## 类型

##### 原始类型：string、boolean、number、bigint、symbol、null和undefined 其他：enum、数组类型(array)、元组类型(tuple)、 any、unknown、void、never

##### number和bigint不兼容；

null 和 undefined 是所有类型的子类型，可以把 null 和 undefined 赋值给其他类型。

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

## 函数重载

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

当TypeScript 编译器开启 noImplicitAny (在表达式和声明上有隐含的any类型时报错)的 配置项时，将不会允许这种函数重载，需要为函数进行类型限制。

## 类型

Array：

**string[],(string|number)[],Array<string|number>**

函数：

**interface FuncType {**

**(x:string,y?:boolean): boolean;**

**}**

Tuple(元组)：

数组一般由同种类型的值组成，有时需要在单个变量中存储不同类型 的 值，这时候就可以使用元组。

**let** x: [string, number?];

**const** point: readonly [number, number] = [10, 20];

point为只读元组，任何企图修改元组中元素的操作都会抛出异常。

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，只能为它赋 予null和undefined（在strictNullChecks未指定为true时）。

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。

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

（2）函数中执行无限循环的代码（死循环）

any：

任何类型都可以被归为 any 类型，是ts类型系统的顶级类型。

unknown：

unknown与any一样，所有类型都可以分配给unknown。unknown与any的最大区别 是：任何类型的值可以赋值给any，同时any类型的值也可以赋值给任何类型。 unknown 任何类型的值都可以赋值给它，但它只能赋值给unknown和any。

Number、String、Boolean、Symbol与number、string、boolean、symbol ：

前者是原始类型的包装对象（对象类型），后者是原始类型。

原始类型兼容对应的对象类型，反过来对象类型不兼容对应的原始类型。

**let** num: number;

**let** Num: Number;

Num = num; *// ok*

num = Num; *// ts(2322)报错*

object、Object 和 {}：

object代表的是所有非原始类型（原始类型：string、boolean、number、bigint、symbol、 null和undefined）。

Object代表所有拥有toString、hasOwnProperty方法的类型（一些原始类型数据， 例如string，number等被调用方法时会暂时变为封装对象共调用，但null与 undefined不行）。

{}空对象类型和Object基本一致。

联合类型

语法：

type SomeType = string | number;

交叉类型

语法：

type Useless = string & number;

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

注意点：

1. 同名属性的类型如果在多个type中的类型不一致，交集取到never
2. 同名属性的类型兼容，比如一个是 number，另一个是 number 的子类 型、数字字面量类型，合并后该属性的类型就是两者中的子类型。

字面量类型：

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

**let** specifiedStr: 'this is string' = 'this is string';

**let** specifiedNum: 1 = 1;

**let** specifiedBoolean: true = true;

等价于（const关键字声明变量没有必要规定类型）：

**const** specifiedStr = 'this is string';

**const** specifiedNum = 1;

**const** specifiedBoolean = true;

通过联合类型与字面量类型相配合，可以严格控制函数入参：

type Direction = 'up' | 'down';

**function** **move**(dir: Direction) { *// ...* }

move('up'); *// ok*

move('right'); *// error*

## 类型推断：

ts中基于赋值表达式推断类型的能力被称之为类型推断，推断的类型等价于手动的规定 类型。

## 类型断言

由于ts的静态类型对某些运行逻辑后产生的数据无能为力，因此一些情况下需要对这 些数据进行类型断言。

（1）语法：

*// 尖括号 语法*

**let** someValue: any = "this is a string";

**let** strLength: number = (<string>someValue).length;

*// as 语法*

**let** someValue: any = "this is a string";

**let** strLength: number = (someValue **as** string).length;

1. 非空断言：

x! 将从 x 值域中排除null和undefined。

若x的类型只规定了null或undefined， 那么排除null和undefined后x的类型为 never。

1. 确定赋值断言

变量在声明后赋值（非null 和非undefined）前被使用会报错，可以使用x!进行声 明，确保这个变量一定有值，避免报错。

**let** x!: number;

initialize();

console.log(2 \* x); *// Ok*

**function** **initialize**() { x = 10 }

1. const 可作为断言，使用 const 断言时，TypeScript 将为它推断出最窄的类型，没 有拓宽。

**// Type is number[]**

**const arr1 = [1, 2, 3];**

**// Type is readonly [1, 2, 3]**

**const arr2 = [1, 2, 3] as const;**

## 接口

接口（Interfaces）用来定义对象的类型，是对行为的抽象，而具体如何行动需要由 类（classes）去实现（implement）。

interface Person {

readonly name: string;

age?: number; // 可选属性

[propName: string]: any; // 任意属性，

}

**let** tom: Person = {

name: 'Tom',

age: 25

};

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。

鸭式辨型法：

实际上是检测对象是否拥有接口定义的所有方法。

类型检查发生在赋值的时候，也即：当函数参数明确规定了类型，可以通过先对 变量进行赋值，然后以该变量为入参（只要包含函数参数规定的类型即可），就 可以避开类型检查。

类型断言：

类型断言的意义是对变量进行类型的自主定义，绕开系统检查。

interface Props {

name: string;

age: number;

money?: number;

}

**let** p: Props = {

name: "兔神",

age: 25,

money: -100000,

girl: false

} **as** Props; *// OK*

类型断言的前提是：被断言的变量必须包含有断言的类型所要求的属性，可多不 可少。

索引签名：

可以使用任意属性来拓宽变量的类型要求。

interface Props {

name: string;

age: number;

money?: number;

[key: string]: any;

}

**let** p: Props = {

name: "兔神",

age: 25,

money: -100000,

girl: false

}; *// OK*

## 接口与类型别名的区别

在大多数的情况下使用接口类型和类型别名的效果等价，但也有不同之处：

TypeScript 的核心原则之一是对值所具有的结构进行类型检查，而接口的作用就是为 这些类型命名和为你的代码或第三方代码定义数据模型。

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

接口一般用于定义**函数和对象**的类型，接口可以定义多次，重复定义合并，类型别名则 不可以。

接口扩展接口：

interface PointX {

x: number

}

interface Point **extends** PointX {

y: number

}

类型扩展类型：

type PointX = {

x: number

}

type Point = PointX | {

y: number

}

接口扩展类型：

type PointX = {

x: number

}

interface Point **extends** PointX {

y: number

}

类型别名扩展接口：

interface PointX {

x: number

}

type Point = PointX & {

y: number

}

## 泛型

使得函数能够支持当前的数据类型，同时也能支持未来的数据类型。

**function** **identity**<**T**>(arg: T): **T** {

**return** arg;

}

泛型T是一个抽象类型，只有在调用的时候才确定它的值。

泛型约束：  
 T在理论上可以是任何类型，但它不同于any，它是一中不确定的类型。

**function** **trace**<**T**>(arg: T): **T** {

console.log(arg.size); *// Error: Property 'size doesn't exist on type 'T'* **return** arg;

}

=>

interface Sizeable { size: number; }

**function** **trace**<**T** **extends** **Sizeable**>(arg: T): **T** {

console.log(arg.size);

**return** arg;

}

## 类