1. 类型介绍

1. 类型是什么?

类型指的是一组具有相同特征的值，如果两个值具有某种共同的特征，就可以说，它们属于同一种类型。

一旦确定某个值的类型，就意味着这个值具有该类型的所有特征，可以进行该类型的所有运算。凡是适用于该类型的地方，都可以用这个值。凡是不是适用该类型的地方，适用这个值都会报错。

2. 如何理解类型

类型其实是人为添加的一种编程约束和用法提示。

3. 目的

在软件开发过程中，为编译器和开发工具提供更多的验证和帮助，从而提高代码质量，减少错误。

ts在开发阶段如果错误会报错，这样有利于提前发现错误，从而避免。另外如果函数中加入类型，具有提示作用，可以告诉开发者如果使用这个函数。

4. 动态类型和静态类型

JS本身就有自己的类型系统，比如数值、字符串，但是JS的类型系统非常弱，没有限制，运算符可以接受各种类型的值，在语法上，JS属于动态类型语言。

因为没有限制，比如先定义一个数值变量，然后再将其修改为字符串，这样也会成功，不容易发现代码错误。

而ts则引入了更强大，更严格的类型系统，属于静态类型语言。就是在定义变量的时候也随之定义其类型，这个类型定义完，就不能再进行修改了。

5. 静态类型优点

- 有利于代码的静态分析

就是不必运行代码，就能确定变量的类型，从而推断出代码有没有错误，这就叫做代码的静态分析。

- 有利于发现错误

每个值、每个变量、运算符都有严格的类型约束，一旦写错，很容易能发现问题。

- IDE支持，语法提示和自动补全
- 提供了代码文档

通过类型就可以代替部分代码文档，通过类型，就可以大致推断代码的作用

- 有利于重构

因为项目越大，代码之间的关联性越大，修改的时候就能越小心。

而有了类型信息后，比如函数，对象的参数，返回值类型保持不变，那就基本技术重构后也能正常运行，不会报错。

6. 静态类型的缺点

- 丧失了动态类型代码的灵活性
- 增加了编程工作量

得编写定义的类型

- 更高的学习成本

类型系统通常比较复杂

- 引入了独立的编译步骤

原生JS代码可以直接在JS引擎运行，而添加类型系统后，就多出了一个独立的编译步骤，要检查类型是否正确，然后再将ts转为js才能运行代码。

- 兼容性

ts依赖js生态，需要用到很多外部模块，但是很多过去js项目没有适配ts。

总的来说这些缺点对于小型的，短期的个人项目并不友好。

2. 基本数据类型

JavaScript 语言（注意，不是 TypeScript）将值分成8种类型。

- boolean
- string
- number
- bigint
- symbol
- object
- undefined
- null

TypeScript 继承了 JavaScript 的类型设计，以上8种类型可以看作 TypeScript 的基本类型。

注意，所有类型的名称都是小写字母，首字母大写的`Number`、`String`、`Boolean`等在 JavaScript 语言中都是内置对象，而不是类型名称。

另外，undefined 和 null 既可以作为值，也可以作为类型，取决于在哪里使用它们。

这8种基本类型是 TypeScript 类型系统的基础，复杂类型由它们组合而成。

1. boolean类型

boolean类型只包含true和false两个布尔值。

```
const x:boolean = true;
const y:boolean = false;
```

2. string类型

包含所有的字符串

```
const x:string = 'hello';
const y:string = `${x} world`;
```

3. number类型

包含所有整数和浮点数。

```
const x:number = 123;
const y:number = 3.14;
const z:number = 0xffff;
```

4. bigint类型

包含最大的大整数,如果被赋值为其他值会报错

```
const x:bigint = 123n;
const y:bigint = 0xffffn;
```

5. symbo类型

包含所有的 Symbol 值。

```
const x:symbol = Symbol();

```

上面，`Symbol()`函数的返回值就是 symbol 类型。

6. object类型

根据 JavaScript 的设计，object 类型包含了所有对象、数组和函数。

```
const x:object = { foo: 123 };
const y:object = [1, 2, 3];
const z:object = (n:number) => n + 1;
```

7. undefined 类型，null 类型

介绍

undefined 和 null 是两种独立类型，它们各自都只有一个值。

undefined 类型只包含一个值`undefined`，表示未定义（即还未给出定义，以后可能会有定义）。

```
let x:undefined = undefined;

```

上面示例中，变量`x`就属于 undefined 类型。两个`undefined`里面，第一个是类型，第二个是值。

null 类型也只包含一个值`null`，表示为空（即此处没有值）。

```
const x:null = null;
```

上面示例中，变量`x`就属于 null 类型。

注意，如果没有声明类型的变量，被赋值为`undefined`或`null`。

在关闭编译设置`noImplicitAny`和`strictNullChecks`时，它们的类型会被推断为`any`。

打开编译设置`strictNullChecks`以后，赋值为`undefined`的变量会被推断为`undefined`类型，赋值为`null`的变量会被推断为`null`类型，而不会被推断为any类型。

特殊性

`undefined`和`null`既是值，又是类型。

作为值，它们有一个特殊的地方：**任何其他类型的变量都可以赋值为`undefined`或`null`**。这样可能造成在编译时不会报错，运行时会报错。

```
let age:number = 24;

age = null;      // 正确
age = undefined; // 正确

const obj:object = undefined;
obj.toString() // 编译不报错，运行就报错
```

TypeScript 提供了一个编译选项`strictNullChecks`。只要打开这个选项，`undefined`和`null`就不能赋值给其他类型的变量（除了自身、`any`类型和`unknown`类型）。

```
// tsc --strictNullChecks app.ts

let age:number = 24;

age = null;      // 报错
age = undefined; // 报错
```

```
let x:any     = undefined;
let y:unknown = null;
```

3. 包装对象类型

1. 什么是包装对象

JavaScript 的8种类型之中，`undefined`和`null`其实是两个特殊值，`object`属于复合类型，剩下的五种属于原始类型（primitive value），代表最基本的、不可再分的值。

- boolean
- string
- number
- bigint

- symbol

上面这五种原始类型的值，都有对应的包装对象（wrapper object）。**所谓"包装对象"，指的是这些值在需要时，会自动产生的对象。**

```
'hello'.charAt(1) // 'e'

```

上面示例中，字符串`hello`执行了`charAt()`方法。但是，**在 JavaScript 语言中，只有对象才有方法，原始类型的值本身没有方法。这行代码之所以可以运行，就是因为在调用方法时，字符串会自动转为包装对象，`charAt()`方法其实是定义在包装对象上。**

这样的设计大大方便了字符串处理，省去了将原始类型的值手动转成对象实例的麻烦。

在这5中原始类型中，其中Bigint()和Symbol()无法直接获取它们的包装对象，因为不能当成构造函数。

Boolean()、Number()和String()可以，执行构造函数之后，都可以直接获取原始类型值的包装对象。

```
const s = new String('hello');
typeof s // 'object'
s.charAt(1) // 'e'
```

2. 字面量类型

每个原始类型的值都有包装对象和字面量两种情况

```
'hello' // 字面量
new String('hello') // 包装对象
```

为了区分这两种情况，TypeScript 对五种原始类型分别提供了大写和小写两种类型。

- Boolean 和 boolean
- String 和 string
- Number 和 number
- BigInt 和 bigint
- Symbol 和 symbol

其中，**大写类型同时包含包装对象和字面量两种情况，小写类型只包含字面量，不包含包装对象。**

```
const s1:String = 'hello'; // 正确
const s2:String = new String('hello'); // 正确

const s3:string = 'hello'; // 正确
const s4:string = new String('hello'); // 报错
```

4. Object类型与object类型

1. Object类型

代表js里面的广义对象，就是所有可以转为对象的值，都是Object类型，基本数据类型也是，因为自动转为包装对象。

几乎包括了所有值，其中不包括undefined和null。

以下都对

```
let obj:Object;

obj = true;
obj = 'hi';
obj = 1;
obj = { foo: 123 };
obj = [1, 2];
obj = (a:number) => a + 1;
```

2. object类型

指的是狭义对象，即用字面量表示的对象，包括对象、数组、函数，不包括基本数据类型

注意点：

无论是大写的`Object`类型，还是小写的`object`类型，都只包含 JavaScript 内置对象原生的属性和方法，**用户自定义的属性和方法都不存在于这两个类型之中，否则会报错。**

```
const o1:Object = { foo: 0 };
const o2:object = { foo: 0 };

o1.toString() // 正确
o1.foo // 报错

o2.toString() // 正确
o2.foo // 报错
```

上面的foo属性是自定义的会报错，toString不会报错，只能针对内置的原生属性和方法。

5. 值类型

ts规定，单个值也是一种类型，称为值类型。比如'hello'，则是一个字符串'hello'类型。

```
let x:'hello';

x = 'hello'; // 正确
x = 'world'; // 报错

```

上面中，变量`x`的类型是字符串`hello`，导致它只能赋值为这个字符串，赋值为其他字符串就会报错。

ts推断

1. `const`命令声明的变量不是对象时

ts推断类型时，遇到**const命令声明的变量不是对象时， 如果代码里面没有注明类型，就会推断该变量是值类型。**

```
// x 的类型是 "https"
const x = 'https';

// y 的类型是 string
const y:string = 'https';

```

上面示例中，变量`x`是`const`命令声明的，TypeScript 就会推断它的类型是值`https`，而不是`string`类型。

这样推断是合理的，因为`const`命令声明的变量，一旦声明就不能改变，相当于常量。值类型就意味着不能赋为其他值。

2. const声明的变量是对象时

则不会推断为值类型，而是根据属性来推断的。

```
// x 的类型是 { foo: number }
const x = { foo: 1 }; // 推断属性foo的类型是number
```

3. 特殊点

当ts推断数值值类型时，会出现子类型和父类型。

比如下面这个代码，左侧会看作是数值5类型，而右侧**4+1**则会看作number类型，而数值5类型，是number的子类型，在ts中子类型可以赋值父类型，父类型不能赋值子类型，所以会报错。这里是两种类型。

```
const x:5 = 4 + 1; // 报错
```

子类型可以赋值给父类型，父类型不能赋值给子类型。

假如说真想让父类型赋值为子类型，要用到**类型断言。**下面语句就是告诉编译器，4+1当做值类型来看待，而不是number类型。

```
const x:5 = (4 + 1) as 5; // 正确
```

6. 联合类型

指定是多个类型组成一个新类型，使用符合`\`表示。

联合类型`A|B`表示，任何一个类型只要属于`A`或`B`，就属于联合类型`A|B`。

```
let x:string|number;

x = 123; // 正确
x = 'abc'; // 正确

```

上面中，变量`x`就是联合类型`string|number`，表示它的值既可以是字符串，也可以是数值。

注意点：

默认打开编译选项strictNullChecks后，其他类型的变量就不能赋值为null或undefined，但是使用联合类型，即使该选项打开了，也能使用undefined和null。

缩小类型

因为联合类型是多个类型组合成的一个新类型，但是当业务操作时，要先将类型缩小化，针对某个数据类型，来执行不同的代码。

代码内通过判断类型，来编写不同代码

```
function printId(
  id:number|string
) {
  if (typeof id === 'string') {
    console.log(id.toUpperCase());
  } else {
    console.log(id);
  }
}
```

7. 交叉类型

指的是多个类型组成一个新类型，使用符合`&`表示。

交叉类型`A&B`表示，任何一个类型必须同时属于`A`和`B`，才属于交叉类型`A&B`，即交叉类型同时满足`A`和`B`的特征。和联合类型相反，联合类型是或的关系，这里是且的关系。

如果一个变量交叉类型不存在，ts会认为是never类型

```
let x:number&string;

```

上面示例中，变量`x`同时是数值和字符串，这当然是不可能的，所以 TypeScript 会认为`x`的类型实际是`never`。

应用

- 表示对象的合成。

同时具有foo、bar两个属性。

```
let obj:
  { foo: string } &
  { bar: string };

obj = {
  foo: 'hello',
  bar: 'world'
};
```

- 为对象添加新属性。新增了一个bar属性

```
type A = { foo: number };

type B = A & { bar: number };
```

8. type命令

用来定义一个类型的别名。就是给一个类型起一个别的名称，一个类型可能很长，可以使用一个简短的别名代替。

```
type Age = number;

let age:Age = 55;
```

这里使用Age代替number类型

块级作用域

具有块级作用域，内部定义的别名，不影响外部。

同一作用域不能同名。

表达式

别名支持使用表达式，也可以在定义一个别名时，使用另一个别名，即别名允许嵌套。

```
type World = "world";
type Greeting = `hello ${World}`;
```

9. typeof运算符

1. 介绍

在js中typeof是一个运算符，**操作的是一个值,返回一个字符串，代表该值的数据类型。**

在ts中typeof操作的**也是一个值，返回的是该值的ts类型，只能用在类型运算之中，不能用在值运算。**

也就是说，同一段代码可能存在两种`typeof`运算符，一种用在值相关的 JavaScript 代码部分，另一种用在类型相关的 TypeScript 代码部分。

```
let a = 1;
let b:typeof a; // 类型运算，类型检查

if (typeof a === 'number') { // 值运算
  b = a;
}
```

注意点：

- 参数不能是一个类型

10. 块级类型声明

ts支持块级类型声明，就是将类型声明在代码块（用大括号表示）里面，并且只在当前代码块有效。

```
if (true) {
  type T = number;
  let v:T = 5;
} else {
  type T = string;
  let v:T = 'hello';
}

```

上面示例中，存在两个代码块，其中分别有一个类型`T`的声明。这两个声明都只在自己的代码块内部有效，在代码块外部无效。

11. 类型的兼容

ts的类型存在兼容关系，某些类型可以兼容其他类型。

```
type T = number|string;

let a:number = 1;
let b:T = a;

```

变量`a`和`b`的类型是不一样的，但是变量`a`赋值给变量`b`并不会报错。这时，我们就认为，`b`的类型兼容`a`的类型。

在ts中，如果类型A的值可以赋值给类型B，那么类型A就称为类型B的子类型，其中子类型可以赋值给父类型，父类型不能赋值给A类型。

之所以有这样的规则，是因为**子类型继承了父类型的所有特征**，所以可以用在父类型的场合。但是，子类型可以有自己扩展的特征，就可能有父类型没有该特征，所以**父类型不能用在子类型**。

12. symbol类型

Symbol 是 ES2015 新引入的一种原始类型的值。它类似于字符串，但是每一个 Symbol 值都是独一无二的，与其他任何值都不相等。

Symbol 值通过`Symbol()`函数生成。在 ts 里面，Symbol 的类型使用`symbol`表示。

变量`x`和`y`的类型都是`symbol`，且都用`Symbol()`生成，但是它们是不相等的。

```
let x:symbol = Symbol();
let y:symbol = Symbol();

x === y // false

```

1. unique symbol

symbol类型包含所有的Symbol值，但是不能表示具体的Symbol值，所以在ts中设计了子类型`unique symbol`，用他表示单个的具体的值。

只能用const定义，因为是独一无二的。

const为变量赋值为Symbol值时，默认变量类型为`unique symbol`，因此可以省略不写。

每个声明为`unique symbol`类型的变量，他们的值都不同，其实也属于值类型。

```
const a:unique symbol = Symbol();
const b:unique symbol = Symbol();

a === b // 报错

```

上面示例中，变量`a`和变量`b`的类型虽然都是`unique symbol`，但其实是两个值类型。不同类型的值肯定是不相等的，所以最后一行就报错了。

unique symbol 类型是 symbol 类型的子类型，所以可以将前者赋值给后者，但是反过来就不行。

```
const a:unique symbol = Symbol();

const b:symbol = a; // 正确

const c:unique symbol = b; // 报错
```

也能用作属性名，这可以保证不会跟其他属性名冲突。如果要把某一个特定的 Symbol 值当作属性名，那么它的类型只能是 unique symbol，不能是 symbol。

也可以用作类（class）的属性值，但只能赋值给类的`readonly static`属性。

```
class C {
  static readonly foo:unique symbol = Symbol();
}

```

上面示例中，静态只读属性`foo`的类型就是`unique symbol`。注意，这时`static`和`readonly`两个限定符缺一不可，这是为了保证这个属性是固定不变的。

2. 类型推断

如果变量声明时没有给出类型，TypeScript 会推断某个 Symbol 值变量的类型。

`let`命令声明的变量，推断类型为 symbol。

```
// 类型为 symbol
let x = Symbol();

```

`const`命令声明的变量，推断类型为 unique symbol。

```
// 类型为 unique symbol
const x = Symbol();

```

但是，`const`命令声明的变量时，被赋值为另一个 symbol 类型的变量，则推断类型为 symbol。

```
let x = Symbol();

// 类型为 symbol
const y = x;

```

`let`命令声明的变量，如果赋值为另一个 unique symbol 类型的变量，则推断类型还是 symbol。

```
const x = Symbol(); // 其实是unique symbol类型

// 类型为 symbol
let y = x;

```

13. any、unknown、never类型

1. any类型

any类型表示没有任何限制，该类型的变量可以赋值任意类型的值。一旦使用any类型，实际上ts会关闭这个变量的类型检查，即使有明显的类型问题。

```
let x:any;
x = 1;
x = 'str';
```

应该尽可能避免使用any类型，否则会失去使用ts的初心。

在实际开发中有两种场合可能会用到any类型

- 出于特殊情况，需要关闭某个变量的类型。
- 为了适配老项目，迁移到ts上来。

1. 类型推断问题

如果开发者没有进行指定类型，ts会自己推断类型，假如ts推断不出来的会当成any类型来处理。

编译时提供了一个编译选项noImplicitAny，打开此选项，如果被推断成any类型，会直接报错。

```
$ tsc --noImplicitAny app.ts
```

注意点：

对于let、var声明变量时，**如果不赋值，不添加类型，即使打开了noImplicitAny也不会报错，** 这个要特别记住，const因为声明时必须初始化，所以没有事情。

2. 污染问题

关闭了类型检查，还会造成一个问题就是可能会污染其他变量。他可以赋值给其他类型的变量（因为没有类型检查），就有可能出错，会把错误留到运行时报错。

```
let x:any = 'hello';
let y:number;

y = x; // 不报错

y * 123 // 不报错
y.toFixed() // 不报错
```

从集合论的角度看，`any` 类型可以看成是所有其他类型的全集，包含了一切可能的类型。TypeScript 将这种类型称为“顶层类型”（top type），意为涵盖了所有下层。

2. unknow类型

为了解决any类型会污染其他变量。

它与`any`含义相同，表示类型不确定，可能是任意类型，但是它的使用有一些限制，不像`any`那样自由，可以视为严格版的`any`。

`unknown`跟`any`的相似之处，在于所有类型的值都可以分配给`unknown`类型。

```
let x:unknown;

x = true; // 正确
x = 42; // 正确
x = 'Hello World'; // 正确
```

1. 与any类型不同点

- 首先，`unknown`类型的变量，不能直接赋值给其他类型的变量（除了`any`类型和`unknown`类型）。

```
let v:unknown = 123;

let v1:boolean = v; // 报错
let v2:number = v; // 报错
```

- 不能直接调用`unknown`类型变量的方法和属性。

```
let v1:unknown = { foo: 123 };
v1.foo  // 报错

let v2:unknown = 'hello';
v2.trim() // 报错

let v3:unknown = (n = 0) => n + 1;
v3() // 报错
```

- `unknown`类型变量能够进行的运算是有限的，只能进行比较运算（运算符`==`、`===`、`!=`、`!==`、`||`、`&&`、`?`）、取反运算（运算符`!`）、`typeof`运算符和`instanceof`运算符这几种，其他运算都会报错。

2. 如何使用unknown类型

使用时将类型缩小才能使用，就是缩小类型范围，确保不会出错。

先进行类型判断，再写业务逻辑

```
let a:unknown = 1;

if (typeof a === 'number') {
  let r = a + 10; // 正确
}
```

在集合论上，`unknown`也可以视为所有其他类型（除了`any`）的全集，所以它和`any`一样，也属于 TypeScript 的顶层类型。

3. never类型

为了保持与集合论的对应关系，以及类型运算的完整性，TypeScript 还引入了“空类型”的概念，即该类型为空，不包含任何值。

由于不存在任何属于“空类型”的值，所以该类型被称为`never`，即不可能有这样的值。**如果被赋值会报错**

重要特点是，**可以赋值给任意其他类型。**

为什么`never`类型可以赋值给任意其他类型呢?

这也跟集合论有关，空集是任何集合的子集。TypeScript 就相应规定，任何类型都包含了`never`类型。因此，`never`类型是任何其他类型所共有的，TypeScript 把这种情况称为“底层类型”（bottom type）。

总之，TypeScript 有两个“顶层类型”（`any`和`unknown`），但是“底层类型”只有`never`唯一一个。