

any 类型, unknown 类型, never 类型

1. any 类型

变量类型一旦设为 `any`, TypeScript 实际上会关闭这个变量的类型检查。即使有明显的类型错误, 只要句法正确, 都不会报错。

```
let x:any = 'hello';
x(1);      // 不报错
x.foo = 100; // 不报错
```

应该尽量避免使用 `any` 类型, 否则就失去了使用 TypeScript 的意义。

实际开发中, `any` 类型主要适用以下两个场合。

- (1) 出于特殊原因, 需要关闭某些变量的类型检查, 就可以把该变量的类型设为 `any`。
- (2) 为了适配以前老的 JavaScript 项目, 让代码快速迁移到 TypeScript, 可以把变量类型设为 `any`。有些年代很久的大型 JavaScript 项目, 尤其是别人的代码, 很难为每一行适配正确的类型, 这时你为那些类型复杂的变量加上 `any`, TypeScript 编译时就不会报错。

`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。

1.1. 类型推断问题

没有指定类型、TypeScript 必须自己推断类型的那些变量, 如果无法推断出类型, TypeScript 就会认为该变量的类型是 `any`。**对于那些类型不明显的变量, 一定要显式声明类型, 防止被推断为 `any`。**

1.2. 污染问题

`any` 类型除了关闭类型检查, 还有一个很大的问题, 就是它会“污染”其他变量。它可以赋值给其他任何类型的变量 (因为没有类型检查), 导致其他变量出错。

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

y = x;      // 不报错

y * 123;    // 不报错
y.toFixed() // 不报错
```

污染其他具有正确类型的变量, 把错误留到运行时, 这就是不宜使用 `any` 类型的另一个主要原因。

2. unknown 类型

为了解决 `any` 类型“污染”其他变量的问题，TypeScript 3.0 引入了 `unknown` 类型。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`。

2.1. unknown 和 any 相似之处

`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
```

2.2. unknown 和 any 不同之处

- `unknown` 类型的变量，不能直接赋值给其他类型的变量。
- 不能直接调用 `unknown` 类型变量的方法和属性。
- `unknown` 类型变量能够进行的运算是有限的。

`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）。克服了 `any` 类型“污染”其他变量问题的一大缺点。

```
let v:unknown = 123;

let v1:boolean = v; // Type 'unknown' is not assignable to type 'boolean'.
let v2:number = v;  // Type 'unknown' is not assignable to type 'number'.
let v3:any = v;
let v4:unknown = v;
```

直接调用 `unknown` 类型变量的属性和方法，或者直接当作函数执行，都会报错。

```
let v1:unknown = { foo: 123 };
let a1:any = { foo: 123 };
v1.foo;      // 'v1' is of type 'unknown'.
a1.foo;

let v2:unknown = 'hello';
let a2:any = 'hello';
v2.trim();   // 'v2' is of type 'unknown'.
a2.trim();

let v3:unknown = (n = 0) => n + 1;
let a3:any = (n = 0) => n + 1;
v3();        // 'v3' is of type 'unknown'.
a3.foo;
```

`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。

```
let a:unknown = 1;

a + 1; // 'a' is of type 'unknown'.
a === 1;
```

怎么才能使用 `unknown` 类型变量呢？

只有经过“类型缩小”，`unknown` 类型变量才可以使用。所谓“类型缩小”，就是缩小 `unknown` 变量的类型范围，确保不会出错。

```
let a:unknown = 1;

if (typeof a === 'number') {
  let r = a + 10; // 正确
  console.log("r", r);
}
```

```
let s:unknown = 'hello';

if (typeof s === 'string') {
  s.length; // 正确
}
```

`unknown` 可以看作是更安全的 `any`。一般来说，凡是需要设为 `any` 类型的地方，通常都应该优先考虑设为 `unknown` 类型。

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

`never` 类型是“空类型”，即该类型为空，不包含任何值。

如果一个变量可能有多种类型（即联合类型），通常需要使用分支处理每一种类型。这时，处理所有可能的类型之后，剩余的情况就属于 `never` 类型。

```
function fn(x:string|number) {
  if (typeof x === 'string') {
    // ...
  } else if (typeof x === 'number') {
    // ...
  } else {
```

```
    x; // never 类型
  }
}
```

参数变量 `x` 可能是字符串，也可能是数值，判断了这两种情况后，剩下的最后那个 `else` 分支里面，`x` 就是 `never` 类型了。

`never` 类型的一个可以赋值给任意其他类型。

```
function f():never {
  throw new Error('Error');
}

let v1:number = f(); // 不报错
let v2:string = f(); // 不报错
let v3:boolean = f(); // 不报错
```

函数 `f()` 会抛错，所以返回值类型可以写成 `never`，即不可能返回任何值。各种其他类型的变量都可以赋值为 `f()` 的运行结果（`never` 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，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）。

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