介绍

为了让程序有价值，我们需要能够处理最简单的数据单元：数字，字符串，结构体，布尔值等。
TypeScript支持与JavaScript几乎相同的数据类型，此外还提供了实用的枚举类型方便我们使用。

布尔值

最基本的数据类型就是简单的true/false值，在JavaScript和TypeScript里叫做`boolean`（其它语言中也一样）。

```
let isDone: boolean = false;
```

数字

和JavaScript一样，TypeScript里的所有数字都是浮点数。
这些浮点数的类型是`number`。
除了支持十进制和十六进制字面量，TypeScript还支持ECMAScript 2015中引入的二进制和八进制字面量。

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
let binaryLiteral: number = 0b1010;
let octalLiteral: number = 0o744;
```

字符串

JavaScript程序的另一项基本操作是处理网页或服务器端的文本数据。
像其它语言里一样，我们使用`string`表示文本数据类型。
和JavaScript一样，可以使用双引号（"）或单引号（'）表示字符串。

```
let name: string = "bob";
name = "smith";
```

你还可以使用*模版字符串*，它可以定义多行文本和内嵌表达式。
这种字符串是被反引号包围（`），并且以`${ expr }`这种形式嵌入表达式

```
let name: string = `Gene`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ name }.

I'll be ${ age + 1 } years old next month.`;
```

这与下面定义`sentence`的方式效果相同：

```
let sentence: string = "Hello, my name is " + name + ".\n\n" +
```

```
    "I'll be " + (age + 1) + " years old next month.";
```

数组

TypeScript像JavaScript一样可以操作数组元素。
有两种方式可以定义数组。
第一种，可以在元素类型后面接上`[]`，表示由此类型元素组成的一个数组：

```
let list: number[] = [1, 2, 3];
```

第二种方式是使用数组泛型，`Array<元素类型>`：

```
let list: Array<number> = [1, 2, 3];
```

元组 Tuple

元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。
比如，你可以定义一对值分别为`string`和`number`类型的元组。

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

当访问一个已知索引的元素，会得到正确的类型：

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

当访问一个越界的元素，会使用联合类型替代：

```
x[3] = 'world'; // OK, 字符串可以赋值给(string | number)类型

console.log(x[5].toString()); // OK, 'string' 和 'number' 都有 toString

x[6] = true; // Error, 布尔不是(string | number)类型
```

联合类型是高级主题，我们会在以后的章节里讨论它。

枚举

`enum`类型是对JavaScript标准数据类型的一个补充。
像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

默认情况下，从`0`开始为元素编号。
你也可以手动的指定成员的数值。

例如，我们将上面的例子改成从1开始编号：

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

或者，全部都采用手动赋值：

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

枚举类型提供的一个便利是你可以由枚举的值得到它的名字。
例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

alert(colorName);
```

任意值

有时候，我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。
这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。
这种情况下，我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。
那么我们可以使用any类型来标记这些变量：

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

在对现有代码进行改写的时候，any类型是十分有用的，它允许你在编译时可选择地包含或移除类型检查。
你可能认为Object有相似的作用，就像它在其它语言中那样。
但是Object类型的变量只是允许你给它赋任意值 - 但是却不能够在它上面调用任意的方法，即便它真的有这些方法：

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Objec
```

当你只知道一部分数据的类型时，any类型也是有用的。
比如，你有一个数组，它包含了不同的类型的数据：

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

空值

某种程度上来说，void类型像是与any类型相反，它表示没有任何类型。
当一个函数没有返回值时，你通常会见到其返回值类型是void：

```
function warnUser(): void {
    alert("This is my warning message");
}
```

声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined和null：

```
let unusable: void = undefined;
```

Null 和 Undefined

TypeScript里，undefined和null两者各自有自己的类型分别叫做undefined和null。
和void相似，它们的本身的类型用处不是很大：

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

默认情况下null和undefined是所有类型的子类型。
就是说你可以把null和undefined赋值给number类型的变量。

然而，当你指定了--strictNullChecks标记，null和undefined只能赋值给void和它们各自。
这能避免*很多*常见的问题。
也许在某处你想传入一个string或null或undefined，你可以使用联合类型string | null | undefined。
再次说明，稍后我们会介绍联合类型。

> 注意：我们鼓励尽可能地使用--strictNullChecks，但在本手册里我们假设这个标记是关闭的。

Never

never类型表示的是那些永不存在的值的类型。
例如，never类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型；
变量也可能是never类型，当它们被永不为真的类型保护所约束时。

never类型是任何类型的子类型，也可以赋值给任何类型；然而，*没有*类型是never的子类型或可以赋值给never类型（除了never本身之外）。
即使any也不可以赋值给never。

下面是一些返回never类型的函数：

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
    throw new Error(message);
}
```

```
// 推断的返回值类型为never
function fail() {
    return error("Something failed");
}

// 返回never的函数必须存在无法达到的终点
function infiniteLoop(): never {
    while (true) {
    }
}
```

类型断言

有时候你会遇到这样的情况，你会比TypeScript更了解某个值的详细信息。
通常这会发生在你清楚地知道一个实体具有比它现有类型更确切的类型。

通过*类型断言*这种方式可以告诉编译器，“相信我，我知道自己在干什么”。
类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。
它没有运行时的影响，只是在编译阶段起作用。
TypeScript会假设你，程序员，已经进行了必须的检查。

类型断言有两种形式。
其一是“尖括号”语法：

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

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

另一个为`as`语法：

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

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

两种形式是等价的。
至于使用哪个大多数情况下是凭个人喜好；然而，当你在TypeScript里使用JSX时，只有`as`语法断言是被允许的。

关于`let`

你可能已经注意到了，我们使用`let`关键字来代替大家所熟悉的JavaScript关键字`var`。
`let`关键字是JavaScript的一个新概念，TypeScript实现了它。
我们会在以后详细介绍它，很多常见的问题都可以通过使用`let`来解决，所以尽可能地使用`let`来代替`var`吧。