简洁的TypeScript之书

Simone Poggiali

简洁的TypeScript之书

《Concise TypeScript Book》全面而简洁地概述了 TypeScript 的功能。它提供了清晰的解释，涵盖了该语言最新版本中的所有方面，从强大的类型系统到高级功能。无论您是初学者还是经验丰富的开发人员，本书都是增强您对 TypeScript 的理解和熟练程度的宝贵资源。

本书完全免费且开源。

如果您发现这本 TypeScript 书籍很有价值并希望做出贡献，请考虑通过 PayPal 支持我的努力。谢谢！

翻译

本书已被翻译成多种语言版本，包括：

中文

下载和网站

您还可以下载 Epub 版本：

https://github.com/gibbok/typescript-book/tree/main/downloads

在线版本可在以下位置获得：

https://gibbok.github.io/typescript-book

目录表

介绍

欢迎来到简洁的TypeScript之书！本指南为您提供有效 TypeScript 开发的基本知识和实践技能。发现编写干净、健壮的代码的关键概念和技

术。无论您是初学者还是经验丰富的开发人员，本书都可以作为在项目中利用 TypeScript 强大功能的综合指南和便捷参考。

本书涵盖了 TypeScript 5.2。

关于作者

Simone Poggiali 是一位经验丰富的高级前端开发人员，自 90 年代以来就热衷于编写专业级代码。在他的国际职业生涯中，他为从初创公司到大型组织的广泛客户提供了众多项目。HelloFresh、Siemens、O2 和 Leroy Merlin 等著名公司都受益于他的专业知识和奉献精神。

您可以通过以下平台联系 Simone Poggiali：

- 领英: https://www.linkedin.com/in/simone-poggiali
- GitHub: https://github.com/gibbok
- 推特: https://twitter.com/gibbok_coding
- 电子邮箱: gibbok.coding gmail.com

TypeScript简介

什么是TypeScript?

TypeScript 是一种基于 JavaScript 构建的强类型编程语言。它最初由 Anders Hejlsberg 于 2012 年设计，目前由 Microsoft 作为开源项目开发和维护。

TypeScript 编译为 JavaScript，并且可以在任何 JavaScript 运行时（例如浏览器或服务器 Node.js）中执行。

TypeScript 支持多种编程范式，例如函数式、泛型、命令式和面向对象。TypeScript 既不是解释型语言，也不是编译型语言。

为什么选择 TypeScript?

TypeScript 是一种强类型语言，有助于防止常见的编程错误，并在程序执行之前避免某些类型的运行时错误。

强类型语言允许开发人员在数据类型定义中指定各种程序约束和行为，从而有助于验证软件的正确性并防止缺陷。这在大规模应用中尤其有价值。

TypeScript 的一些好处

- 静态类型，可选强类型
- 类型推断
- 能使用ES6和ES7的新功能
- 跨平台和跨浏览器兼容性 * IntelliSense 工具支持

TypeScript 和 JavaScript

TypeScript是用`.ts`或`.tsx`文件编写的, 而JavaScript是用`.js`或`.jsx`文件编写的。

扩展名为.tsx或.jsx的文件可以包含 JavaScript 语法扩展 JSX，该扩展在 React 中用于 UI 开发。

就语法而言，TypeScript 是 JavaScript (ECMAScript 2015) 的类型化超集。所有 JavaScript 代码都是有效的 TypeScript 代码，但反之则不然。

例如，考虑 JavaScript 文件中具有.js扩展名的函数，如下所示：

```
const sum = (a, b) => a + b;
```

该函数可以通过将文件扩展名更改为 .TypeScript 来转换和使用.ts。但是，如果同一个函数使用 TypeScript 类型进行注释，则未经编译就无法在任何 JavaScript 运行时中执行。如果未编译以下 TypeScript 代码，将会产生语法错误

```
const sum = (a: number, b: number): number => a + b;
```

TypeScript 旨在通过让开发人员使用类型注释定义意图来检测编译期间运行时可能发生的异常。此外，如果没有提供类型注释，TypeScript 也可以捕获问题。例如，以下代码片段未指定任何 TypeScript 类型：

```
const items = [{ x: 1 }, { x: 2 }];
const result = items.filter(item => item.y);
```

在这种情况下，TypeScript 检测到错误并报告：

```
类型 '{ x: number; }' 上不存在属性 'y' 。
```

TypeScript 的类型系统很大程度上受到 JavaScript 运行时行为的影响。例如，加法运算符 (+) 在 JavaScript 中可以执行字符串连接或数字加法，在 TypeScript 中以相同的方式建模：

```
const result = '1' + 1; // 结果是string类型
```

TypeScript 背后的团队经过深思熟虑，决定将 JavaScript 的异常使用标记为错误。例如，考虑以下有效的 JavaScript 代码：

```
const result = 1 + true; // 在JavaScript中, 结果等于2
```

但是，TypeScript 会抛出错误：

```
运算符"+"不能应用于类型"number"和"boolean"。
```

出现此错误的原因是 TypeScript 严格强制执行类型兼容性，在这种情况下，它标识了数字和布尔值之间的无效操作。

TypeScript 代码生成

TypeScript 编译器有两个主要职责：检查类型错误和编译为 JavaScript。这两个过程是相互独立的。类型不会影响 JavaScript 运行时中代码的执行，因为它们在编译过程中会被完全擦除。即使存在类型错误，TypeScript 仍然可以输出 JavaScript。以下是存在类型错误的 TypeScript 代码示例：

```
const add = (a: number, b: number): number => a + b;
const result = add('x', 'y'); // "字符串"类型的参数不可赋值
给"数字"类型的参数
```

但是，它仍然可以生成可执行的 JavaScript 输出：

```
'use strict';
const add = (a, b) => a + b;
const result = add('x', 'y'); // xy
```

无法在运行时检查 TypeScript 类型。例如：

```
interface Animal {
    name: string;
}
interface Dog extends Animal {
    bark: () => void;
}
interface Cat extends Animal {
    meow: () => void;
}
const makeNoise = (animal: Animal) => {
    if (animal instanceof Dog) {
        // "Dog"仅指一种类型，但在这里用作值。
        // ...
    }
};
```

由于编译后类型被删除，因此无法在 JavaScript 中运行此代码。为了在运行时识别类型，我们需要使用另一种机制。TypeScript 提供了多种选项，其中常见的一个是“标签联合（tagged union）”。例如：

```
interface Dog {
    kind: 'dog'; // 标签联合
    bark: () => void;
}
interface Cat {
    kind: 'cat'; // 标签联合
    meow: () => void;
}
```

```
type Animal = Dog | Cat;

const makeNoise = (animal: Animal) => {
    if (animal.kind === 'dog') {
        animal.bark();
    } else {
        animal.meow();
    }
};

const dog: Dog = {
    kind: 'dog',
    bark: () => console.log('bark'),
};
makeNoise(dog);
```

属性"kind"是一个可以在运行时用来区分 JavaScript 中的对象的值。

运行时的值也可能具有与类型声明中声明的类型不同的类型。例如，如果开发人员误解了 API 类型并对其进行了错误注释。

TypeScript 是 JavaScript 的超集，因此"class"关键字可以在运行时用作类型和值。

```
class Animal {
    constructor(public name: string) {}
}
class Dog extends Animal {
    constructor(public name: string, public bark: () =>
void) {
        super(name);
    }
}
class Cat extends Animal {
    constructor(public name: string, public meow: () =>
void) {
        super(name);
    }
}
type Mammal = Dog | Cat;
```

```
const makeNoise = (mammal: Mammal) => {
    if (mammal instanceof Dog) {
        mammal.bark();
    } else {
        mammal.meow();
    }
};

const dog = new Dog('Fido', () => console.log('bark'));
makeNoise(dog);
```

在 JavaScript 中，“类”具有”prototype”属性，“instanceof”运算符可用于测试构造函数的原型属性是否出现在对象原型链中的任何位置。

TypeScript 对运行时性能没有影响，因为所有类型都将被删除。然而，TypeScript 确实引入了一些构建时间开销。

现在的现代 JavaScript（降级）

TypeScript 可以将代码编译为自 ECMAScript 3 (1999) 以来任何已发布的 JavaScript 版本。这意味着 TypeScript 可以将代码从最新的 JavaScript 功能转换为旧版本，这一过程称为降级。这允许使用现代 JavaScript，同时保持与旧运行时环境的最大兼容性。

值得注意的是，在转换为旧版本 JavaScript 的过程中，TypeScript 可能会生成与本机实现相比会产生性能开销的代码。

以下是一些可以在 TypeScript 中使用的现代 JavaScript 功能：

- ECMAScript 模块，而不是 AMD 风格的”define”回调或 CommonJS 的”require”语句。
- 用类代替原型。
- 变量声明使用”let”或”const”而不是”var”。
- “for-of”循环或”.forEach”而不是传统的”for”循环。
- 用箭头函数代替函数表达式。
- 解构赋值。
- 简写属性/方法名称和计算属性名称。

- 默认函数参数。

通过利用这些现代 JavaScript 功能，开发人员可以在 TypeScript 中编写更具表现力和简洁的代码。

TypeScript 入门

安装

Visual Studio Code 为 TypeScript 语言提供了出色的支持，但不包含 TypeScript 编译器。要安装 TypeScript 编译器，您可以使用包管理器，例如 npm 或yarn：

```
npm install typescript --save-dev
```

或者

```
yarn add typescript --dev
```

确保提交生成的锁定文件，以确保每个团队成员使用相同版本的 TypeScript。

要运行TypeScript编译器，可以使用以下命令

```
npx tsc
```

或者

```
yarn tsc
```

建议按项目安装 TypeScript，而不是全局安装，因为它提供了更可预测的构建过程。但是，对于一次性情况，您可以使用以下命令：

```
npx tsc
```

或者安装到全局：

```
npm install -g typescript
```

如果您使用的是 Microsoft Visual Studio，则可以在 NuGet 中为 MSBuild 项目获取作为包的 TypeScript。在 NuGet 包管理器控制台中，运行以下命令：

```
Install-Package Microsoft.TypeScript.MSBuild
```

在 TypeScript 安装过程中，会安装两个可执行文件："tsc"作为 TypeScript 编译器，"tsserver"作为 TypeScript 独立服务器。独立服务器包含编译器和语言服务，编辑器和 IDE 可以利用它们来提供智能代码补全。

此外，还有几种兼容 TypeScript 的转译器可用，例如 Babel（通过插件）或 swc。这些转译器可用于将 TypeScript 代码转换为其他目标语言或版本。

配置

可以使用 tsc CLI 选项或利用位于项目根目录中名为 tsconfig.json 的专用配置文件来配置 TypeScript。

要生成预填充推荐设置的 tsconfig.json 文件，您可以使用以下命令：

```
tsc --init
```

在本地执行tsc命令时，TypeScript 将使用最近的 tsconfig.json 文件中指定的配置来编译代码。

以下是使用默认设置运行的 CLI 命令的一些示例：

```
tsc main.ts // 将一个特定的文件 (main.ts) 编译成 JavaScript
tsc src/*.ts // 将 'src' 文件夹下任意的 .ts 文件编译成
JavaScript
tsc app.ts util.ts --outfile index.js // 将 2 个 TypeScript
文件 (app.ts 和 util.ts) 编译成 1 个 JavaScript 文件
(index.js)
```

TypeScript 的配置文件

tsconfig.json 文件用于配置 TypeScript 编译器 (tsc)。通常，它与文件一起添加到项目的根目录中package.json。

注意：

- tsconfig.json 即使是 json 格式也接受注释。
- 建议使用此配置文件而不是命令行选项。

在以下链接中，您可以找到完整的文档及其配置示例：

https://www.typescriptlang.org/tsconfig

http://json.schemastore.org/tsconfig

以下列出了常见且有用的配置：

target

“target”属性用于指定 TypeScript 应发出/编译到哪个版本的 JavaScript ECMAScript 版本。对于现代浏览器，ES6是一个不错的选择，对于较旧的浏览器，建议使用ES5。

lib

“lib”属性用于指定编译时要包含哪些库文件。TypeScript 自动包含”目标”属性中指定功能的 API，但可以根据特定需求省略或选择特定库。例如，如果您正在开发服务器项目，则可以排除”DOM”库，该库仅在浏览器环境中有用。

strict

“strict”属性可以提供更强有力的保证并增强类型安全性。建议始终将此属性包含在项目的 tsconfig.json 文件中。启用”strict”属性允许 TypeScript：

- 触发每个源文件的代码使用”use strict”。
- 在类型检查过程中考虑”null”和”undefined”
- 当不存在类型注释时禁用”any”类型的使用。
- 在使用”this”表达式时引发错误，否则”this”会被视为任意类型。

module

“module”属性设置编译程序支持的模块系统。在运行时，模块加载器用于根据指定的模块系统定位并执行依赖项。 JavaScript 中最常见的模块加载器是用于服务器端应用程序的 Node.js 的CommonJS和用于基于浏览器的 Web 应用程序中的 AMD 模块的 RequireJS。 TypeScript 可以为各种模块系统生成代码，包括 UMD、System、ESNext、ES2015/ES6 和 ES2020。

注意：应根据目标环境和该环境中可用的模块加载机制来选择模块系统。

moduleResolution

“moduleResolution”属性指定模块解析策略。对现代TypeScript代码使用”node”，“classic”仅用于旧版本的 TypeScript（1.6 之前）。

esModuleInterop

“esModuleInterop”属性允许从未使用”default”属性导出的 CommonJS 模块导入默认值，此属性提供了一个兼容以确保生成的 JavaScript 的兼容性。启用此选项后，我们可以使用 `import MyLibrary from "my-library"` 而不是 `import * as MyLibrary from "my-library"`。

jsx

“jsx”属性仅适用于 ReactJS 中使用的 .tsx 文件，并控制 JSX 构造如何编译为 JavaScript。一个常见的选项是”preserve”，它将编译为 .jsx 文件，保持 JSX 不变，以便可以将其传递给 Babel 等不同工具进行进一步转换。

skipLibCheck

“skipLibCheck”属性将阻止 TypeScript 对整个导入的第三方包进行类型检查。此属性将减少项目的编译时间。TypeScript 仍会根据这些包提供的类型定义检查您的代码。

files

“files”属性向编译器指示必须始终包含在程序中的文件列表。

include

“include”属性向编译器指示我们想要包含的文件列表。此属性允许类似 glob 的模式，例如 “*” *表示任何子目录，*”” 表示任何文件名，“?” 表示可选字符。

exclude

“exclude”属性向编译器指示不应包含在编译中的文件列表。这可以包括”node_modules”等文件或测试文件 注意：tsconfig.json 允许注释。

importHelpers

TypeScript 在为某些高级或低级 JavaScript 功能生成代码时使用帮助程序代码。 默认情况下，这些助手会在使用它们的文件中复制。 `importHelpers` 选项从 `tslib` 模块导入这些帮助器，从而使 JavaScript 输出更加高效。

迁移到 TypeScript 的建议

对于大型项目，建议采用逐渐过渡的方式，其中 TypeScript 和 JavaScript 代码最初共存。只有小型项目才能一次性迁移到 TypeScript。

此转变的第一步是将 TypeScript 引入构建链过程。这可以通过使用”allowJs”编译器选项来完成，该选项允许 .ts 和 .tsx 文件与现有 JavaScript 文件共存。由于当 TypeScript 无法从 JavaScript 文件推断类型时，它会回退到变量的”any”类型，因此建议在迁移开始时在编译器选项中禁用”noImplicitAny”。

第二步是确保您的 JavaScript 测试与 TypeScript 文件一起工作，以便您可以在转换每个模块时运行测试。如果您正在使用 Jest，请考虑使用 ts-jest，它允许您使用 Jest 测试 TypeScript 项目。

第三步是在项目中包含第三方库的类型声明。 这些声明可以第三方库的类型声明文件或专门的声明包中找到，你能通过 https://www.typescriptlang.org/dt/search 搜索并安装它们。:

```
npm install --save-dev @types/package-name or yarn add --
dev @types/package-name.
```

第四步是使用自下而上的方法逐个模块地迁移，遵循从叶开始的依赖关系图。这个想法是开始转换不依赖于其他模块的模块。要可视化依赖关系图，您可以使用该madge工具。

有一些对于转换成 TypeScript 比较友好的模块（外部 API 或规范相关的实用函数和代码），比如Swagger、GraphQL 或 JSONSchema 自动生成 TypeScript 类型定义，并使用在您的项目中。

当没有可用的规范或官方架构时，您可以从原始数据生成类型，例如服务器返回的 JSON。但是，建议从规范而不是数据生成类型，以避免丢失边缘情况。

在迁移过程中，不要进行代码重构，而只专注于向模块添加类型。

第五步是启用”noImplicitAny”，这将强制所有类型都是已知和定义的，从而为您的项目提供更好的 TypeScript 体验。

在迁移过程中，您可以使用该@ts-check指令，该指令在 JavaScript 文件中启用 TypeScript 类型检查。该指令提供了宽松版本的类型检查，最初可用于识别 JavaScript 文件中的问题。当@ts-check包含在文件中

时，TypeScript 将尝试使用 JSDoc 风格的注释来推断定义。但是，仅在迁移的早期阶段考虑使用 JSDoc 注释。

考虑在你的tsconfig.json文件中将 `noEmitOnError` 设置为 false，即使报告错误，这也将允许您输出 JavaScript 源代码。

探索类型系统

TypeScript 的语言服务

TypeScript 的语言服务, 也被称为 tsserver，提供了各种功能，例如错误报告、诊断、保存时编译、重命名、跳转到定义、补全列表、签名帮助等。 它主要由集成开发环境 (IDE) 使用来提供 IntelliSense 支持。它与 Visual Studio Code 无缝集成，并由 Conquer of Completion (Coc) 等工具使用。

开发人员可以利用专用 API 并创建自己的自定义语言服务插件来增强 TypeScript 编辑体验。这对于实现特殊的 linting 功能或启用自定义模板语言的自动完成特别有用。

现实世界中的自定义插件的一个示例是”typescript-styled-plugin”，它为样式组件中的 CSS 属性提供语法错误报告和 IntelliSense 支持。

有关更多信息和快速入门指南，您可以参考 GitHub 上的官方 TypeScript Wiki： https://github.com/microsoft/TypeScript/wiki/

结构类型

TypeScript 基于结构类型系统。这意味着类型的兼容性和等效性由类型的实际结构或定义决定，而不是由其名称或声明位置决定，如 C# 或 C 等主要类型系统中那样。

TypeScript 的结构类型系统是基于 JavaScript 的动态 duck 类型系统在运行时的工作方式而设计的。

以下示例是有效的 TypeScript 代码。正如您所观察到的，“X”和”Y”具有相同的成员”a”，尽管它们具有不同的声明名称。类型由其结构决定，在这种情况下，由于结构相同，因此它们是兼容且有效的。

```
type X = {
    a: string;
};
type Y = {
    a: string;
};
const x: X = { a: 'a' };
const y: Y = x; // 有效
```

TypeScript 的基本比较规则

TypeScript 比较过程是递归的，并在任何级别嵌套的类型上执行。

如果”Y”至少具有与”X”相同的成员，则类型”X”与”Y”兼容。

```
type X = {
    a: string;
};
const y = { a: 'A', b: 'B' }; // 有效, 至少它拥有相同的成员 X
const r: X = y;
```

函数参数按类型进行比较，而不是按名称进行比较：

```
type X = (a: number) => void;
type Y = (a: number) => void;
let x: X = (j: number) => undefined;
let y: Y = (k: number) => undefined;
y = x; // 有效
x = y; // 有效
```

函数返回类型必须相同：

```
type X = (a: number) => undefined;
type Y = (a: number) => number;
let x: X = (a: number) => undefined;
```

```ts
let y: Y = (a: number) => 1;
y = x; // 无效
x = y; // 无效
```

源函数的返回类型必须是目标函数的返回类型的子类型：

```ts
let x = () => ({ a: 'A' });
let y = () => ({ a: 'A', b: 'B' });
x = y; // 有效
y = x; // 无效，缺少 b 成员
```

允许丢弃函数参数，因为这是 JavaScript 中的常见做法，例如使用“Array.prototype.map()”：

```ts
[1, 2, 3].map((element, _index, _array) => element + 'x');
```

因此，以下类型声明是完全有效的：

```ts
type X = (a: number) => undefined;
type Y = (a: number, b: number) => undefined;
let x: X = (a: number) => undefined;
let y: Y = (a: number) => undefined; // 缺少 b 参数
y = x; // 有效
```

源类型的任何附加可选参数都是有效的：

```ts
type X = (a: number, b?: number, c?: number) => undefined;
type Y = (a: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
y = x; // 有效
x = y; // 有效
```

目标类型的任何可选参数在源类型中没有对应的参数都是有效的并且不是错误：

```ts
type X = (a: number) => undefined;
type Y = (a: number, b?: number) => undefined;
let x: X = a => undefined;
let y: Y = a => undefined;
```

```
y = x; // 有效
x = y; // 有效
```

其余参数被视为无限系列的可选参数：

```
type X = (a: number, ...rest: number[]) => undefined;
let x: X = a => undefined; // 有效
```

如果重载签名与其实现签名兼容，则具有重载的函数有效：

```
function x(a: string): void;
function x(a: string, b: number): void;
function x(a: string, b?: number): void {
    console.log(a, b);
}
x('a'); // 有效
x('a', 1); // 有效

function y(a: string): void; // 无效，不兼容重载的签名
function y(a: string, b: number): void;
function y(a: string, b: number): void {
    console.log(a, b);
}
y('a');
y('a', 1);
```

如果源参数和目标参数可赋值给超类型或子类型（Bivariance 双变），则函数参数比较成功。

```
// 超类
class X {
    a: string;
    constructor(value: string) {
        this.a = value;
    }
}
// 子类
class Y extends X {}
// 子类
class Z extends X {}
```

```ts
type GetA = (x: X) => string;
const getA: GetA = x => x.a;

// 双变（Bivariance） 确实接收超类
console.log(getA(new X('x'))); // 有效
console.log(getA(new Y('Y'))); // 有效
console.log(getA(new Z('z'))); // 有效
```

枚举与数字具有可比性和有效性，反之亦然，但比较不同枚举类型的枚举值是无效的。

```ts
enum X {
    A,
    B,
}
enum Y {
    A,
    B,
    C,
}
const xa: number = X.A; // 有效
const ya: Y = 0; // 有效
X.A === Y.A; // 无效
```

类的实例需要对其私有成员和受保护成员进行兼容性检查：

```ts
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}

class Y {
    private a: string;
    constructor(value: string) {
        this.a = value;
    }
}
```

```
let x: X = new Y('y'); // 无效
```

比较检查不考虑不同的继承层次结构，例如：

```
class X {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
class Y extends X {
    public a: string;
    constructor(value: string) {
        super(value);
        this.a = value;
    }
}
class Z {
    public a: string;
    constructor(value: string) {
        this.a = value;
    }
}
let x: X = new X('x');
let y: Y = new Y('y');
let z: Z = new Z('z');
x === y; // 有效
x === z; // 有效即使 z 来自不同的继承层次结构
```

泛型根据应用泛型参数后的结果类型使用其结构进行比较，仅将最终结果作为非泛型类型进行比较。

```
interface X<T> {
    a: T;
}
let x: X<number> = { a: 1 };
let y: X<string> = { a: 'a' };
x === y; // 无效，因为最终结构中使用了类型参数
```

```ts
interface X<T> {}
const x: X<number> = 1;
const y: X<string> = 'a';
x === y; // 有效，因为最终结构中没有使用类型参数
```

当泛型未指定其类型参数时，所有未指定的参数都将被视为带有"any"的类型：

```ts
type X = <T>(x: T) => T;
type Y = <K>(y: K) => K;
let x: X = x => x;
let y: Y = y => y;
x = y; // 有效
```

记住：

```ts
let a: number = 1;
let b: number = 2;
a = b; // 有效，一切都可以赋值给自己

let c: any;
c = 1; // 有效，所有类型都可以赋值给any

let d: unknown;
d = 1; // 有效，所有类型都可以赋值给unknown

let e: unknown;
let e1: unknown = e; // 有效，unknown只能赋值给自己和any
let e2: any = e; // 有效
let e3: number = e; // 无效

let f: never;
f = 1; // 无效，所有类型不能赋值给never

let g: void;
let g1: any;
g = 1; // 无效，void不可赋值给除"any"之外的任何内容或从任何内容赋值
g = g1; // 有效
```

请注意，当启用”strictNullChecks”时，“null”和”undefined”的处理方式与”void”类似；否则，它们类似于”never”。

类型作为集合

在 TypeScript 中，类型是一组可能的值。该集合也称为类型的域。类型的每个值都可以被视为集合中的一个元素。类型建立了集合中的每个元素必须满足才能被视为该集合的成员的约束。TypeScript 的主要任务是检查并验证一组是否是另一组的子集。

TypeScript 支持各种类型的集合：

Set term	TypeScript	Notes
空集	never	“never” 包含除自身之外的任何类型
单元素集	undefined / null / literal type	
有限集	boolean / union	
无限集	string / number / object	
通用集	any / unknown	每个元素都是”any”的成员，每个集合都是它的子集/“unknown”是”any”的类型安全对应项

这里有几个例子：

TypScript	Set term	Example
never	∅ (空集)	const x: never = ‘x’; // 错误: ’string’类似不能赋值给’never’类型

TypScript	Set term	Example
Literal type	单元素集	type X = ‘X’;
		type Y = 7;
Value assignable to T	Value ∈ T (属于)	type XY = ‘X’ | ‘Y’;
		const x: XY = ‘X’;
T1 assignable to T2	T1 ⊆ T2 (子集)	type XY = ‘X’ | ‘Y’;
		const x: XY = ‘X’;
		const j: XY = ‘J’; // 类型“‘J’” 不能赋值给 ‘XY’ 类型.
T1 extends T2	T1 ⊆ T2 (子集)	type X = ‘X’ extends string ? true : false;
T1 | T2	T1 ∪ T2 (并集)	type XY = ‘X’ | ‘Y’;
		type JK = 1 | 2;
T1 & T2	T1 ∩ T2 (交集)	type X = { a: string }
		type Y = { b: string }
		type XY = X & Y
		const x: XY = { a: ‘a’, b: ‘b’ }
unknown	通用集	const x: unknown = 1

并集 (T1 | T2) 创建一个更广泛的集合（两者）：

```
type X = {
    a: string;
};
type Y = {
    b: string;
};
type XY = X | Y;
const r: XY = { a: 'a', b: 'x' }; // 有效
```

交集（T1 & T2）创建一个更窄的集合（仅共享）：

```
type X = {
    a: string;
};
type Y = {
    a: string;
    b: string;
};
type XY = X & Y;
const r: XY = { a: 'a' }; // 无效
const j: XY = { a: 'a', b: 'b' }; // 有效
```

在这种情况下，关键字extends可以被视为”的子集”。它为类型设置约束。与泛型一起使用的扩展将泛型视为无限集，并将其限制为更具体的类型。请注意，这extends与 OOP 意义上的层次结构无关（TypScript 中没有这个概念）。TypeScript 使用集合并且没有严格的层次结构，事实上，如下面的示例所示，两种类型可以重叠，而不会成为另一种类型的子类型（TypScript 考虑对象的结构和形状）。

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
interface Z extends Y {
    c: string;
}
```

```
const z: Z = { a: 'a', b: 'b', c: 'c' };
interface X1 {
    a: string;
}
interface Y1 {
    a: string;
    b: string;
}
interface Z1 {
    a: string;
    b: string;
    c: string;
}
const z1: Z1 = { a: 'a', b: 'b', c: 'c' };

const r: Z1 = z; // 有效
```

赋值类型：类型声明和类型断言

在 TypeScript 中可以通过不同的方式赋值类型：

类型声明

在下面的示例中，我们使用 x:X(“:Type”) 来声明变量 x 的类型。

```
type X = {
    a: string;
};

// 类型声明
const x: X = {
    a: 'a',
};
```

如果变量不是指定的格式，TypeScript 将报告错误。例如：

```
type X = {
    a: string;
};
```

```
const x: X = {
    a: 'a',
    b: 'b', // 错误: 对象字面量只能指定已知属性
};
```

类型断言

可以使用as关键字添加断言。这告诉编译器开发人员拥有有关类型的更多信息并消除可能发生的任何错误。

例如：

```
type X = {
    a: string;
};
const x = {
    a: 'a',
    b: 'b',
} as X;
```

在上面的示例中，使用 as 关键字将对象 x 断言为类型 X。这通知 TypeScript 编译器该对象符合指定的类型，即使它具有类型定义中不存在的附加属性 b。

类型断言在需要指定更具体类型的情况下非常有用，尤其是在使用 DOM 时。例如：

```
const myInput = document.getElementById('my_input') as
HTMLInputElement;
```

此处，类型断言 HTMLInputElement 用于告诉 TypeScript getElementById 的结果应被视为 HTMLInputElement。类型断言还可以用于重新映射键，如下面使用模板文字的示例所示：

```
type J<Type> = {
    [Property in keyof Type as `prefix_${string &
        Property}`]: () => Type[Property];
};
```

```
type X = {
    a: string;
    b: number;
};
type Y = J<X>;
```

在此示例中，类型 J 使用带有模板文字的映射类型来重新映射 Type 的键。它创建新属性，并在每个键上添加 prefix_ ，它们对应的值是返回原始属性值的函数。

值得注意的是，当使用类型断言时，TypeScript 不会执行多余的属性检查。因此，当预先知道对象的结构时，通常最好使用类型声明。

非空断言

此断言是使用后缀表达式!运算符应用的，它告诉 TypeScript 值不能为 null 或未定义。

```
let x: null | number;
let y = x!; // number
```

环境声明

环境声明是描述 JavaScript 代码类型的文件，它们的文件名格式为.d.ts.. 它们通常被导入并用于注释现有的 JavaScript 库或向项目中的现有 JS 文件添加类型。

许多常见的库类型可以在以下位置找到：
https://github.com/DefinitelyTyped/DefinitelyTyped/

```
npm install --save-dev @types/library-name
```

对于您定义的环境声明，您可以使用”三斜杠”引用导入：

```
/// <reference path="./library-types.d.ts" />
```

即使在 JavaScript 文件中，您也可以通过 `// @ts-check` 使用环境声明。

属性检测和多余属性检测

TypeScript 基于结构类型系统，但过多的属性检查是 TypeScript 的一个属性，它允许它检查对象是否具有类型中指定的确切属性。

例如，在将对象字面量赋值给变量或将它们作为参数传递给函数的多余属性时，会执行多余属性检查。

```
type X = {
    a: string;
};
const y = { a: 'a', b: 'b' };
const x: X = y; // 有效，因为结构类型
const w: X = { a: 'a', b: 'b' }; // 无效，因为多余属性检测
```

弱类型

当一个类型只包含一组全可选属性时，该类型被认为是弱类型：

```
type X = {
    a?: string;
    b?: string;
};
```

当没有重叠时，TypeScript 认为将任何内容赋值给弱类型是错误的，例如，以下会引发错误：

```
type Options = {
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;

fn({ c: 'c' }); // 无效
```

尽管不推荐，但如果需要，可以使用类型断言绕过此检查：

```ts
type Options = {
    a?: string;
    b?: string;
};
const fn = (options: Options) => undefined;
fn({ c: 'c' } as Options); // 有效
```

或者通过将unknown索引签名添加到弱类型：

```ts
type Options = {
    [prop: string]: unknown;
    a?: string;
    b?: string;
};

const fn = (options: Options) => undefined;
fn({ c: 'c' }); // 有效
```

严格的对象字面量检测 (Freshness)

严格的对象字面量检查（有时称为“新鲜度”）是 TypeScript 中的一项功能，有助于捕获多余或拼写错误的属性，否则这些属性在正常结构类型检查中会被忽视。

创建对象字面量时，TypeScript 编译器认为它是“新鲜的”。 如果将对象字面量分配给变量或作为参数传递，并且对象字面量指定目标类型中不存在的属性，则 TypeScript 将引发错误。

然而，当扩展对象文字或使用类型断言时，“新鲜感”就会消失。

下面举一些例子来说明：

```ts
type X = { a: string };
type Y = { a: string; b: string };

let x: X;
x = { a: 'a', b: 'b' }; // 严格的对象字面量检查：无效的赋值
var y: Y;
y = { a: 'a', bx: 'bx' }; // 严格的对象字面量检查：无效的赋值
```

```ts
const fn = (x: X) => console.log(x.a);

fn(x);
fn(y); // 类型加宽：没有错误，结构类型兼容

fn({ a: 'a', bx: 'b' }); // 严格的对象字面量检查：无效的参数

let c: X = { a: 'a' };
let d: Y = { a: 'a', b: '' };
c = d; // 类型加宽：没有严格的对象字面量检查
```

类型推断

当在以下期间未提供注释时，TypeScript 可以推断类型：

- 变量初始化
- 成员初始化。
- 设置参数的默认值。
- 函数返回类型。

例如：

```ts
let x = 'x'; // 推断的类型是 string
```

TypeScript 编译器分析值或表达式并根据可用信息确定其类型。

更高级的推断

当在类型推断中使用多个表达式时，TypeScript 会查找”最佳常见类型”。例如：

```ts
let x = [1, 'x', 1, null]; // 类型推断为: (string | number | null)[]
```

如果编译器找不到最佳通用类型，它将返回联合类型。例如：

```
let x = [new RegExp('x'), new Date()]; // 类型推断为:
(RegExp | Date)[]
```

TypeScript 利用基于变量位置的”上下文类型”来推断类型。在下面的示例中，编译器知道它的e类型是MouseEvent，因为在lib.d.ts 文件中定义了click事件类型，该文件包含各种常见 JavaScript 构造和 DOM 的环境声明：

```
window.addEventListener('click', function (e) {}); // e 的
类型被推断为 MouseEvent
```

类型加宽

类型加宽是 TypeScript 将类型分配给未提供类型注释时初始化的变量的过程。它允许从窄到宽的类型，但反之则不然。在以下示例中：

```
let x = 'x'; // TypeScript 推断为字符串，一种宽类型
let y: 'y' | 'x' = 'y'; // y 类型是字面量类型的联合
y = x; // 无效，字符串不可分配给类型 'x' | 'y'。
```

TypeScript根据初始化期间提供的单个值（x），将 `string` 赋予给 `x`，这是一个扩展的示例。

TypeScript 提供了控制加宽过程的方法，例如使用”const”。

常量

在声明变量时使用 `const` 关键字会导致 TypeScript 中的类型推断范围更窄。

For example:

```
const x = 'x'; // TypeScript 将 x 的类型推断为 'x'，一种较窄的
类型
let y: 'y' | 'x' = 'y';
y = x; // 有效: x的类型推断为 'x'
```

通过使用 const 声明变量 x，其类型被缩小为特定的文字值"x"。由于 x 的类型被缩小，因此可以将其赋值给变量 y 而不会出现任何错误。可以推断类型的原因是因为 const 变量无法重新分配，因此它们的类型可以缩小到特定的文字类型，在本例中为字面量类型"x"。

类型参数的 const 修饰符

从 TypeScript 5.0 版本开始，可以 const 在泛型类型参数上指定属性。这可以推断出最精确的类型。让我们看一个不使用 const 的示例：

```
function identity<T>(value: T) {
    // 这里没有const
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // 类型推断为:
{ a: string; b: string; }
```

正如您所看到的，属性a和b是通过 类型推断出来的string 。

现在，让我们看看 const 版本的差异：

```
function identity<const T>(value: T) {
    // 对类型参数使用 const 修饰符
    return value;
}
const values = identity({ a: 'a', b: 'b' }); // 类型推断为:
{ a: "a"; b: "b"; }
```

现在我们可以看到属性 a 和 b 被推断为const，因此 a 和 b被视为字符串文字而不仅仅是 string 类型。

常量断言

此功能允许您根据变量的初始化值声明具有更精确的文字类型的变量，这向编译器表明该值应被视为不可变文字。 这里有一些例子：

在单个属性上：

```
const v = {
    x: 3 as const,
};
v.x = 3;
```

在整个对象上：

```
const v = {
    x: 1,
    y: 2,
} as const;
```

这在定义元组的类型时特别有用：

```
const x = [1, 2, 3]; // number[]
const y = [1, 2, 3] as const; // 只读数组 [1, 2, 3]
```

显式类型注释

我们可以具体地传递一个类型，在下面的示例中，属性x的类型是number：

```
const v = {
    x: 1, // 推断类型: number (加宽了)
};
v.x = 3; // 有效
```

我们可以通过使用字面量类型的联合使类型注释更加具体：

```
const v: { x: 1 | 2 | 3 } = {
    x: 1, // x 现在是字面量的联合类型： 1 | 2 | 3
};
v.x = 3; // 有效
v.x = 100; // 无效的
```

类型缩小

类型缩小是 TypeScript 中的一个过程，其中一般类型缩小为更具体的类型。当 TypeScript 分析代码并确定某些条件或操作可以细化类型信息时，就会发生这种情况。

缩小类型可以通过不同的方式发生，包括：

条件

通过使用条件语句（比如 `if` 或 `switch`），TypeScript 可以根据条件的结果缩小类型范围。例如：

```
let x: number | undefined = 10;

if (x !== undefined) {
    x += 100; // 由于条件判断，类型被缩小为 number
}
```

抛错或者返回

抛出错误或从分支提前返回可用于帮助 TypeScript 缩小类型范围。例如：

```
let x: number | undefined = 10;

if (x === undefined) {
    throw 'error';
}
x += 100;
```

在 TypeScript 中缩小类型范围的其他方法包括：

- `instanceof` 操作: 用于检查对象是否是特定类的实例。
- `in` 操作: 用于检查对象中是否存在属性。
- `typeof` 操作: 用于在运行时检查值的类型。
- 内部函数，比如: `Array.isArray()`: 用于检查值是否为数组。

可区分联合

使用”可区分联合”是 TypeScript 中的一种模式，其中向对象添加显式”标签”以区分联合内的不同类型。该模式也称为”标记联合”。在以下示例中，“tag”由属性”type”表示：

```
type A = { type: 'type_a'; value: number };
type B = { type: 'type_b'; value: string };

const x = (input: A | B): string | number => {
    switch (input.type) {
        case 'type_a':
            return input.value + 100; // 类型为 A
        case 'type_b':
            return input.value + 'extra'; // 类型为 B
    }
};
```

用户定义的类型保护

在 TypeScript 无法确定类型的情况下，可以编写一个称为”用户定义类型保护”的辅助函数。在下面的示例中，我们将在应用某些过滤后利用类型谓词来缩小类型范围：

```
const data = ['a', null, 'c', 'd', null, 'f'];

const r1 = data.filter(x => x != null); // 类型为 (string | null)[], TypeScript 不能准确推断类型

const isValid = (item: string | null): item is string => item !== null; // 自定义类型保护

const r2 = data.filter(isValid); // 类型现在为 string[], 通过使用断言类型保护，我们能够缩小类型
```

原始类型

TypeScript 支持 7 种基本类型。原始数据类型是指不是对象并且没有任何与其关联的方法的类型。在 TypeScript 中，所有原始类型都是不可变的，这意味着它们的值一旦分配就无法更改。

string

原始 `string` 类型存储文本数据，并且值始终是双引号或单引号的。

```
const x: string = 'x';
const y: string = 'y';
```

如果字符串被反引号 (`) 字符包围，则字符串可以跨越多行：

```
let sentence: string = `xxx,
   yyy`;
```

boolean

TypeScript 中的数据 `boolean` 类型存储二进制值，或者true或false。

```
const isReady: boolean = true;
```

number

TypeScript 中的数据类型 `number` 用 64 位浮点值表示。类型 `number` 可以表示整数和分数。TypeScript 还支持十六进制、二进制和八进制，例如：

```
const decimal: number = 10;
const hexadecimal: number = 0xa00d; // 十六进制数以 0x 开始
const binary: number = 0b1010; // 二进制数以 0b 开始
const octal: number = 0o633; // 八进制数以 0o 开始
```

bigInt

bigInt 表示无法用 number 表示的非常大的数值 (253 – 1)。

bigInt 可以通过调用内置函数 BigInt() 或添加 n 到任何整数数字字面量的末尾来创建：

```
const x: bigint = BigInt(9007199254740991);
const y: bigint = 9007199254740991n;
```

笔记：

- bigInt 值不能与 number 混合，也不能与内置的 Math 一起使用，它们必须强制为相同的类型。
- 仅当目标配置为 ES2020 或更高版本时，“bigInt”值才可用。

symbol

JavaScript 有一个原始函数 Symbol()，它创建一个全局唯一的引用。

```
let sym = Symbol('x'); // symbol 类型
```

null and undefined

null和 undefined 类型都表示没有值或不存在任何值。

undefined 类型意味着该值未分配或初始化，或者指示无意中缺少值。

null 类型意味着我们知道该字段没有值，因此值不可用，这表明故意不存在值。

Array

array 是一种可以存储多个相同类型或不同类型的值的数据类型。可以使用以下语法定义它：

```ts
const x: string[] = ['a', 'b'];
const y: Array<string> = ['a', 'b'];
const j: Array<string | number> = ['a', 1, 'b', 2];
```

TypeScript 使用以下语法支持只读数组：

```ts
const x: readonly string[] = ['a', 'b']; // 只读修饰符
const y: ReadonlyArray<string> = ['a', 'b'];
const j: ReadonlyArray<string | number> = ['a', 1, 'b',
2];
j.push('x'); // 有效
```

TypeScript 支持数组和只读数组：

```ts
const x: [string, number] = ['a', 1];
const y: readonly [string, number] = ['a', 1];
```

any

数据 any 类型字面上代表”任何”值，当 TypeScript 无法推断类型或未指定时，它是默认值。

使用 any 时，TypeScript编译器会跳过类型检查，因此 any 使用时不存在类型安全。通常，当发生错误时不要使用 any 静默编译器，而是专注于修复错误，因为使用 any 它可能会破坏契约，并且我们会失去 TypeScript 自动完成的好处。

在从 JavaScript 逐步迁移到 TypeScript 的过程中，该 any 类型可能很有用，因为它可以让编译器保持沉默。

对于新项目，请使用 TypeScript 配置 noImplicitAny ，该配置使 TypeScript 能够在any使用或推断时发出错误。

any 通常是错误的来源，它可以掩盖类型的实际问题。尽可能避免使用它。

类型注释

在使用 `var` 、 `let` 和 `const` 声明变量时，可以选择添加类型：

```
const x: number = 1;
```

TypeScript 在推断类型方面做得很好，尤其是简单类型时，因此在大多数情况下这些声明是不必要的。

在函数上可以向参数添加类型注释：

```
function sum(a: number, b: number) {
    return a + b;
}
```

以下是使用匿名函数（所谓的 lambda 函数）的示例：

```
const sum = (a: number, b: number) => a + b;
```

当参数存在默认值时可以避免这些注释：

```
const sum = (a = 10, b: number) => a + b;
```

可以将返回类型注释添加到函数中：

```
const sum = (a = 10, b: number): number => a + b;
```

这对于更复杂的函数尤其有用，因为在实现之前编写显式返回类型可以帮助更好地思考该函数。

通常考虑注释类型签名，但不注释主体局部变量，并始终将类型添加到对象字面量中。

可选属性

对象可以通过在属性名称末尾添加问号 ? 来指定可选属性：

```ts
type X = {
    a: number;
    b?: number; // 可选的
};
```

当属性是可选的时，可以指定默认值

```ts
type X = {
    a: number;
    b?: number;
};
const x = ({ a, b = 100 }: X) => a + b;
```

只读属性

是否可以通过使用修饰符来防止对属性进行写入，readonly 以确保该属性不能被重写，但不提供任何完全不变性的保证：

```ts
interface Y {
    readonly a: number;
}

type X = {
    readonly a: number;
};

type J = Readonly<{
    a: number;
}>;

type K = {
    readonly [index: number]: string;
};
```

索引签名

在 TypeScript 中，我们可以使用 `string` 、`number` 和 `symbol` 作为索引签名：

```
type K = {
    [name: string | number]: string;
};
const k: K = { x: 'x', 1: 'b' };
console.log(k['x']);
console.log(k[1]);
console.log(k['1']); // 同 k[1] 的结果相同
```

请注意，JavaScript 会自动将 `number` 的索引转换相同值的 'string'索引，比如 `k[1]` 和 `k["1"]` 返回相同值。

扩展类型

可以扩展 `interface`（从另一种类型复制成员）：

```
interface X {
    a: string;
}
interface Y extends X {
    b: string;
}
```

还可以从多种 `interface` 进行扩展：

```
interface A {
    a: string;
}
interface B {
    b: string;
}
interface Y extends A, B {
```

```
    y: string;
}
```

该 extends 关键字仅适用于 interface，因为 type 使用交集：

```
type A = {
    a: number;
};
type B = {
    b: number;
};
type C = A & B;
```

可以使用 interface 来扩展类 type，但反之则不然：

```
type A = {
    a: string;
};
interface B extends A {
    b: string;
}
```

字面量类型

文字类型是来自集体类型的单个元素集，它定义了一个非常精确的值，即 JavaScript 原始数据。

TypeScript 中的文字类型是数字、字符串和布尔值。

示例如下：

```
const a = 'a'; //  字符串字面量类型
const b = 1; //  数字字面量类型
const c = true; //  布尔字面量类型
```

字符串、数字和布尔字面量类型用于联合、类型保护和类型别名。在下面的示例中，您可以看到类型别名联合，O可以是指定的唯一值，而不是任何其他字符串：

```ts
type O = 'a' | 'b' | 'c';
```

字面量推断

字面量推断是 TypeScript 中的一项功能，允许根据变量或参数的值推断其类型。

在下面的示例中，我们可以看到 TypeScript 认为x文字类型是因为该值以后不能随时更改，而y被推断为字符串，因为它以后可以随时修改。

```ts
const x = 'x'; // x 为字面量类型, 因为值不能改变
let y = 'y'; // string, 我们能改变这个值
```

在下面的示例中，我们可以看到 `o.x` 被推断为 `string`（而不是字面量的a），因为 TypeScript 认为该值可以在以后随时更改。

```ts
type X = 'a' | 'b';

let o = {
    x: 'a', // 这是一个更宽的 string
};

const fn = (x: X) => `${x}-foo`;

console.log(fn(o.x)); // 'string' 类型的参数不能赋值给 'X' 类型的参数
```

正如你所看到的代码在传递 `o.x` 给 `fn` 作为一个狭窄类型时，抛出了一个错误。

我们能通过使用 `const` 或者 `X` 来借助类型推断解决这个问题：

```ts
let o = {
    x: 'a' as const,
};
```

or:

```
let o = {
    x: 'a' as X,
};
```

严格空检查

strictNullChecks 是一个 TypeScript 编译器选项，强制执行严格的 null 检查。启用此选项后，只有在变量和参数已使用联合类型 null | undefined 显式声明为该类型时，才可以对其进行赋值null 或者 undefined。如果变量或参数未显式声明为可为空，TypeScript 将生成错误以防止潜在的运行时错误。

枚举

在 TypeScript 中，枚举是一组命名常量值。

```
enum Color {
    Red = '#ff0000',
    Green = '#00ff00',
    Blue = '#0000ff',
}
```

枚举可以用不同的方式定义：

数字枚举

在 TypeScript 中，数字枚举是一个枚举，其中每个常量都分配有一个数值，默认从 0 开始。

```
enum Size {
    Small, // 值从 0 开始
    Medium,
    Large,
}
```

可以通过显式分配来指定自定义值：

```
enum Size {
    Small = 10,
    Medium,
    Large,
}
console.log(Size.Medium); // 11
```

字符串枚举

在 TypeScript 中，字符串枚举是每个常量都分配有一个字符串值的枚举。

```
enum Language {
    English = 'EN',
    Spanish = 'ES',
}
```

注意：TypeScript 允许使用异构枚举，其中字符串和数字成员可以共存。

常量枚举

TypeScript 中的常量枚举是一种特殊类型的枚举，其中所有值在编译时都是已知的，并且在使用枚举的任何地方都会内联，从而产生更高效的代码。

```
const enum Language {
    English = 'EN',
    Spanish = 'ES',
}
console.log(Language.English);
```

将被编译成：

```
console.log('EN' /* Language.English */);
```

注意：常量枚举具有硬编码值，擦除枚举，这在独立库中可能更有效，但通常是不可取的。此外，常量枚举不能有计算成员。

反向映射

在 TypeScript 中，枚举中的反向映射是指从值中检索枚举成员名称的能力。默认情况下，枚举成员具有从名称到值的正向映射，但可以通过为每个成员显式设置值来创建反向映射。当您需要按枚举成员的值查找枚举成员，或者需要迭代所有枚举成员时，反向映射非常有用。需要注意的是，只有数字类型的枚举成员会生成反向映射，字符串类型的枚举成员则不会。

以下枚举：

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    F = 'fail',
}
```

编译为：

```
'use strict';
var Grade;
(function (Grade) {
    Grade[(Grade['A'] = 90)] = 'A';
    Grade[(Grade['B'] = 80)] = 'B';
    Grade[(Grade['C'] = 70)] = 'C';
    Grade['F'] = 'fail';
})(Grade || (Grade = {}));
```

由此可见，对数字类型的枚举成员，可以从枚举值映射回枚举名称，但对字符串类型的枚举成员无法这样做。

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
```

```
    F = 'fail',
}
const myGrade = Grade.A;
console.log(Grade[myGrade]); // A
console.log(Grade[90]); // A

const failGrade = Grade.F;
console.log(failGrade); // fail
console.log(Grade[failGrade]); // 因为索引表达式的类型不是
'number'，所以元素是隐式的 'any' 类型。
```

环境枚举

TypeScript 中的环境枚举是一种在声明文件 (*.d.ts) 中定义的枚举类型，没有关联的实现。它允许您定义一组命名常量，这些常量可以在不同文件中以类型安全的方式使用，而无需在每个文件中导入实现细节。

计算成员和常量成员

在 TypeScript 中，计算成员是枚举的成员，其值在运行时计算，而常量成员是其值在编译时设置且在运行时无法更改的成员。常规枚举中允许使用计算成员，而常规枚举和常量枚举中都允许使用常量成员。

```
// 常量成员
enum Color {
    Red = 1,
    Green = 5,
    Blue = Red + Green,
}
console.log(Color.Blue); // 6 编译时生成

// 计算成员
enum Color {
    Red = 1,
    Green = Math.pow(2, 2),
    Blue = Math.floor(Math.random() * 3) + 1,
}
console.log(Color.Blue); // 运行时生成的随机数
```

枚举由包含其成员类型的联合表示。每个成员的值可以通过常量或非常量表达式确定，拥有常量值的成员被分配字面量类型。为了说明这一点，请考虑类型 E 及其子类型 E.A、E.B 和 E.C 的声明。在本例中，E 表示联合 E.A | E.B | E.C 。

```
const identity = (value: number) => value;

enum E {
    A = 2 * 5, // 数字字面量
    B = 'bar', // 字符串字面量
    C = identity(42), // 不透明计算
}

console.log(E.C); //42
```

缩小范围

TypeScript 缩小范围是细化条件块内变量类型的过程。这在使用联合类型时很有用，其中一个变量可以有多个类型。

TypeScript 可识别多种缩小类型范围的方法：

typeof 类型保护

typeof 类型保护是 TypeScript 中的一种特定类型保护，它根据变量的内置 JavaScript 类型检查变量的类型。

```
const fn = (x: number | string) => {
    if (typeof x === 'number') {
        return x + 1; // x 是数字
    }
    return -1;
};
```

真实性缩小

TypeScript 中的真实性缩小是通过检查变量是真还是假来相应地缩小其类型来实现的。

```
const toUpperCase = (name: string | null) => {
    if (name) {
        return name.toUpperCase();
    } else {
        return null;
    }
};
```

相等缩小

TypeScript 中的相等缩小通过检查变量是否等于特定值来相应缩小其类型。

它与switch语句和等号运算符（例如===、!==、==和!=）结合使用来缩小类型范围。

```
const checkStatus = (status: 'success' | 'error') => {
    switch (status) {
        case 'success':
            return true
        case 'error':
            return null
    }
};
```

In运算符缩小

TypeScript 中的 in 运算符缩小范围是一种根据变量类型中是否存在属性来缩小变量类型的方法。

```
type Dog = {
    name: string;
    breed: string;
```

```
};

type Cat = {
    name: string;
    likesCream: boolean;
};

const getAnimalType = (pet: Dog | Cat) => {
    if ('breed' in pet) {
        return 'dog';
    } else {
        return 'cat';
    }
};
```

instanceof 缩小

TypeScript 中的 instanceof 运算符缩小是一种根据变量的构造函数缩小变量类型的方法，方法是检查对象是否是某个类或接口的实例。

```
class Square {
    constructor(public width: number) {}
}
class Rectangle {
    constructor(
        public width: number,
        public height: number
    ) {}
}
function area(shape: Square | Rectangle) {
    if (shape instanceof Square) {
        return shape.width * shape.width;
    } else {
        return shape.width * shape.height;
    }
}
const square = new Square(5);
const rectangle = new Rectangle(5, 10);
```

```
console.log(area(square)); // 25
console.log(area(rectangle)); // 50
```

赋值

使用赋值缩小 TypeScript 是一种根据分配给变量的值来缩小变量类型的方法。当为变量分配值时，TypeScript 会根据分配的值推断其类型，并缩小变量的类型以匹配推断的类型。

```
let value: string | number;
value = 'hello';
if (typeof value === 'string') {
    console.log(value.toUpperCase());
}
value = 42;
if (typeof value === 'number') {
    console.log(value.toFixed(2));
}
```

控制流分析

TypeScript 中的控制流分析是一种静态分析代码流以推断变量类型的方法，允许编译器根据分析结果根据需要缩小这些变量的类型。

在 TypeScript 4.4 之前，代码流分析仅适用于 if 语句中的代码，但从 TypeScript 4.4 开始，它还可以应用于条件表达式和通过 const 变量间接引用的判别式属性访问。

例如：

```
const f1 = (x: unknown) => {
    const isString = typeof x === 'string';
    if (isString) {
        x.length;
    }
};
```

```
const f2 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar';
bar: number }
) => {
    const isFoo = obj.kind === 'foo';
    if (isFoo) {
        obj.foo;
    } else {
        obj.bar;
    }
};
```

一些未发生缩小的示例：

```
const f1 = (x: unknown) => {
    let isString = typeof x === 'string';
    if (isString) {
        x.length; // 错误，没有缩小，因为 isString 不是常量
    }
};

const f6 = (
    obj: { kind: 'foo'; foo: string } | { kind: 'bar';
bar: number }
) => {
    const isFoo = obj.kind === 'foo';
    obj = obj;
    if (isFoo) {
        obj.foo; // 错误，没有缩小，因为 obj 在函数体中被赋值
    }
};
```

注意：在条件表达式中最多分析五个间接级别。

类型谓词

TypeScript 中的类型谓词是返回布尔值的函数，用于将变量的类型缩小为更具体的类型。

```
const isString = (value: unknown): value is string =>
typeof value === 'string';

const foo = (bar: unknown) => {
    if (isString(bar)) {
        console.log(bar.toUpperCase());
    } else {
        console.log('not a string');
    }
};
```

可区分联合

TypeScript 中的可区分联合是一种联合类型，它使用称为判别式的公共属性来缩小联合的可能类型集。

```
type Square = {
    kind: 'square'; // 判别式
    size: number;
};

type Circle = {
    kind: 'circle'; // 判别式
    radius: number;
};

type Shape = Square | Circle;

const area = (shape: Shape) => {
    switch (shape.kind) {
        case 'square':
            return Math.pow(shape.size, 2);
```

```
        case 'circle':
            return Math.PI * Math.pow(shape.radius, 2);
    }
};

const square: Square = { kind: 'square', size: 5 };
const circle: Circle = { kind: 'circle', radius: 2 };

console.log(area(square)); // 25
console.log(area(circle)); // 12.566370614359172
```

never 类型

当变量缩小为不能包含任何值的类型时，TypeScript 编译器将推断该变量必须属于该never类型。这是因为 never 类型代表永远无法生成的值。

```
const printValue = (val: string | number) => {
    if (typeof val === 'string') {
        console.log(val.toUpperCase());
    } else if (typeof val === 'number') {
        console.log(val.toFixed(2));
    } else {
        // val 在这里的类型为 never，因为它只能是字符串或数字
        const neverVal: never = val;
        console.log(`Unexpected value: ${neverVal}`);
    }
};
```

详尽性检查

详尽性检查是 TypeScript 中的一项功能，可确保在 `switch` 语句或 `if` 语句中处理可区分联合的所有可能情况。

```
type Direction = 'up' | 'down';
```

```
const move = (direction: Direction) => {
    switch (direction) {
        case 'up':
            console.log('Moving up');
            break;
        case 'down':
            console.log('Moving down');
            break;
        default:
            const exhaustiveCheck: never = direction;
            console.log(exhaustiveCheck); // 这行永远不会被执
行
    }
};
```

该 never 类型用于确保默认情况是详尽的，并且如果将新值添加到 Direction 类型而未在 switch 语句中进行处理，则 TypeScript 将引发错误。

对象类型

在 TypeScript 中，对象类型描述对象的形状。它们指定对象属性的名称和类型，以及这些属性是必需的还是可选的。

在 TypeScript 中，您可以通过两种主要方式定义对象类型：

通过指定对象属性的名称、类型和可选性来定义对象的形状的接口。

```
interface User {
    name: string;
    age: number;
    email?: string;
}
```

类型别名与接口类似，定义了对象的形状。但是，它还可以基于现有类型或现有类型的组合创建新的自定义类型。这包括定义联合类型、交集类型和其他复杂类型。

```ts
type Point = {
    x: number;
    y: number;
};
```

也可以匿名定义类型：

```ts
const sum = (x: { a: number; b: number }) => x.a + x.b;
console.log(sum({ a: 5, b: 1 }));
```

元组类型（匿名）

元组类型是一种表示具有固定数量的元素及其相应类型的数组的类型。元组类型以固定顺序强制执行特定数量的元素及其各自的类型。当您想要表示具有特定类型的值的集合时，元组类型非常有用，其中数组中每个元素的位置都有特定的含义。

```ts
type Point = [number, number];
```

命名元组类型（已标记）

元组类型可以包含每个元素的可选标签或名称。 这些标签用于提高可读性和工具帮助，不会影响您可以使用它们执行的操作。

```ts
type T = string;
type Tuple1 = [T, T];
type Tuple2 = [a: T, b: T];
type Tuple3 = [a: T, T]; // 命名元组加匿名元组
```

固定长度元组

固定长度元组是一种特定类型的元组，它强制执行特定类型的固定数量的元素，并且一旦定义元组就不允许对其长度进行任何修改。

当您需要表示具有特定数量的元素和特定类型的值的集合，并且您希望确保元组的长度和类型不会无意中更改时，固定长度元组非常有用。

```
const x = [10, 'hello'] as const;
x.push(2); // 错误
```

联合类型

联合类型是一种表示值的类型，该值可以是多种类型之一。联合类型使用 | 表示 每种可能类型之间的符号。

```
let x: string | number;
x = 'hello'; // 有效
x = 123; // 有效
```

交集类型

交集类型是表示具有两种或多种类型的所有属性的值的类型。交叉类型在每种类型之间使用 & 符号表示。

```
type X = {
    a: string;
};

type Y = {
    b: string;
};

type J = X & Y; // 交集

const j: J = {
    a: 'a',
    b: 'b',
};
```

类型索引

类型索引是指能够通过预先未知的键来定义可以索引的类型，使用索引签名来指定未显式声明的属性的类型。

```
type Dictionary<T> = {
    [key: string]: T;
};
const myDict: Dictionary<string> = { a: 'a', b: 'b' };
console.log(myDict['a']); // 返回 a
```

值的类型

TypeScript 中的”Type from Value”是指通过类型推断从值或表达式自动推断出类型。

```
const x = 'x'; // TypeScript 可以自动推断变量的类型是 string
```

Func 返回值的类型

Func Return 中的类型是指根据函数的实现自动推断函数的返回类型的能力。这允许 TypeScript 无需显式类型注释即可确定函数返回值的类型。

```
const add = (x: number, y: number) => x + y; // TypeScript 可以推断函数的返回类型是数字
```

模块的类型

模块的类型是指使用模块的导出值自动推断其类型的能力。当模块导出特定类型的值时，TypeScript 可以使用该信息在将该值导入到另一个模块时自动推断该值的类型。

```ts
export const add = (x: number, y: number) => x + y;
// index.ts
import { add } from 'calc';
const r = add(1, 2); // r 是 number 类型
```

映射类型

TypeScript 中的映射类型允许您通过使用映射函数转换每个属性来基于现有类型创建新类型。通过映射现有类型，您可以创建以不同格式表示相同信息的新类型。要创建映射类型，您可以使用运算符访问现有类型的属性 keyof ，然后更改它们以生成新类型。在以下示例中：

```ts
type MyMappedType<T> = {
    [P in keyof T]: T[P][];
};
type MyType = {
    foo: string;
    bar: number;
};
type MyNewType = MyMappedType<MyType>;
const x: MyNewType = {
    foo: ['hello', 'world'],
    bar: [1, 2, 3],
};
```

我们定义 MyMappedType 来映射 T 的属性，创建一个新类型，其中每个属性都是其原始类型的数组。使用它，我们创建 MyNewType 来表示与 MyType 相同的信息，但每个属性都是一个数组。

映射类型修饰符

TypeScript 中的映射类型修饰符支持对现有类型中的属性进行转换：

- `readonly` 或 `+readonly`：这会将映射类型中的属性呈现为只读。
- `-readonly`：这允许映射类型中的属性是可变的。
- `?`：这将映射类型中的属性指定为可选。

例子：

```
type ReadOnly<T> = { readonly [P in keyof T]: T[P] }; // 所有属性标记为只读

type Mutable<T> = { -readonly [P in keyof T]: T[P] }; // 所有标记为可变的属性

type MyPartial<T> = { [P in keyof T]?: T[P] }; // 所有标记为可选的属性
```

条件类型

条件类型是一种创建依赖于条件的类型的方法，其中要创建的类型是根据条件的结果确定的。它们是使用 extends 关键字和三元运算符来定义的，以便有条件地在两种类型之间进行选择。

```
type IsArray<T> = T extends any[] ? true : false;

const myArray = [1, 2, 3];
const myNumber = 42;

type IsMyArrayAnArray = IsArray<typeof myArray>; // true 类型
type IsMyNumberAnArray = IsArray<typeof myNumber>; // false 类型
```

分配条件类型

分布式条件类型是一种功能，通过单独对联合的每个成员应用转换，允许类型分布在类型的联合上。当使用映射类型或高阶类型时，这尤其有用。

```
type Nullable<T> = T extends any ? T | null : never;
type NumberOrBool = number | boolean;
```

```
type NullableNumberOrBool = Nullable<NumberOrBool>; // number | boolean | null
```

infer 条件类型中的类型推断

infer 关键字在条件类型中使用，用于从依赖于泛型参数的类型中推断（提取）泛型参数的类型。这允许您编写更灵活且可重用的类型定义。

```
type ElementType<T> = T extends (infer U)[] ? U : never;
type Numbers = ElementType<number[]>; // number
type Strings = ElementType<string[]>; // string
```

预定义条件类型

在 TypeScript 中，预定义的条件类型是语言提供的内置条件类型。它们旨在根据给定类型的特征执行常见的类型转换。

Exclude<UnionType, ExcludedType>: 此类型从 Type 中删除可分配给 ExcludedType 的所有类型。

Extract<Type, Union>: 此类型从 Union 中提取可分配给 Type 的所有类型。

NonNullable<Type>: 此类型从 Type 中删除 null 和 undefined。

ReturnType<Type>: 此类型提取函数 Type 的返回类型。

Parameters<Type>: 该类型提取函数类型的参数类型。

Required<Type>: 此类型使 Type 中的所有属性成为必需。

Partial<Type>: 此类型使 Type 中的所有属性都是可选的。

Readonly<Type>: 此类型使 Type 中的所有属性变为只读。

模板联合类型

模板联合类型可用于合并和操作类型系统内的文本，例如：

```
type Status = 'active' | 'inactive';
type Products = 'p1' | 'p2';
type ProductId = `id-${Products}-${Status}`; // "id-p1-active" | "id-p1-inactive" | "id-p2-active" | "id-p2-inactive"
```

任意类型

any 类型是一种特殊类型（通用超类型），可用于表示任何类型的值（基元、对象、数组、函数、错误、符号）。它通常用于编译时未知值类型的情况，或者使用来自没有 TypeScript 类型的外部 API 或库的值时。

通过使用任何类型，您向 TypeScript 编译器指示值应该不受任何限制地表示。为了最大限度地提高代码中的类型安全性，请考虑以下事项：

- 将 any 的使用限制在类型确实未知的特定情况下。
- 不要从函数返回 any 类型，因为使用该函数会在代码中失去类型安全性，从而削弱类型安全性。
- 如果您需要使编译器保持沉默，请使用 @ts-ignore 而不是 any。

```
let value: any;
value = true; // 有效
value = 7; // 有效
```

未知类型

在 TypeScript 中，未知类型表示未知类型的值。与允许任何类型值的 any 类型不同，unknown 需要在以特定方式使用它之前进行类型检查或断言，因此在未首先断言或缩小到更具体的类型的情况下，不允许对 unknown 进行任何操作 。

unknown 类型只能分配给任何类型和未知类型本身，它是any 的类型安全替代方案。

```
let value: unknown;

let value1: unknown = value; // 有效
let value2: any = value; // 有效
let value3: boolean = value; // 无效
let value4: number = value; // 无效

const add = (a: unknown, b: unknown): number | undefined
=>
    typeof a === 'number' && typeof b === 'number' ? a + b
: undefined;
console.log(add(1, 2)); // 3
console.log(add('x', 2)); // undefined
```

空类型

void 类型用于指示函数不返回值。

```
const sayHello = (): void => {
    console.log('Hello!');
};
```

Never类型

never 类型表示从未出现过的值。它用于表示从不返回或抛出错误的函数或表达式。

例如无限循环：

```
const infiniteLoop = (): never => {
    while (true) {
        // 做点什么
```

```
    }
};
```

抛出错误：

```
const throwError = (message: string): never => {
    throw new Error(message);
};
```

never 类型对于确保类型安全和捕获代码中的潜在错误很有用。当与其他类型和控制流语句结合使用时，它可以帮助 TypeScript 分析和推断更精确的类型，例如：

```
type Direction = 'up' | 'down';
const move = (direction: Direction): void => {
    switch (direction) {
        case 'up':
            // 向上移动
            break;
        case 'down':
            // 向下移动
            break;
        default:
            const exhaustiveCheck: never = direction;
            throw new Error(`Unhandled direction:
${exhaustiveCheck}`);
    }
};
```

接口及类型

通用语法

在 TypeScript 中，接口定义对象的结构，指定对象必须具有的属性或方法的名称和类型。在 TypeScript 中定义接口的常用语法如下：

```
interface InterfaceName {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
}
```

类型定义也类似:

```
type TypeName = {
    property1: Type1;
    // ...
    method1(arg1: ArgType1, arg2: ArgType2): ReturnType;
    // ...
};
```

`interface InterfaceName` 或者 `type TypeName`: 定义接口的名称。`property1: Type1`: 指定接口的属性及其相应的类型。可以定义多个属性，每个属性用分号分隔。`method1(arg1: ArgType1, arg2: ArgType2): ReturnType;`: 指定接口的方法。方法用其名称进行定义，后跟括号中的参数列表和返回类型。可以定义多个方法，每个方法用分号分隔。

接口示例:

```
interface Person {
    name: string;
    age: number;
    greet(): void;
}
```

类型示例:

```
type TypeName = {
    property1: string;
    method1(arg1: string, arg2: string): string;
};
```

在 TypeScript 中，类型用于定义数据的形状并强制执行类型检查。在 TypeScript 中定义类型有几种常见的语法，具体取决于具体的用例。这

里有些例子：

基本类型

```
let myNumber: number = 123; // 数字类型
let myBoolean: boolean = true; // 布尔类型
let myArray: string[] = ['a', 'b']; // 字符串数组
let myTuple: [string, number] = ['a', 123]; // 元组
```

对象和接口

```
const x: { name: string; age: number } = { name: 'Simon',
age: 7 };
```

并集和交集类型

```
type MyType = string | number; // 并集
let myUnion: MyType = 'hello'; // 可以是字符串
myUnion = 123; // 或者是一个数字

type TypeA = { name: string };
type TypeB = { age: number };
type CombinedType = TypeA & TypeB; // 交集
let myCombined: CombinedType = { name: 'John', age: 25 };
// 对象同时有name和age属性
```

内置原始数据类型

TypeScript 有几个内置的原属数据类型，可用于定义变量、函数参数和返回类型：

`number`: 表示数值，包括整数和浮点数。 `string`: 代表文本数据。 `boolean`: 代表逻辑值，可以是 true 或 false。 `null`: 表示没有值。 `undefined`: 表示尚未赋值或未定义的值。 `symbol`: 代表唯一标识符。符号通常用作对象属性的键。 `bigint`: 表示任意精度整数。 `any`: 代表

动态或未知类型。any 类型的变量可以保存任何类型的值，并且它们绕过类型检查。 `void`: 表示不存在任何类型。它通常用作不返回值的函数的返回类型。 `never`: 表示从未出现过的值的类型。它通常用作引发错误或进入无限循环的函数的返回类型。

常见的内置JS对象

TypeScript 是 JavaScript 的超集，它包含所有常用的内置 JavaScript 对象。您可以在 Mozilla 开发者网络 (MDN) 文档网站上找到这些对象的详细列表： https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

以下是一些常用的内置 JavaScript 对象的列表：

- Function
- Object
- Boolean
- Error
- Number
- BigInt
- Math
- Date
- String
- RegExp
- Array
- Map
- Set
- Promise
- Intl

重载

TypeScript 中的函数重载允许您为单个函数名称定义多个函数签名，从而使您能够定义可以多种方式调用的函数。这是一个例子：

```
// 重载
function sayHi(name: string): string;
function sayHi(names: string[]): string[];

// 实现
function sayHi(name: unknown): unknown {
    if (typeof name === 'string') {
        return `Hi, ${name}!`;
    } else if (Array.isArray(name)) {
        return name.map(name => `Hi, ${name}!`);
    }
    throw new Error('Invalid value');
}

sayHi('xx'); // 有效
sayHi(['aa', 'bb']); // 有效
```

这是在 class 中使用函数重载的另一个示例:

```
class Greeter {
    message: string;

    constructor(message: string) {
        this.message = message;
    }

    // 重载
    sayHi(name: string): string;
    sayHi(names: string[]): ReadonlyArray<string>;

    // 实现
    sayHi(name: unknown): unknown {
        if (typeof name === 'string') {
            return `${this.message}, ${name}!`;
        } else if (Array.isArray(name)) {
            return name.map(name => `${this.message},
${name}!`);
        }
        throw new Error('value is invalid');
    }
```

```
}
console.log(new Greeter('Hello').sayHi('Simon'));
```

合并与扩展

合并和扩展是指与使用类型和接口相关的两个不同概念。

合并允许您将多个同名声明合并到一个定义中，例如，当您多次定义同名接口时：

```
interface X {
    a: string;
}

interface X {
    b: number;
}

const person: X = {
    a: 'a',
    b: 7,
};
```

扩展是指扩展或继承现有类型或接口以创建新类型或接口的能力。它是一种向现有类型添加附加属性或方法而不修改其原始定义的机制。例子：

```
interface Animal {
    name: string;
    eat(): void;
}

interface Bird extends Animal {
    sing(): void;
}

const dog: Bird = {
    name: 'Bird 1',
```

```
    eat() {
        console.log('Eating');
    },
    sing() {
        console.log('Singing');
    },
};
```

类型和接口之间的差异

声明合并（增强）：

接口支持声明合并，这意味着您可以定义多个具有相同名称的接口，TypeScript 会将它们合并为具有组合属性和方法的单个接口。 另一方面，类型不支持声明合并。 当您想要添加额外的功能或自定义现有类型而不修改原始定义或修补丢失或不正确的类型时，这可能会很有帮助。

```
interface A {
    x: string;
}
interface A {
    y: string;
}
const j: A = {
    x: 'xx',
    y: 'yy',
};
```

扩展其他类型/接口：

类型和接口都可以扩展其他类型/接口，但语法不同。 对于接口，您可以使用“extends”关键字从其他接口继承属性和方法。 但是，接口无法扩展像联合类型这样的复杂类型。

```
interface A {
    x: string;
    y: number;
}
```

```
interface B extends A {
    z: string;
}
const car: B = {
    x: 'x',
    y: 123,
    z: 'z',
};
```

对于类型，您可以使用 & 运算符将多个类型合并为单个类型（交集）。

```
interface A {
    x: string;
    y: number;
}

type B = A & {
    j: string;
};

const c: B = {
    x: 'x',
    y: 123,
    j: 'j',
};
```

并集和交集类型：

在定义并集和交集类型时，类型更加灵活。 通过“type”关键字，您可以使用“|”运算符轻松创建联合类型，并使用“&”运算符创建交集类型。 虽然接口也可以间接表示联合类型，但它们没有对交集类型的内置支持。

```
type Department = 'dep-x' | 'dep-y'; // 并集

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

type Employee = {
```

```
    id: number;
    department: Department;
};

type EmployeeInfo = Person & Employee; // 交集
```

接口示例：

```
interface A {
    x: 'x';
}
interface B {
    y: 'y';
}

type C = A | B; // 接口的并集
```

Class

通用语法

TypeScript 中使用关键字 `class` 来定义类。下面，您可以看到一个示例：

```
class Person {
    private name: string;
    private age: number;
    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
    public sayHi(): void {
        console.log(
            `Hello, my name is ${this.name} and I am
${this.age} years old.`
        );
    }
}
```

class 关键字用于定义名为 Person 的类。

该类有两个私有属性：类型名称 string 和类型年龄 number。

构造函数是使用 constructor 关键字定义的。它将姓名和年龄作为参数并将它们分配给相应的属性。

该类有一个 public 名为 sayHi 的方法，用于记录问候消息。

要在 TypeScript 中创建类的实例，可以使用 new 关键字，后跟类名，然后使用括号 ()。例如：

```
const myObject = new Person('John Doe', 25);
myObject.sayHi(); // 输出：Hello, my name is John Doe and I
am 25 years old.
```

构造函数

构造函数是类中的特殊方法，用于在创建类的实例时初始化对象的属性。

```
class Person {
    public name: string;
    public age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log(
            `Hello, my name is ${this.name} and I'm
${this.age} years old.`
        );
    }
}
```

```
const john = new Person('Simon', 17);
john.sayHello();
```

可以使用以下语法重载构造函数：

```
type Sex = 'm' | 'f';

class Person {
    name: string;
    age: number;
    sex: Sex;

    constructor(name: string, age: number, sex?: Sex);
    constructor(name: string, age: number, sex: Sex) {
        this.name = name;
        this.age = age;
        this.sex = sex ?? 'm';
    }
}

const p1 = new Person('Simon', 17);
const p2 = new Person('Alice', 22, 'f');
```

在 TypeScript 中，可以定义多个构造函数重载，但只能有一个必须与所有重载兼容的实现，这可以通过使用可选参数来实现。

```
class Person {
    name: string;
    age: number;

    constructor();
    constructor(name: string);
    constructor(name: string, age: number);
    constructor(name?: string, age?: number) {
        this.name = name ?? 'Unknown';
        this.age = age ?? 0;
    }

    displayInfo() {
        console.log(`Name: ${this.name}, Age:
```

```
${this.age}`);
    }
}

const person1 = new Person();
person1.displayInfo(); // Name: Unknown, Age: 0

const person2 = new Person('John');
person2.displayInfo(); // Name: John, Age: 0

const person3 = new Person('Jane', 25);
person3.displayInfo(); // Name: Jane, Age: 25
```

私有和受保护的构造函数

在 TypeScript 中，构造函数可以标记为私有或受保护，这限制了它们的可访问性和使用。

私有构造函数：只能在类本身内调用。私有构造函数通常用于以下场景：您想要强制执行单例模式或将实例的创建限制为类中的工厂方法

受保护的构造函数：当您想要创建一个不应直接实例化但可以由子类扩展的基类时，受保护的构造函数非常有用。

```
class BaseClass {
    protected constructor() {}
}

class DerivedClass extends BaseClass {
    private value: number;

    constructor(value: number) {
        super();
        this.value = value;
    }
}

// 尝试直接实例化基类将导致错误
// const baseObj = new BaseClass(); // 错误：类"BaseClass"的
```

```
构造函数受到保护。

// 创建派生类的实例
const derivedObj = new DerivedClass(10);
```

访问修饰符

访问修饰符 `private`、`protected` 和 `public` 用于控制 TypeScript 类中类成员（例如属性和方法）的可见性和可访问性。这些修饰符对于强制封装以及建立访问和修改类内部状态的边界至关重要。

修饰符 `private` 仅限制对包含类中的类成员的访问。

修饰符 `protected` 允许访问包含类及其派生类中的类成员。

修饰符 `public` 提供对类成员的不受限制的访问，允许从任何地方访问它。

Get 与 Set

Getter 和 Setter 是特殊方法，允许您定义类属性的自定义访问和修改行为。它们使您能够封装对象的内部状态，并在获取或设置属性值时提供附加逻辑。在 TypeScript 中，getter 和 setter 分别使用 `get` 和 `set` 关键字定义。这是一个例子：

```
class MyClass {
    private _myProperty: string;

    constructor(value: string) {
        this._myProperty = value;
    }
    get myProperty(): string {
        return this._myProperty;
    }
    set myProperty(value: string) {
        this._myProperty = value;
    }
}
```

类中的自动访问器

TypeScript 版本 4.9 添加了对自动访问器的支持，这是即将推出的 ECMAScript 功能。它们类似于类属性，但使用"accessor"关键字声明。

```
class Animal {
    accessor name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

自动访问器被"脱糖"为私有get访问set器，在无法访问的属性上运行。

```
class Animal {
    #__name: string;

    get name() {
        return this.#__name;
    }
    set name(value: string) {
        this.#__name = name;
    }

    constructor(name: string) {
        this.name = name;
    }
}
```

this

在 TypeScript 中，`this` 关键字指的是类的方法或构造函数中的当前实例。它允许您在类自己的范围内访问和修改类的属性和方法。它提供了一种在对象自己的方法中访问和操作对象内部状态的方法。

```
class Person {
    private name: string;
    constructor(name: string) {
```

```
        this.name = name;
    }
    public introduce(): void {
        console.log(`Hello, my name is ${this.name}.`);
    }
}

const person1 = new Person('Alice');
person1.introduce(); // Hello, my name is Alice.
```

参数属性

参数属性允许您直接在构造函数参数中声明和初始化类属性，从而避免样板代码，例如：

```
class Person {
    constructor(
        private name: string,
        public age: number
    ) {
        // 构造函数中的"private"和"public"关键字自动声明并初始化相应的类属性。
    }
    public introduce(): void {
        console.log(
            `Hello, my name is ${this.name} and I am ${this.age} years old.`
        );
    }
}
const person = new Person('Alice', 25);
person.introduce();
```

抽象类

抽象类在 TypeScript 中主要用于继承，它们提供了一种定义可由子类继承的公共属性和方法的方法。当您想要定义常见行为并强制子类实现某

些方法时，这非常有用。它们提供了一种创建类层次结构的方法，其中抽象基类为子类提供共享接口和通用功能。

```
abstract class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    abstract makeSound(): void;
}

class Cat extends Animal {
    makeSound(): void {
        console.log(`${this.name} meows.`);
    }
}

const cat = new Cat('Whiskers');
cat.makeSound(); // 输出：Whiskers meows.
```

使用泛型

具有泛型的类允许您定义可以与不同类型一起使用的可重用类。

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }

    setItem(item: T): void {
        this.item = item;
```

```
    }
}

const container1 = new Container<number>(42);
console.log(container1.getItem()); //  42

const container2 = new Container<string>('Hello');
container2.setItem('World');
console.log(container2.getItem()); // World
```

装饰器

装饰器提供了一种添加元数据、修改行为、验证或扩展目标元素功能的机制。它们是在运行时执行的函数。多个装饰器可以应用于一个声明。

装饰器是实验性功能，以下示例仅与使用 ES6 的 TypeScript 版本 5 或更高版本兼容。

对于 5 之前的 TypeScript 版本，应在您的 `tsconfig.json` 中使用使 `experimentalDecorators` 或在命令行中使用 `--experimentalDecorators` 来启用它们（但以下示例不起作用）。

装饰器的一些常见用例包括：

- 观察属性变化。
- 观察方法调用。
- 添加额外的属性或方法。
- 运行时验证。
- 自动序列化和反序列化。
- 日志记录。
- 授权和认证。
- 错误防护。

注意：版本 5 的装饰器不允许装饰参数。

装饰器的类型：

类装饰器

类装饰器对于扩展现有类非常有用，例如添加属性或方法，或者收集类的实例。在下面的示例中，我们添加一个 toString 将类转换为字符串表示形式的方法。

```
type Constructor<T = {}> = new (...args: any[]) => T;

function toString<Class extends Constructor>(
    Value: Class,
    context: ClassDecoratorContext<Class>
) {
    return class extends Value {
        constructor(...args: any[]) {
            super(...args);
            console.log(JSON.stringify(this));
            console.log(JSON.stringify(context));
        }
    };
}

@toString
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    greet() {
        return 'Hello, ' + this.name;
    }
}
const person = new Person('Simon');
/* Logs:
{"name":"Simon"}
{"kind":"class","name":"Person"}
*/
```

属性装饰器

属性装饰器对于修改属性的行为非常有用，例如更改初始化值。在下面的代码中，我们有一个脚本将属性设置为始终大写：

```
function upperCase<T>(
    target: undefined,
    context: ClassFieldDecoratorContext<T, string>
) {
    return function (this: T, value: string) {
        return value.toUpperCase();
    };
}

class MyClass {
    @upperCase
    prop1 = 'hello!';
}

console.log(new MyClass().prop1); // 日志：HELLO!
```

方法装饰器

方法装饰器允许您更改或增强方法的行为。下面是一个简单记录器的示例：

```
function log<This, Args extends any[], Return>(
    target: (this: This, ...args: Args) => Return,
    context: ClassMethodDecoratorContext<
        This,
        (this: This, ...args: Args) => Return
    >
) {
    const methodName = String(context.name);

    function replacementMethod(this: This, ...args: Args):
Return {
        console.log(`LOG: Entering method
'${methodName}'.`);
```

```
        const result = target.call(this, ...args);
        console.log(`LOG: Exiting method
'${methodName}'.`);
        return result;
    }

    return replacementMethod;
}

class MyClass {
    @log
    sayHello() {
        console.log('Hello!');
    }
}

new MyClass().sayHello();
```

它记录：

```
LOG: Entering method 'sayHello'.
Hello!
LOG: Exiting method 'sayHello'.
```

Getter 和 Setter 装饰器

getter 和 setter 装饰器允许您更改或增强类访问器的行为。例如，它们对于验证属性分配很有用。这是 getter 装饰器的一个简单示例：

```
function range<This, Return extends number>(min: number,
max: number) {
    return function (
        target: (this: This) => Return,
        context: ClassGetterDecoratorContext<This, Return>
    ) {
        return function (this: This): Return {
            const value = target.call(this);
            if (value < min || value > max) {
                throw 'Invalid';
            }
```

```
            Object.defineProperty(this, context.name, {
                value,
                enumerable: true,
            });
            return value;
        };
    };
}

class MyClass {
    private _value = 0;

    constructor(value: number) {
        this._value = value;
    }
    @range(1, 100)
    get getValue(): number {
        return this._value;
    }
}

const obj = new MyClass(10);
console.log(obj.getValue); // 有效: 10

const obj2 = new MyClass(999);
console.log(obj2.getValue); // 抛出异常: Invalid!
```

装饰器元数据

装饰器元数据简化了装饰器在任何类中应用和利用元数据的过程。 他们可以访问上下文对象上的新元数据属性，该属性可以充当基元和对象的密钥。 可以通过”Symbol.metadata”在类上访问元数据信息。

元数据可用于各种目的，例如调试、序列化或使用装饰器的依赖项注入。

```
//@ts-ignore
Symbol.metadata ??= Symbol('Symbol.metadata'); // 简单的兼容性填充
```

```
type Context =
    | ClassFieldDecoratorContext
    | ClassAccessorDecoratorContext
    | ClassMethodDecoratorContext; // 上下文对象包含属性元数
据: 装饰器元数据

function setMetadata(_target: any, context: Context) {
    // 使用基本类型值设置元数据对象
    context.metadata[context.name] = true;
}

class MyClass {
    @setMetadata
    a = 123;

    @setMetadata
    accessor b = 'b';

    @setMetadata
    fn() {}
}

const metadata = MyClass[Symbol.metadata]; // 获取元数据对象
信息

console.log(JSON.stringify(metadata)); //
{"bar":true,"baz":true,"foo":true}
```

继承

继承是指一个类可以从另一个类（称为基类或超类）继承属性和方法的机制。派生类也称为子类或子类，可以通过添加新的属性和方法或重写现有的属性和方法来扩展和专门化基类的功能。

```
class Animal {
    name: string;

    constructor(name: string) {
```

```
        this.name = name;
    }

    speak(): void {
        console.log('The animal makes a sound');
    }
}

class Dog extends Animal {
    breed: string;

    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }

    speak(): void {
        console.log('Woof! Woof!');
    }
}

// 创建基类的一个实例
const animal = new Animal('Generic Animal');
animal.speak(); // The animal makes a sound

// 创建派生类的一个实例
const dog = new Dog('Max', 'Labrador');
dog.speak(); // Woof! Woof!"
```

TypeScript 不支持传统意义上的多重继承，而是允许从单个基类继承。TypeScript 支持多种接口。接口可以定义对象结构的契约，类可以实现多个接口。这允许类从多个源继承行为和结构。

```
interface Flyable {
    fly(): void;
}

interface Swimmable {
    swim(): void;
}
```

```
class FlyingFish implements Flyable, Swimmable {
    fly() {
        console.log('Flying...');
    }

    swim() {
        console.log('Swimming...');
    }
}

const flyingFish = new FlyingFish();
flyingFish.fly();
flyingFish.swim();
```

TypeScript 中的关键字 class 与 JavaScript 类似，通常被称为语法糖。它是在 ECMAScript 2015 (ES6) 中引入的，旨在提供更熟悉的语法，以基于类的方式创建和使用对象。然而，值得注意的是，TypeScript 作为 JavaScript 的超集，最终会编译为 JavaScript，而 JavaScript 的核心仍然是基于原型的。

静态成员

TypeScript 有静态成员。要访问类的静态成员，可以使用类名后跟一个点，而不需要创建对象。

```
class OfficeWorker {
    static memberCount: number = 0;

    constructor(private name: string) {
        OfficeWorker.memberCount++;
    }
}

const w1 = new OfficeWorker('James');
const w2 = new OfficeWorker('Simon');
const total = OfficeWorker.memberCount;
console.log(total); // 2
```

属性初始化

在 TypeScript 中初始化类的属性有多种方法：

内嵌：

在下面的示例中，创建类的实例时将使用这些初始值。

```
class MyClass {
    property1: string = 'default value';
    property2: number = 42;
}
```

在构造函数中：

```
class MyClass {
    property1: string;
    property2: number;

    constructor() {
        this.property1 = 'default value';
        this.property2 = 42;
    }
}
```

使用构造函数参数：

```
class MyClass {
    constructor(
        private property1: string = 'default value',
        public property2: number = 42
    ) {
        // 无需显式地将值分配给属性。
    }
    log() {
        console.log(this.property2);
    }
}
const x = new MyClass();
x.log();
```

方法重载

方法重载允许一个类有多个名称相同但参数类型不同或参数数量不同的方法。这允许我们根据传递的参数以不同的方式调用方法。

```
class MyClass {
    add(a: number, b: number): number; // 重载签名 1
    add(a: string, b: string): string; // 重载签名 2

    add(a: number | string, b: number | string): number |
string {
        if (typeof a === 'number' && typeof b ===
'number') {
            return a + b;
        }
        if (typeof a === 'string' && typeof b ===
'string') {
            return a.concat(b);
        }
        throw new Error('Invalid arguments');
    }
}

const r = new MyClass();
console.log(r.add(10, 5)); // 日志: 15
```

泛型

泛型允许您创建可与多种类型一起使用的可重用组件和函数。使用泛型，您可以参数化类型、函数和接口，从而允许它们对不同类型进行操作，而无需事先显式指定它们。

泛型允许您使代码更加灵活和可重用。

泛型类型

要定义泛型类型，可以使用尖括号 (<>) 来指定类型参数，例如：

```
function identity<T>(arg: T): T {
    return arg;
}
const a = identity('x');
const b = identity(123);

const getLen = <T,>(data: ReadonlyArray<T>) =>
data.length;
const len = getLen([1, 2, 3]);
```

泛型类

泛型也可以应用于类，这样它们就可以通过使用类型参数来处理多种类型。这对于创建可重用的类定义非常有用，这些定义可以在保持类型安全的同时对不同的数据类型进行操作。

```
class Container<T> {
    private item: T;

    constructor(item: T) {
        this.item = item;
    }

    getItem(): T {
        return this.item;
    }
}

const numberContainer = new Container<number>(123);
console.log(numberContainer.getItem()); // 123

const stringContainer = new Container<string>('hello');
console.log(stringContainer.getItem()); // hello
```

泛型约束

可以使用关键字 extends 后跟类型参数必须满足的类型或接口来约束泛型参数。

在下面的示例中，T 必须正确包含 length 才能有效：

```
const printLen = <T extends { length: number }>(value: T):
void => {
    console.log(value.length);
};

printLen('Hello'); // 5
printLen([1, 2, 3]); // 3
printLen({ length: 10 }); // 10
printLen(123); // 无效
```

3.4 RC 版中引入的泛型的一个有趣功能是高阶函数类型推断，它引入了传播泛型类型参数：

```
declare function pipe<A extends any[], B, C>(
    ab: (...args: A) => B,
    bc: (b: B) => C
): (...args: A) => C;

declare function list<T>(a: T): T[];
declare function box<V>(x: V): { value: V };

const listBox = pipe(list, box); // <T>(a: T) => { value:
T[] }
const boxList = pipe(box, list); // <V>(x: V) => { value:
V }[]
```

此功能允许更轻松地键入安全的无点风格编程，这在函数式编程中很常见。

泛型上下文缩小

泛型上下文缩小是 TypeScript 中的机制，允许编译器根据使用泛型参数的上下文来缩小泛型参数的类型，在条件语句中使用泛型类型时非常有用：

```
function process<T>(value: T): void {
    if (typeof value === 'string') {
```

```
        // Value 的类型被缩小到 'string' 类型
        console.log(value.length);
    } else if (typeof value === 'number') {
        // Value 的类型被缩小到 'number' 类型
        console.log(value.toFixed(2));
    }
}

process('hello'); // 5
process(3.14159); // 3.14
```

擦除的结构类型

在 TypeScript 中，对象不必匹配特定的、精确的类型。例如，如果我们创建一个满足接口要求的对象，我们就可以在需要该接口的地方使用该对象，即使它们之间没有显式连接。例子：

```
type NameProp1 = {
    prop1: string;
};

function log(x: NameProp1) {
    console.log(x.prop1);
}

const obj = {
    prop2: 123,
    prop1: 'Origin',
};

log(obj); // 有效
```

命名空间

在 TypeScript 中，命名空间用于将代码组织到逻辑容器中，防止命名冲突并提供一种将相关代码分组在一起的方法。使用关键字 export 允许

在”外部”模块中访问名称空间。

```
export namespace MyNamespace {
    export interface MyInterface1 {
        prop1: boolean;
    }
    export interface MyInterface2 {
        prop2: string;
    }
}

const a: MyNamespace.MyInterface1 = {
    prop1: true,
};
```

Symbols

符号是一种原始数据类型，表示不可变值，保证在程序的整个生命周期中全局唯一。

符号可以用作对象属性的键，并提供一种创建不可枚举属性的方法。

```
const key1: symbol = Symbol('key1');
const key2: symbol = Symbol('key2');

const obj = {
    [key1]: 'value 1',
    [key2]: 'value 2',
};

console.log(obj[key1]); // value 1
console.log(obj[key2]); // value 2
```

在 WeakMap 和 WeakSet 中，现在允许符号作为键。

三斜杠指令

三斜杠指令是特殊注释，为编译器提供有关如何处理文件的说明。这些指令以三个连续斜杠 (///) 开头，通常放置在 TypeScript 文件的顶部，对运行时行为没有影响。

三斜杠指令用于引用外部依赖项、指定模块加载行为、启用/禁用某些编译器功能等等。几个例子：

引用声明文件：

```
/// <reference path="path/to/declaration/file.d.ts" />
```

指明模块格式：

```
/// <amd|commonjs|system|umd|es6|es2015|none>
```

启用编译器选项，在以下示例中严格模式：

```
///
<strict|noImplicitAny|noUnusedLocals|noUnusedParameters>
```

类型操作

从类型创建类型

是否可以通过组合、操作或转换现有类型来创建新类型。

交集类型 (&):

允许您将多种类型组合成单一类型：

```
type A = { foo: number };
type B = { bar: string };
type C = A & B; // A和B的交集
const obj: C = { foo: 42, bar: 'hello' };
```

联合类型 (|):

允许您定义可以是以下几种类型之一的类型

```
type Result = string | number;
const value1: Result = 'hello';
const value2: Result = 42;
```

映射类型:

允许您转换现有类型的属性以创建新类型:

```
type Mutable<T> = {
    readonly [P in keyof T]: T[P];
};
type Person = {
    name: string;
    age: number;
};
type ImmutablePerson = Mutable<Person>; // 属性变为只读
```

条件类型:

允许您根据某些条件创建类型:

```
type ExtractParam<T> = T extends (param: infer P) => any ?
P : never;
type MyFunction = (name: string) => number;
type ParamType = ExtractParam<MyFunction>; // string
```

索引访问类型

在 TypeScript 中，可以使用索引访问和操作另一个类型中的属性类型 `Type[Key]`。

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

```ts
type AgeType = Person['age']; // number

type MyTuple = [string, number, boolean];
type MyType = MyTuple[2]; // boolean
```

工具类型

可以使用几种内置工具来操作类型，下面列出了最常用的：

Awaited<T>

构造一个递归解包 Promise 的类型。

```ts
type A = Awaited<Promise<string>>; // string
```

Partial<T>

构造一个类型，并将 T 的所有属性设置为可选。

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

type A = Partial<Person>; // { name?: string | undefined;
age?: number | undefined; }
```

Required<T>

构造一个类型，并将 T 的所有属性设置为必需。

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

```ts
type A = Required<Person>; // { name: string; age: number;
}
```

Readonly<T>

构造一个类型，并将 T 的所有属性设置为只读。

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

type A = Readonly<Person>;

const a: A = { name: 'Simon', age: 17 };
a.name = 'John'; // 无效
```

Record<K, T>

构造一个具有类型 T 的一组属性 K 的类型。

```ts
type Product = {
    name: string;
    price: number;
};

const products: Record<string, Product> = {
    apple: { name: 'Apple', price: 0.5 },
    banana: { name: 'Banana', price: 0.25 },
};

console.log(products.apple); // { name: 'Apple', price:
0.5 }
```

Pick<T, K>

通过从 T 中选取指定属性 K 来构造类型。

```ts
type Product = {
    name: string;
    price: number;
};

type Price = Pick<Product, 'price'>; // { price: number; }
```

Omit<T, K>

通过从 T 中省略指定属性 K 来构造类型。

```ts
type Product = {
    name: string;
    price: number;
};

type Name = Omit<Product, 'price'>; // { name: string; }
```

Exclude<T, U>

通过从 T 中排除类型 U 的所有值来构造类型。

```ts
type Union = 'a' | 'b' | 'c';
type MyType = Exclude<Union, 'a' | 'c'>; // b
```

Extract<T, U>

通过从 T 中提取类型 U 的所有值来构造类型。

```ts
type Union = 'a' | 'b' | 'c';
type MyType = Extract<Union, 'a' | 'c'>; // a | c
```

NonNullable<T>

通过从 T 中排除 null 和 undefined 来构造类型。

```ts
type Union = 'a' | null | undefined | 'b';
type MyType = NonNullable<Union>; // 'a' | 'b'
```

Parameters<T>

提取函数类型 T 的参数类型。

```ts
type Func = (a: string, b: number) => void;
type MyType = Parameters<Func>; // [a: string, b: number]
```

ConstructorParameters<T>

提取构造函数类型 T 的参数类型。

```ts
class Person {
    constructor(
        public name: string,
        public age: number
    ) {}
}
type PersonConstructorParams =
ConstructorParameters<typeof Person>; // [name: string,
age: number]
const params: PersonConstructorParams = ['John', 30];
const person = new Person(...params);
console.log(person); // Person { name: 'John', age: 30 }
```

ReturnType<T>

提取函数类型 T 的返回类型。

```ts
type Func = (name: string) => number;
type MyType = ReturnType<Func>; // number
```

InstanceType<T>

提取类类型 T 的实例类型。

```
class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    sayHello() {
        console.log(`Hello, my name is ${this.name}!`);
    }
}

type PersonInstance = InstanceType<typeof Person>;

const person: PersonInstance = new Person('John');

person.sayHello(); // Hello, my name is John!
```

ThisParameterType<T>

从函数类型 T 中提取”this”参数的类型。

```
interface Person {
    name: string;
    greet(this: Person): void;
}
type PersonThisType = ThisParameterType<Person['greet']>;
// Person
```

OmitThisParameter<T>

从函数类型 T 中删除”this”参数。

```
function capitalize(this: String) {
    return this[0].toUpperCase +
this.substring(1).toLowerCase();
}
```

```
type CapitalizeType = OmitThisParameter<typeof
capitalize>; // () => string
```

ThisType<T>

作为上下文类型 this 的一部分。

```
type Logger = {
    log: (error: string) => void;
};

let helperFunctions: { [name: string]: Function } &
ThisType<Logger> = {
    hello: function () {
        this.log('some error'); // 有效，因为"log"是"this"的
一部分
        this.update(); // 无效
    },
};
```

Uppercase<T>

将输入类型 T 的名称设为大写。

```
type MyType = Uppercase<'abc'>; // "ABC"
```

Lowercase<T>

将输入类型 T 的名称设为小写。

```
type MyType = Lowercase<'ABC'>; // "abc"
```

Capitalize<T>

输入类型 T 的名称大写。

```
type MyType = Capitalize<'abc'>; // "Abc"
```

Uncapitalize<T>

将输入类型 T 的名称取消大写。

```
type MyType = Uncapitalize<'Abc'>; // "abc"
```

其他

错误和异常处理

TypeScript 允许您使用标准 JavaScript 错误处理机制捕获和处理错误：

Try-Catch-Finally 块：

```
try {
    // 可能会抛出异常的代码
} catch (error) {
    // 处理错误
} finally {
    // 总是会执行的代码, finally 是可选的
}
```

您还可以处理不同类型的错误：

```
try {
    // 可能会抛出不同类型错误的代码
} catch (error) {
    if (error instanceof TypeError) {
        // 处理 TypeError
    } else if (error instanceof RangeError) {
        // 处理 RangeError
    } else {
        // 处理其他的错误
    }
}
```

自定义错误类型：

可以通过扩展 Error 来指定更具体的错误 class：

```
class CustomError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'CustomError';
    }
}

throw new CustomError('This is a custom error.');
```

混合类

Mixin 类允许您将多个类的行为组合并组合成一个类。它们提供了一种重用和扩展功能的方法，而不需要深层继承链。

```
abstract class Identifiable {
    name: string = '';
    logId() {
        console.log('id:', this.name);
    }
}
abstract class Selectable {
    selected: boolean = false;
    select() {
        this.selected = true;
        console.log('Select');
    }
    deselect() {
        this.selected = false;
        console.log('Deselect');
    }
}
class MyClass {
    constructor() {}
}

// 扩展 MyClass 以包含可识别和可选择的行为
interface MyClass extends Identifiable, Selectable {}
```

```
// 将 mixins 应用于类的函数
function applyMixins(source: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.prototype).forEach(name
=> {
            let descriptor =
Object.getOwnPropertyDescriptor(
                baseCtor.prototype,
                name
            );
            if (descriptor) {
                Object.defineProperty(source.prototype,
name, descriptor);
            }
        });
    });
}

// 将 mixins 应用到 MyClass
applyMixins(MyClass, [Identifiable, Selectable]);
let o = new MyClass();
o.name = 'abc';
o.logId();
o.select();
```

异步语言特性

由于 TypeScript 是 JavaScript 的超集，因此它内置了 JavaScript 的异步语言功能，例如：

Promises：

Promise 是一种处理异步操作及其结果的方法，使用 `.then()`和等方法 `.catch()` 来处理成功和错误条件。

要了解更多信息： https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Async/await:

Async/await 关键字是一种为处理 Promise 提供看起来更同步的语法的方法。`async` 关键字用于定义异步函数，并且 `await` 关键字在异步函数中使用以暂停执行，直到 Promise 被解决或拒绝。

要了解更多信息： https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

TypeScript 很好地支持以下 API：

Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Web Workers: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

Shared Workers: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker

WebSocket: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

迭代器和生成器

TypeScript 很好地支持交互器和生成器。

迭代器是实现迭代器协议的对象，提供了一种逐个访问集合或序列元素的方法。它是一个包含指向迭代中下一个元素的指针的结构。他们有一个 `next()` 方法返回序列中的下一个值以及指示序列是否为 的布尔值 `done` 。

```
class NumberIterator implements Iterable<number> {
    private current: number;

    constructor(
```

```
        private start: number,
        private end: number
    ) {
        this.current = start;
    }

    public next(): IteratorResult<number> {
        if (this.current <= this.end) {
            const value = this.current;
            this.current++;
            return { value, done: false };
        } else {
            return { value: undefined, done: true };
        }
    }

    [Symbol.iterator](): Iterator<number> {
        return this;
    }
}

const iterator = new NumberIterator(1, 3);

for (const num of iterator) {
    console.log(num);
}
```

生成器是使用 function* 简化迭代器创建的语法定义的特殊函数。它们使用 yield 关键字来定义值的序列，并在请求值时自动暂停和恢复执行。

生成器使创建迭代器变得更加容易，并且对于处理大型或无限序列特别有用。

例子：

```
function* numberGenerator(start: number, end: number):
Generator<number> {
    for (let i = start; i <= end; i++) {
        yield i;
```

```
    }
}

const generator = numberGenerator(1, 5);

for (const num of generator) {
    console.log(num);
}
```

TypeScript 还支持异步迭代器和异步生成器。

要了解更多信息：

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Iterator

TsDocs JSDoc 参考

使用 JavaScript 代码库时，可以通过使用 JSDoc 注释和附加注释来提供类型信息，帮助 TypeScript 推断正确的类型。

例子：

```
/**
 * Computes the power of a given number
 * @constructor
 * @param {number} base – The base value of the expression
 * @param {number} exponent – The exponent value of the
expression
 */
function power(base: number, exponent: number) {
    return Math.pow(base, exponent);
}
power(10, 2); // function power(base: number, exponent:
number): number
```

此链接提供了完整文档：
https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html

从版本 3.7 开始，可以从 JavaScript JSDoc 语法生成 .d.ts 类型定义。
更多信息可以在这里找到：
https://www.typescriptlang.org/docs/handbook/declaration-files/dts-from-js.html

@types

@types 组织下的包是特殊的包命名约定，用于为现有 JavaScript 库或模块提供类型定义。例如使用：

```
npm install --save-dev @types/lodash
```

将在您当前的项目中安装 `lodash` 的类型定义。

要为 @types 包的类型定义做出贡献，请向 https://github.com/DefinitelyTyped/DefinitelyTyped 提交pr请求。

JSX

JSX (JavaScript XML) 是 JavaScript 语言语法的扩展，允许您在 JavaScript 或 TypeScript 文件中编写类似 HTML 的代码。它通常在 React 中用来定义 HTML 结构。

TypeScript 通过提供类型检查和静态分析来扩展 JSX 的功能。

要使用 JSX，您需要在文件 `tsconfig.json` 中设置 `jsx` 编译器选项。
两个常见的配置选项：

- “preserve”: 触发 .jsx 文件且 JSX 不变. 此选项告诉 TypeScript 按原样保留 JSX 语法，而不是在编译过程中对其进行转换。 如果您有单独的工具（例如 Babel）来处理转换，则可以使用此选项。
- “react”: 启用 TypeScript 的内置 JSX 转换。 将使用 React.createElement 。

所有选项均可在此处使用：
https://www.typescriptlang.org/tsconfig#jsx

ES6 模块

TypeScript 确实支持 ES6 (ECMAScript 2015) 和许多后续版本。这意味着您可以使用 ES6 语法，例如箭头函数、模板文字、类、模块、解构等等。

要在项目中启用 ES6 功能，您可以在 `tsconfig.json` 中指定 `target` 属性。

配置示例：

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "es6",
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist"
  },
  "include": ["src"]
}
```

ES7 求幂运算符

求幂 (**) 运算符计算通过将第一个操作数进行第二个操作数的幂获得的值。它的功能与 `Math.pow()` 类似，但增加了接受 BigInts 作为操作数的功能。TypeScript 完全支持在 `tsconfig.json` 文件中设置 `target` 为 `es2016`或更大版本来使用此运算符。

```
console.log(2 ** (2 ** 2)); // 16
```

for-await-of 语句

这是 TypeScript 完全支持的 JavaScript 功能，它允许您从目标版本 `es2018` 迭代异步可迭代对象。

```
async function* asyncNumbers():
AsyncIterableIterator<number> {
    yield Promise.resolve(1);
    yield Promise.resolve(2);
    yield Promise.resolve(3);
}

(async () => {
    for await (const num of asyncNumbers()) {
        console.log(num);
    }
})();
```

New target 元属性

您可以在 TypeScript 中使用 `new.target` 元属性，该属性使您能够确定是否使用 new 运算符调用函数或构造函数。它允许您检测对象是否是由于构造函数调用而创建的。

```
class Parent {
    constructor() {
        console.log(new.target); // 记录用于创建实例的构造函数
    }
}

class Child extends Parent {
    constructor() {
        super();
    }
}

const parentX = new Parent(); // [Function: Parent]
const child = new Child(); // [Function: Child]
```

动态导入表达式

可以使用 TypeScript 支持的动态导入 ECMAScript 建议有条件地加载模块或按需延迟加载模块。

TypeScript 中动态导入表达式的语法如下：

```
async function renderWidget() {
    const container = document.getElementById('widget');
    if (container !== null) {
        const widget = await import('./widget'); // 动态导
入
        widget.render(container);
    }
}

renderWidget();
```

“tsc –watch”

此命令使用 `--watch` 参数启动 TypeScript 编译器，能够在修改 TypeScript 文件时自动重新编译它们。

```
tsc --watch
```

从 TypeScript 4.9 版本开始，文件监控主要依赖于文件系统事件，如果无法建立基于事件的观察程序，则会自动诉诸轮询。

默认声明

当为变量或参数分配默认值时，将使用默认声明。这意味着如果没有为该变量或参数提供值，则将使用默认值。

```
function greet(name: string = 'Anonymous'): void {
    console.log(`Hello, ${name}!`);
}
greet(); // Hello, Anonymous!
greet('John'); // Hello, John!
```

可选链

可选的链接运算符 `?.` 与常规点运算符 (`.`) 一样用于访问属性或方法。但是，它通过优雅处理 `undefined` 和 `null` 来终止表达式并返回 `undefined`，而不是抛出错误。

```
type Person = {
    name: string;
    age?: number;
    address?: {
        street?: string;
        city?: string;
    };
};

const person: Person = {
    name: 'John',
};

console.log(person.address?.city); // undefined
```

空合并运算符

如果 `??` 左侧是 `null` 或者 `undefined` ，则空合并运算符返回右侧值，否则，它返回左侧值。

```
const foo = null ?? 'foo';
console.log(foo); // foo

const baz = 1 ?? 'baz';
const baz2 = 0 ?? 'baz';
console.log(baz); // 1
console.log(baz2); // 0
```

模板字符串类型

模板字符串类型允许在类型级别操作字符串值并基于现有字符串生成新的字符串类型。它们对于从基于字符串的操作创建更具表现力和更精确

的类型非常有用。

```
type Department = 'enginnering' | 'hr';
type Language = 'english' | 'spanish';
type Id = `${Department}-${Language}-id`; // "enginnering-english-id" | "enginnering-spanish-id" | "hr-english-id" | "hr-spanish-id"
```

函数重载

函数重载允许您为同一函数名定义多个函数签名，每个函数签名具有不同的参数类型和返回类型。当您调用重载函数时，TypeScript 使用提供的参数来确定正确的函数签名：

```
function makeGreeting(name: string): string;
function makeGreeting(names: string[]): string[];

function makeGreeting(person: unknown): unknown {
    if (typeof person === 'string') {
        return `Hi ${person}!`;
    } else if (Array.isArray(person)) {
        return person.map(name => `Hi, ${name}!`);
    }
    throw new Error('Unable to greet');
}

makeGreeting('Simon');
makeGreeting(['Simone', 'John']);
```

递归类型

递归类型是可以引用自身的类型。 这对于定义具有分层或递归结构（可能无限嵌套）的数据结构非常有用，例如链表、树和图。

```
type ListNode<T> = {
    data: T;
    next: ListNode<T> | undefined;
};
```

递归条件类型

可以使用 TypeScript 中的逻辑和递归来定义复杂的类型关系。让我们简单地分解一下：

条件类型：允许您基于布尔条件定义类型：

```
type CheckNumber<T> = T extends number ? 'Number' : 'Not a number';
type A = CheckNumber<123>; // 'Number'
type B = CheckNumber<'abc'>; // 'Not a number'
```

递归：是指在自己的定义中引用自身的类型定义：

```
type Json = string | number | boolean | null | Json[] | { [key: string]: Json };

const data: Json = {
    prop1: true,
    prop2: 'prop2',
    prop3: {
        prop4: [],
    },
};
```

递归条件类型结合了条件逻辑和递归。这意味着类型定义可以通过条件逻辑依赖于自身，从而创建复杂且灵活的类型关系。

```
type Flatten<T> = T extends Array<infer U> ? Flatten<U> : T;

type NestedArray = [1, [2, [3, 4], 5], 6];
type FlattenedArray = Flatten<NestedArray>; // 2 | 3 | 4 | 5 | 1 | 6
```

Node 中的 ECMAScript 模块支持

Node.js 从 15.3.0 版本开始添加了对 ECMAScript 模块的支持，而 TypeScript 从 4.7 版本开始增加了对 Node.js 的 ECMAScript 模块支

持。可以通过将 tsconfig.json 文件中的module属性的值设置为 nodenext 来启用此支持。这是一个例子：

```
{
  "compilerOptions": {
    "module": "nodenext",
    "outDir": "./lib",
    "declaration": true
  }
}
```

Node.js 支持两种模块文件扩展名：.mjs 的ES 模块和 .cjs 的 CommonJS 模块。TypeScript 中的等效文件扩展名适用 .mts 于 ES 模块和 .cts 于CommonJS 模块。当 TypeScript 编译器将这些文件转译为 JavaScript 时，它将分别创建 .mjs 和 .cjs 文件。

如果您想在项目中使用 ES 模块，可以type在 package.json 文件中将该属性设置为"module"。这指示 Node.js 将项目视为 ES 模块项目。

此外，TypeScript 还支持 .d.ts 文件中的类型声明。这些声明文件为用 TypeScript 编写的库或模块提供类型信息，允许其他开发人员通过 TypeScript 的类型检查和自动完成功能来利用它们。

断言函数

在 TypeScript 中，断言函数是根据返回值指示特定条件验证的函数。在最简单的形式中，断言函数检查提供的谓词，并在谓词计算结果为 false 时引发错误。

```
function isNumber(value: unknown): asserts value is number
{
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
}
```

或者可以声明为函数表达式：

```
type AssertIsNumber = (value: unknown) => asserts value is
number;
const isNumber: AssertIsNumber = value => {
    if (typeof value !== 'number') {
        throw new Error('Not a number');
    }
};
```

断言函数与类型保护有相似之处。类型保护最初是为了执行运行时检查并确保值的类型在特定范围内而引入的。具体来说，类型保护是一个计算类型谓词并返回指示谓词是真还是假的布尔值的函数。这与断言函数略有不同，断言函数的目的是在不满足谓词时抛出错误而不是返回false。

类型保护示例：

```
const isNumber = (value: unknown): value is number =>
typeof value === 'number';
```

可变参数元组类型

可变元组类型是 TypeScript 4.0 版本中引入的一个功能，让我们通过回顾什么是元组来开始学习它们：

元组类型是一个具有定义长度的数组，并且每个元素的类型已知：

```
type Student = [string, number];
const [name, age]: Student = ['Simone', 20];
```

术语”可变参数”意味着不定数量（接受可变数量的参数）。

可变参数元组是一种元组类型，它具有以前的所有属性，但确切的形状尚未定义：

```
type Bar<T extends unknown[]> = [boolean, ...T, number];

type A = Bar<[boolean]>; // [boolean, boolean, number]
type B = Bar<['a', 'b']>; // [boolean, 'a', 'b', number]
type C = Bar<[]>; // [boolean, number]
```

在前面的代码中我们可以看到元组形状是由T传入的泛型定义的。

可变参数元组可以接受多个泛型，这使得它们非常灵活：

```
type Bar<T extends unknown[], G extends unknown[]> =
[...T, boolean, ...G];

type A = Bar<[number], [string]>; // [number, boolean,
string]
type B = Bar<['a', 'b'], [boolean]>; // ["a", "b",
boolean, boolean]
```

使用新的可变参数元组，我们可以使用：

- 元组类型语法中的扩展现在可以是通用的，因此即使我们不知道我们正在操作的实际类型，我们也可以表示元组和数组上的高阶操作
- 其余元素可以出现在元组中的任何位置。

例子：

```
type Items = readonly unknown[];

function concat<T extends Items, U extends Items>(
    arr1: T,
    arr2: U
): [...T, ...U] {
    return [...arr1, ...arr2];
}

concat([1, 2, 3], ['4', '5', '6']); // [1, 2, 3, "4", "5",
"6"]
```

装箱类型

装箱类型是指用于将基本类型表示为对象的包装对象。这些包装器对象提供了原始值无法直接使用的附加功能和方法。

当你访问原始 `string` 上的 `charAt` 或者 `normalize` 方法时，JavaScript 将其包装在 `String` 类型的对象中，调用该方法，然后丢弃

该对象

示范：

```
const originalNormalize = String.prototype.normalize;
String.prototype.normalize = function () {
    console.log(this, typeof this);
    return originalNormalize.call(this);
};
console.log('\u0041'.normalize());
```

TypeScript 通过为原语及其相应的对象包装器提供单独的类型来表示这种区别：

- string => String
- number => Number
- boolean => Boolean
- symbol => Symbol
- bigint => BigInt

通常不需要盒装类型。避免使用装箱类型，而是使用基元类型，例如 `string` 代替 `String`。

TypeScript 中的协变和逆变

协变和逆变用于描述在处理类型的继承或赋值时关系如何工作。

协变意味着类型关系保留继承或赋值的方向，因此如果类型 A 是类型 B 的子类型，则类型 A 的数组也被视为类型 B 的数组的子类型。这里需要注意的重要事项是维持子类型关系，这意味着协变接受子类型但不接受超类型。

逆变意味着类型关系颠倒了继承或赋值的方向，因此如果类型 A 是类型 B 的子类型，则类型 B 的数组被视为类型 A 数组的子类型。子类型关系颠倒了，这意味着该逆变接受超类型但不接受子类型。

注意：双变量意味着同时接受超类型和子类型。

示例：假设我们有一个适合所有动物的空间和一个专门适合狗的单独空间。

在协方差中，您可以将所有狗放入动物空间中，因为狗是一种动物。但你不能把所有的动物都放在狗的空间里，因为可能还有其他动物混在一起。

在逆变中，您不能将所有动物放入狗空间中，因为动物空间也可能包含其他动物。然而，你可以把所有的狗都放在动物空间里，因为所有的狗也是动物。

```
// 协变示例
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
}

class Dog extends Animal {
    breed: string;
    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }
}

let animals: Animal[] = [];
let dogs: Dog[] = [];

// 协变允许将子类型（狗）数组分配给超类型（动物）数组
animals = dogs;
dogs = animals; // 无效: 'Animal[]' 不能赋值给 'Dog[]'

// 逆变示例
type Feed<in T> = (animal: T) => void;

let feedAnimal: Feed<Animal> = (animal: Animal) => {
    console.log(`Animal name: ${animal.name}`);
};
```

```
let feedDog: Feed<Dog> = (dog: Dog) => {
    console.log(`Dog name: ${dog.name}, Breed:
${dog.breed}`);
};

// 逆变允许将超类型（动物）回调赋值给子类型（狗）回调
feedDog = feedAnimal;
feedAnimal = feedDog; // 无效: Type 'Feed<Dog>' 不能赋值给
'Feed<Animal>'.
```

在 TypeScript 中，数组的类型关系是协变的，而函数参数的类型关系是逆变的。这意味着 TypeScript 同时表现出协变和逆变，具体取决于上下文。

类型参数的可选方差注释

从 TypeScript 4.7.0 开始，我们可以使用out和in关键字来具体说明方差注释。

对于协变，使用out关键字：

```
type AnimalCallback<out T> = () => T; // 此处 T 是协变的
```

对于逆变，使用in关键字：

```
type AnimalCallback<in T> = (value: T) => void; // 此处 T
是逆变的
```

模板字符串模式索引签名

模板字符串模式索引签名允许我们使用模板字符串模式定义灵活的索引签名。 此功能使我们能够创建可以使用特定字符串键模式进行索引的对象，从而在访问和操作属性时提供更多控制和特异性。

TypeScript 4.4 版开始允许符号和模板字符串模式的索引签名。

```ts
const uniqueSymbol = Symbol('description');

type MyKeys = `key-${string}`;

type MyObject = {
    [uniqueSymbol]: string;
    [key: MyKeys]: number;
};

const obj: MyObject = {
    [uniqueSymbol]: 'Unique symbol key',
    'key-a': 123,
    'key-b': 456,
};

console.log(obj[uniqueSymbol]); // Unique symbol key
console.log(obj['key-a']); // 123
console.log(obj['key-b']); // 456
```

satisfies操作符

satisfies 允许您检查给定类型是否满足特定接口或条件。换句话说，它确保类型具有特定接口所需的所有属性和方法。这是确保变量适合类型定义的一种方法。

下面是一个示例：

```ts
type Columns = 'name' | 'nickName' | 'attributes';

type User = Record<Columns, string | string[] |
undefined>;

// `User`的类型注释
const user: User = {
    name: 'Simone',
    nickName: undefined,
    attributes: ['dev', 'admin'],
};
```

```ts
// 在以下几行中，TypeScript 将无法正确推断
user.attributes?.map(console.log); // 'string | string[]'
中不存在属性 'map'。'string' 中不存在属性 'map'。
user.nickName; // string | string[] | undefined

// 类型断言 `as`
const user2 = {
    name: 'Simon',
    nickName: undefined,
    attributes: ['dev', 'admin'],
} as User;

// 这里也一样的， TypeScript 将无法正确推断
user2.attributes?.map(console.log); //'string | string[]'
中不存在属性 'map'。'string' 中不存在属性 'map'。
user2.nickName; // string | string[] | undefined

// 使用"satisfies"运算符我们现在可以正确推断类型
const user3 = {
    name: 'Simon',
    nickName: undefined,
    attributes: ['dev', 'admin'],
} satisfies User;

user3.attributes?.map(console.log); // TypeScript 推断正确:
string[]
user3.nickName; // TypeScript 推断正确: undefined
```

仅类型导入和导出

仅类型导入和导出允许您导入或导出类型，而无需导入或导出与这些类型关联的值或函数。 这对于减小捆绑包的大小很有用。

要使用仅类型导入，您可以使用`import type`关键字。

TypeScript 允许在仅类型导入中使用声明和实现文件扩展名（.ts、.mts、.cts 和 .tsx），无论`allowImportingTsExtensions`设置如何。

例如：

```
import type { House } from './house.ts';
```

以下是支持的形式：

```
import type T from './mod';
import type { A, B } from './mod';
import type * as Types from './mod';
export type { T };
export type { T } from './mod';
```

使用声明和显式资源管理

“using”声明是块范围的、不可变的绑定，类似于”const”，用于管理一次性资源。 当使用值初始化时，该值的”Symbol.dispose”方法将被记录，并随后在退出封闭块作用域时执行。

这是基于 ECMAScript 的资源管理功能，该功能对于在对象创建后执行基本的清理任务非常有用，例如关闭连接、删除文件和释放内存。

笔记：

- 由于最近在 TypeScript 5.2 版中引入，大多数运行时缺乏本机支持。 您将需要以下功能的填充：`Symbol.dispose`、`Symbol.asyncDispose`、`DisposableStack`、`AsyncDisposableStack`、`SuppressedError`。
- 此外，您需要按如下方式配置 tsconfig.json：

```
{
    "compilerOptions": {
        "target": "es2022",
        "lib": ["es2022", "esnext.disposable", "dom"]
    }
}
```

例子：

```ts
//@ts-ignore
Symbol.dispose ??= Symbol('Symbol.dispose'); // 简单的兼容性填充

const doWork = (): Disposable => {
    return {
        [Symbol.dispose]: () => {
            console.log('disposed');
        },
    };
};

console.log(1);

{
    using work = doWork(); // 资源被声明
    console.log(2);
} // 资源被释放（例如，`work[Symbol.dispose]()` 被执行）

console.log(3);
```

该代码将记录：

```
1
2
disposed
3
```

符合处置条件的资源必须遵守 Disposable 接口：

```ts
// lib.esnext.disposable.d.ts
interface Disposable {
    [Symbol.dispose](): void;
}
```

“using”声明在堆栈中记录资源处置操作，确保它们以与声明相反的顺序处置：

```ts
{
    using j = getA(),
```

```
        y = getB();
    using k = getC();
} // 先释放 `C`, 然后 `B`, 然后 `A`.
```

即使发生后续代码或异常，也保证会释放资源。 这可能会导致处置可能引发异常，并可能抑制另一个异常。 为了保留有关被抑制错误的信息，引入了一个新的本机异常”SuppressedError”。

使用声明等待

“await using”声明处理异步一次性资源。 该值必须具有”Symbol.asyncDispose”方法，该方法将在块末尾等待。

```
async function doWorkAsync() {
    await using work = doWorkAsync(); // 资源被声明
} // // 资源被释放 (例如, `await work[Symbol.asyncDispose]()` 被执行)
```

对于异步可处置资源，它必须遵守”Disposable”或”AsyncDisposable”接口：

```
// lib.esnext.disposable.d.ts
interface AsyncDisposable {
    [Symbol.asyncDispose](): Promise<void>;
}

//@ts-ignore
Symbol.asyncDispose ??= Symbol('Symbol.asyncDispose'); // Simple polify

class DatabaseConnection implements AsyncDisposable {
    // 当对象被异步释放时会被调用的方法
    [Symbol.asyncDispose]() {
        // Close the connection and return a promise
        return this.close();
    }

    async close() {
        console.log('Closing the connection...');
```

```
        await new Promise(resolve => setTimeout(resolve,
1000));
        console.log('Connection closed.');
    }
}

async function doWork() {
    // 创建一个新的连接，并在其超出作用域时进行异步释放
    await using connection = new DatabaseConnection(); //
资源被声明
    console.log('Doing some work...');
} // 资源被释放（例如，`await
connection[Symbol.asyncDispose]()` 被执行）

doWork();
```

代码日志：

```
Doing some work...
Closing the connection...
Connection closed.
```

语句中允许使用”using”和”await using”声明：“for”、“for-in”、“for-of”、“for-await-of”、“switch”。