2022年2月18日 / #JAVASCRIPT

面向初学者的 TypeScript 完全指南

译者：luojiyin

作者：Danny Adams (英语)

原文： Learn TypeScript – The Ultimate Beginners Guide

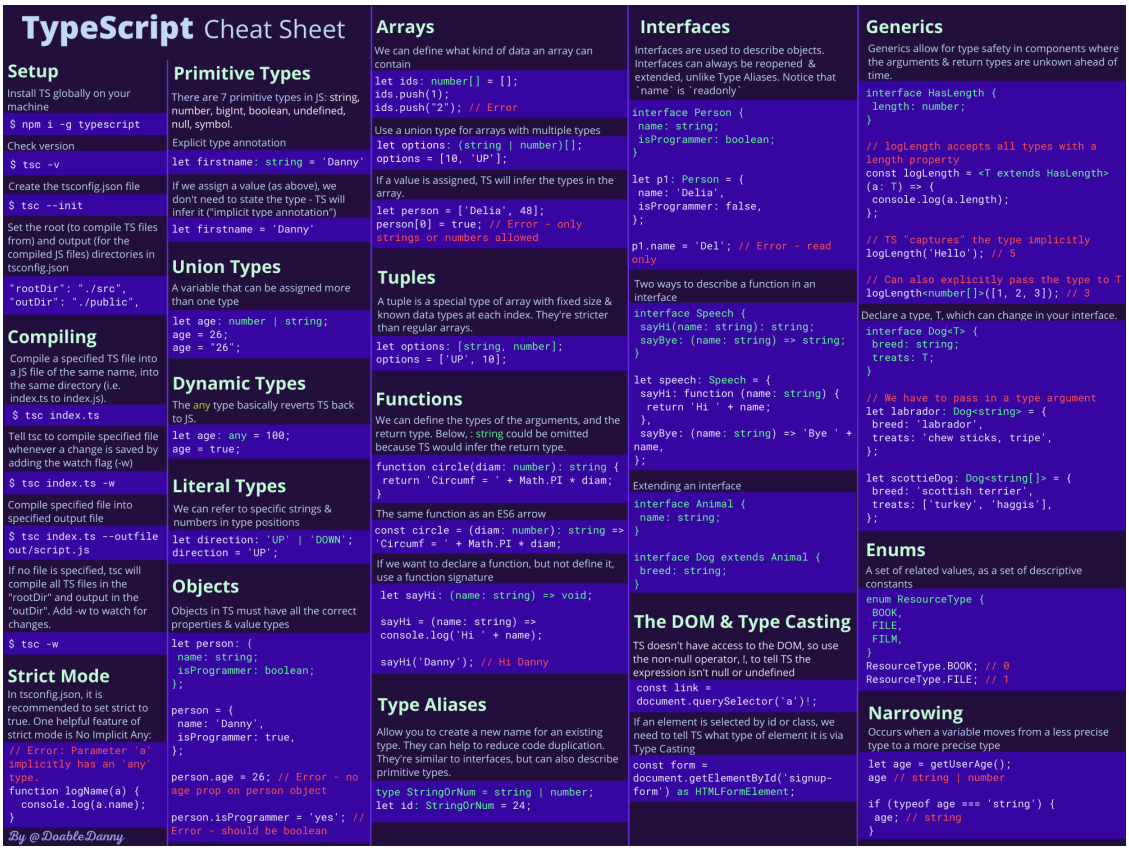
在过去的几年里，TypeScript变得越来越流行，许多工作现在都要求开发人员了解TypeScript。

即使你不打算使用TypeScript，学习它也会让你对JavaScript有更好的理解--让你成为更好的开发者。

在本文中，你将了解到：

- 什么是TypeScript，为什么要学习它？
- 如何使用 TypeScript 设置项目
- TypeScript的所有主要概念（类型、接口、泛型、类型转换，以及更多...）
- 如何在React中使用TypeScript

我还制作了一个PDF 格式TypeScript手册和海报，将这篇文章总结为一页，便于你快速查找和修改概念/语法。



PDF 格式 TypeScript 手册

什么是TypeScript

TypeScript是JavaScript的超集，意味着它能做JavaScript所做的一切，但有一些附加功能。

使用TypeScript的主要原因是为JavaScript添加静态类型。静态类型意味着变量的类型在程序中的任何时候都不能被改变。它可以防止大量的bug!

另一方面，JavaScript是一种动态类型的语言，意味着变量可以改变类型。这里有一个例子：

```
// JavaScript
let foo = "hello";
foo = 55; // foo 的类型已从字符串变为数字 - 没问题

// TypeScript
let foo = "hello";
foo = 55; // 错误 - foo 无法从字符串变为数字
```

TypeScript不能被浏览器理解，所以它必须由TypeScript编译器（TSC）编译成JavaScript，我们很快会讨论这个问题。

TypeScript值得吗

为什么要使用TypeScript

- 研究表明，TypeScript可以发现15%的常见错误。
- 可读性--更容易看到代码应该做什么。而在团队中工作时，更容易看到其他开发人员的意图。
- 它很受欢迎--了解TypeScript将使你能够申请到更多好工作。

下面是我写的一篇短文，展示了TypeScript如何防止烦人的Bug。

TypeScript的缺点

- TypeScript的编写时间比JavaScript长，因为你必须指定类型，所以对于较小的单独项目，可能不值得使用它。
- TypeScript必须进行编译--这可能需要时间，特别是在大型项目中

但是，你必须花更多的时间来写更精确的代码和编译，这将使你的代码中的错误减少得更多。

对于许多项目--尤其是中大型项目--TypeScript将为你节省大量的时间和麻烦。

如果你已经知道JavaScript，TypeScript也不会太难学。它是你武器库中的一个重要工具。

如何设置TypeScript项目

安装Node和TypeScript编译器

首先，确保你的机器上全局安装了Node。

然后通过运行以下命令在你的机器上全局安装TypeScript编译器。

```
npm i -g typescript
```

检查安装是否成功（如果成功，它将返回版本号）：

如何编译TypeScript

打开你的文本编辑器，创建一个TypeScript文件（例如，index.ts）。

编写一些JavaScript或TypeScript。

```
let sport = 'football';

let id = 5;
```

现在我们可以用以下命令将其编译成JavaScript：

```
tsc index
```

TSC将把代码编译成JavaScript，并在一个名为index.js的文件中输出。

```
var sport = 'football';
var id = 5;
```

如果你想指定输出文件的名称：

```
tsc index.ts --outfile file-name.js
```

```
tsc index.ts -w
```

TypeScript的一个有趣之处在于，当你在编码时，它会在文本编辑器中报告错误，但它总是会编译你的代码，无论是否有错误。

例如，下面的内容会使TypeScript立即报告错误：

```
var sport = 'football';
var id = 5;

id = '5'; // Error: Type 'string' is not assignable to
type 'number'.
```

但如果我们尝试用 `tsc index` 来编译这段代码，尽管有错误，但代码仍然可以编译。

这是TypeScript的一个重要特性：它假定开发者知道得更多。即使有一个TypeScript错误，它也不会妨碍你编译代码。它告诉你有一个错误，但这取决于你是否对它做了什么。

如何设置ts配置文件

ts 配置文件应该在你项目的根目录下。在这个文件中，我们可以指定根文件、编译器选项以及我们希望 TypeScript 在检查我们项目时有多严格。

首先，创建 ts 配置文件：

```
tsc --init
```

你现在应该在项目根部有一个 `tsconfig.json` 文件。

```
{
    "compilerOptions": {
        ...
        /* Modules */
        "target": "es2016", // 改为 “ES2015” 以编译为 ES6
        "rootDir": "./src", // 从何处编译
        "outDir": "./public", // 编译到何处（通常是要部署到网络服务器的

        /* JavaScript Support */
        "allowJs": true, // 允许编译 JavaScript 文件
        "checkJs": true, // 检查 JavaScript 文件类型并报告错误

        /* Emit */
        "sourceMap": true, // 为生成的 JavaScript 文件创建源映射文件
         "removeComments": true, // 不要生成注释
    },
    "include": ["src"] // 确保只编译 src 中的文件
}
```

编译所有内容并观察变化：

```
tsc -w
```

> *注意：当输入文件在命令行中被指定时（例如，* `tsc index`*），* `tsconfig.json` *文件被忽略。*

TypeScript中的类型

原始类型

在JavaScript中，原始数据类型是指不属于对象且没有方法的数据。有7种原始数据类型。

- bigint
- boolean
- undefined
- null
- symbol

基元（Primitives）是不可变的（immutable）：它们不能被改变。重要的是，不要将基元本身与分配给基元值的变量相混淆。变量可以被重新分配一个新值，但现有值不能像对象、数组和函数那样被改变。

这是一个例子：

```
let name = 'Danny';
name.toLowerCase();
console.log(name); // Danny - 字符串方法没有改变字符串

let arr = [1, 3, 5, 7];
arr.pop();
console.log(arr); // [1, 3, 5] - 数组方法改变了数组

name = 'Anna' // 赋值给基元一个新值（而不是改变值）
```

在JavaScript中，所有的原始值（除了null和undefined）都有对应的对象，这些对象包裹着原始值。这些包装对象是String、Number、BigInt、Boolean和Symbol。这些包装对象提供了允许原始值被操纵的方法。

回到TypeScript，我们可以在声明一个变量后添加 `：type` （称为"类型注解"或"类型签名"）来设置我们希望变量的类型。例子。

```
let hasDog: boolean = true;

let unit: number; // Declare variable without assigning a value
unit = 5;
```

但通常最好不要明确说明类型，因为TypeScript会自动推断变量的类型（类型推论）：

```
let id = 5; // TS 知道它是一个数字
let firstname = 'danny'; // TS 知道它是一个字符串
let hasDog = true; // TS 知道它是布尔值

hasDog = 'yes'; // ERROR
```

我们也可以将一个变量设定为能够成为联合类型。**联合类型是一个可以被分配到多个类型的变量**：

```
let age: string | number;
age = 26;
age = '26';
```

引用类型

在JavaScript中，几乎“所有东西”都是一个对象。事实上（而且令人困惑的是），如果用 new 关键字定义的话，字符串、数字和布尔都可以成为对象：

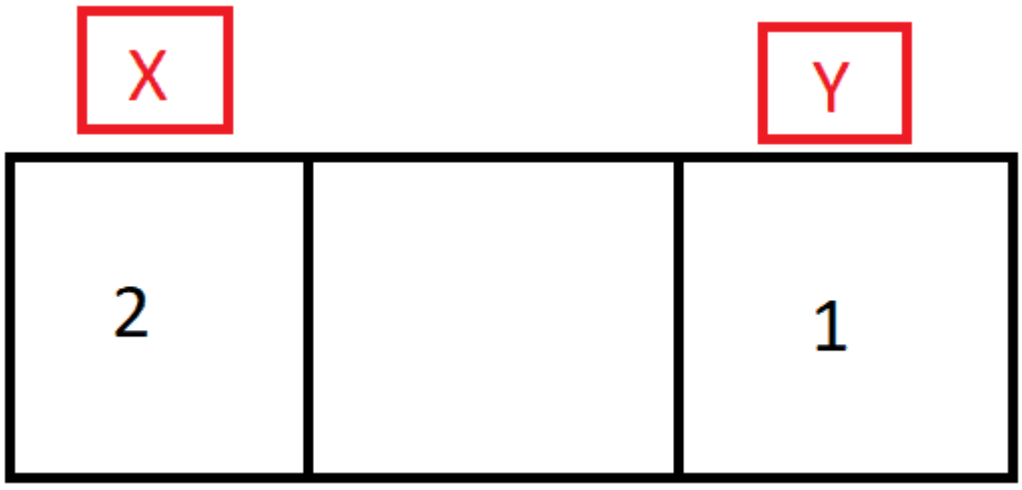
但是，当我们谈论JavaScript中的引用类型时，我们指的是数组、对象和函数。

注意事项：原始类型与引用类型

对于那些从未研究过原始类型与引用类型的人来说，让我们来讨论一下其根本区别。

如果一个基元类型被分配给一个变量，我们可以认为该变量是**包含**基元值的。每个基元值都存储在内存中的一个唯一位置。

如果我们有两个变量x和y，并且它们都包含原始数据，那么它们就完全相互独立。

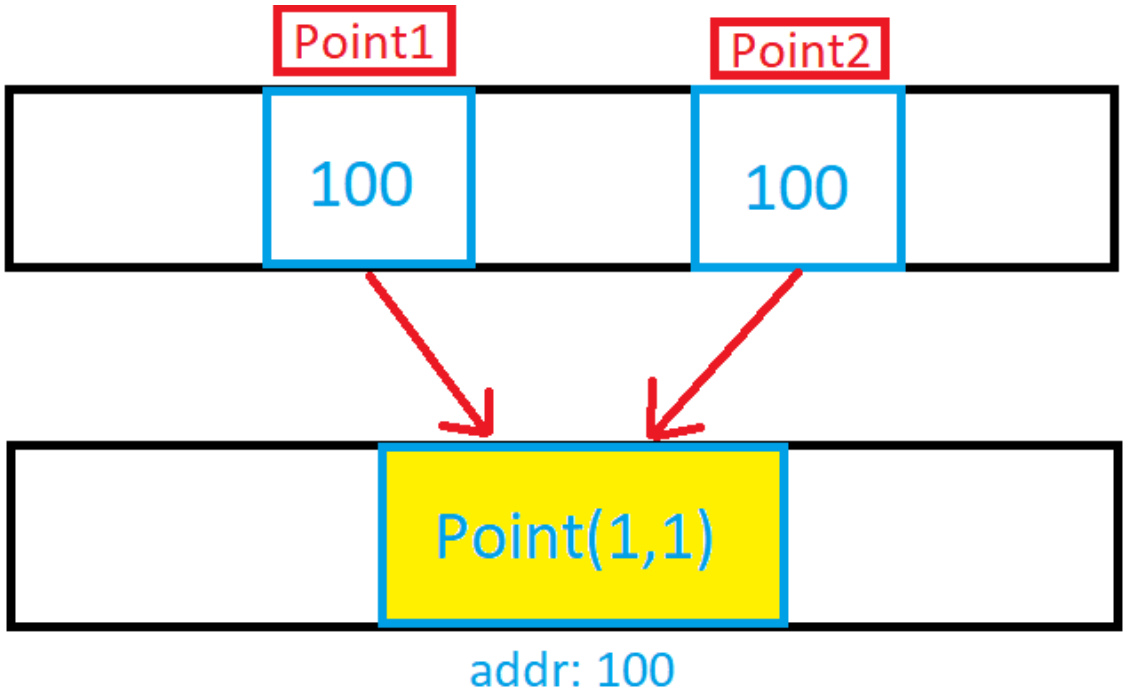


X和Y都包含唯一的独立原始数据

```
let x = 2;
let y = 1;

x = y;
```

而引用类型则不是这样。引用类型指向的是存储对象的一个内存位置。



point1和point2包含一个对存储对象的地址的引用。

```
let point1 = { x: 1, y: 1 };
let point2 = point1;

point1.y = 100;
console.log(point2.y); // 100 (point1和point2多指向的是存储点对象的同
```

这是对主要类型与参考类型的一个快速概述。如果你需要更深入的解释，请看这篇文章。原始类型与引用类型。

```
let ids: number[] = [1, 2, 3, 4, 5]; // 只能包含数字
let names: string[] = ['Danny', 'Anna', 'Bazza']; // 只能包含字符串
let options: boolean[] = [true, false, false]; 只能包含真或假
let books: object[] = [
  { name: 'Fooled by randomness', author: 'Nassim Taleb' },
  { name: 'Sapiens', author: 'Yuval Noah Harari' },
]; // 只能包含对象
let arr: any[] = ['hello', 1, true]; // 这基本上是将 TypeScript 还原

ids.push(6);
ids.push('7'); // 错误："字符串"类型的参数不能分配给"数字"类型的参数。
```

你可以使用联合类型来定义包含多种类型的数组：

```
let person: (string | number | boolean)[] = ['Danny', 1, true];
person[0] = 100;
person[1] = {name: 'Danny'} // Error - person array can't contain
```

如果你用一个值初始化一个变量，没有必要明确说明类型，因为TypeScript会推断出它的类型：

```
let person = ['Danny', 1, true]; // This is identical to above ex
person[0] = 100;
person[1] = { name: 'Danny' }; // Error - person array can't cont
```

一种特殊类型的数组可以在TypeScript中定义。元组（Tuples）。**元组是一个具有固定大小和已知数据类型的数组，**它们比普通数组更

```
let person: [string, number, boolean] = ['Danny', 1, true];
person[0] = 100; // 错误 - 索引 0 的值只能是字符串
```

TypeScript中的对象

TypeScript中的对象必须有所有正确的属性和值类型：

```
// 使用特定对象类型方法声明名为 person 的变量
let person: {
  name: string;
  location: string;
  isProgrammer: boolean;
};

// 将 person 赋值为给一个具有所有必要属性和值类型的对象
person = {
  name: 'Danny',
  location: 'UK',
  isProgrammer: true,
};

person.isProgrammer = 'Yes'; // 错误：应为布尔值


person = {
  name: 'John',
  location: 'US',
};
// 错误：缺少 isProgrammer 属性
```

当定义一个对象的签名时，你通常会使用一个 **接口（interface）**。如果我们需要检查多个对象是否具有相同的特定属性和价值类型，这一点很有用：

```
  isProgrammer: boolean;
}

let person1: Person = {
  name: 'Danny',
  location: 'UK',
  isProgrammer: true,
};

let person2: Person = {
  name: 'Sarah',
  location: 'Germany',
  isProgrammer: false,
};
```

我们也可以用函数签名来声明函数属性。我们可以使用老式的普通JavaScript function(`sayHi`)，或者ES6的箭头函数（ `sayBye` ）来做这件事：

```
interface Speech {
  sayHi(name: string): string;
  sayBye: (name: string) => string;
}

let sayStuff: Speech = {
  sayHi: function (name: string) {
    return `Hi ${name}`;
  },
  sayBye: (name: string) => `Bye ${name}`,
};

console.log(sayStuff.sayHi('Heisenberg')); // Hi Heisenberg
console.log(sayStuff.sayBye('Heisenberg')); // Bye Heisenberg
```

注意在 `sayStuff` 对象中， `sayHi` 或 `sayBye` 可以被赋予一个箭头函数或一个普通的JavaScript函数--TypeScript并不关心。

```
// 定义一个名为 circle 的函数，该函数接收一个数字类型的 diam 变量，并返回-
function circle(diam: number): string {
  return 'The circumference is ' + Math.PI * diam;
}

console.log(circle(10)); // The circumference is 31.415926535897
```

同样的函数，但使用ES6箭头函数：

```
const circle = (diam: number): string => {
  return 'The circumference is ' + Math.PI * diam;
};

console.log(circle(10)); // The circumference is 31.415926535897
```

请注意，没有必要明确说明`circle'是一个函数；TypeScript会推断它。TypeScript也会推断出函数的返回类型，所以也不需要说明。不过，如果函数很大，有些开发者喜欢明确说明返回类型，以使其清晰明了。

```
// 使用显式类型
const circle: Function = (diam: number): string => {
  return 'The circumference is ' + Math.PI * diam;
};

// 推断类型--TypeScript 认为 circle 是一个总是返回字符串的函数，因此无需
const circle = (diam: number) => {
```

我们可以在一个参数后面加一个问号，使其成为可选参数，不一定需要传入此参数。还注意到下面 c 是一个联合类型，可以是数字或字符串：

```
const add = (a: number, b: number, c?: number | string) => {
  console.log(c);

  return a + b;
};

console.log(add(5, 4, 'I could pass a number, string, or nothing
// I could pass a number, string, or nothing here!
// 9
```

一个不返回任何东西的函数被说成是返回void--完全没有任何返回值。下面，已经明确说明了void的返回类型。但是，这也是没有必要的，因为TypeScript会推断出它。

```
const logMessage = (msg: string): void => {
  console.log('This is the message: ' + msg);
};

logMessage('TypeScript is superb'); // This is the message: Type
```

如果我们想声明一个函数变量，但不定义它（说清楚它的作用），**就使用一个函数签名。** 下面，函数 `sayHello` 必须跟在冒号后面的签名：

```
// 定义函数，满足其签名
sayHello = (name) => {
  console.log('Hello ' + name);
};

sayHello('Danny'); // Hello Danny
```

动态(任意)类型

使用 any 类型，我们基本上可以将TypeScript还原成JavaScript：

```
let age: any = '100';
age = 100;
age = {
  years: 100,
  months: 2,
};
```

建议尽量避免使用 “any” 类型，因为它妨碍了TypeScript的工作--并可能导致错误。

类型别名

类型别名可以减少代码的重复，使我们的代码保持干燥。下面，我们可以看到 PersonObject 类型别名已经防止了重复，并且作为一个单一的真理来源，说明一个人的对象应该包含哪些数据。

```
type StringOrNumber = string | number;

type PersonObject = {
  name: string;
  id: StringOrNumber;
};
```

```
};

const person2: PersonObject = {
  name: 'Delia',
  id: 2,
};

const sayHello = (person: PersonObject) => {
  return 'Hi ' + person.name;
};

const sayGoodbye = (person: PersonObject) => {
  return 'Seeya ' + person.name;
};
```

DOM和类型转换

TypeScript并不像JavaScript那样可以访问DOM。这意味着，每当我们试图访问DOM元素时，TypeScript从不确定它们是否真的存在。

下面的例子显示了这个问题：

```
const link = document.querySelector('a');

console.log(link.href); // 错误:  对象可能为“空”。TypeScript 无法确定
```

通过非空断言操作符（！），我们可以明确地告诉编译器，一个表达式的值不是“空”或“未定义”。当编译器不能确定地推断出类型，但我们比编译器拥有更多的信息时，这就很有用。

```
console.log(link.href); // www.freeCodeCamp.org
```

请注意，我们不必说明 `link` 变量的类型。这是因为TypeScript可以清楚地看到（通过类型推论）它是 `HTMLAnchorElement` 类型。

但是，如果我们需要通过它的类或id来选择一个DOM元素呢？TypeScript不能推断出类型，因为它可能是任何东西。

```
const form = document.getElementById('signup-form');

console.log(form.method);
// 错误:  对象可能为“空”。
// 错误:  “HTMLElement” 类型中不存在属性 “method”。
```

以上，我们得到了两个错误。我们需要告诉TypeScript，我们确定 `form` 存在，而且我们知道它是 `HTMLFormElement` 的类型。我们通过类型转换来实现这一点：

```
const form = document.getElementById('signup-form') as HTMLFormEl

console.log(form.method); // post
```

使用TypeScript，感到工作上的愉悦！

TypeScript也有一个内置的事件对象。所以，如果我们在表单中添加一个提交事件监听器，如果我们调用任何不属于Event对象的方法，TypeScript会给我们一个错误。看看TypeScript有多酷--它可以在我们犯了拼写错误时告诉我们：

```
form.addEventListener('submit', (e: Event) => {
  e.preventDefault(); // prevents the page from refreshing

  console.log(e.tarrget); // 错误:  事件类型中不存在属性 "tarrget"。你
});
```

TypeScript中的类

我们可以在一个类中定义每一块数据具体的类型：

```
class Person {
  name: string;
  isCool: boolean;
  pets: number;

  constructor(n: string, c: boolean, p: number) {
    this.name = n;
    this.isCool = c;
    this.pets = p;
  }

  sayHello() {
    return `Hi, my name is ${this.name} and I have ${this.pets} p
  }
}

const person1 = new Person('Danny', false, 1);
const person2 = new Person('Sarah', 'yes', 6); // 错误:“字符串”类型

console.log(person1.sayHello()); // Hi, my name is Danny and I ha
```

然后我们可以创建一个 people 数组，其中只包括由 Person 类构建的对象：

我们可以在类的属性中添加访问修饰语。TypeScript也提供了一个新的访问修饰符，叫做 只读（readonly）。

```
class Person {
  readonly name: string; // 该属性不可变，只能被读取
  private isCool: boolean; // 只能访问或修改这个类中的方法
  protected email: string; // 可以访问和修改这个类和子类
  public pets: number; // 可从任何地方（包括类之外）访问或修改

  constructor(n: string, c: boolean, e: string, p: number) {
    this.name = n;
    this.isCool = c;
    this.email = e;
    this.pets = p;
  }

  sayMyName() {
    console.log(`Your not Heisenberg, you're ${this.name}`);
  }
}

const person1 = new Person('Danny', false, 'dan@e.com', 1);
console.log(person1.name); // Fine
person1.name = 'James'; // Error: read only
console.log(person1.isCool); // Error: private property - only ac
console.log(person1.email); // Error: protected property - only a
console.log(person1.pets); // Public property - so no problem
```

我们可以通过这样构建类（constructing class）的属性来使我们的代码更加简洁：

```
class Person {
  constructor(
```

```
  ) {}

  sayMyName() {
    console.log(`Your not Heisenberg, you're ${this.name}`);
  }
}

const person1 = new Person('Danny', false, 'dan@e.com', 1);
console.log(person1.name); // Danny
```

以上述方式编写，属性会在构造函数中自动分配--省去了我们把它们全部写出来的麻烦。

注意，如果我们省略了访问修饰符，默认情况下，该属性将是公共的（public）。

类也可以被扩展（extend），就像在普通的JavaScript中一样：

```
class Programmer extends Person {
  programmingLanguages: string[];

  constructor(
    name: string,
    isCool: boolean,
    email: string,
    pets: number,
    pL: string[]
  ) {
    // The super call must supply all parameters for base (Persor
    super(name, isCool, email, pets);
    this.programmingLanguages = pL;
  }
}
```

查看 TypeScript 官方文档了解更多关于类的信息。

在JavaScript中，一个模块只是一个包含相关代码的文件。功能可以在模块之间被导入和导出，使代码保持良好的组织。

TypeScript也支持模块。TypeScript文件将被编译成多个JavaScript文件。

在 `tsconfig.json` 文件中，改变以下选项以支持现代的导入和导出：

```
"target": "es2016",
"module": "es2015"
```

（不过，对于Node项目来说，你很可能需要 `"模块": "CommonJS"` - Node还不支持现代的导入/导出。）

现在，在你的HTML文件中，将脚本的导入改为模块类型：

```
<script type="module" src="/public/script.js"></script>
```

我们现在可以使用ES6导入和导出文件了：

```
// src/hello.ts
export function sayHi() {
  console.log('Hello there!');
}

// src/script.ts
import { sayHi } from './hello.js';
```

注意：总是作为一个JavaScript文件导入，即使在TypeScript文件中。

TypeScript中的接口

接口定义了一个对象：

```
interface Person {
  name: string;
  age: number;
}

function sayHi(person: Person) {
  console.log(`Hi ${person.name}`);
}

sayHi({
  name: 'John',
  age: 48,
}); // Hi John
```

你也可以使用一个类型别名来定义一个对象类型：

```
type Person = {
  name: string;
  age: number;
};

function sayHi(person: Person) {
  console.log(`Hi ${person.name}`);
}

sayHi({
  name: 'John',
```

或者可以匿名地定义一个对象类型：

```
function sayHi(person: { name: string; age: number }) {
  console.log(`Hi ${person.name}`);
}

sayHi({
  name: 'John',
  age: 48,
}); // Hi John
```

接口与类型别名非常相似，在许多情况下，你可以使用这两者。关键的区别是，类型别名不能被重新打开以添加新的属性，而接口总是可以扩展的。

下面的例子取自TypeScript docs。

扩展一个接口:

```
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear: Bear = {
  name: "Winnie",
  honey: true,
}
```

```
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: boolean
}

const bear: Bear = {
  name: "Winnie",
  honey: true,
}
```

在现有接口上添加新字段：

```
interface Animal {
  name: string
}

// 重新打开 Animal interface 添加新字段
interface Animal {
  tail: boolean
}

const dog: Animal = {
  name: "Bruce",
  tail: true,
}
```

这里有一个关键的区别：一个类型在被创建后不能被改变。

```
type Animal = {
  name: string
}
```

作为一条经验法则，TypeScript文档推荐使用接口来定义对象，直到你需要使用一个类型的功能。

接口也可以定义函数签名：

```
interface Person {
  name: string
  age: number
  speak(sentence: string): void
}

const person1: Person = {
  name: "John",
  age: 48,
  speak: sentence => console.log(sentence),
}
```

你可能想知道为什么在上面的例子中我们要使用一个接口而不是一个类。

使用接口的一个好处是，它只被TypeScript使用，而不是JavaScript。这意味着它不会被编译，也不会给你的JavaScript增加臃肿。类是JavaScript的特性，所以它会被编译。

另外，类本质上是一个**对象工厂**（也就是说，一个对象应该是什么样子的蓝图（blueprint），然后实现），而接口是一个仅用于**类型检查的结构**。

虽然一个类可能有初始化的属性和方法来帮助创建对象，但一个接口基本上定义了一个对象可以拥有的属性和类型。

方法：

```
interface HasFormatter {
  format(): string;
}

class Person implements HasFormatter {
  constructor(public username: string, protected password: string

  format() {
    return this.username.toLocaleLowerCase();
  }
}

// 必须是实现 HasFormatter 接口的对象
let person1: HasFormatter;
let person2: HasFormatter;

person1 = new Person('Danny', 'password123');
person2 = new Person('Jane', 'TypeScripter1990');

console.log(person1.format()); // danny
```

确保 people 是一个实现 HasFormatter 的对象数组（确保每个人都有一样的 format method）：

```
let people: HasFormatter[] = [];
people.push(person1);
people.push(person2);
```

TypeScript中的字面类型

```
// 每个位置都有一个字面类型的联合类型
let favouriteColor: 'red' | 'blue' | 'green' | 'yellow';

favouriteColor = 'blue';
favouriteColor = 'crimson'; // ERROR: Type '"crimson"' is not ass
```

泛型

泛型允许你创建一个可以在多种类型上工作的组件，而不是单一的类型，**这有助于使组件更容易重复使用**。

让我们通过一个例子来告诉你这意味着什么……

addID "函数接受任何对象，并返回一个新的对象，该对象具有所有的属性和值，加上一个 "id "属性，其值在0到1000之间。简而言之，它给任何对象一个ID。

```
 const addID = (obj: object) => {
  let id = Math.floor(Math.random() * 1000);

  return { ...obj, id };
};

let person1 = addID({ name: 'John', age: 40 });

console.log(person1.id); // 271
console.log(person1.name); // ERROR: Property 'name' does not ex
```

正如你所看到的，当我们试图访问 name 属性时，TypeScript给出了一个错误。这是因为当我们传入一个对象到 addID 时，我们没有指

那么，我们怎样才能向 addID 传递任何对象，但仍然告诉TypeScript这个对象有哪些属性和值呢？我们可以使用一个 *generic*，其中 T 被称为 *type parameter*：

```
// <T> is just the convention - e.g. we could use <X> or <A>
const addID = <T>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);

  return { ...obj, id };
};
```

这有什么作用？现在，当我们把一个对象传给 addID 时，我们已经告诉TypeScript捕捉类型--所以 T 变成我们传入的任何类型。 addID 现在会知道我们传入的对象有哪些属性。

但是，我们现在有一个问题：任何东西都可以被传入 addID ，TypeScript将捕获类型，并报告没有问题：

```
let person1 = addID({ name: 'John', age: 40 });
let person2 = addID('Sally'); // Pass in a string - no problem

console.log(person1.id); // 271
console.log(person1.name); // John

console.log(person2.id);
console.log(person2.name); // ERROR: Property 'name' does not ex
```

当我们传入一个字符串时，TypeScript没有发现问题。它只在我们试图访问 name 属性时报告了一个错误。因此，我们需要一个约束：

```
const addID = <T extends object>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);

  return { ...obj, id };
};

let person1 = addID({ name: 'John', age: 40 });
let person2 = addID('Sally'); // 错误: “字符串”类型的参数不能分配给“对
```

这个错误被直接抓住了--完美.....好吧，不完全是。在JavaScript中，数组是对象，所以我们仍然可以通过传入数组来解决这个问题：

```
let person2 = addID(['Sally', 26]); // Pass in an array - no prob

console.log(person2.id); // 824
console.log(person2.name); // Error: Property 'name' does not ex
```

我们可以通过说对象参数应该有一个带有字符串值的名称属性来解决这个问题：

```
const addID = <T extends { name: string }>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);

  return { ...obj, id };
};

let person2 = addID(['Sally', 26]); // ERROR: argument should hav
```

```
// 在下文中，我们明确说明了括号中的参数类型。
let person1 = addID<{ name: string; age: number }>({ name: 'John
```

在参数和返回类型提前未知的情况下，泛型允许你在组件中实现类型安全。

在TypeScript中，当我们想描述两个值之间的对应关系时，可以使用泛型。 在上面的例子中，返回类型与输入类型相关。我们用一个_generic_来描述这个对应关系。

另一个例子。如果我们需要一个接受多种类型的函数，使用泛型比使用 any 类型更好。下面显示了使用`any'的问题：

```
function logLength(a: any) {
  console.log(a.length); // No error
  return a;
}

let hello = 'Hello world';
logLength(hello); // 11

let howMany = 8;
logLength(howMany); // undefined（但没有 TypeScript 错误--我们当然希
```

我们可以尝试使用泛型：

```
function logLength<T>(a: T) {
  console.log(a.length); // 错误: TypeScript 无法确定 `a` 是一个具有
```

至少我们现在得到了一些反馈，我们可以用它来约束我们的代码。

解决方案：使用一个扩展了接口的泛型，确保传入的每个参数都有一个长度属性：

```
interface hasLength {
  length: number;
}

function logLength<T extends hasLength>(a: T) {
  console.log(a.length);
  return a;
}

let hello = 'Hello world';
logLength(hello); // 11

let howMany = 8;
logLength(howMany); // 错误：数字没有 length 属性
```

我们也可以写一个函数，参数是一个元素数组，这些元素都有一个 length 属性：

```
interface hasLength {
  length: number;
}

function logLengths<T extends hasLength>(a: T[]) {
  a.forEach((element) => {
    console.log(element.length);
  });
}
```

```
];

logLengths(arr);
// 29
// 4
// 30
```

泛型是TypeScript的一个很好的功能！

带有接口的泛型

当我们不知道一个对象中的某个值会是什么类型时，我们可以使用一个泛型来传递类型：

```
// The type, T, will be passed in
interface Person<T> {
  name: string;
  age: number;
  documents: T;
}

// 我们必须输入 `documents` 的类型，这里是字符串数组
const person1: Person<string[]> = {
  name: 'John',
  age: 48,
  documents: ['passport', 'bank statement', 'visa'],
};

// 同样，我们实现了 `Person` 接口，并输入了 `documents` 类型--这里是字符
const person2: Person<string> = {
  name: 'Delia',
  age: 46,
  documents: 'passport, P45',
};
```

允许我们定义或声明相关值的集合，可以是数字或字符串，作为一组命名的常量。

```
enum ResourceType {
  BOOK,
  AUTHOR,
  FILM,
  DIRECTOR,
  PERSON,
}

console.log(ResourceType.BOOK); // 0
console.log(ResourceType.AUTHOR); // 1

// To start from 1
enum ResourceType {
  BOOK = 1,
  AUTHOR,
  FILM,
  DIRECTOR,
  PERSON,
}

console.log(ResourceType.BOOK); // 1
console.log(ResourceType.AUTHOR); // 2
```

默认情况下，枚举（enums）是基于数字的--它们将字符串值存储为数字。但是它们也可以是字符串：

```
enum Direction {
  Up = 'Up',
  Right = 'Right',
  Down = 'Down',
  Left = 'Left',
}
```

当我们有一组相关的常量时，枚举（Enums）很有用。例如，与其在整个代码中使用非描述性的数字，枚举可以通过描述性的常量使代码更具可读性。

枚举（Enums）也可以防止bug，因为当你输入枚举的名称时，智能感知（intellisense）会弹出，并给出可以选择的可能选项列表。

TypeScript严格模式

建议在 `tsconfig.json` 文件中启用所有严格的类型检查操作。这将导致TypeScript报告更多的错误，但将有助于防止许多错误悄悄进入你的应用程序。

```
// tsconfig.json
"strict": true
```

让我们讨论一下严格模式所做的几件事：没有隐含的any类型定义，以及严格的空值检查（null checks）。

没有隐含的any

在下面的函数中，TypeScript已经推断出参数 `a` 是 `any` 类型。正如你所看到的，当我们传入一个数字到这个函数，并试图通过打印 `name` 属性值时，没有报告错误。这不太好。

```
function logName(a) {
  // No error??
  console.log(a.name);
}
```

在打开 "noImplicitAny "选项的情况下，如果我们没有明确说明 a 的类型，TypeScript将立即标记一个错误：

```
// ERROR: Parameter 'a' implicitly has an 'any' type.
function logName(a) {
  console.log(a.name);
}
```

严格的空值检查（null checks）

当 "strictNullChecks "选项为 `false` 时，没有启用，TypeScript会忽略 `null` 和 `undefined` 。这可能在运行时出现意外的错误。

当 `strictNullChecks` 设置为 `true` 时，变量被定义为 `null` 和 `undefined` 类型，如果你把它们赋给一个期望有具体数值的变量（例如， `string` ），你会得到一个类型错误（type error）。

```
let whoSangThis: string = getSong();

const singles = [
  { song: 'touch of grey', artist: 'grateful dead' },
  { song: 'paint it black', artist: 'rolling stones' },
];

const single = singles.find((s) => s.song === whoSangThis);

console.log(single.artist);
```

上面， `singles.find` 不能保证它能找到这 `song` ，但我们写的代码就好像它总是能找到。

```
const getSong = () => {
  return 'song';
};

let whoSangThis: string = getSong();

const singles = [
  { song: 'touch of grey', artist: 'grateful dead' },
  { song: 'paint it black', artist: 'rolling stones' },
];

const single = singles.find((s) => s.song === whoSangThis);

console.log(single.artist); // ERROR: Object is possibly 'undefi
```

TypeScript基本上是在告诉我们在使用 `single` 之前要确保它的存在。我们需要先检查它是否为 `null` 或 `undefined`：

```
if (single) {
  console.log(single.artist); // rolling stones
}
```

TypeScript中的type narrowing

在TypeScript程序中，**变量可以从一个不太精确的类型转移到一个更精确的类型。** 这个过程称为类型缩小（type narrowing）。

下面是一个简单的例子，显示了当我们使用if语句和 `typeof` 时，TypeScript如何将不太具体的 `string | number` 的类型缩小到更具体的类型：

```
    // 在这个代码块中，TypeScript 将 `val` 视为字符串，所以我们可以在 `v
    return val.concat(' ' + val);
  }

  // 这里 TypeScript 知道 `val` 是一个数字
  return val + val;
}

console.log(addAnother('Woooo')); // Woooo Woooo
console.log(addAnother(20)); // 40
```

另一个例子：下面，我们定义了一个名为 `allVehicles` 的联合类型，它可以是 `Plane` 或 `Train` 的类型。

```
interface Vehicle {
  topSpeed: number;
}

interface Train extends Vehicle {
  carriages: number;
}

interface Plane extends Vehicle {
  wingSpan: number;
}

type PlaneOrTrain = Plane | Train;

function getSpeedRatio(v: PlaneOrTrain) {
  // 这里，我们想要返回 topSpeed/carriages 或 topSpeed/wingSpan
  console.log(v.carriages); // ERROR: 'carriages' doesn't exist o
}
```

由于函数 `getSpeedRatio` 要处理多种类型，我们需要一种方法来区分 `v` 是 `Plane` 还是 `Train` 。我们可以通过给这两种类型一个共同的

```
// 现在，所有列车的类型属性都必须等于 “Train”
interface Train extends Vehicle {
  type: 'Train';
  carriages: number;
}

// 现在，所有列车的类型属性都必须等于 “Plane”
interface Plane extends Vehicle {
  type: 'Plane';
  wingSpan: number;
}

type PlaneOrTrain = Plane | Train;
```

现在，我们和TypeScript可以缩小 v 的类型了：

```
function getSpeedRatio(v: PlaneOrTrain) {
  if (v.type === 'Train') {
    // TypeScript 现在知道 `v` 绝对是 `Train`。它已将类型从较不具体的 `
    return v.topSpeed / v.carriages;
  }

  // 如果不是 Train，TypeScript 就会缩小范围，认为 `v` 必须是 Plane -
  return v.topSpeed / v.wingSpan;
}

let bigTrain: Train = {
  type: 'Train',
  topSpeed: 100,
  carriages: 20,
};

console.log(getSpeedRatio(bigTrain)); // 5
```

TypeScript与React

- create-react-app (TS setup)
- Gatsby (TS setup)
- Next.js (TS setup)

如果你需要更多定制的React-TypeScript配置，你可以设置Webpack（一个模块打包器--module bundler）并自己配置 `tsconfig.json` 。但大多数时候，一个框架会完成这项工作。

例如，要用TypeScript设置create-react-app，只需运行：

```
npx create-react-app my-app --template typescript

# or

yarn create react-app my-app --template typescript
```

在src文件夹中，我们现在可以创建以 `.ts` （普通TypeScript文件）或 `.tsx` （TypeScript with React）为扩展名的文件，用TypeScript编写我们的组件。然后，这将编译成JavaScript文件存储在public文件夹中。

使用 TypeScript 的 React props

下面，我们说 `Person` 应该是一个React功能组件（functional component），它接收一个props对象，其props `name` 应该是一个字符串， `age` 应该是一个数字。

```
// src/components/Person.tsx
import React from 'react';
```

```
}> = ({ name, age }) => {
  return (
    <div>
      <div>{name}</div>
      <div>{age}</div>
    </div>
  );
};

export default Person;
```

但大多数开发者更愿意使用一个接口（interface）来指定 `prop type`：

```
interface Props {
  name: string;
  age: number;
}

const Person: React.FC<Props> = ({ name, age }) => {
  return (
    <div>
      <div>{name}</div>
      <div>{age}</div>
    </div>
  );
};
```

然后我们可以把这个组件（component）导入到 `App.tsx` 中。如果我们没有提供必要的props，TypeScript会给出一个错误。

```
import React from 'react';
import Person from './components/Person';

const App: React.FC = () => {
```

```
  );
};

export default App;
```

下面是几个例子，说明我们可以有哪些prop types：

```
interface PersonInfo {
  name: string;
  age: number;
}

interface Props {
  text: string;
  id: number;
  isVeryNice?: boolean;
  func: (name: string) => string;
  personInfo: PersonInfo;
}
```

使用TypeScript编写React hooks

useState()

我们可以通过使用角括号声明一个状态变量应该是什么类型。下面，如果我们省略了角括号，TypeScript会推断出 cash 是一个数字（number）。所以，如果想让它也成为空值，我们必须指定：

```
const Person: React.FC<Props> = ({ name, age }) => {
  const [cash, setCash] = useState<number | null>(1);

  setCash(null);

  return (
```

```
  );
};
```

useRef()

`useRef` 返回一个可变的对象（mutable object），在组件的生命周期内存在。我们可以告诉TypeScript这个ref对象应该指向什么，下面我们说这个prop应该是一个 `HTMLInputElement`：

```
const Person: React.FC = () => {
  // Initialise .current property to null
  const inputRef = useRef<HTMLInputElement>(null);

  return (
    <div>
      <input type='text' ref={inputRef} />
    </div>
  );
};
```

关于React与TypeScript的更多信息，请查看这些awesome React-TypeScript cheatsheets。

有用的资源和进一步阅读

- The official TypeScript docs
- The Net Ninja's TypeScript video series (awesome!)
- Ben Awad's TypeScript with React video
- Narrowing in TypeScript (TS的一个非常有趣的功能，你应该学习一下)
- Function overloads

感谢你阅读本文

希望这对你有用。如果你走到这里，你现在知道了TypeScript的主要基础知识，可以开始在你的项目中使用它。

同样，你也可以下载我的one-page TypeScript cheat sheet PDF 或者订购实物海报。

更多关于我的信息，你可以在推特和 YouTube上关注我。

干杯!

译者：luojiyin

阅读 更多文章。

作者：Danny Adams (英语)

I am a fullstack web developer focused on React, NextJS, TypeScript, Node, and PHP. Currently freelancing fulltime with WordPress.

学习编程 – 3,000 小时免费课程

你可以点击此处免税捐款。

精选文章

about:blank 是什么意思
反恶意软件服务
AppData 文件夹
MongoDB Atlas 教程
更新 NPM 依赖
Python 获取时间
React 个人作品网站
撤销 Git Add
CMD 删除文件
Nano 怎么保存退出
打开 .dat 文件
Windows10 产品密钥
Windows 10 屏幕亮度
Python 字符串转数字
谷歌恐龙游戏
Git Clone 指定分支
媒体查询范围
OSI 七层网络
Git 删除分支
HTML5 模板
Node 最新版本
Git 切换分支
JSON 注释
Git 命令
CSS 使用 SVG 图片
JS 字符串反转
forEach 遍历数组
Event Loop 执行顺序
HTML 表格代码
学习编程

移动应用

我们的慈善机构