any 类型，unknown 类型，never 类型

本章介绍 TypeScript 的三种特殊类型，它们可以作为学习 TypeScript 类型系统的起点。

any 类型

基本含义

any 类型表示没有任何限制，该类型的变量可以赋予任意类型的值。

typescript

```
let x: any;

x = 1; // 正确
x = "foo"; // 正确
x = true; // 正确
```

上面示例中，变量 `x` 的类型是 `any` ，就可以被赋值为任意类型的值。

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。

typescript

```
let x: any = "hello";

x(1); // 不报错
x.foo = 100; // 不报错
```

上面示例中，变量 `x` 的值是一个字符串，但是把它当作函数调用，或者当作对象读取任意属性，TypeScript 编译时都不报错。原因就是 `x` 的类型是 `any` ，TypeScript 不对其进行类型检查。

由于这个原因，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。

总之，TypeScript 认为，只要开发者使用了 any 类型，就表示开发者想要自己来处理这些代码，所以就不对 any 类型进行任何限制，怎么使用都可以。

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。

类型推断问题

对于开发者没有指定类型、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 。

typescript

```
function add(x, y) {
  return x + y;
}

add(1, [1, 2, 3]); // 不报错
```

上面示例中，函数 add() 的参数变量 x 和 y ，都没有足够的信息，TypeScript 无法推断出它们的类型，就会认为这两个变量和函数返回值的类型都是 any 。以至于后面就不再对函数 add() 进行类型检查了，怎么用都可以。

这显然是很糟糕的情况，所以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 。

TypeScript 提供了一个编译选项 noImplicitAny ，打开该选项，只要推断出 any 类型就会报错。

bash

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

上面命令使用了 noImplicitAny 编译选项进行编译，这时上面的函数 add() 就会报错。

这里有一个特殊情况，即使打开了 `noImplicitAny` ，使用 `let` 和 `var` 命令声明变量，但不赋值也不指定类型，是不会报错的。

```typescript
var x; // 不报错
let y; // 不报错
```

上面示例中，变量 `x` 和 `y` 声明时没有赋值，也没有指定类型，TypeScript 会推断它们的类型为 `any` 。这时即使打开了 `noImplicitAny` ，也不会报错。

```typescript
let x;

x = 123;
x = { foo: "hello" };
```

上面示例中，变量 `x` 的类型推断为 `any` ，但是不报错，可以顺利通过编译。

由于这个原因，建议使用 `let` 和 `var` 声明变量时，如果不赋值，就一定要显式声明类型，否则可能存在安全隐患。

`const` 命令没有这个问题，因为 JavaScript 语言规定 `const` 声明变量时，必须同时进行初始化（赋值）。

```typescript
const x; // 报错
```

上面示例中， `const` 命令声明的 `x` 是不能改变值的，声明时必须同时赋值，否则报错，所以它不存在类型推断为 `any` 的问题。

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。

```typescript
let x: any = "hello";
let y: number;

y = x; // 不报错
```

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

上面示例中，变量 x 的类型是 any ，实际的值是一个字符串。变量 y 的类型是 number ，表示这是一个数值变量，但是它被赋值为 x ，这时并不会报错。然后，变量 y 继续进行各种数值运算，TypeScript 也检查不出错误，问题就这样留到运行时才会暴露。

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。

---

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 。

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。

typescript

```
let x: unknown;

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

上面示例中，变量 x 的类型是 unknown ，可以赋值为各种类型的值。这与 any 的行为一致。

unknown 类型跟 any 类型的不同之处在于，它不能直接使用。主要有以下几个限制。

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）。

typescript

```
let v: unknown = 123;

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

上面示例中，变量 v 是 unknown 类型，赋值给 any 和 unknown 以外类型的变量都会报错，这就避免了污染问题，从而克服了 any 类型的一大缺点。

其次，不能直接调用 unknown 类型变量的方法和属性。

typescript

```
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
```

上面示例中，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。

再次， 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。

typescript

```
let a: unknown = 1;

a + 1; // 报错
a === 1; // 正确
```

上面示例中， unknown 类型的变量 a 进行加法运算会报错，因为这是不允许的运算。但是，进行比较运算就是可以的。

那么，怎么才能使用 unknown 类型变量呢?

答案是只有经过“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。

typescript

```
let a: unknown = 1;

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

上面示例中， `unknown` 类型的变量 `a` 经过 `typeof` 运算以后，能够确定实际类型是 `number` ，就能用于加法运算了。这就是“类型缩小”，即将一个不确定的类型缩小为更明确的类型。

下面是另一个例子。

typescript

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

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

上面示例中，确定变量 `s` 的类型为字符串以后，才能调用它的 `length` 属性。

这样设计的目的是，只有明确 `unknown` 变量的实际类型，才允许使用它，防止像 `any` 那样可以随意乱用，“污染”其他变量。类型缩小以后再使用，就不会报错。

总之， `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。

---

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。

typescript

```
let x: never;
```

上面示例中，变量 `x` 的类型是 `never` ，就不可能赋给它任何值，否则都会报错。

`never` 类型的使用场景，主要是在一些类型运算之中，保证类型运算的完整性，详见后面章节。另外，不可能返回值的函数，返回值的类型就可以写成 `never` ，详见《函数》一章。

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。

typescript

```typescript
function fn(x: string | number) {
  if (typeof x === "string") {
    // ...
  } else if (typeof x === "number") {
    // ...
  } else {
    x; // never 类型
  }
}
```

上面示例中，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` 类型的一个重要特点是，可以赋值给任意其他类型。

typescript

```typescript
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
```

上面示例中，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。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。

最后更新: 2023/8/13 15:25

Previous page
基本用法

Next page
类型系统