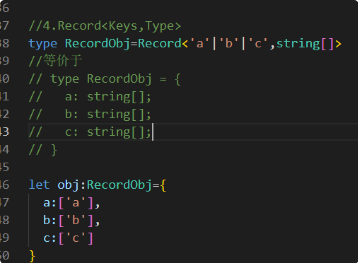
## **Pick工具类型有两个类型变量：1.表示选择谁的属性 2 表示选择哪几个属性。**

## **其中第二个属性变量，如果只选择一个则只传入该属性名即可。**

## **第二个类型变量传入的属性只能是第一个类型变量中存在的属性。**

## **构造出来的新类型PickProps，只有id和title两个属性类型。**

## **4.泛型工具类型：Record<Keys,Type>构造一个对象类型，属性键为Keys，属性类型为Type。**



## **解释：**

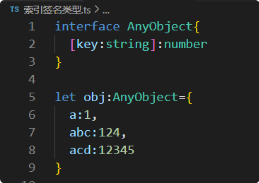
## **Record工具类型有两个类型变量：1 表示对象有哪些属性 2 表示对象属性的类型。**

## **构建的新对象类型RecordObj表示：这个对象又三个属性分别为a/b/c，属性值的类型都是string[]。**

## **4.5 索引签名类型**

## **绝大多数情况下，我们都可以在使用对象前就确定对象的结构，并为对象添加准确的类型。**

## **使用场景：当无法确定对象中有哪些属性（或者说对象中可以出现任意多个属性），此时，就用到索引签名类型了。**



## **解释：**

## **使用[key:string]来约束该接口中允许出现的属性名称。表示只要是string类型的属性名称，都可以出现在对象中。**

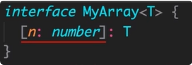
## **这样，对象obj中就可以出现任意多个属性（比如，a,b等）。**

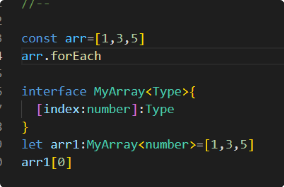
## **key只是一个占位符，可以换成任意合法的变量名称。**

## **隐藏的前置知识：JS中对象（{}）的键是string类型的。**

## **在JS数组中是一类特殊的对象，特殊在数组的键（索引）是数值类型。**

## **并且，数组也可以出现任意多个元素。所以，在数组对应的泛型接口中国，也用到了索引签名类型。**





## **解释：**

## **MyArray接口模拟原生的数组接口，并使用[n:number]来作为索引签名类型。**

## **该索引签名类型表示：只要是number类型的键（索引）都可以出现在数组中，或者说数组中可以有任意多个元素。**

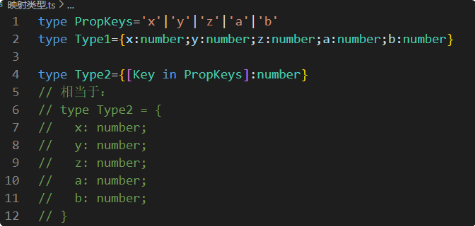
## **同时也符合数组索引是number类型这一前提。**

## **4.6 映射类型**

## **映射类型：基于旧类型创建新类型（对象类型），减少重复，提升开发效率。**

## **比如：类型PropKeys有x/y/z，另一个类型Type1中也有x/y/z，并且Type1中x/y/z的类型相同；**

## **这样书写没错，但x/y/z重复书写了两次。像这种情况，就可以使用映射类型来进行简化。**



## **解释：**

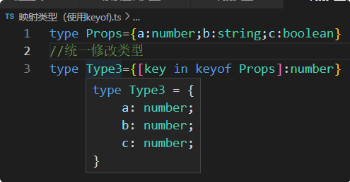
## **映射类型是基于索引签名类型的，所以，该语法类似于索引签名类型，也使用了[]。**

## **Key in PropKeys表示Key可以是PropKeys联合类型中的任意一个，类似于forin（let k in obj）。**

## **使用映射类型创建的新对象类型Type2和类型Type1结构完全相同。**

## **注意：映射类型只能在类型别名中使用，不能在接口中使用。、**

## **映射类型除了根据联合类型创建新类型外，还可以根据对象类型来创建：**

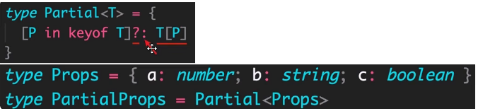


## **解释：**

## **首先，先执行keyof Props获取到对象类型Props中所有键的联合类型即，'a'|'b'|'c'。**

## **然后，Key in ...就表示Key可以是Props中所有键名称中的任意一个。**

## **实际上，前面讲到的泛型工具类型（比如，Partial<Type>）都是基于映射类型实现的。**



## **解释：**

## **keyof T 即keyof Props 表示获取Props的所有键，也就是：'a'|'b'|'c'。**

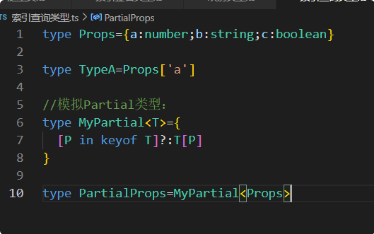
## **在[]后面添加?（问号），表示将这些属性变为可选的，以此来实现Partial的功能。**

## **冒号后面的T[P]表示获取T中每个键对应的类型。比如，如果是'a'，则类型是number；如果是'b'，则类型是string。**

## **最终，新类型PartialProps和旧类型Props结构完全相同，只是让所有类型变得可选了。**

## **刚刚用到的T[P]语法，在TS中叫做索引查询（访问）类型。**

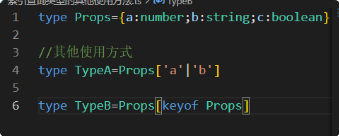
## **作用：用来查询属性的类型。**



## **解释：Props['a']表示查询类型Props中属性'a'对应的类型number，所以，TypeA的类型为number。**

## **注意，[]中的属性必须存在于被查询属性中，否则就会报错。**

## **索引查询类型的其他使用方式：同时查询多个索引的类型。**



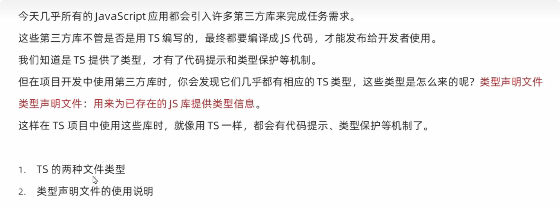
## **解释：使用字符串字面量的联合类型，获取属性a和b对应的类型，结果为string | number。**

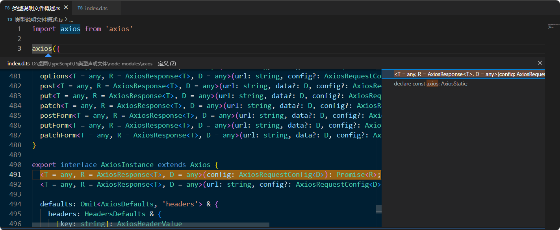
## **使用keyof操作符获取Props中所有键对应的类型，结果为：string | number | boolean。**

* 文档

**5.TypeScript类型声明文件**

## **概述：**





## **TS中有两种文件类型：1 .ts文件 2 .d.ts文件。**

* **.ts文件：**

## **既包含类型信息又可执行代码。**

## **可以被编译为.js文件，然后，执行代码。**

## **用途：编写程序代码的地方。**

* **.d.ts文件：**

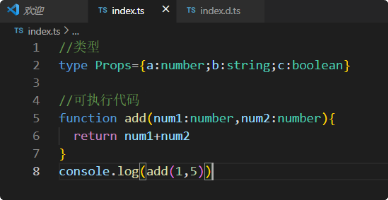
## **只包含类型信息的类型声明文件。**

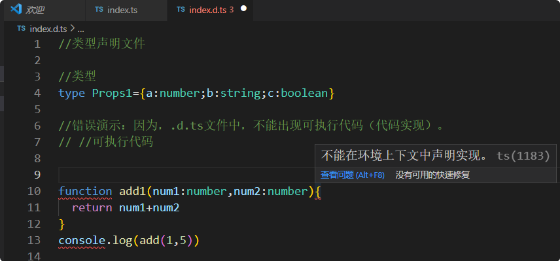
## **不会生成.js文件，仅用于提供类型信息。**

## **用途：为JS提供类型信息。**

## **总结：.ts是implementation（代码实现文件）；.d.ts文件时declaration（类型声明文件）。**

## **然后要为JS库提供类型信息，要使用.d.ts文件。**





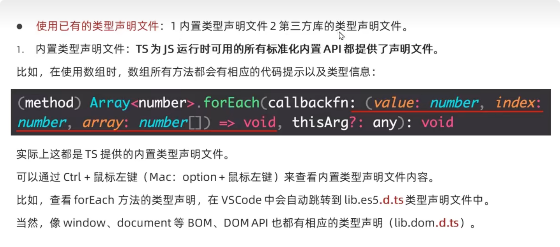
## **5.2 类型声明文件的使用说明**

## **在使用TS开发项目时。类型声明文件的使用包括以下两种方式：**

## **使用已有的类型声明文件**

## **创建自己的类型声明文件**

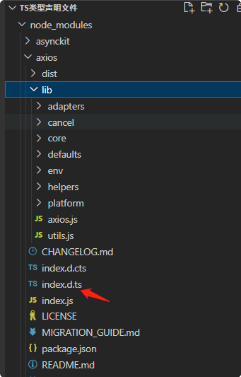
## **学习顺序：先会用（别人的）再会写（自己的）。**



## **2.第三方库的类型声明文件：目前，几乎所有常用的第三方库都有相应的类型声明文件。**

## **第三方库的类型声明文件有两种存在形式：1 库自带类型声明文件 2 Definitely Typed提供。**

## **1.库自带类型说明文件：比如，axios。**



## **解释:这种情况下，正常导入该库，TS就会自动加载库自己的类型声明文件，以提供该库的类型声明。**

## **2.由Definitely Typed提供。**

## **Definitely Typed是一个GitHub仓库，用来提供高质量TypeScript类型声明。<https://github.com/DefinitelyTyped/DefinitelyTyped>**

## **可以通过npm/yarn来下载仓库提供的TS类型声明包，这些包的名称格式为：@types/\*。**

## **比如，@types/react、@types/lodash等。**

## **说明在实际项目开发时，如果你使用的第三方库没有自带的声明文件，VScode会给出明确的提示。**



## **使用'npm i 包'下载包，然后使用明确提示中的提示下载:'npm i -D @types/lodash。**

## **解释：当安装@types/\*类型声明包后，TS也会自动加载该类声明包，以提供该库的类型声明。**

## **补充：TS官方文档提供了一个页面，可以来查询@types/\*库。这个页面已经没有了，官网提示npm已经包含了提示，不再被需要了**

* **创建自己的类型声明文件：1 项目内共享类型 2 已有JS文件提供类型声明。**

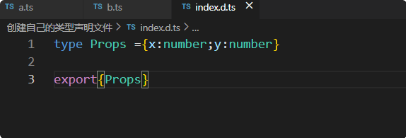
## **1.项目内共享类型：如果多个.ts文件中都用到同一个类型，此时可以创建.d.ts文件提供该类型，实现类型共享。**

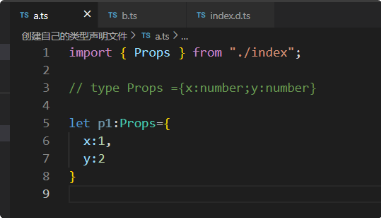
## **操作步骤：**

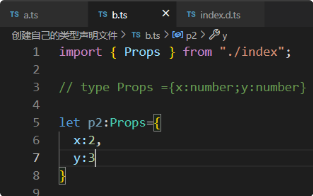
## **创建index.d.ts类型声明文件。**

## **创建需要共享的类型，并使用export导出（TS中的类型也可以使用import/export实现模块化功能）。**

## **在需要使用共享类型的.ts文件中，通过import导入即可（.d.ts后缀导入时，直接省略）。**







## **5.2 类型声明文件的使用说明**

* **创建自己的类型声明文件：1 项目内共享类型 2 已有JS文件提供类型声明。**

## **2.为已有JS文件提供类型声明：**

## **在将JS项目迁移到TS项目时，为了让已有的.js文件有类型声明。**

## **成为库作者，创建库给其他人使用。**

## **注意：类型声明文件的编写与模块化方式相关，不同的模块化方式有不同的写法。但由于历史原因，JS模块化的发展经历过多种变化（AMD、CommonJS、UMD、ESModule等），而TS支持各种模块化形式的类型声明。这就导致类型声明文件相关内容又多又杂。**

## **演示：基于最新的ESModule（import/export）来为已有.js文件，创建类型声明文件。**

## **开发环境准备：使用webpack搭建，通过ts-loader处理.ts文件。**

## **2.为已有JS文件提供类型声明：**

## **说明：TS项目也可以使用.js文件。**

## **说明：在导入.js文件时，TS会自动加载与.js同名的.d.ts文件，以提供类型声明。**

## **declare关键字：用于类型声明，为其它地方（比如.js文件）已存在的变量声明类型，而不是创建一个新的变量。**

## **对于type、interface等这些明确的TS类型的（只能在TS中使用的），可以省略declare关键字。**

## **对于let、function等具有双重含义（在TS、JS中都能用），应该使用declare关键字，明确指定此处用于类型声明。**

