作者：李立超

版本：1.1.1

更新日期：2023-08-18

代码仓库地址：https://gitee.com/ymhold/typescript.git

1. TypeScript概述

TypeScript是一种由微软开发的开源编程语言。它构建于JavaScript之上，是JavaScript的超集，这意味着所有的JS代码都是合法的TS代码（虽然有些JS代码会在编译时报出类型错误，当并不影响其正常执行）！和JavaScript的动态弱类型不同，TypeScript 提供了静态强类型功能。你可以在编写代码时为变量、参数和返回值指定明确的类型。这有助于捕获潜在的错误，例如：传递了错误的数据类型或尝试调用不存在的方法或属性，这些错误在TypeScript都可轻松避免。

TypeScript 在 JavaScript 的基础上增加了一系列新的特性和功能，其中最显著的特性就是其静态类型系统。以下是 TypeScript 相对于 JavaScript 提供的一些核心特性：

1. **静态类型系统**：这是 TypeScript 最主要的特性。你可以为变量、函数参数和函数返回值指定类型。这有助于在开发过程中就能及时发现并纠正错误，大大降低了运行时出错的概率。
2. **接口 (Interfaces)**：TypeScript 允许你定义接口来保证对象满足某种特定的结构。
3. **类 (Classes)**：虽然 JavaScript ES6 引入了类的概念，但 TypeScript 在此基础上增加了访问修饰符（如 `private` 和 `protected`）和抽象类等特性。
4. **泛型 (Generics)**：泛型允许你创建可重用的组件，这些组件可以支持多种类型的数据。
5. **高级类型特性**：如交叉类型、联合类型、类型别名、类型守卫、映射类型等。
6. **枚举 (Enums)**：枚举是一种为一组数值或其他值设置友好名称的方式。
7. **源码映射 (Source Maps)**：使你可以在原始 TypeScript 代码中进行调试，而不是生成的 JavaScript 代码。
8. **对 ES6+ 特性的支持**：TypeScript 支持最新的 JavaScript 特性，并允许将其转译为旧版本的 JavaScript 以确保兼容性。
9. **丰富的配置选项**：通过 `tsconfig.json` 文件，开发者可以细粒度地控制 TypeScript 编译器的行为。

总的来说，TypeScript 为开发大型、复杂的 JavaScript 项目提供了更强大、更安全的工具集，而这些在纯 JavaScript 开发中是不容易实现的。

为什么需要TypeScript

JavaScript 是一门历史悠久的编程语言。自1995年由 Netscape 的 Brendan Eich 创造以来，已经成为 Web 开发中不可或缺的一部分。起初，它只是为了给早期的 Web 页面增添一些简单的交互功能，但如今，它已经发展成为一个功能强大、广泛使用的语言，拥有庞大的生态系统和社区支持。

JavaScript 的历史悠久和其广泛的应用并不意味着它是完美的。与所有编程语言一样，JavaScript 有其自身的限制和问题。其中动态弱类型系统所带来的问题尤为严重！JavaScript 是一门动态弱类型语言，这意味着变量的类型在运行时才被确定，且可以随时更改。这给开发带来了一定的灵活性，但也增加了因类型错误导致的运行时错误的风险。正是因为动态弱类型才导致了JavaScript中的一系列的问题。

1. 变量可以任意赋值修改，调用方法或属性时编辑器无法预测变量的类型，导致错误直到代码运行时才抛出。
2. 变量没有明确的类型，无法快速的了解代码中变量和函数所需参数的类型，导致项目重构变得极为困难。
3. 由于缺乏静态类型系统，编辑器无法在程序员编写代码时给出恰当的提示，增加了开发难度。

微软认识到了JavaScript在大型应用开发中由于其动态类型特性可能带来的挑战和问题。为了解决这些问题，并为开发者提供一个更安全、更强大、具有更好工具支持的开发环境，微软推出了TypeScript。

TypeScript的主要特性就是它的静态类型系统。这意味着开发者可以在代码中为变量、函数参数和返回值定义类型，而这些类型信息在编译时会被检查，帮助捕获可能的错误。这样，TypeScript能够确保代码的类型安全性，避免运行时的类型错误。但TypeScript并不仅仅是类型系统。它还引入了其他JavaScript不具备的语言特性，如接口、枚举、泛型等，使得开发者可以编写更加结构化、更易维护的代码。微软的目标是让TypeScript成为大型应用开发的首选语言，同时也确保它与现有的JavaScript生态系统完全兼容。这意味着开发者可以逐步迁移到TypeScript，不需要从头开始。这种创新意味着开发者可以充分利用JavaScript的所有优点，同时享受TypeScript提供的安全性和其他高级特性。

如何使用TypeScript

要开始使用 TypeScript，你需要按照以下步骤操作：

1. 安装 Node.js：
首先，你需要安装 Node.js，因为 TypeScript 是一个 Node.js 的包。你可以访问Node.js官网 https://nodejs.org/来下载和安装 Node.js。安装完成后，你可以通过在命令行中输入 `node -v` 来验证安装是否成功，如果安装成功，它会显示你安装的 Node.js 的版本。

2. 安装 TypeScript：
通过 Node.js 的包管理器 npm，你可以非常容易地安装 TypeScript。在命令行中输入以下命令：

```
npm install -g typescript
```

`-g` 表示全局安装，这样你可以在任何地方都可以使用 TypeScript。安装完成后，你可以通过在命令行中输入 `tsc -v` 来验证安装是否成功，如果安装成功，它会显示你安装的 TypeScript 的版本。

3. 编写 TypeScript 代码：
现在，你可以开始编写 TypeScript 代码了。创建一个新的 `.ts` 文件，比如 `hello.ts`，然后写入以下代码：

```
function hello(name: string) {
  console.log(`Hello, ${name}!`);
}

hello("TypeScript");
```

这段代码定义了一个函数 `hello`，接受一个类型为 `string` 的参数 `name`，然后输出一条问候信息。

4. 编译 TypeScript 代码：
在命令行中输入以下命令来编译你的 TypeScript 代码：

```
tsc hello.ts
```

这会生成一个新的 JavaScript 文件 `hello.js`。TypeScript 编译器将 TypeScript 代码转换为 JavaScript 代码，这样它就可以在任何 JavaScript 环境中运行了。

5. 运行编译后的 JavaScript 代码：
最后，你可以使用 Node.js 来运行编译后的 JavaScript 代码：

```
node hello.js
```

你会看到命令行中输出了 `Hello, TypeScript!`，这就表示你成功地使用 TypeScript 编写并运行了代码。

除此之外，也可以通过在线编辑器来运行TS代码，比如：官方的代码编辑器：https://www.typescriptlang.org/play。这是 TypeScript 官方提供的在线代码编辑器。你可以在这里编写 TypeScript 代码，并看到它实时编译为 JavaScript 的结果。此外，TypeScript Playground 还提供了许多高级功能，比如查看 AST（抽象语法树），下载编译后的代码等。

tsc

`tsc`是TypeScript编译器的命令行工具，全称为TypeScript Compiler。你可以使用它将TypeScript源代码转换（编译）为JavaScript代码。

以下是一些基本的`tsc`命令：

- `tsc --init`：创建一个新的`tsconfig.json`文件。`tsconfig.json`文件是TypeScript项目的配置文件，它包含了编译选项和项目设置。
- `tsc`：没有任何参数的时候，这个命令会查找当前目录及其子目录下的`tsconfig.json`文件，然后根据这个文件的配置进行编译。
- `tsc <file>`：当给出一个TypeScript文件名时，`tsc`会编译这个文件，并生成一个同名的JavaScript文件。例如，`tsc hello.ts`会生成`hello.js`。
- `tsc -w`或`tsc --watch`：在监视模式下运行编译器。在此模式下，`tsc`会监视源文件的更改，并在每次更改时重新编译它们。
- `tsc --target <version>`：设置目标JavaScript版本。例如，`tsc --target ES2015`将会把TypeScript源码编译为ES2015版本的JavaScript代码。

通过`tsc --help`可以获取`tsc`的全部命令和选项。

tsconfig.json

`tsconfig.json`是一个包含了 TypeScript 项目配置的 JSON 文件。这个文件定义了项目中的根文件和编译选项，使你可以更精细地控制 TypeScript 项目的编译过程。

以下是一个基础的`tsconfig.json`文件的例子：

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "strict": true,
        "esModuleInterop": true
    },
    "include": ["src"],
    "exclude": ["node_modules"]
}
```

在这个例子中：

- `"compilerOptions"` 属性是用来设置 TypeScript 编译器选项的。
  - `"target"` 设置了编译后的代码需要支持的 ECMAScript 版本。
  - `"module"` 设置了编译后的代码使用的模块系统。
  - `"strict"` 设置了是否启用所有的严格类型检查选项。
  - `"esModuleInterop"` 允许我们导入 CommonJS 模块。
- `"include"` 属性是一个文件或文件夹的数组，告诉编译器应该编译哪些文件。在这个例子中，所有在 `src` 文件夹中的文件都会被编译。
- `"exclude"` 属性也是一个文件或文件夹的数组，告诉编译器应该忽略哪些文件。在这个例子中，`node_modules` 文件夹中的所有文件都会被忽略。

你可以在TypeScript 官方文档（https://www.typescriptlang.org/tsconfig）中找到 `tsconfig.json` 文件的所有可能选项。

默认文件解析

当你使用 TypeScript 进行项目编译而没有提供任何文件输入，或者没有在 `tsconfig.json` 中指定 `files` 或 `include` 选项时，TypeScript 编译器将进行默认的文件解析。以下是 TypeScript 默认文件解析的特点：

1. **包括哪些文件**:
   - 所有在项目目录（包含 `tsconfig.json` 的目录）及其所有子目录中的 `.ts` 和 `.tsx` 文件都会被包括在编译上下文中。
   - `.d.ts` 文件也会被自动包括，用于获取类型信息。
2. **排除哪些文件/目录**:
   - 文件和目录名以 `.` 开头的会被默认忽略。
   - `node_modules`、`bower_components`、`jspm_packages` 目录会被默认忽略。
   - 如果 `outDir` 选项在 `tsconfig.json` 中被指定，那么它所指定的输出目录也会被排除。
   - 你也可以使用 `exclude` 选项来明确指定要排除的文件或目录。但如果你没有指定 `exclude`，上述的默认排除规则会生效。
3. **如果既指定了 `files` 又指定了 `include`**:
   - 这种情况下，只有明确列出的文件和通过模式匹配的文件会被包括进来。

总之，TypeScript 默认文件解析是基于 `tsconfig.json` 文件所在位置的。它会包括所有的 `.ts` 和 `.tsx` 文件，除非它们符合上述的排除规则或者你明确使用了 `exclude` 选项进行排除。如果你想对文件解析进行更细致的控制，你可以使用 `files`、`include` 和 `exclude` 选项。

files

在 `tsconfig.json` 中，可以通过 `files` 选项来指定需要编译的文件列表。当你只想编译特定的文件，而不是依靠 TypeScript 的默认文件解析或其他 `include` 和 `exclude` 规则时，可以使用这个选项。

以下是关于 `files` 的一些关键点：

1. **基本用法**：`files` 是一个字符串数组，其中每个字符串都是一个文件的相对或绝对路径。

```
{
  "files": [
    "src/main.ts",
    "src/utilities/helpers.ts"
  ]
}
```

2. **明确指定**：与 `include` 和 `exclude` 选项不同，`files` 选项不支持文件模式匹配或通配符。
3. **与其他选项的关系**：如果 `files` 和 `include` 选项都被指定，那么结果是这两个列表的并集。换句话说，两个列表中的所有文件都会被包括进来。如果有 `exclude` 选项，那么它仍然会从这两个列表的并集中排除文件。
4. **自动关联**：即使在 `files` 列表中没有明确列出某个文件，但如果它被已列出文件通过 `import` 语句引用，它仍然会被包括在编译中。

总的来说，`files` 选项在某些特定场景中是有价值的，尤其是当你只想编译某个项目中的特定文件，或当你希望对编译的文件有更精确的控制时。但在大多数情况下，使用 `include` 和 `exclude` 选项可能更加方便和灵活。

include

`include` 是 `tsconfig.json` 文件中的一个选项，它允许你指定一个文件的模式数组，以确定哪些文件应该被包含在编译中。

以下是 `include` 选项的一些特点和用法：

1. **模式匹配**:
   - `include` 接受一个 glob 模式数组，用来匹配文件。常见的模式有 `*` (匹配0个或多个字符) 和 `?` (匹配1个字符)。
   - 你可以使用 `**` 来匹配任意深度的子目录。例如：`**/*.ts` 会匹配所有 `.ts` 文件，不论它们在哪个子目录中。
2. **默认行为**:
   - 如果 `files` 和 `include` 都没有指定，TypeScript 默认会包括当前目录及其子目录中的所有 `.ts`，`.tsx` 和 `.d.ts` 文件。
   - 如果 `include` 被指定，那么默认的文件包括规则就会被 `include` 中的规则所覆盖。
3. **与 `exclude` 一同使用**:
   - 使用 `include` 和 `exclude` 一起可以更精确地控制要编译的文件。首先，`include` 中的模式被解析，然后从这些匹配的文件中排除掉 `exclude` 中指定的模式匹配的文件。
4. **示例**:

```
{
    "include": ["src/**/*.ts", "typings/**/*.d.ts"],
    "exclude": ["node_modules"]
}
```

在上述的配置中，编译器会包括 `src` 目录及其所有子目录中的 `.ts` 文件以及 `typings` 目录及其所有子目录中的 `.d.ts` 文件，但是会排除 `node_modules` 目录。

注意：`include` 中的模式是相对于 `tsconfig.json` 文件的位置的。

exclude

`exclude` 是 `tsconfig.json` 文件中的一个选项，它告诉 TypeScript 编译器忽略某些文件或目录。这是在编译过程中排除不需要或不应该编译的文件的方法。

以下是一些关于 `exclude` 的要点：

1. **功能**: 它用于定义一个文件或目录的数组，这些文件或目录及其子内容不应被 TypeScript 编译器处理。
2. **默认值**: 如果没有提供 `exclude` 选项，但是提供了 `include` 选项，那么默认的 `exclude` 值会是 `["node_modules", "bower_components", "jspm_packages", "<outDir>"]`。
3. **与 include 的关系**: `exclude` 选项始终会排除某些文件，即使它们已经被 `include` 选项包括。
4. **路径和模式**: 和 `include` 选项一样，你可以为 `exclude` 使用相对或绝对路径、以及 glob 模式。例如: `["**/test/*.test.ts"]` 会排除任何目录下的所有 `.test.ts` 文件。
5. **注意**: 即使你已经在 `exclude` 中列出了文件或目录，如果该文件在其他 TypeScript 文件中被明确引用，或者被列在 `files` 选项中，它仍然会被编译。

举一个简单的例子：

```
{
  "compilerOptions": {
    // ...其他编译选项
  },
  "include": ["src/**/*.ts"],
  "exclude": ["src/tests/**/*"]
}
```

在上面的配置中，我们告诉 TypeScript 只处理 `src` 目录下的 `.ts` 文件，但是忽略 `src/tests` 目录及其所有子内容。

extends

一个 TypeScript 项目可以包含多个 `tsconfig.json` 配置文件。这在大型项目中尤其有用，你可能需要对不同部分的代码应用不同的编译器选项。比如，你可能希望测试代码能够使用一些开发环境下的特性，但是产品代码需要被编译到更低版本的 ECMAScript 来支持更广泛的浏览器。

在这种情况下，你可以创建多个 `tsconfig.json` 文件，每个文件针对项目的不同部分。TypeScript 编译器可以使用 `-p` 或 `--project` 参数来指定要使用的配置文件，如 `tsc -p tsconfig.prod.json`。

还可以在一个 `tsconfig.json` 文件中使用 `"extends"` 属性来继承另一个 `tsconfig.json` 文件中的配置，然后覆盖或添加其他选项。例如：

```
{
    "extends": "./tsconfig.base.json",
    "compilerOptions": {
        "outDir": "./dist",
        "rootDir": "./src"
    }
}
```

在这个例子中，新的 `tsconfig.json` 文件继承了 `tsconfig.base.json` 中的所有配置，然后设置了新的 `outDir` 和 `rootDir` 选项。这个特性在你想要共享一组基本配置，但对某些配置进行覆盖或添加的情况下非常有用。

调用过程

当你调用 `tsc`（TypeScript Compiler）命令时，它会进行一系列的步骤来将 TypeScript 源码转换成 JavaScript 代码：

1. **解析**：编译器会首先解析 TypeScript 代码，将源码转换为一个抽象语法树（AST）。这个过程也会检查代码中的语法错误。
2. **类型检查**：在生成 AST 之后，编译器会进行类型检查。这个步骤会查找类型错误，例如尝试将字符串赋值给一个数值类型的变量。
3. **转换**：编译器接着会将 TypeScript 的 AST 转换为一个新的 AST，这个新的 AST 是 JavaScript 的 AST。这个过程会移除 TypeScript 特有的语法（如类型注解和接口）并可能根据目标 JavaScript 版本进行必要的语法转换（如将 ES6 的箭头函数转换为 ES5 的函数表达式）。
4. **生成代码**：最后，编译器会将 JavaScript 的 AST 转换为可执行的 JavaScript 代码。这个步骤也会生成源映射（source maps）以支持调试（如果你在编译选项中启用了 source map）。

以上就是 `tsc` 命令执行时的主要步骤。在实际的开发中，你可能并不需要关心这些详细的步骤，但了解它们可以帮助你更好地理解 TypeScript 是如何工作的。

静态类型检查

"Static type-checking"（静态类型检查）是在编程语言中，在代码运行之前进行类型检查的过程。这种类型检查方式是在编译时进行的，而不是在运行时。这意味着，如果你在代码中犯了类型错误（比如，尝试将一个字符串赋值给一个期望数字的变量），那么这个错误将在你运行代码之前被捕获。

静态类型检查的主要优点是：

1. **错误检测：** 它可以在编译时而不是运行时捕获错误，这样可以尽早地发现并修复问题，而不是在运行程序时突然遇到错误。
2. **代码可读性：** 通过在代码中明确指出变量和函数的类型，其他的开发者可以更好地理解代码的行为和意图。
3. **自动完成和重构：** 在有静态类型信息的情况下，开发者工具（如IDE和编辑器）可以提供更好的自动完成和重构功能。

JavaScript是一个动态类型的语言，这意味着它的类型检查是在运行时进行的。但TypeScript添加了静态类型检查，这使得在JavaScript的灵活性基础上，还能享受到静态类型检查带来的好处。

TypeScript 编译器会进行以下几种主要的检查：

1. **类型检查**：这是 TypeScript 的核心特性，它会检查变量和值的类型是否匹配，函数调用是否正确，等等。例如，如果你试图将字符串赋值给一个数字类型的变量，TypeScript 会报告错误。
2. **标识符检查**：TypeScript 会检查你使用的所有变量、函数、类等是否已经被定义。如果你试图使用一个未定义的变量，TypeScript 会报告错误。
3. **属性检查**：当你访问一个对象的属性时，TypeScript 会检查该属性是否存在。例如，如果你试图访问一个不存在的属性，TypeScript 会报告错误。
4. **函数和方法调用检查**：当你调用一个函数或方法时，TypeScript 会检查你是否正确地传递了所有必需的参数，是否传递了正确类型的参数，等等。
5. **模块和命名空间检查**：TypeScript 会检查你导入的模块或使用的命名空间是否存在，并且检查你是否正确地导出和导入了模块成员。
6. **null 和 undefined 检查**：如果你启用了 `strictNullChecks` 选项，TypeScript 会更严格地检查 null 和 undefined 的使用。
7. **泛型和类型参数检查**：当你使用泛型或类型参数时，TypeScript 会检查你是否正确地使用了它们。
8. **额外的严格检查**：如果你启用了 `strict` 选项，TypeScript 会进行更多的严格检查，例如不允许隐式的 any 类型，检查函数是否有返回值，等等。

以上这些检查帮助开发者写出更安全、更可靠的代码，也是 TypeScript 受到欢迎的重要原因之一。

类型工具

"Types for Tooling"（类型工具） 是 TypeScript 的一个关键概念，它指的是 TypeScript 的类型系统如何帮助开发者提高编程效率和代码质量。这个概念主要有以下几个方面：

1. **代码提示和自动补全**：当你在编写代码时，TypeScript 可以根据变量和函数的类型提供智能的代码提示和自动补全。这不仅可以帮助你更快地编写代码，还可以避免因拼写错误等问题引发的错误。
2. **类型检查和错误提示**：TypeScript 会在编译阶段进行类型检查，如果发现类型错误或其他问题，它会立即报告错误。这可以在代码运行之前就发现和修复错误，从而提高代码质量。
3. **代码导航和重构**：TypeScript 可以根据类型信息提供强大的代码导航和重构功能。例如，你可以轻松地找到一个变量或函数的定义位置，或者安全地重命名一个变量或函数，而不用担心会破坏代码的正确性。
4. **API 文档和签名提示**：当你使用一个函数或方法时，TypeScript 可以显示该函数或方法的 API 文档和签名信息。这可以帮助你理解和正确使用 API。
5. **集成开发环境（IDE）支持**：许多现代 IDE，如 Visual Studio Code、WebStorm 等，都有对 TypeScript 的良好支持。在这些 IDE 中，你可以享受到上述所有的 "Types for Tooling" 功能，以及其他许多高级功能，如代码格式化、代码质量检查等。

总的来说，类型工具 是 TypeScript 的一个重要优势，它可以极大地提高开发者的编程效率和代码质量。

2. 基本类型

类型注解

类型注解是 TypeScript 中的一种重要特性。通过在变量或函数的声明中指定类型信息，我们可以告诉 TypeScript 编译器这些变量期望的类型是什么。编译器会使用这些信息来进行静态类型检查，这有助于在编译时就发现并修复错误。

类型注解的基本语法是在变量名或函数参数名后面加上冒号和类型，如下所示：

```
let name: string;
let age: number;
let isStudent: boolean;

function greet(person: string, age: number): string {
  return `Hello ${person}, you are ${age} years old.`;
}
```

在上面的代码中，`name` 的类型被注解为 `string`，`age` 的类型被注解为 `number`，`isStudent` 的类型被注解为 `boolean`。而 `greet` 函数则期望接收两个参数，第一个是 `string` 类型，第二个是 `number` 类型，同时，函数返回的结果也应为 `string` 类型。

如果我们试图给这些变量赋予与其类型注解不符的值，或者以错误的参数类型调用函数，TypeScript 编译器就会报错。例如，以下代码将会在编译时产生错误：

```
name = 123; // Error: Type 'number' is not assignable to type 'string'.
greet(123, 'John'); // Error: Argument of type 'number' is not assignable to
parameter of type 'string'.
```

这样，类型注解可以帮助我们确保变量和函数在使用时能遵循正确的类型，从而提高代码的可读性和可维护性。

类型推断

类型推断（Type Inference）。这意味着当你在没有明确指出类型的情况下声明变量或赋值时，TypeScript 编译器会自动推断出变量的类型。

例如：

```
let num = 2; // 类型推断为 'number'

let message = 'Hello World'; // 类型推断为 'string'
```

在上述代码中，虽然我们没有显式地为变量 `num` 和 `message` 指定类型，但是由于初始赋值，TypeScript 编译器能够推断出 `num` 的类型为 `number`，`message` 的类型为 `string`。

值得注意的是，TypeScript 的类型推断并不总是完全准确，有时我们需要使用类型注解或类型断言来帮助编译器确定准确的类型。

常用类型

原始值：string、number、boolean

在TypeScript中，`string`，`number`和`boolean`是基础的数据类型，以下是对这三种类型的详细描述：

1. **String（字符串）**：在TypeScript中，您可以使用双引号（" "）或单引号（' '）来表示字符串，也可以使用模板字符串（反引号包围的字符串，`${expression}`）来创建多行文本和插入表达式。例如：

```
let name: string = 'bob';
name = "smith";
let sentence: string = `Hello, my name is ${ name }.
I'll be ${ age + 1 } years old next month.`;
```

2. **Number（数字）**：在TypeScript中，所有的数字都是浮点数。除了十进制和十六进制，ES6中引入的二进制和八进制也可以使用TypeScript。例如：

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

3. **Boolean（布尔值）**：最基础的数据类型就是简单的真值逻辑，它可以是`true`或`false`。例如：

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

这些类型是JavaScript的基础类型，并且TypeScript只是为它们添加了类型注解功能。这对于强类型语言的开发人员在理解和使用JavaScript这种动态语言时，提供了很大的帮助。

数组

在 TypeScript 中，你可以使用两种方式来声明数组类型：

1. **类型 + 方括号 []**：表示由某种类型元素构成的数组。

```
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: string[] = ['a', 'b', 'c'];
```

在这个例子中，`number[]`表示由`number`类型元素构成的数组，`string[]`表示由`string`类型元素构成的数组。

2. **泛型 Array<元素类型>**：表示由某种类型元素构成的数组。

```
let numbers: Array<number> = [1, 2, 3, 4, 5];
let strings: Array<string> = ['a', 'b', 'c'];
```

在这个例子中，`Array<number>` 表示由 `number` 类型元素构成的数组，`Array<string>` 表示由 `string` 类型元素构成的数组。

两种声明方式是等价的，你可以根据个人习惯选择使用哪一种。不过在大多数情况下，类型 + 方括号 `[]` 的写法更为常见。

另外，数组中也可以包含多种类型的元素。在这种情况下，你可以使用联合类型（union types）来声明数组类型。例如：

```
let array: (number | string)[] = [1, 'a', 2, 'b'];
```

在这个例子中，`(number | string)[]` 表示数组中的元素可以是 `number` 类型，也可以是 `string` 类型。

any

在 TypeScript 中，`any` 类型是一种特殊的类型，它允许你在编译时跳过类型检查。

当你声明一个变量为 `any` 类型，你可以为这个变量赋任何值，也可以调用该变量上的任何方法，而 TypeScript 编译器都不会报错。这意味着，使用 `any` 类型，你可以写出和纯 JavaScript 一样动态和灵活的代码。

```
let anything: any = 'hello';
anything = 42;
anything = [1, 2, 3];
anything.foo.bar(); // 不会报错
```

虽然 `any` 类型非常灵活，但是过度使用 `any` 类型将剥夺 TypeScript 提供的类型检查功能。因此，除非有特殊的需要，否则最好尽可能少地使用 `any` 类型。

unknown

在TypeScript中，`unknown` 类型被称为顶级类型（top type），这意味着它可以代表所有可能的类型。它与 `any` 类型相似，但更安全，因为我们不能对 `unknown` 类型的值执行大多数操作，直到我们对其类型进行某种类型检查。

例如，考虑以下函数，它接受一个类型为 `unknown` 的参数：

```
function printValue(value: unknown) {
    console.log(value);
}

printValue(123); // OK
printValue('hello'); // OK
printValue({ a: 1 }); // OK
```

在这个例子中，`printValue`函数可以接受任何类型的参数，因为参数类型被定义为`unknown`。然而，我们不能对`value`执行除了赋值和比较之外的任何操作，除非我们首先执行类型检查。

```
function validateValue(value: unknown) {
    if (typeof value === 'string') {
        console.log(value.toUpperCase()); // OK, because we know value is a string
here
    }

    console.log(value.toFixed(2)); // Error, value could be anything
}
```

在上面的例子中，我们只能在检查了`value`是否为字符串之后才能安全地调用`toUpperCase`方法。如果我们试图在没有进行类型检查的情况下调用`toFixed`方法，TypeScript将会报错。

因此，`unknown`类型允许我们在类型安全的情况下接受任何类型的值，这使得它在编写库或者其他需要处理未知类型的代码时非常有用。

对象类型

在 TypeScript 中，除了可以使用原始类型（如`number`、`string`、`boolean`等）来声明变量，你还可以使用对象类型来描述更复杂的数据结构。

最基本的对象类型是一个普通的 JavaScript 对象，它可以由`{}`符号来表示。你可以在`{}`符号中定义对象的属性及其类型。

例如：

```
let obj: { a: number; b: string } = { a: 1, b: 'hello' };
```

上述代码中，变量`obj`的类型是一个对象类型，该对象有两个属性，`a`属性的类型为`number`，`b`属性的类型为`string`。

此外，你还可以使用接口（`interface`）或类（`class`）来定义更复杂的对象类型：

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

let person: Person = { name: 'Alice', age: 20 };
```

```
class Animal {
  name: string;
  age: number;
}

let animal: Animal = new Animal();
animal.name = 'Dog';
animal.age = 5;
```

在上述两个例子中，`Person` 和 `Animal` 都定义了一个具有 `name` 和 `age` 属性的对象类型。这样，你就可以使用这些自定义的对象类型来声明变量，使得代码更具有可读性和可维护性。

在 TypeScript 中，对象类型的属性可以被声明为可选的。这意味着，这个属性可能不存在于对象中。你可以在属性名后面加上 `?` 符号来表示这个属性是可选的。

例如，假设我们有一个表示人的接口 `Person`，它有一个必需的 `name` 属性和一个可选的 `age` 属性：

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

在这个例子中，`age` 属性是可选的。这意味着，你可以创建一个只有 `name` 属性的 `Person` 对象：

```
let person: Person = { name: 'Alice' };
```

但是，如果你试图访问 `person` 对象的 `age` 属性，TypeScript 将不会报错，因为 `age` 属性是可选的。然而，如果你试图访问 `person` 对象的一个不存在的属性，比如 `height`，那么 TypeScript 就会报错，因为 `height` 属性没有在 `Person` 接口中被定义：

```
let height = person.height;  // Error: Property 'height' does not exist on type
'Person'
```

这种方式可以让你更灵活地处理对象，不需要为每个可能存在的属性都声明一个字段，同时还可以保持类型安全。

object

在TypeScript中，`object`类型代表非原始类型，也就是除了`number`，`string`，`boolean`，`bigint`，`symbol`，`null`，或`undefined`之外的类型。

例如，你可以定义一个接受object类型的函数：

```
function create(o: object): void {
  //...
}

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

在上面的例子中，`create`函数接受一个类型为`object`的参数。你可以传入一个对象或者`null`，但如果你尝试传入`number`，`string`，`boolean`或者`undefined`，TypeScript将会报错。

然而需要注意的是，`object`类型的使用通常在你希望接受任何对象值时才有意义，而不是某种具体类型。如果你知道更具体的类型，你应该使用那个更具体的类型。例如，如果你知道一个对象有一个叫做`name`的`string`类型的属性，你应该使用`{ name: string }`类型，而不是`object`类型。

联合类型

在 TypeScript 中，联合类型（Union Types）用于表示变量可以是多种类型中的一种。联合类型使用管道符号（`|`）来分隔多个类型。例如，我们可以表示一个变量可以是`string`或`number`类型：

```
let id: string | number;
```

在这个例子中，变量`id`可以被赋予字符串或数字值：

```
id = '123';  // OK
id = 123;    // OK
```

联合类型特别有用，因为 JavaScript 是动态类型的语言，有些函数可能接受多种类型的参数。例如，你可能有一个函数，它接受一个字符串（代表元素的 ID）或者一个数字（代表元素在数组中的索引）：

```
function getElement(id: string | number) {
  // ...
}
```

在这个函数中，`id` 参数可以是 `string` 或 `number` 类型。

请注意，使用联合类型时，你只能访问所有类型共有的属性和方法。如果你尝试访问一个不存在于所有类型中的属性或方法，TypeScript 将会报错。

要在 TypeScript 中调用联合类型中某个类型独有的方法或属性，你需要使用类型断言或类型守卫来明确告诉 TypeScript 你正在使用的具体类型。

1. 类型断言：使用类型断言（Type Assertion）可以告诉编译器，你确定你知道你正在处理的具体类型。在 TypeScript 中，你可以使用 `<Type>` 或 `as Type` 的语法来进行类型断言。

   ```
   let value: number | string;
   value = 'myString';

   // 使用 <Type> 语法
   let length1: number = (<string>value).length;

   // 使用 as Type 语法
   let length2: number = (value as string).length;
   ```

2. 类型守卫：类型守卫（Type Guard）是一种检查联合类型变量当前所持有的类型的方式。最常见的类型守卫方式是使用 `typeof` 或 `instanceof` 关键字，还可以使用自定义的类型守卫函数。

   ```
   function getLength(value: number | string): number {
      if (typeof value === 'string') {
         // 在这个代码块中，TypeScript 知道 value 是 string 类型
         return value.length;
      } else {
         // 在这个代码块中，TypeScript 知道 value 是 number 类型
         return value.toString().length;
      }
   }
   ```

以上两种方式都可以确保你在调用联合类型变量的特定类型方法或属性时，不会出现类型错误。

类型别名

在 TypeScript 中，类型别名（Type Aliases）可以用来为一个类型表达式创建新的名字。类型别名并不创建新的类型，而只是创建一个新的名字来引用那个类型。类型别名可以用于简化复杂类型的书写，也可以用于创建可重用的类型。

你可以使用 `type` 关键字来定义类型别名。例如：

```
type StringOrNumber = string | number;
```

在这个例子中，`StringOrNumber` 是 `string | number` 的类型别名。现在你可以在任何需要使用 `string | number` 的地方使用 `StringOrNumber`。

类型别名可以用于任何类型，不仅仅是联合类型。你也可以为原始类型、字面量类型、接口类型、交叉类型、元组类型等等创建类型别名。

例如，你也可以为一个对象类型创建类型别名：

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

这个 `User` 类型可以在需要使用这个对象类型的地方进行重用。例如，你可以在函数参数中使用它：

```
function greet(user: User) {
  return `Hello, ${user.name}!`;
}
```

接口

在 TypeScript 中，接口（Interfaces）是用来定义复杂的类型的一种工具。接口主要用于定义对象的结构。一个接口可以描述对象的属性、方法，甚至可以描述函数的签名。

接口是 TypeScript 特有的语法，JavaScript 中没有接口这个概念。

下面是一个基本的接口示例：

```
interface Person {
  firstName: string;
  lastName: string;
}

function greet(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Jane", lastName: "User" };

console.log(greet(user));
```

在这个例子中，`Person` 接口定义了一个类型，这个类型包含两个字符串属性：`firstName` 和 `lastName`。`greet` 函数的参数 `person` 被注解为 `Person` 类型，这意味着我们只能向 `greet` 函数传入符合 `Person` 类型的对象。

接口可以被类（classes）实现（implements），也可以被扩展（extends）。接口还可以用来描述函数类型，索引类型（indexable types），还可以与类型别名一起工作，提供了更为强大的类型定义能力。

尽管类型别名和接口在许多情况下可以互换使用，但它们的一个关键区别在于接口是开放的，可以随时添加新的属性，而类型别名一旦创建，就无法更改。

1. **接口的扩展**：接口可以通过extends关键字进行扩展，创建一个包含父接口所有属性的新接口。

```
interface Animal {
  name: string;
}

interface Bear extends Animal {
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```

2. **类型别名的扩展**：类型别名可以通过交集操作符(&)扩展已存在的类型，但是并不能像接口那样直接使用extends关键字进行扩展。

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

type Bear = Animal & {
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```

3. **接口的修改**：接口是开放的，你可以在任何地方添加新的属性。这就意味着你可以在不同的地方定义同名的接口，TypeScript 会将它们视为同一接口的不同部分，并将这些部分合并在一起。

```
interface Window {
  title: string;
}

interface Window {
  ts: TypeScriptAPI;
}

const src = 'const a = "Hello World"';
window.ts.transpileModule(src, {});
```

4. **类型别名的修改**：类型别名一旦创建，就无法更改。如果你试图给一个已经存在的类型别名添加新的字段，TypeScript 会报错。

```
type Window = {
  title: string;
}

type Window = {
  ts: TypeScriptAPI;
}

 // Error: Duplicate identifier 'Window'.
```

总的来说，类型别名和接口各有优势。当你需要一个不能改变的类型，或者需要表示联合类型、交叉类型、原始类型时，可以使用类型别名。当你需要一个可以在任何地方添加新属性的类型，或者需要描述类的公共接口时，可以使用接口。

类型断言

类型断言（Type Assertions）是 TypeScript 的一种语法，允许你在编程时指定一个更具体的类型。在运行时，类型断言不会有任何影响，仅在编译时产生作用。

在 TypeScript 中，你可以使用两种语法进行类型断言：尖括号语法和as语法。

1. 尖括号语法：

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
```

在上述示例中，我们知道 `someValue` 实际上是一个字符串，所以我们想要获取它的长度属性。由于 `someValue` 的类型是 `any`，TypeScript 不会假定它有 `.length` 属性。于是我们使用 `<string>` 语法将 `someValue` 断言为 `string` 类型，这样我们就可以访问 `.length` 属性了。

2. as语法：

```
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;
```

在上述示例中，我们使用 `as string` 语法将 `someValue` 断言为 `string` 类型。这与使用 `<string>` 完全等价。

注意，在使用 JSX 时，只有 `as` 语法断言是被允许的。

类型断言的前提是你确切地知道你在做什么，因为它将会跳过类型检查器的部分检查。所以在使用类型断言时一定要小心，尽量避免错误地断言类型。

字面量类型

字面量类型是一种特殊的子类型，它们的取值范围被限定在一个具体的值。在 TypeScript 中，你可以使用字面量类型来具体地描述一个值的取值范围。字面量类型包括字符串字面量类型、数字字面量类型和布尔字面量类型。

1. **字符串字面量类型**：它用来约束取值只能是某几个字符串中的一个。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

在上述示例中，`Easing` 类型只能取 `ease-in`、`ease-out` 或 `ease-in-out` 中的一个。

2. **数字字面量类型**：它与字符串字面量类型类似，只是它的取值被约束在某些特定的数字上。

```
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;
```

在上述示例中，`DiceRoll` 类型的取值只能是 1 到 6 之间的整数。

3. **布尔字面量类型**：布尔字面量类型可以是 `true` 或 `false`。

```
type IsTrue = true;
```

在上述示例中，`IsTrue` 类型的取值只能是 `true`。

字面量类型在某些特定的情况下非常有用，比如当你需要描述一个值的取值范围时，或者当你需要对某些特定的值进行类型检查时。

字面量推断

在 TypeScript 中，Literal Inference（字面量推断）是指 TypeScript 编译器根据字面量的值来推断其类型的功能。

当你使用 `let` 或 `var` 声明一个变量并直接赋值，TypeScript 通常会推断出一个“宽”类型，如 `string`、`number` 或 `boolean`。然而，如果你使用 `const` 来声明变量，TypeScript 将推断出一个“窄”类型，即字面量类型。例如：

```
let x = "hello";   // type of x is string
const y = "world"; // type of y is "world"
```

在上面的代码中，`x` 的类型被推断为 `string`，而 `y` 的类型被推断为 `"world"`。

当你使用字面量对象或数组时，TypeScript 会推断出这些对象或数组的结构，并将它们的类型设置为对应的属性或元素的类型：

```
const point = { x: 0, y: 0 }; // type of point is { x: number; y: number; }
const numbers = [1, 2, 3];    // type of numbers is number[]
```

在这些例子中，`point` 的类型被推断为 `{ x: number; y: number; }`，`numbers` 的类型被推断为 `number[]`。

注意，你可以使用 `as const` 断言来创建只读的字面量类型，这将使得 TypeScript 推断出更具体的类型：

```
const str = "hello" as const;  // type of str is "hello"
const arr = [10, "hello"] as const; // type of arr is readonly [10, "hello"]
```

在这些例子中，`str` 的类型被推断为 `"hello"`，而 `arr` 的类型被推断为 `readonly [10, "hello"]`。

null和undefined

在TypeScript中，`null`和`undefined`都有各自的类型，分别名为`null`和`undefined`。它们并不是非常有用，因为你只能为它们赋予各自的类型值。在 TypeScript 中，默认情况下`null`和`undefined`是所有其它类型的子类型，这就意味着你可以将`null`和`undefined`赋值给它们，比如说`number`类型的变量。

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

// These would cause an error in --strictNullChecks mode
let num: number = undefined;
let str: string = null;
```

然而，在严格的空检查(`--strictNullChecks`)模式下，`null`和`undefined`只能赋值给它们自己的类型以及`void`。这可以避免很多常见的问题。例如，如果你有一个可能为`null`的字符串，你可以使用`string | null`来表示。

```
let s: string | null = "Hello";
s = null; // OK
```

这个模式可以帮助捕获许多常见的错误，因为你现在必须显式地处理`null`和`undefined`。例如，如果你有一个可能为`null`的对象，你在访问它的属性之前必须检查它是否为`null`。

```
function doSomething(x: string | null) {
    if (x === null) {
        // do nothing
    } else {
        console.log("Hello, " + x.toUpperCase());
    }
}
```

在这个例子中，如果你试图直接调用`x.toUpperCase()`，TypeScript 会报错，因为`x`可能为`null`。

> `strictNullChecks`是 TypeScript 的一个编译选项，该选项可以使得 TypeScript 对`null`和`undefined`值的处理更加严格。具体来说，当你开启`strictNullChecks`选项后，`null`和`undefined`将只能被赋值给它们各自的类型以及`void`类型，同时它们也不能赋值给其它任何类型的变量。
>
> 在没有开启`strictNullChecks`的情况下，`null`和`undefined`是所有类型的子类型，你可以将它们赋值给任何类型的变量。但是这样做可能会导致一些错误，因为大部分情况下，我们并不希望变量的值为`null`或`undefined`。

> 当你开启 `strictNullChecks` 后，如果你的变量可能会有 `null` 或 `undefined` 的值，你需要显式地使用联合类型来表示。例如，如果你有一个可能为 `null` 的字符串变量，你可以这样声明它：`let s: string | null;`。
>
> `strictNullChecks` 选项可以帮助我们在编译时期就发现可能的空引用错误，而不是在运行时才发现这些错误。

void

在 TypeScript 中，`void` 是一个重要的类型，主要用于表示函数没有返回值。你可以在函数声明中使用 `void` 关键字来标明该函数没有返回值。这里是一个例子：

```
function warnUser(): void {
    console.log("This is a warning message");
}
```

在这个例子中，`warnUser` 函数没有返回任何值，所以它的返回类型被设置为 `void`。

需要注意的是，`void` 类型的变量在 TypeScript 中并没有太大的用处，因为你只能为它赋予 `undefined` 和 `null` 这两个值。

`void` 与 `any` 和 `unknown` 不同，`void` 表示函数没有返回值，而 `any` 和 `unknown` 用于表示类型不确定或者可以是任何类型。

在返回类型为void的上下文类型化中，并不强制函数不返回某些东西。换句话说，具有void返回类型的上下文函数类型（类型voidFunc = () => void），在实现时，可以返回任何其他值，但这些值将被忽略。

因此，以下类型为() => void的实现都是有效的：

```
type voidFunc = () => void;

const f1: voidFunc = () => {
  return true;
};

const f2: voidFunc = () => true;

const f3: voidFunc = function () {
  return true;
};
```

当将这些函数的返回值赋值给另一个变量时，它将保留void类型：

```
const v1 = f1();

const v2 = f2();

const v3 = f3();
```

这种行为的存在是为了让以下代码有效，即使Array.prototype.push返回一个数字，而Array.prototype.forEach方法期望一个返回类型为void的函数。

```
const src = [1, 2, 3];
const dst = [0];

src.forEach((el) => dst.push(el));
```

还有一个特殊的情况需要注意，当一个字面量函数定义有一个void返回类型，那么这个函数必须不返回任何东西。

```
function f2(): void {
  // @ts-expect-error
  return true;
}

const f3 = function (): void {
  // @ts-expect-error
  return true;
};
```

never

在 TypeScript 中，`never` 类型表示的是那些永不存在的值的类型。也就是说，如果你声明一个变量是 `never` 类型，那么你就不能给这个变量赋予任何值。

在实际开发中，`never` 类型通常用于以下两种情况：

1. 在函数内部，我们使用 `throw new Error('some error')` 抛出一个错误，使函数无法正常结束，则你可以将这个函数的返回值类型定义为 `never`。

```
function throwError(message: string): never {
    throw new Error(message);
}
```

在上述代码中，函数 `throwError` 的返回类型是 `never`，这意味着这个函数永远不可能有返回值。

2. 当 TypeScript 类型系统检测到一个条件永远也不会为真时，它也会推断出这个类型为 `never`。

```
function checkExhaustiveness(val: never): never {
    throw new Error(`Unexpected value: ${val}`);
}

type Fruit = 'apple' | 'orange';

function processFruit(fruit: Fruit) {
    switch (fruit) {
        case 'apple':
            eatApple();
            break;
        case 'orange':
            eatOrange();
            break;
        default:
            // The following error can help catch when you've forgotten to handle a
case
            checkExhaustiveness(fruit);
    }
}
```

在上述代码中，如果我们忘记处理一种水果，那么 TypeScript 就会在 `checkExhaustiveness(fruit)` 这一行抛出错误，因为它期望 `fruit` 的类型为 `never`，也就是说，这个分支永远都不应该被执行到。如果我们确实忘记处理一种水果，那么 `fruit` 的类型就不会是 `never`，从而导致 TypeScript 报错。

非空断言

在 TypeScript 中，非空断言运算符（Postfix !）是一个后缀运算符，可以用于告诉 TypeScript 编译器一个表达式总是会返回非空值。

当你明确知道一个表达式不会产生 null 或 undefined 的时候，可以使用非空断言运算符。例如，当你在处理 DOM 元素或者有一些默认值时，可能会经常使用这个运算符。

这是一个使用非空断言运算符的示例：

```
function myFunction(elementId: string): void {
  const el = document.getElementById(elementId);

  // the "!" at the end asserts that el is non-null
  el!.innerText = 'Hello, world!';
}
```

在这个例子中，`document.getElementById` 会返回 HTMLElement 或 null。但是如果你确定 elementId 总是会匹配一个存在的元素，就可以使用 `!` 后缀，这样 TypeScript 就不会报错说 `el` 可能为 null。

请注意，滥用非空断言运算符可能会导致问题，因为它基本上是在告诉编译器忽略可能的 null 或 undefined。所以，在你确定表达式绝对不会返回 null 或 undefined 的时候，才应该使用非空断言运算符。

类型收窄

"Narrowing"（类型收窄）是 TypeScript 中的一个重要概念。当你有一个更宽泛的类型（如 `any` 或 `unknown`）的值，然后尝试将其限制（收窄）为更具体的类型时，你就在做类型收窄。

这是 TypeScript 用来进行类型检查的一种方式，帮助你在编译时捕获可能的错误。类型收窄通常通过使用类型守卫（如 `typeof` 检查，`instanceof` 检查，或用户自定义的类型守卫函数），或使用类型断言来实现。

下面是一个简单的例子，使用 `typeof` 来收窄类型：

```
let value: any = "Hello, TypeScript!";
if (typeof value === "string") {
  console.log(value.toUpperCase()); // OK, because value is narrowed to string
}
```

在这个例子中，`value` 的初始类型是 `any`。但是在 `typeof value === "string"` 的检查之后，`value` 的类型被收窄为 `string`，因此可以安全地调用 `toUpperCase()` 方法。

类型收窄是 TypeScript 提供的强大功能之一，它可以帮助你编写更安全，更具可维护性的代码。

控制流分析

TypeScript 是一种强类型的编程语言，它的类型系统有助于我们编写更健壮、更安全的代码。在 TypeScript 中，类型收窄用于在代码的特定部分中，将变量的类型从更广泛的类型收窄为更具体的类型。那么，我们如何让 TypeScript 知道某个变量在某个特定的代码块中具有特定的类型？这就涉及到一个被称为“控制流分析”的过程。

控制流分析的主要目标是理解代码在运行时的行为，并据此推断出变量的类型。让我们通过一个简单的例子来说明这个过程：

```
function padLeft(padding: number | string, input: string) {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
  }
  return padding + input;
}
```

在这个 `padLeft` 函数中，我们首先定义了一个带有两个参数的函数。第一个参数 `padding` 的类型为 `number | string`，第二个参数 `input` 的类型为 `string`。我们的目标是根据 `padding` 的类型返回一个新的字符串。

让我们来看看这个函数的控制流：

1. 首先，我们进入函数体。此时，`padding` 的类型是 `number | string`，因为这是它的原始类型。
2. 然后，我们看到一个 `if` 语句，它用 `typeof` 检查 `padding` 是否是 `number` 类型。这个 `if` 语句导致了控制流的分支。
3. 如果 `padding` 是 `number` 类型，那么我们将进入 `if` 语句块。在这个语句块中，TypeScript 使用 `typeof` 检查作为类型守卫，收窄 `padding` 的类型为 `number`。然后我们生成一个由空格组成的字符串（长度由 `padding` 确定），并将 `input` 连接在其后。最后，我们返回这个新的字符串。这个语句块执行完后，函数就结束了。
4. 如果 `padding` 不是 `number` 类型，那么我们会跳过 `if` 语句块，执行 `else` 语句块（这里省略了 `else`）。由于 `padding` 可能的类型只有 `number` 和 `string`，并且 `number` 类型已经在 `if` 语句块中被处理了，所以在 `else` 语句块中，`padding` 的类型就被收窄为 `string`。我们直接将 `padding` 与 `input` 连接，并返回结果。

这就是控制流分析在这个函数中的运用。通过控制流分析，我们能够在不同的代码路径中推断出变量的更精确的类型。

详细解释控制流分析的运行机制，我们可以说它是通过跟踪代码的可能路径来收窄变量的类型。在这个过程中，有几个关键的概念需要理解：分支（branch）、合并（merge）和收窄（narrowing）。

在更复杂的情况下，比如循环和函数调用，TypeScript 也有相应的处理方式。例如，对于循环，TypeScript 会分析循环体内的代码，并根据这些代码收窄循环变量的类型。

最后，我们需要认识到控制流分析的重要性，并理解它是如何为 TypeScript 的类型安全性提供基础的。虽然 TypeScript 的类型系统提供了强大的工具，但它仍然依赖于我们编写的代码。因此，理解并正确使用控制流分析可以帮助我们编写出更健壮、更安全的 TypeScript 代码。

typeof 类型守卫

`typeof` 类型守卫是TypeScript内置的一种类型守卫机制，它允许我们根据变量的 `typeof` 值来收窄变量的类型。

在JavaScript中，`typeof` 运算符可以返回一个字符串，表示其操作数的类型。例如，`typeof 123` 返回 `"number"`，`typeof "hello"` 返回 `"string"`。

在TypeScript中，当 `typeof` 用在 `if`、`switch`、`while` 等控制流语句中时，TypeScript会根据 `typeof` 的结果来自动收窄变量的类型。这被称为 `typeof` 类型守卫。

以下是一个例子：

```
function padLeft(padding: number | string, input: string): string {
    if (typeof padding === "number") {
        // 在这个 if 分支中，padding 的类型被收窄为 number
        return new Array(padding + 1).join(" ") + input;
    }
    // 在这个 else 分支中，padding 的类型被收窄为 string
    return padding + input;
}
```

在上面的例子中，我们使用 `typeof padding === "number"` 来检查 `padding` 是否为数字。在 `if` 分支中，TypeScript知道 `padding` 的类型只能是 `number`，所以我们可以安全地对它进行数字操作。在 `else` 分支中，TypeScript知道 `padding` 的类型只能是 `string`，所以我们可以安全地对它进行字符串操作。

需要注意的是，`typeof` 类型守卫可以用于 `string`，`number`，`boolean`，`symbol`，`bigint`，`function`，和 `undefined` 类型。对于其他类型，例如自定义的类，`typeof` 将返回 `"object"`，并且不能用来收窄这些类型。

真值收窄

真值收窄是 TypeScript 中的一种类型收窄方法，通过与布尔类型进行比较（通常在 `if` 或 `while` 语句中），来推断出更具体的类型。

在 JavaScript（以及 TypeScript）中，某些值在布尔环境中被认为是“假”，如：`0`，`NaN`，`null`，`undefined`，空字符串 `''`。这些值被称为“falsy”值。其他所有的值被认为是“真”，这些值被称为“truthy”。

在进行真值收窄时，TypeScript 会根据这些规则判断在某个特定点上，变量是否可能为 `null` 或 `undefined`，从而更精确地推断出它的类型。例如：

```
function myFunction(x: string | undefined) {
  if (x) {
    // 在这里，TypeScript 知道 x 一定是 string
    console.log(x.toUpperCase());
  } else {
    // 在这里，TypeScript 知道 x 可能是 undefined
    console.log("No string given");
  }
}
```

在这个例子中，当我们进入 `if` 语句的代码块时，TypeScript 已经通过真实性收窄排除了 `x` 是 `undefined` 的可能性。因此，它知道 `x` 一定是 `string` 类型，可以安全地调用 `toUpperCase()` 方法。

同样地，当我们进入 `else` 语句的代码块时，TypeScript 知道 `x` 只能是 `undefined`，因为所有的 `string` 值都被前面的 `if` 语句处理了。

相等性收窄

相等性收窄（Equality Narrowing）是指使用某些形式的相等性比较（例如 `===`、`!==`、`==` 或 `!=`）来改变变量或参数的类型。根据等号的两边，TypeScript 可以推断出变量或参数在相等性检查之后的类型。

值得注意的是，`==` 和 `!=` 也可以用于相等性收窄，但是由于 JavaScript 在使用这两个运算符时会进行隐式类型转换，可能会导致一些不易察觉的错误。因此，通常建议在 TypeScript 和 JavaScript 中使用严格的相等性运算符 `===` 和 `!==`。

以下是一个使用相等性收窄的示例：

```
```

```ts
type Shape = Circle | Square;

interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    // 在这个代码块中，TypeScript 知道 shape 是 Circle 类型
    return Math.PI * shape.radius ** 2;
  } else {
    // 在这个代码块中，TypeScript 知道 shape 是 Square 类型
    return shape.sideLength ** 2;
  }
}
```

在上述代码中，我们首先定义了一个 `Shape` 类型，它是 `Circle` 和 `Square` 类型的联合。然后，我们定义了一个 `getArea` 函数，它接收一个 `Shape` 类型的参数。

在 `getArea` 函数中，我们使用相等性检查 `shape.kind === "circle"` 来判断 `shape` 的具体类型。如果 `shape.kind` 等于 `"circle"`，那么在该 `if` 代码块中，TypeScript 就会知道 `shape` 的类型必须是 `Circle`；否则，在 `else` 代码块中，TypeScript 就会知道 `shape` 的类型必须是 `Square`。

因此，在每个代码块中，我们可以安全地访问和使用 `shape` 的属性（例如 `shape.radius` 或 `shape.sideLength`），因为 TypeScript 已经知道 `shape` 的具体类型。

in 运算符收窄

in 运算符收窄 是 TypeScript 中的另一种类型收窄方法。当你使用 `in` 运算符检查一个属性是否存在于一个对象中时，TypeScript 可以根据检查结果收窄变量的类型。

下面是一个示例：

```ts
function example(x: { a: number } | { b: string }) {
  if ("a" in x) {
    // 在这个代码块中，TypeScript 知道 x 必须有一个 a 属性
    console.log(x.a); // ok
  } else {
    // 在这个代码块中，TypeScript 知道 x 必须有一个 b 属性
    console.log(x.b); // ok
  }
}
```

在上述代码中，`if` 语句中的 `in` 运算符检查 `"a" in x`，使 TypeScript 能够推断出在该 `if` 代码块中 `x` 的类型必须是 `{ a: number }`。相应地，它也可以推断出在 `else` 代码块中，`x` 必须是 `{ b: string }` 类型（因为所有可能有 `a` 属性的类型都已在 `if` 语句中被处理）。因此，在每个代码块中，我们可以安全地访问特定于那种类型的属性，如 `x.a` 或 `x.b`。

instanceof收窄

`instanceof` 在 JavaScript 中是一个常见的运算符，它用于测试构造函数的 prototype 属性是否出现在某个实例对象的原型链上。在 TypeScript 中，你也可以使用 `instanceof` 运算符进行类型收窄。这主要用于自定义类或内置类（例如 `Date`，`RegExp`，`Array` 等）。例如：

```
class MyError extends Error {
  constructor(message?: string) {
    super(message);
    this.name = "MyError";
  }
}

function handleError(e: Error) {
  if (e instanceof MyError) {
    // 在这个代码块中，TypeScript 知道 e 是 MyError 类型
    console.log(e.name);
  } else {
    // 在这个代码块中，TypeScript 知道 e 是一个 Error 类型，
    // 但不能确定是不是 MyError 类型
    console.log(e.message);
  }
}
```

在上述代码中，我们首先定义了一个自定义错误类 `MyError`，它继承自内置的 `Error` 类。然后，我们定义了一个 `handleError` 函数，它接收一个 `Error` 类型的参数。

在 `handleError` 函数中，我们使用 `instanceof` 运算符来判断 `e` 的具体类型。如果 `e` 是 `MyError` 的实例，那么在该 `if` 代码块中，TypeScript 就会知道 `e` 的类型必须是 `MyError`；否则，在 `else` 代码块中，TypeScript 只能确定 `e` 是 `Error` 类型，但不能确定它是否是 `MyError` 类型。

因此，在每个代码块中，我们可以安全地访问和使用 `e` 的属性（例如 `e.name` 或 `e.message`），因为 TypeScript 已经知道 `e` 的具体类型。

赋值收窄

赋值收窄（Assignment Narrowing）是 TypeScript 中一种常见的类型收窄方式。当我们将一个新的值赋给一个变量时，TypeScript 将会使用这个新的值来更新该变量的类型。

让我们看下面这个例子：

```
let x = Math.random() < 0.5 ? 10 : "hello world!";
//  这里，x 的类型是 number | string
```

```
x = 1;
// 现在，TypeScript 知道 x 的类型是 number

console.log(x);
// 在这里，x 的类型是 number

x = "goodbye!";
// 现在，TypeScript 知道 x 的类型是 string

console.log(x);
// 在这里，x 的类型是 string
```

在这个例子中，我们首先创建了一个名为 `x` 的变量，它的类型是 `number | string`，这是由三元操作符决定的（`Math.random() < 0.5 ? 10 : "hello world!"`）。

然后，我们将一个 `number` 类型的值（`1`）赋值给 `x`。此时，TypeScript 更新了 `x` 的类型，现在 `x` 的类型是 `number`。因此，在 `console.log(x)` 的地方，我们知道 `x` 的类型是 `number`。

再然后，我们将一个 `string` 类型的值（`"goodbye!"`）赋值给 `x`。同样，TypeScript 更新了 `x` 的类型，现在 `x` 的类型是 `string`。因此，在下一个 `console.log(x)` 的地方，我们知道 `x` 的类型是 `string`。

这就是赋值收窄的过程：当我们给变量赋新的值时，TypeScript 会使用新值的类型来更新变量的类型。

类型谓词

类型谓词在 TypeScript 中是一个特殊的返回值类型，它可以在运行时检查一个对象是否符合某个特定的类型。类型谓词通常形如 `arg is T`，其中 `arg` 是函数参数的名称，`T` 是一个类型。如果函数返回 `true`，那么 TypeScript 就会将 `arg` 识别为类型 `T`。

这是一个使用类型谓词的示例：

```
interface Fish {
  swim: () => void;
}

interface Bird {
  fly: () => void;
}

function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

let pet: Fish | Bird;

// 使用类型谓词
if (isFish(pet)) {
```

```
    pet.swim();
  } else {
    pet.fly();
  }
```

在这个例子中，`isFish` 函数的返回类型是 `pet is Fish`，这是一个类型谓词。当 `isFish(pet)` 返回 `true` 时，TypeScript 会知道 `pet` 的类型应该是 `Fish`，这样就能安全地访问 `swim` 方法。相反，如果 `isFish(pet)` 返回 `false`，那么 TypeScript 就知道 `pet` 的类型应该是 `Bird`，这样就能安全地访问 `fly` 方法。

类型谓词对于编写类型安全的代码是非常有用的，特别是当你需要区分类型联合中的类型时。

断言函数

断言函数允许开发者定义一个返回值为 `asserts condition` 的函数，其中 `condition` 是一个类型谓词。这种断言函数可以在运行时验证某个条件是否为真，如果不为真，就抛出一个错误。

这种断言函数可以帮助 TypeScript 在编译时期更好地理解代码的控制流，并能够避免某些潜在的运行时错误。

以下是一个断言函数的例子：

```
function assertIsArray(arg: any): asserts arg is Array<any> {
  if (!Array.isArray(arg)) {
    throw new Error('Not an array!');
  }
}

let value: any = getSomeValue(); // getSomeValue 是一个返回 any 类型值的函数
assertIsArray(value); // 如果 value 不是数组，这里会抛出错误
value.push(1); // 这里 TypeScript 知道 value 一定是数组，所以这样的调用是安全的
```

在这个例子中，`assertIsArray` 函数就是一个断言函数。它的返回类型是 `asserts arg is Array<any>`，表示如果 `arg` 不是数组，就抛出一个错误。在调用 `assertIsArray(value)` 后，TypeScript 可以推断出 `value` 一定是数组（因为如果不是，程序就会在此处抛出错误并停止执行），所以后面的 `value.push(1)` 是安全的。

这种断言函数是一种非常强大的工具，它们可以帮助你编写更安全、更健壮的代码。

可辨识联合

可辨识联合，也被称为标签联合或代数数据类型，是 TypeScript 中的一个强大特性，它能让我们将一种特定类型的变量收窄到更具体的类型。

可辨识联合主要由三个部分组成：

1. **可辨识的属性**：一个类型守卫的共享属性，通常是一个字面量类型的属性，这个属性就是"标签"。
2. **类型别名**：这个类型别名包含了那些具有可辨识特性的类型。

3. **类型守卫**：这个类型守卫根据可辨识的特性，让 TypeScript 收窄可能的类型。

这里是一个可辨识联合的示例：

```
// 这是我们的可辨识特性
type ShapeType = "circle" | "square";

// 这是我们的具有可辨识特性的类型
interface Circle {
  type: ShapeType;
  radius: number;
}

interface Square {
  type: ShapeType;
  sideLength: number;
}

// 这是我们的类型别名
type Shape = Circle | Square;

// 这是我们的类型守卫
function getArea(shape: Shape) {
  switch (shape.type) {
    case "circle":
      // 在这个分支中，shape 被收窄为 Circle
      return Math.PI * shape.radius ** 2;
    case "square":
      // 在这个分支中，shape 被收窄为 Square
      return shape.sideLength ** 2;
  }
}
```

在这个示例中，`type` 属性是可辨识属性，`Circle` 和 `Square` 是具有可辨识属性的类型，`Shape` 是类型别名，`getArea` 函数中的 `switch` 语句是类型守卫。

穷尽性检查

穷尽性检查是 TypeScript 中一个非常有用的特性，它允许你检查是否已经覆盖了所有可能的情况，特别适用于辨识联合类型。

在处理辨识联合类型的 switch 语句中，我们可能希望确保所有可能的情况都被处理。如果忘记了处理某种情况，我们希望 TypeScript 能发出警告。这就是穷尽性检查的作用。

以下是一个简单的例子：

```
type Shape =
  | { kind: 'circle', radius: number }
```

```
  | { kind: 'square', sideLength: number };

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2;
    case 'square':
      return shape.sideLength ** 2;
    default:
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

在这个例子中，`Shape` 是一个 discriminated unions 类型，它可以是一个 `{ kind: 'circle', radius: number }` 或者 `{ kind: 'square', sideLength: number }`。在 `getArea` 函数中，我们用一个 switch 语句处理所有可能的 `Shape`。如果有一个新的 `Shape` 类型被添加，但是我们忘记在 `getArea` 函数中处理，那么 TypeScript 就会在 `_exhaustiveCheck` 处给出一个编译错误，因为新的 `Shape` 类型不能赋值给 `never` 类型。

这种用法称为穷尽性检查，它帮助我们保证处理了所有可能的情况，从而避免在运行时出现错误。

3. 函数

在 TypeScript 中，函数是非常重要的构建模块，用于封装一段可以重复使用的代码逻辑。和 JavaScript 一样，TypeScript 的函数可以是命名函数或匿名函数，可以是函数声明或函数表达式，还可以是箭头函数。此外，TypeScript 还为函数提供了一些额外的特性，以提高开发的效率和代码的可读性。

Function

在 TypeScript 中，`Function` 是所有函数的父类型，它是一个非常广泛的类型。如果一个变量被声明为 `Function` 类型，那么我们可以对这个变量进行任何函数操作，而 TypeScript 编译器都不会产生错误。

例如：

```
let fn: Function;

// These are all valid
fn = function () { return 10; };
fn = function (name: string) { return name; };
fn = () => 'Hello World';
```

在上述代码中，虽然 `fn` 的类型为 `Function`，但我们可以赋给它任何函数，无论这个函数的形式如何，都不会导致 TypeScript 报错。

然而，尽管 `Function` 类型在某些情况下可能会有用，但在可能的情况下，你应该尽量避免使用它，因为它并没有提供任何有关函数参数或返回值的信息。这意味着如果你尝试调用这个函数，你将不得不假设它的参数类型和返回值类型。这种假设可能会导致运行时错误，因为 TypeScript 编译器无法帮助你检查这些错误。

更好的做法是使用具体的函数类型，比如 `(arg1: number, arg2: number) => number`，这样 TypeScript 就可以提供更强大的类型检查和错误提示。

函数类型表达式

在 TypeScript 中，函数类型表达式（Function Type Expressions）是一种用来定义函数的类型方式。这个类型包括参数的类型和函数返回的类型。以下是一个简单的例子：

```
type GreetFunction = (a: string) => void;
```

在这个例子中，`GreetFunction` 的类型是一个函数，它接受一个字符串参数，没有返回值（返回 `void`）。你可以使用这个类型来给其他函数或变量提供类型标注，如：

```
let greet: GreetFunction = function(name) {
    console.log(`Hello, ${name}`);
};
```

使用函数类型表达式有助于我们创建可重用的函数类型，并提供了更清晰的函数签名。它们常常和接口（Interfaces）或类型别名（Type Aliases）一起使用，以提供更强大的类型定义能力。

调用签名

在 TypeScript 中，调用签名（Call Signatures）用于描述函数或方法的类型。它指定了函数的参数类型和返回值类型。这对于在对象类型、接口、类型别名等地方定义函数类型非常有用。以下是一个调用签名的例子：

```
type DescriberFunction = {
  (input: string): string
};
```

在这个例子中，`DescriberFunction` 类型的对象是一个接受一个字符串参数并返回字符串的函数。这样的对象可以像下面这样定义：

```
let myFunction: DescriberFunction = function(s) {
  return `Hello, ${s}`;
};
```

注意，调用签名的参数名是可选的，它们并不是实际函数中的参数名，只是为了提供类型信息和提高可读性。因此，上面的调用签名可以简化为：

```
type DescriberFunction = {
  (a: string): string
};
```

然而，提供参数名可以使得调用签名更容易理解，特别是对于接受多个参数的函数。

构造签名

构造签名是 TypeScript 中一种特殊的调用签名，用于描述一个函数或对象的构造函数签名。这种签名允许我们描述一个函数或对象在使用 new 运算符进行构造时需要的参数类型以及它所返回的类型。

这在描述像 JavaScript 中的构造函数或者类这样的可构造的对象时非常有用。

以下是一个使用构造签名的示例：

```
interface StringConstructor {
  new (value?: any): String;
  (value?: any): string;
  readonly prototype: String;
}

declare var String: StringConstructor;
```

在这个例子中，`StringConstructor` 接口有一个构造签名，表示当我们使用 new 关键字和这个接口的类型（在这里是 `String`）进行构造时，我们可以传入一个任意类型的参数（`value?: any`），并且它将返回一个 `String` 类型的对象。

这是一种强大的类型工具，允许我们在 TypeScript 中精确地描述 JavaScript 中的构造函数和类的行为。

可选参数

在 TypeScript 中，函数的参数可以是可选的。这意味着您可以选择是否提供该参数。在函数声明中，可以通过在参数名后面添加一个问号（`?`）来将参数标记为可选。例如：

```
function greet(name?: string) {
    if (name === undefined) {
        return 'Hello!';
    } else {
        return `Hello, ${name}!`;
    }
}
```

在这个例子中，`name` 参数是可选的。这意味着你可以不传入 `name` 参数来调用 `greet` 函数，就像这样：

```
console.log(greet()); // prints: Hello!
```

你当然也可以提供 `name` 参数：

```
console.log(greet('Alice')); // prints: Hello, Alice!
```

需要注意的是，可选参数必须位于所有必需参数之后。换句话说，你不能在一个必需参数之后放一个可选参数，然后再放一个必需参数。这将会导致一个语法错误。例如，以下代码就是错误的：

```
// 错误：必需参数不能位于可选参数后。
function greet(name?: string, age: number) {
    // ...
}
```

rest 参数

在 TypeScript 中，如果你想要一个函数接受任意数量的参数，你可以使用 rest 参数。Rest 参数是一种将任意数量的参数作为数组收集起来的方法。你可以通过在参数名前添加三个点 (`...`) 来声明一个 rest 参数。

例如，下面的 `sum` 函数接受任意数量的数字参数，并返回它们的总和：

```
function sum(...nums: number[]): number {
  return nums.reduce((a, b) => a + b, 0);
}

console.log(sum(1, 2, 3, 4)); // Outputs: 10
```

在这个例子中，`...nums` 是一个 rest 参数，它表示 "任意数量的 `nums` 参数"。这些参数将作为一个数组收集起来，所以在函数体中，我们可以使用数组的 `reduce` 方法来计算所有数字的总和。

注意，每个函数最多只能有一个 rest 参数，而且必须是参数列表中的最后一个参数。

泛型函数

泛型函数是一种特殊类型的函数，它可以适应多种数据类型。

在编程中，有时我们可能希望函数接受多种类型的参数，但又希望保留参数类型之间的某种关系，这时我们可以使用泛型。

以下是一个泛型函数的例子：

```
function identity<T>(arg: T): T {
  return arg;
}

let output = identity<string>("myString");
```

在这个例子中，我们定义了一个名为 `identity` 的泛型函数，该函数接受一个参数 `arg`，这个参数的类型是可变的，可以是 `string`，`number`，`Array` 等任何类型，由调用函数时决定。同时，这个函数返回的类型与输入的参数类型相同。

推断

对于函数和泛型，类型推断也起到重要作用。例如，对于一个泛型函数，如果没有明确指定类型参数，TypeScript 会根据提供的参数来推断类型。

例如：

```
function identity<T>(arg: T): T {
    return arg;
}

let output = identity("myString");  // TypeScript 会推断出 T 是字符串类型 (string)
```

在这个例子中，我们没有明确指定类型参数 `T`，但是因为我们传入了一个字符串 `"myString"` 作为参数，所以 TypeScript 能够推断出 `T` 是 `string` 类型。

约束

在 TypeScript 中，约束是指你可以在泛型函数中对类型参数设置某些限制。

例如，假设你想创建一个可以处理所有种类的数组的函数，但你想确保数组中的元素都有 `length` 属性。在这种情况下，你可以在类型参数上添加一个约束，表示该类型参数必须包含 `length` 属性。

下面是如何实现这个需求的例子：

```
function getLength<T extends { length: number }>(arg: T): number {
    return arg.length;
}

let arr = [1, 2, 3];
console.log(getLength(arr)); // 输出: 3

let str = "hello";
console.log(getLength(str)); // 输出: 5
```

在这个例子中，类型参数 `T` 是有约束的：`T` 必须是包含 `length` 属性的类型。所以，该函数可以接受任何具有 `length` 属性的值，例如数组或字符串。

注意，如果你试图传入一个没有 `length` 属性的值，比如数字或者布尔值，TypeScript 就会报错。

```
getLength(123); // Error: Argument of type '123' is not assignable to parameter of type '{ length: number; }'.
```

这就是在 TypeScript 中对类型参数设置约束的方法。

小问题

观察下边的代码看看有什么问题：

```
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };
  }
}
```

这段代码试图定义一个名为 `minimumLength` 的函数，该函数接受一个具有 `length` 属性的对象 `obj` 和一个数字 `minimum` 作为参数，如果 `obj` 的 `length` 属性大于等于 `minimum`，则返回 `obj`，否则返回一个新对象，其 `length` 属性等于 `minimum`。

但是这段代码存在一个问题。在 TypeScript 中，`Type` 是一个泛型，可以是任何类型，只要它有一个 `number` 类型的 `length` 属性。这意味着 `Type` 可能是一个数组，一个字符串，或者任何自定义的具有 `length` 属性的对象。在函数的 `else` 分支中，尝试返回一个只有 `length` 属性的对象，这并不能保证与输入的 `Type` 类型相同，因此在类型检查上会产生错误。

以下是修改后的代码，使其能够通过类型检查：

```
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type | { length: number } {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };
  }
}
```

在这个版本中，函数的返回类型被修改为 `Type | { length: number }`，表明函数可能返回与输入的 `Type` 类型相同的对象，也可能返回一个只有 `length` 属性的对象。这样可以确保类型检查的正确性，但是使用这个函数的时候可能需要处理返回值可能不同的问题。

指定类型参数

指定类型参数是在使用泛型函数或泛型类时，手动指定泛型类型的过程。这是 TypeScript 中处理泛型时的一个重要概念。

大多数情况下，当你使用泛型函数或泛型类时，TypeScript 编译器可以通过类型推断来自动确定泛型的类型。例如：

```
function identity<Type>(arg: Type): Type {
  return arg;
}

let output = identity("myString");  // type of output will be 'string'
```

在上述示例中，TypeScript 能够通过你传递给 `identity` 函数的参数 `"myString"` 推断出 `Type` 应为 `string` 类型。因此，变量 `output` 的类型也被推断为 `string`。

然而，有时你可能需要手动指定泛型的类型。例如，你的函数可能需要接收多个参数，但只有其中一部分参数用于确定泛型的类型。在这种情况下，你可以使用尖括号（`<>`）来手动指定类型参数，如下例所示：

```
let output = identity<string>("myString");
```

在这个例子中，我们明确地告诉 TypeScript 我们想要 `Type` 类型为 `string`，而不依赖于类型推断。

值得注意的是，虽然手动指定类型参数在某些情况下可能很有用，但大多数情况下你可以（也应该）依赖 TypeScript 的类型推断，它可以帮助你写出更清晰、更简洁的代码。

编写良好的泛型函数指南：

编写泛型函数是有趣的，但使用过多的类型参数或在不需要的地方使用约束可能会使推断不那么成功，这会让你的函数的调用者感到沮丧。

1. **向下推送类型参数**
   这里有两种编写函数的方式，它们看起来相似：

```
function firstElement1<Type>(arr: Type[]) {
  return arr[0];
}

function firstElement2<Type extends any[]>(arr: Type) {
  return arr[0];
}

// a: number (good)
const a = firstElement1([1, 2, 3]);
// b: any (bad)
const b = firstElement2([1, 2, 3]);
```

   这两者可能一开始看起来是相同的，但是 firstElement1 是编写此函数的更好的方式。它的推断返回类型是 Type，但是 firstElement2 的推断返回类型是 any，因为 TypeScript 必须使用约束类型来解析 arr[0] 表达式，而不是在调用时“等待”解析元素。

   > 规则：尽可能使用类型参数本身，而不是约束它

2. **使用更少的类型参数**
   下面是另一对类似的函数：

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}
```

   我们创建了一个类型参数 Func，它并没有将两个值关联起来。这总是一个警告信号，因为它意味着想要指定类型参数的调用者必须手动指定一个额外的类型参数，而这没有任何理由。Func 并没有做任何事情，只是让函数更难读和理解！

   > 规则：总是尽可能使用更少的类型参数

3. **类型参数应出现两次**
   有时我们忘记了一个函数可能不需要是泛型：

```
function greet<Str extends string>(s: Str) {
  console.log("Hello, " + s);
}

greet("world");
```

我们同样可以编写一个更简单的版本：

```
function greet(s: string) {
  console.log("Hello, " + s);
}
```

请记住，类型参数是用于关联多个值的类型。如果一个类型参数在函数签名中只使用一次，那么它并不关联任何东西。这包括推断的返回类型；例如，如果 Str 是 greet 的推断返回类型的一部分，那么它将关联参数类型和返回类型，因此即使在编写的代码中只出现一次，也会被使用两次。

> 规则：如果一个类型参数只出现在一个位置，强烈建议你重新考虑是否真的需要它

函数重载

函数重载（Function Overloads）是 TypeScript 提供的一种特性，允许我们为同一个函数提供多个函数类型定义。可以理解为，同一个函数，接收不同数量或类型的参数时，会执行不同的逻辑。这样做的好处是让 TypeScript 能更准确地推断出函数的返回值类型。

这是一些例子：

```
// 函数重载声明
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
// 函数实现
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
    if (d !== undefined && y !== undefined) {
        return new Date(y, mOrTimestamp, d);
    } else {
        return new Date(mOrTimestamp);
    }
}
const d1 = makeDate(12345678); // d1 类型为 Date
const d2 = makeDate(5, 5, 2023); // d2 类型为 Date
```

在上述代码中，`makeDate` 函数定义了两种调用方式，一种接收一个 number 类型参数，一种接收三个 number 类型参数。在这两种情况下，函数的实现逻辑是不同的。因此，我们通过函数重载为每种情况都定义了一个函数类型。这样 TypeScript 就能根据传入的参数类型和数量，推断出对应的函数类型，以及相应的返回值类型。

函数重载的声明和实现是分开的。最后一项（实现签名）不能有类型注解，因为类型应由重载的签名列表决定。同时，实现签名也不会出现在重载列表中。

请注意，虽然 TypeScript 会根据重载的定义去检查你的函数调用，但实际上当运行时 JavaScript 并不会做这样的检查。这就意味着你需要在函数体内自行检查参数，并进行相应的处理。

this

在 JavaScript 中，`this` 关键字在函数内部的值会根据函数的调用方式而变化。然而，在 TypeScript 中，我们可以显式指定 `this` 的类型，从而在函数体内部获得更强的类型检查。

你可以在函数参数列表的开头位置声明 `this` 的类型。这样的声明不会影响函数的运行时行为，只是为 TypeScript 提供类型信息。

以下是一个示例：

```
interface Deck {
  suits: string[];
  cards: number[];
  createCardPicker(this: Deck): () => Card;
}

interface Card {
  suit: string;
  card: number;
}

let deck: Deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  // NOTE: The function now explicitly specifies that its callee must be of type
Deck
  createCardPicker: function(this: Deck) {
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();
```

```
alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

在这个例子中，`createCardPicker` 方法在其参数列表中声明了 `this` 应为 `Deck` 类型。这样，当你在 `createCardPicker` 方法内部使用 `this` 时，TypeScript 会知道 `this.suits` 和 `this.cards` 是存在的。

需要注意的是，`this` 的类型只能在非箭头函数中指定。箭头函数使用包含它们的函数或模块的 `this` 值，所以不能在箭头函数的参数列表中指定 `this` 的类型。

4.对象类型

可选属性

在TypeScript中，接口或类型的属性可以被标记为可选。这意味着，这个属性可以存在，也可以不存在。我们使用 `?` 符号来表示一个属性是可选的。例如：

```
interface User {
  id: number;
  name: string;
  age?: number;  // age 是可选属性
}
```

在这个 `User` 接口中，`age` 是一个可选属性。这意味着，我们可以创建一个符合 `User` 接口的对象，而不必提供 `age` 属性：

```
const user: User = {
  id: 1,
  name: "Alice",
  // age 属性被忽略了，但是这没有问题，因为 age 是可选的
};
```

当然，如果你想提供 `age` 属性，也是可以的：

```
const user: User = {
  id: 2,
  name: "Bob",
  age: 30,  // 这是可以的，因为 age 是可选的
};
```

如果你试图访问一个对象的可选属性，而该属性并不存在，那么你将得到 `undefined`。这是因为在JavaScript中，访问一个不存在的属性总是会返回 `undefined`。例如：

```
console.log(user.age);  // 输出: undefined
```

这就是TypeScript中的可选属性。它们允许你在对象中选择性地包含某些属性，同时保持类型安全。

只读属性

在 TypeScript 中，`readonly` 是一个重要的修饰符，可以应用于类或接口的属性，来保证它们不会被重新赋值。如果你试图更改一个被标记为 `readonly` 的属性，TypeScript 将在编译时给出错误提示。注意，`readonly` 只在编译时起作用，在运行时不会对代码行为产生影响。

1. 基本使用：你可以使用 `readonly` 修饰符来创建只读属性。只读属性必须在它们被声明时或在构造函数中初始化。

```
interface SomeType {
  readonly prop: string;
}
```

2. 对于对象类型的 `readonly` 属性：如果 `readonly` 修饰符应用于对象类型的属性，那么你仍然可以更改该对象的内部状态，但不能重新赋值整个对象。

```
interface Home {
  readonly resident: { name: string; age: number };
}
```

在上述代码中，我们不能改变 `home.resident` 的引用，但我们可以更改 `resident` 对象内的属性值，例如 `home.resident.age++` 是允许的。

3. 移除 `readonly`：虽然 `readonly` 属性不能被重新赋值，但可以通过特定的方式绕过这个限制。例如，使用映射类型可以创建一个新的类型，它与原始类型具有相同的属性，但所有属性都变成了可写的。

需要注意的是，即使一个对象的某个属性被标记为 `readonly`，但如果该对象被赋值给一个没有 `readonly` 限制的相同类型的变量，那么通过这个新变量，我们仍然可以改变那个 `readonly` 属性的值。

此外，`readonly` 修饰符对于标识开发者的意图，以及帮助避免在开发过程中不必要的错误是很有用的。尽管 `readonly` 只在编译时起作用，但通过阻止不必要的更改，它能够帮助我们编写出更加可靠和可维护的代码。

索引签名

在 TypeScript 中，索引签名（Index Signatures）是一种用于描述对象或者类的索引（或键）的类型和值的类型的方式。它们可以让你更灵活地创建对象或类的实例，而不是只能使用预定义的属性。

下面是索引签名的一个简单示例：

```
interface StringDictionary {
  [index: string]: string;
}

let myDict: StringDictionary = {};
myDict["hello"] = "world";
// myDict[42] = "world"; // Error: An index signature parameter type must be
'string' or 'number'.
```

在这个例例中，我们定义了一个接口 `StringDictionary`，它有一个字符串索引签名，表示这个接口的实例对象可以使用任意字符串作为键，但值必须是字符串。

需要注意的是，在 TypeScript 中，索引签名的键只能是 `string` 、 `number` 、 `symbol` 和它们的联合类型。同时，所有明确定义的属性都必须与索引签名返回的类型兼容。例如：

```
interface StringDictionaryWithNumber {
  [index: string]: string | number;
  length: number; // OK, length is a number
  name: string; // OK, name is a string
}
```

在这个例子中，接口 `StringDictionaryWithNumber` 的索引签名表示它可以用任意字符串作为键，值可以是字符串或者数字。同时，它有两个显式定义的属性： `length` 和 `name` ，它们的类型分别是 `number` 和 `string` ，这与索引签名的返回类型兼容，因此这个接口定义是正确的。

额外属性检查

在 TypeScript 中，当对象字面量被赋值给一个变量或作为函数参数时，"额外属性检查"（Excess Property Checks）会起作用。这个特性是为了捕获可能的类型错误，并提供更精确的类型安全性。

这是怎么工作的呢？当你把一个对象字面量赋值给一个具有特定类型的变量时，TypeScript 会检查这个对象字面量是否包含目标类型不包含的属性。如果存在这样的额外属性，TypeScript 会给出一个错误。

例如，假设我们有如下的接口：

```
interface SquareConfig {
  color?: string;
  width?: number;
}
```

然后，我们尝试将包含额外属性 `opacity` 的对象字面量赋值给一个 `SquareConfig` 类型的变量：

```
let mySquare: SquareConfig = { color: "red", width: 100, opacity: 0.5 };
```

在这种情况下，TypeScript 会提示错误，因为 `SquareConfig` 类型不包含 `opacity` 属性。这就是额外属性检查。

需要注意的是，额外属性检查仅在直接将对象字面量赋值给变量或传递给函数时生效。如果你先将对象字面量赋值给一个新变量，然后将这个新变量赋值给另一个具有特定类型的变量，额外属性检查就不会生效。这是因为，在赋值给新变量时，TypeScript 无法预知你接下来要做什么，因此它不会执行额外属性检查。

然而，这并不意味着你可以通过这种方式来“规避”额外属性检查。事实上，额外属性检查的目的是帮助你捕获可能的错误。如果你发现自己在尝试规避额外属性检查，这可能意味着你需要重新审视你的类型声明。

总的来说，额外属性检查是 TypeScript 的一种特性，目的是提供更强的类型安全性。如果你的代码中出现了额外属性检查的错误，这通常意味着你的类型声明可能需要调整。

如果你希望允许对象有额外的属性，你可以在接口中使用索引签名。例如：

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}
```

在这个接口中，`[propName: string]: any;` 是一个索引签名。这意味着 `SquareConfig` 可以有任意数量的额外属性，只要它们的键是字符串。这些额外属性的值的类型都是 `any`，意味着可以是任何类型。

这样，就可以在 `SquareConfig` 类型的对象中添加任意额外的属性，而不会收到 TypeScript 的错误提示。

```
let mySquare: SquareConfig = { color: "red", width: 100, opacity: 0.5 };
```

在这个例子中，`opacity` 是一个额外的属性，但由于 `SquareConfig` 接口中定义了索引签名，TypeScript 不会给出错误。

注意，虽然这样可以让你在对象中添加额外的属性，但这样做可能会损失一些类型安全性。如果可能，最好的做法仍然是定义精确的类型，而不是使用 `any` 类型。

接口继承

在 TypeScript 中，你可以使用 `extends` 关键字创建一个新的类型，这个新类型包含了一个现有类型的所有属性，然后再添加新的属性或修改现有的属性。这通常被称为扩展一个类型。以下是一个简单的例子：

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

interface Employee extends Person {
  salary: number;
}

let employee: Employee = {
  name: "Alice",
  age: 25,
  salary: 50000,
};
```

在以上的代码中，`Employee` 是一个新的类型，它通过添加 `salary` 属性来扩展 `Person`。因此，类型为 `Employee` 的对象必须包含 `name`，`age`，和 `salary` 属性。

当你想创建一个类型的更具体版本，而不必重新定义所有的属性时，扩展类型的过程非常有用。例如，你可能有一个基本的 `Address` 类型，你想将它扩展为一个更具体的 `AddressWithUnit` 类型，该类型还包括 `unit` 属性：

```
interface BasicAddress {
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

在以上代码中，`AddressWithUnit` 通过添加 `unit` 属性来扩展 `BasicAddress`。因此，类型为 `AddressWithUnit` 的对象必须包含 `street`，`city`，`country`，`postalCode`，和 `unit` 属性。这在减少你需要编写的类型声明样板文件，以及在表示多个不同声明可能以某种方式相关时非常有帮助。

接口也可以同时扩展多个类型。例如：

```
interface Shape {
```

```
  area: number;
}

interface Solid {
  volume: number;
}

interface SolidShape extends Shape, Solid { }

let cube: SolidShape = {
  area: 36,
  volume: 27,
};
```

在上述代码中，`SolidShape` 同时扩展了 `Shape` 和 `Solid`，这意味着类型为 `SolidShape` 的对象必须包含 `area` 和 `volume` 属性。

交叉类型

"交叉类型"是 TypeScript 中的一个重要概念，主要用于组合现有的对象类型。交叉类型是使用 `&` 运算符定义的。

以下是一个简单的例子：

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

type ColorfulCircle = Colorful & Circle;
```

在这里，我们将 `Colorful` 和 `Circle` 进行了交叉，生成了一个新的类型，这个新类型具有 `Colorful` 和 `Circle` 的所有成员。

接下来，我们可以定义一个函数 `draw`，该函数接受一个参数，其类型为 `Colorful & Circle`：

```
function draw(circle: Colorful & Circle) {
  console.log(`Color was ${circle.color}`);
  console.log(`Radius was ${circle.radius}`);
}

// 正确
draw({ color: "blue", radius: 42 });

// 错误
draw({ color: "red", raidus: 42 });
```

在以上的代码中，`draw` 函数接受一个参数，这个参数的类型必须同时满足 `Colorful` 和 `Circle` 的类型，即这个参数必须有 `color` 和 `radius` 这两个属性。如果我们尝试将一个不符合这个类型的对象传递给 `draw` 函数（如第二个 `draw` 函数调用），TypeScript 将给出一个类型错误。

继承和交叉类型

"接口继承"和"交叉类型"是两种相似但实际上有微妙区别的组合类型的方式。在接口中，我们可以使用 `extends` 关键字从其他类型扩展，而在交叉类型中，我们可以通过类型别名来命名结果。这两者之间的主要区别在于如何处理冲突，这通常是你在选择使用接口还是交叉类型的类型别名时的主要考虑因素。

当你有两个类型，并且它们在同名属性上有冲突时，如果你使用接口去扩展它们，TypeScript 会将它们视为一个错误。例如：

```
interface X {
  c: string;
}
interface Y {
  c: number;
}

interface Z extends X, Y {} // 错误，X 和 Y 不兼容
```

在上面的代码中，`X` 和 `Y` 都有一个名为 `c` 的属性，但它们的类型不同，所以在尝试扩展它们以创建 `Z` 时，TypeScript 会报错。

然而，如果你尝试使用交叉类型，TypeScript 会允许它：

```ts
interface X {
  c: string;
}
interface Y {
  c: number;
}

type Z = X & Y; // 允许，但是 c 的类型为 `never`
```

在这种情况下，`c` 的类型将为 `never`，这是因为 `string` 和 `number` 的交集是 `never`。这是 TypeScript 的一种方式，表示某种类型在运行时永远不可能存在。在这种情况下，它是正确的，因为一个值不能同时是 `string` 和 `number`。

所以，总的来说，接口和交叉类型在处理类型冲突时的行为不同，这通常是你在选择使用哪一个时的决定因素。如果你希望在有冲突时得到错误，你可能会选择使用接口。如果你希望允许冲突并产生新的更严格的类型，你可能会选择使用交叉类型。

泛型对象

让我们设想一种名为 `Box` 的类型，它可以包含任何值——字符串、数字、长颈鹿，无论什么。

```ts
interface Box {
  contents: any;
}
```

现在，属性 `contents` 被指定为 `any` 类型，这是可行的，但可能导致后续的错误。

我们可以选择使用 `unknown`，但那将意味着在我们已经知道 `contents` 类型的情况下，我们需要进行预防性检查，或者使用可能出错的类型断言。

```ts
interface Box {
  contents: unknown;
}

let x: Box = {
  contents: "hello world",
};

// 我们可以检查 'x.contents'
if (typeof x.contents === "string") {
  console.log(x.contents.toLowerCase());
}

// 或者我们可以使用类型断言
console.log((x.contents as string).toLowerCase());
```

一种类型安全的方法是为每一种 `contents` 类型建立不同的 `Box` 类型。

```ts
interface NumberBox {
  contents: number;
}

interface StringBox {
  contents: string;
}

interface BooleanBox {
  contents: boolean;
}
```

但这意味着我们需要为这些类型创建不同的函数，或者为函数重载。

```ts
function setContents(box: StringBox, newContents: string): void;
function setContents(box: NumberBox, newContents: number): void;
function setContents(box: BooleanBox, newContents: boolean): void;
function setContents(box: { contents: any }, newContents: any) {
  box.contents = newContents;
}
```

这将产生大量的样板代码。此外，我们可能还需要引入新的类型和重载。这让人感到挫败，因为我们的 `Box` 类型和重载实际上是完全相同的。

相反，我们可以创建一个声明类型参数的泛型 `Box` 类型。

```ts
interface Box<Type> {
  contents: Type;
}
```

你可能将其解读为 "类型为 `Type` 的 `Box` 是一个 `contents` 的类型为 `Type` 的东西"。稍后，当我们引用 `Box` 时，我们需要在 `Type` 的位置提供一个类型参数。

```ts
let box: Box<string>;
```

把 `Box` 看作一个真实类型的模板，其中 `Type` 是一个将被替换为其他类型的占位符。当 TypeScript 看到 `Box<string>` 时，它将把 `Box<Type>` 中的每一个 `Type` 实例替换为 `string`，并最终处理类似 `{ contents: string }` 的东西。换言之，`Box<string>` 和我们之前的 `StringBox` 的工作方式完全一样。

```ts
interface Box<Type> {
  contents: Type;
}
interface StringBox {
  contents: string;
}

let boxA: Box<string> = { contents: "hello" };
boxA.contents;

let boxB: StringBox = { contents: "world" };
boxB.contents;
```

`Box` 是可重用的，因为 `Type` 可以被替换为任何东西。这意味着当我们需要为新类型的 `Box` 时，我们不需要声明一个新的 `Box` 类型（尽管我们肯定可以如果我们想要的话）。

```ts
interface Box<Type> {
  contents: Type;
}

interface Apple {
  // ....
}

// 和 '{ contents: Apple }' 相同。
type AppleBox = Box<Apple>;
```

这也意味着我们可以通过使用泛型函数来完全避免重载。

```ts
function setContents<Type>(box: Box<Type>, newContents: Type) {
  box.contents = newContents;
}
```

值得注意的是，类型别名也可以是泛型的。我们可以使用类型别名来定义新的 `Box<Type>` 接口，就像这样：

```ts
type Box<Type> = {
  contents: Type;
};
```

由于类型别名（与接口不同）可以描述除对象类型之外的更多类型，我们也可以使用它们来编写其他种类的泛型辅助类型。

```
type

 OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];

type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;

type OneOrManyOrNullStrings = OneOrManyOrNull<string>;
```

只读数组

ReadonlyArray 是一个特殊的类型，用于描述不应被更改的数组。

```
function doStuff(values: ReadonlyArray<string>) {
  // 我们可以从 'values' 中读取...
  const copy = values.slice();
  console.log(`第一个值是 ${values[0]}`);

  // ...但我们不能改变 'values'。
  values.push("hello!");
}
```

就像用于属性的 readonly 修饰符一样，它主要是我们可以用于表达意图的工具。当我们看到一个返回 ReadonlyArrays 的函数时，它告诉我们我们不应该改变内容，当我们看到一个消费 ReadonlyArrays 的函数时，它告诉我们我们可以将任何数组传递给该函数，而不用担心它会改变其内容。

不像 Array，我们不能使用 ReadonlyArray 构造函数。

```
new ReadonlyArray("red", "green", "blue");
// 'ReadonlyArray' only refers to a type, but is being used as a value here.
```

相反，我们可以将普通的 Arrays 分配给 ReadonlyArrays。

```
const roArray: ReadonlyArray<string> = ["red", "green", "blue"];
```

就像 TypeScript 为 Array 提供了 Type[] 的简写语法，它也为 ReadonlyArray 提供了 readonly Type[] 的简写语法。

```
function doStuff(values: readonly string[]) {
  // 我们可以从 'values' 中读取...
  const copy = values.slice();
  console.log(`第一个值是 ${values[0]}`);

  // ...但我们不能改变 'values'。
  values.push("hello!");
Property 'push' does not exist on type 'readonly string[]'.
}
```

需要注意的最后一点是，不像 readonly 属性修饰符，普通的 Arrays 和 ReadonlyArrays 之间的可分配性并不是双向的。

```
let x: readonly string[] = [];
let y: string[] = [];

x = y;
y = x;
//The type 'readonly string[]' is 'readonly' and cannot be assigned to the mutable
type 'string[]'.
```

元组

元组类型是另一种数组类型，它知道它包含的元素数量，以及在特定位置包含哪些类型。

```
type StringNumberPair = [string, number];
```

这里，StringNumberPair 是一个包含字符串和数字的元组类型。就像 ReadonlyArray 一样，它在运行时没有表示，但对 TypeScript 来说是重要的。对于类型系统，StringNumberPair 描述了索引 0 包含字符串，索引 1 包含数字的数组。

```
function doSomething(pair: [string, number]) {
  const a = pair[0];
  const b = pair[1];
  // ...
}

doSomething(["hello", 42]);
```

如果我们试图索引超过元素的数量，我们将得到一个错误。

```
function doSomething(pair: [string, number]) {
  // ...
  const c = pair[2];
  // Tuple type '[string, number]' of length '2' has no element at index '2'.
}
```

我们也可以使用 JavaScript 的数组解构来解构元组。

```
function doSomething(stringHash: [string, number]) {
  const [inputString, hash] = stringHash;
  console.log(inputString);
  console.log(hash);
}
```

元组类型在重度约定式的 API 中非常有用，其中每个元素的含义都是“显然的”。这给了我们在解构它们时对变量命名的灵活性。在上面的例子中，我们能够将元素 0 和 1 命名为我们想要的任何名字。

然而，由于不是每个用户都持有同样的观点，所以使用带有描述性属性名称的对象可能更适合你的 API。

除了这些长度检查外，这样的简单元组类型等价于声明了特定索引的属性的数组版本，并用数字文字类型声明长度。

```
interface StringNumberPair {
  // specialized properties
  length: 2;
  0: string;
  1: number;

  // Other 'Array<string | number>' members...
  slice(start?: number, end?: number): Array<string | number>;
}
```

另一个你可能感兴趣的是，元组可以通过在元素类型后面写出一个问号 (?) 来拥有可选属性。可选的元组元素只能出现在末尾，并且也会影响长度的类型。

```
type Either2dOr3d = [number, number, number?];

function setCoordinate(coord: Either2dOr3d) {
  const [x, y, z] = coord;
  console.log(`Provided coordinates had ${coord.length} dimensions`);
}
```

元组也可以有剩余元素，这必须是一个数组/元组类型。

```
type StringNumberBooleans = [string, number, ...boolean[]];
type StringBooleansNumber = [string, ...boolean[], number];
type BooleansStringNumber = [...boolean[], string, number];
```

StringNumberBooleans 描述了一个元组，其前两个元素分别是字符串和数字，但其后可能有任意数量的布尔值。
StringBooleansNumber 描述了一个元组，其第一个元素是字符串，然后是任意数量的布尔值，最后是一个数字。
BooleansStringNumber 描述了一个元组，其开始元素是任意数量的布尔值，最后是一个字符串，然后是一个数字。
一个带有剩余元素的元组没有设定的“长度” - 它只有在不同位置的一组已知元素。

```
const a: StringNumberBooleans = ["hello", 1];
const b: StringNumberBooleans = ["beautiful", 2, true];
const c: StringNumberBooleans = ["world", 3, true, false, true, false, true];
```

为什么可选和剩余元素可能有用呢？好吧，它让 TypeScript 能够将元组与参数列表相对应。元组类型可以用在剩余参数和参数中，这样以下面的：

```
function readButtonInput(...args: [string, number, ...boolean[]]) {
  const [name, version, ...input] = args;
  // ...
}
```

基本上等价于：

```
function readButtonInput(name: string, version: number, ...input: boolean[]) {
  // ...
}
```

当你想要使用剩余参数获取可变数量的参数，并且你需要一定数量的元素，但你不想引入中间变量时，这非常方便。

只读元组

关于元组类型的最后一点说明 - 元组类型有只读的变体，可以通过在它们前面添加只读修饰符来指定 - 就像数组的简写语法一样。

```
function doSomething(pair: readonly [string, number]) {
  // ...
}
```

正如你可能期望的，TypeScript 不允许写入只读元组的任何属性。

```
function doSomething(pair: readonly [string, number]) {
  pair[0] = "hello!";
  // Cannot assign to '0' because it is a read-only property.
}
```

元组在大多数代码中倾向于被创建并保持不变，因此尽可能将类型注解为只读元组是一个好的默认设置。这也很重要，因为具有 const 断言的数组字面量将被推断为只读元组类型。

```
let point = [3, 4] as const;

function distanceFromOrigin([x, y]: [number, number]) {
  return Math.sqrt(x ** 2 + y ** 2);
}

distanceFromOrigin(point);
// Argument of type 'readonly [3, 4]' is not assignable to parameter of type
'[number, number]'.
// The type 'readonly [3, 4]' is 'readonly' and cannot be assigned to the mutable
type '[number, number]'.
```

这里，distanceFromOrigin 从不修改它的元素，但是它期望一个可变的元组。由于 point 的类型被推断为 readonly [3, 4]，它与 [number, number] 不兼容，因为那种类型不能保证 point 的元素不会被改变。

keyof

"keyof" 是 TypeScript 中的一个关键字，被称为类型查询操作符，它可以用于获取某种类型的所有键的集合。

下面是一个简单的例子：

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

type PersonKeys = keyof Person; // "name" | "age" | "location"
```

在上述例子中，我们首先定义了一个 `Person` 接口，然后使用 `keyof` 创建了一个新的类型 `PersonKeys`，它表示 `Person` 接口的所有键的集合。所以 `PersonKeys` 类型等价于 `"name" | "age" | "location"`。

这个特性特别有用，当你需要动态访问对象的属性，或者创建更复杂的映射类型时。例如，如果你想要一个函数，它接受一个对象和对象的键，然后返回该键对应的值，你可以这样做：

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

let person: Person = {name: "Bob", age: 20, location: "USA"};

let personName = getProperty(person, "name"); // personName 的类型为 string
let personAge = getProperty(person, "age"); // personAge 的类型为 number
```

在这个例子中，我们使用 `keyof` 和泛型来创建一个类型安全的函数 `getProperty`，它能根据给定的键返回对象的相应属性。注意，在 `getProperty` 的类型定义中，`K extends keyof T` 确保了你只能传递对象的实际属性名作为键。

typeof

在 TypeScript 中，`typeof` 关键字有两个主要的用途：一种是在运行时检查一个变量的类型，这与 JavaScript 的用法一致；另一种是在编译时获取一个变量或表达式的类型，这是 TypeScript 特有的功能。

在 TypeScript 中，`typeof` 可以在类型的上下文中使用，来获取一个变量或表达式的类型。例如：

```
let s = "hello";
let n: typeof s;

n = "world";  // OK
n = 10;  // Error: Type '10' is not assignable to type 'string'
```

在这个例子中，`typeof s` 表示获取变量 `s` 的类型（在这个例子中是 `string`）。然后我们声明了一个新的变量 `n`，它的类型是 `typeof s`，也就是 `string`。这意味着我们只能给 `n` 赋值字符串类型的值。

这个特性在你需要捕获复杂类型，或者想要使用已有变量的类型来声明新变量时，非常有用。例如，你可以用它来捕获和复制一个对象的完整类型：

```
let foo = { a: 1, b: "hello" };

type FooType = typeof foo;  // { a: number, b: string }

let bar: FooType;
bar = { a: 2, b: "world" };  // OK
bar = { a: 2 };  // Error: Property 'b' is missing in type '{ a: number }'
```

这里，`typeof foo` 获取了 `foo` 的完整类型，然后我们用这个类型来声明了一个新的变量 `bar`。这样可以保证 `bar` 必须有和 `foo` 相同的结构。

索引访问类型

在TypeScript中，索引访问类型是一种通过索引类型查询和索引访问操作符来操作类型的方式。这种类型让我们能够以一种安全的方式动态获取对象的属性。

例如，假设我们有一个名为 `Person` 的类型：

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

我们可以通过索引访问类型，动态地访问 `Person` 类型的属性类型：

```
type Name = Person["name"];  // string
type Age = Person["age"];  // number
type Location = Person["location"];  // string
```

在这个例子中，`Person["name"]` 会返回 `name` 属性的类型，也就是 `string`。

索引访问类型非常有用，因为它允许我们根据已经存在的类型来获取新的类型。它也可以和 `keyof` 类型操作符结合使用，创建更加复杂的类型。

例如，我们可以创建一个类型，该类型接受一个对象类型 `T` 和该对象的键 `K`，返回该键对应的属性的类型：

```
type PropertyType<T, K extends keyof T> = T[K];

type PersonName = PropertyType<Person, "name">;  // string
```

在这个例子中，`PropertyType` 类型使用了两个类型参数：`T` 和 `K`。`K extends keyof T` 表示 `K` 必须是 `T` 的键。然后，`T[K]` 返回 `K` 在 `T` 中对应的属性的类型。

条件类型

在TypeScript中，条件类型是一种选择两种可能的类型之间的方式，这取决于给定的条件。

一般的形式如下：

```
T extends U ? X : Y
```

这个类型可以读作：如果 `T` 可以赋值给 `U`，那么该类型是 `X`，否则该类型是 `Y`。

让我们看一个简单的例子：

```
type IsString<T> = T extends string ? true : false;

type A = IsString<number>;  // false
type B = IsString<string>;  // true
```

在上述例子中，`IsString<T>` 是一个条件类型。如果 `T` 是 `string`，则结果类型为 `true`，否则为 `false`。

条件类型经常与 `keyof` 和映射类型等其他TypeScript特性一起使用，以创建复杂的类型逻辑。

例如，以下是一个 `Diff<T, U>` 类型，它只返回不在 `U` 中的 `T` 的键：

```
type Diff<T, U> = T extends U ? never : T;

type T1 = Diff<"a" | "b" | "c", "a" | "e">;  // "b" | "c"
```

在这个例子中，`Diff<T, U>` 遍历 `T` 中的每个类型，并检查它是否可以赋值给 `U`。如果可以，返回 `never` 类型（表示不存在的类型）；否则，返回该类型。因此，`Diff<"a" | "b" | "c", "a" | "e">` 的结果是 `"b" | "c"`。

映射类型

在 TypeScript 中，映射类型（Mapped Types）允许你根据旧的类型生成新的类型。它们在处理旧类型的所有属性时非常有用。例如，你可能想让旧类型的所有属性都变为可选的或只读的。

以下是一个只读的映射类型的示例：

```
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
}
```

在这个示例中，`Readonly<T>` 是一个新的类型，它将类型 `T` 的所有属性都变为了只读的。`P in keyof T` 是类型变量 `P` 遍历 `T` 中所有属性名的语法，`T[P]` 是索引访问类型，用于访问 `T` 的属性类型。

类似地，你也可以创建一个将所有属性变为可选的映射类型：

```
type Partial<T> = {
    [P in keyof T]?: T[P];
}
```

在这个例子中，`Partial<T>` 是一个新的类型，它将类型 `T` 的所有属性都变为了可选的。

在 TypeScript 标准库中，已经内置了 `Readonly`，`Partial` 等常用的映射类型，可以直接使用。

在某些情况下，映射类型可以大大提高代码的灵活性和复用性，减少重复代码的编写。

映射修改器

在TypeScript 4.1及以上版本中，可以使用映射修饰符来更改映射类型中属性的修饰符。目前有两种映射修饰符：`readonly` 和 `?`。

- `readonly`：可以添加或移除属性的 `readonly` 修饰符。
- `?`：可以添加或移除属性的可选性（optional）。

这是一个例子：

```
type MutableRequired<T> = {
  -readonly [P in keyof T]-?: T[P];
};
type ReadonlyPartial<T> = {
  +readonly [P in keyof T]+?: T[P];
};

type Foo = {
  readonly a: number;
  b?: string;
};

type A = MutableRequired<Foo>;  // { a: number; b: string; }
type B = ReadonlyPartial<Foo>;  // { readonly a?: number; readonly b?: number; }
```

在这个例子中：

- `MutableRequired<T>` 是一个映射类型，用于移除 `T` 中所有属性的 `readonly` 修饰符，并使所有属性都成为必需的（移除 `?`）。
- `ReadonlyPartial<T>` 是一个映射类型，用于添加 `T` 中所有属性的 `readonly` 修饰符，并使所有属性都成为可选的（添加 `?`）。

映射修饰符 `-readonly`，`+readonly`，`-?` 和 `+?` 分别用于移除和添加 `readonly` 和可选修饰符。当然，如果属性本来就没有 `readonly` 或可选修饰符，添加的 `-readonly` 或 `-?` 就不会有任何效果；相应地，如果属性本来就是 `readonly` 或可选的，添加的 `+readonly` 或 `+?` 也不会有任何效果。

通过 as 实现键名重新映射

在TypeScript 4.1及更高版本中，您可以使用as子句在映射类型中重新映射键：

```
type MappedTypeWithNewProperties<Type> = {
    [Properties in keyof Type as NewKeyType]: Type[Properties]
}
```

您可以利用像模板字面量类型这样的特性，从先前的属性名称创建新的属性名称：

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

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

type LazyPerson = Getters<Person>;
// 类型LazyPerson为:  { getName: () => string; getAge: () => number; getLocation: ()
=> string; }
```

您可以通过生成never的条件类型来过滤掉键：

```
// 移除 'kind' 属性
type RemoveKindField<Type> = {
    [Property in keyof Type as Exclude<Property, "kind">]: Type[Property]
};

interface Circle {
    kind: "circle";
    radius: number;
}

type KindlessCircle = RemoveKindField<Circle>;
// 类型KindlessCircle为:  { radius: number; }
```

您可以映射任意联合，不仅仅是字符串 | 数字 | 符号的联合，还可以是任何类型的联合：

```ts
type EventConfig<Events extends { kind: string }> = {
    [E in Events as E["kind"]]: (event: E) => void;
}

type SquareEvent = { kind: "square", x: number, y: number };
type CircleEvent = { kind: "circle", radius: number };

type Config = EventConfig<SquareEvent | CircleEvent>
// 类型Config为:  { square: (event: SquareEvent) => void; circle: (event:
CircleEvent) => void; }
```

模板文字类型

模板文字类型（Template Literal Types）是 TypeScript 4.1 中引入的新特性。这个特性允许我们使用模板文字字符串（这是一种在 JavaScript 中已经存在的特性）来定义类型。这提供了一个创建复合类型，或者根据其他类型派生新类型的强大工具。

模板文字类型非常强大，因为它们可以与联合类型和条件类型一起使用，这允许我们创建非常复杂的类型。

基本的使用方法如下：

```ts
type World = "world";

type Greeting = `hello ${World}`;  // "hello world"
```

在上面的例子中，`Greeting` 类型等价于字符串 "hello world"。

如果我们使用联合类型，那么结果就会是一个联合类型：

```ts
type World = "world" | "mars" | "venus";

type Greeting = `hello ${World}`;  // "hello world" | "hello mars" | "hello venus"
```

在这个例子中，`Greeting` 类型等价于 "hello world" | "hello mars" | "hello venus"。

模板文字类型是 TypeScript 类型系统的一个强大工具，它为我们提供了更多的方式来创建和组合类型。

字符串联合类型

在 TypeScript 中，字符串联合类型（String Unions）允许你在类型系统中声明一个变量只能是某几个字符串之一。这在约束函数参数或者对象属性的值非常有用，可以避免拼写错误或者使用了不正确的值。

而模板字面量类型（Template Literal Types）则进一步拓展了字符串联合类型的功能，它允许你在类型中使用字符串模板，来根据其他类型来生成新的类型。

举个例子，你有一个函数 `makeWatchedObject`，这个函数接收一个对象，并向这个对象添加一个 `on` 方法，用于监听对象属性的变化。

这个 `on` 方法接收两个参数：一个事件名（eventName）和一个回调函数（callback）。事件名应该是由传入对象的属性名后面加上 "Changed" 构成的，而回调函数接收一个新的属性值作为参数。

你可以这样定义这个 `on` 方法的类型：

```
type PropEventSource<Type> = {
    on(eventName: `${string & keyof Type}Changed`, callback: (newValue: any) =>
void): void;
};

declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;
```

在这里，`${string & keyof Type}Changed` 是一个模板字面量类型。它表示所有 `Type` 的属性名后面加上 "Changed" 的字符串。这样，`eventName` 就被约束为只能是这些字符串之一。

因此，如果你尝试使用一个不正确的事件名，TypeScript 就会给出错误提示：

```
person.on("firstName", () => {});  // Error: Argument of type '"firstName"' is not
assignable to parameter of type '"firstNameChanged" | "lastNameChanged" |
"ageChanged"'.

person.on("frstNameChanged", () => {});  // Error: Argument of type
'"frstNameChanged"' is not assignable to parameter of type '"firstNameChanged" |
"lastNameChanged" | "ageChanged"'.
```

这就是模板字面量类型和字符串联合类型的一种组合使用方式，可以有效地帮助我们在类型系统中捕捉到错误。

在模板文字类型中进行类型推断

当然可以，以下是整理后的内容：

使用模板文字进行类型推断

在为对象的属性设置监听事件时，模板文字类型可以为我们提供强大的类型推断能力。例如，当属性 `firstName` 发生改变时，我们希望监听到的事件名为 `firstNameChanged`，同时期望事件的回调函数接收一个类型为 `string` 的参数。

下面的代码示例展示了如何利用TypeScript的模板文字类型来实现这一功能：

```
type PropEventSource<Type> = {
    on<Key extends string & keyof Type>
```

```
        (eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void):
void;
};

declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;

const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26
});

person.on("firstNameChanged", newName => {
    console.log(`新的名字是 ${newName}`);
});

person.on("ageChanged", newAge => {
    if (newAge < 0) {
        console.warn("警告! 年龄为负数");
    }
});
```

解释：

1. `on` 方法被定义为一个泛型方法，允许我们从事件名称中推断属性名。
2. 当我们监听 `firstNameChanged` 事件时，TypeScript会推断 `Key` 为 `firstName`，因此回调函数的参数 `newName` 的类型自然为 `string`。
3. 同样地，对于 `ageChanged` 事件，TypeScript推断出 `Key` 为 `age`，使得回调函数的参数 `newAge` 的类型为 `number`。

这种推断方法确保了代码的类型安全性，避免了常见的类型错误。

字符串操作工具

在 TypeScript 4.1 及以后的版本中，引入了几个内置的字符串操作工具类型（Intrinsic String Manipulation Types）。这些类型可以帮助你在类型系统层面操作字符串。以下是一些内置的字符串操作类型：

1. `Uppercase<S>`：将字符串字面量 `S` 中的所有字符转换为大写。
2. `Lowercase<S>`：将字符串字面量 `S` 中的所有字符转换为小写。
3. `Capitalize<S>`：将字符串字面量 `S` 的首字母转换为大写。
4. `Uncapitalize<S>`：将字符串字面量 `S` 的首字母转换为小写。

请注意，这些类型只对字符串字面量有效。如果你使用的是普通的字符串类型（例如：`string`），那么这些操作类型不会产生任何效果。例如，`Uppercase<string>` 仍然是 `string`。

这是一些使用示例：

```
type T1 = Uppercase<'Hello'>;  // 'HELLO'
type T2 = Lowercase<'Hello'>;  // 'hello'
type T3 = Capitalize<'hello'>; // 'Hello'
type T4 = Uncapitalize<'Hello'>; // 'hello'
```

这些内置的字符串操作类型在与模板文字类型一起使用时非常有用，可以帮助你根据现有的类型信息生成新的字符串类型。

5. 类

TypeScript对于在ES2015中引入的`class`关键字提供完整支持。

就像其他的 JavaScript 语言特性一样，TypeScript 添加了类型注解和其他语法来让你能够表达类和其他类型之间的关系。

类成员

这是一个最基本的类，一个空类：

```
class Point {}
```

这个类目前还不是很有用，所以让我们开始添加一些成员。

字段

字段声明在类上创建一个公有的可写属性：

```
class Point {
  x: number;
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

就像其他位置一样，类型注解是可选的，但是如果没有指定，那么将是隐式的 any。

字段也可以有初始化器；这些将在类被实例化时自动运行：

```
class Point {
  x = 0;
  y = 0;
}

const pt = new Point();
// 打印 0, 0
console.log(`${pt.x}, ${pt.y}`);
```

就像 const，let，和 var 一样，类属性的初始化器将被用来推断它的类型：

```
const pt = new Point();
pt.x = "0";
// 错误：类型 'string' 不能赋值给 'number' 类型。
```

strictPropertyInitialization

strictPropertyInitialization 设置控制是否需要在构造函数中初始化类字段。

```
class BadGreeter {
  name: string;
 // 错误：属性 'name' 没有初始化器，也没有在构造函数中明确赋值。
}
```

```
class GoodGreeter {
  name: string;

  constructor() {
    this.name = "hello";
  }
}
```

注意，字段需要在构造函数本身中初始化。TypeScript 不会分析你从构造函数中调用的方法来检测初始化，因为派生类可能会覆盖这些方法并且不去初始化成员。

如果你打算通过构造函数以外的方式去初始化字段（例如，可能有一个外部库正在为你填充类的一部分），你可以使用明确赋值断言运算符，!：

```
class OKGreeter {
  // 没有初始化，但是没有错误
  name!: string;
}
```

readonly

字段可能会被 readonly 修饰符修饰。这将阻止在构造函数之外的地方对字段进行赋值。

```
class Greeter {
  readonly name: string = "world";

  constructor(otherName?: string) {
    if (otherName !== undefined) {
      this.name = otherName;
    }
  }

  err() {
    this.name = "not ok";
    // 错误: 不能给 'name' 赋值，因为它是只读属性。
  }
}
const g = new Greeter();
g.name = "also not ok";
// 错误: 不能给 'name' 赋值，因为它是只读属性。
```

构造函数

类构造器与函数非常相似。你可以添加带有类型注解的参数、默认值和重载：

```
class Point {
  x: number;
  y: number;

  // 带有默认值的普通签名
  constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
  }
}
```

```
class Point {
  // 重载
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // 待定
  }
}
```

类构造函数签名和函数签名之间只有一些微小的差异：

- 构造函数不能有类型参数，这些类型参数应该位于外部类声明上，我们将在后面学习。
- 构造函数不能有返回类型注解， 总是返回类实例类型。

Super调用

正如在JavaScript中一样，如果你有一个基类，你需要在你的构造器体中调用`super();`之后才能使用任何`this.`成员：

```
class Base {
  k = 4;
}

class Derived extends Base {
  constructor() {
    // 在ES5中打印错误的值；在ES6中抛出异常
    console.log(this.k);
    // 'super' must be called before accessing 'this' in the constructor of a
derived class.
    super();
  }
}
```

忘记调用super是在JavaScript中容易犯的错误，但TypeScript会告诉你什么时候需要这样做。

方法

方法是类上的一个函数属性。方法可以使用所有与函数和构造函数相同的类型注解：

```
class Point {
  x = 10;
  y = 10;

  scale(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

除了标准的类型注解，TypeScript对方法没有添加任何新的东西。

注意，方法体内访问字段和其他方法仍然必须通过 `this.` 。在方法体内的无限定名字总是引用包围作用域内的某个东西：

```
let x: number = 0;

class C {
  x: string = "hello";

  m() {
    // 这是试图修改第1行的'x'，而不是类属性
    x = "world";
    // Type 'string' is not assignable to type 'number'.
  }
}
```

Getters / Setters

类也可以拥有访问器：

```
class C {
  _length = 0;
  get length() {
    return this._length;
  }
  set length(value) {
    this._length = value;
  }
}
```

> 注意，如果在get/set操作期间不需要添加额外的逻辑，那么具有无额外逻辑的字段支持的get/set对在JavaScript中很少有用。如果你不需要在get/set操作中添加额外的逻辑，那么公开公共字段是可以的。

TypeScript对访问器有一些特殊的推断规则：

- 如果存在get但不存在set，属性会自动变为readonly
- 如果未指定setter参数的类型，它会从getter的返回类型中推断出来
- getters和setters必须具有相同的成员可见性
- 自TypeScript 4.3起，可以让访问器的获取和设置有不同的类型。

```
class Thing {
  _size = 0;

  get size(): number {
    return this._size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);

    // Don't allow NaN, Infinity, etc

    if (!Number.isFinite(num)) {
      this._size = 0;
      return;
    }

    this._size = num;
  }
}
```

索引签名

类可以声明索引签名；这些与其他对象类型的索引签名工作方式相同：

```
class MyClass {
  [s: string]: boolean | ((s: string) => boolean);

  check(s: string) {
    return this[s] as boolean;
  }
}
```

因为索引签名类型也需要捕获方法的类型，所以这些类型不容易有用地使用。通常，最好在其他地方而不是在类实例本身上存储索引数据。

类的继承

像其他具有面向对象特性的语言一样，JavaScript中的类可以从基类继承。

implements

你可以使用 `implements` 检查一个类是否满足特定的接口。如果一个类没有正确实现它，会发出错误：

```
interface Pingable {
  ping(): void;
}

class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}

class Ball implements Pingable {
  // Class 'Ball' incorrectly implements interface 'Pingable'.
  // Property 'ping' is missing in type 'Ball' but required in type 'Pingable'.
  pong() {
    console.log("pong!");
  }
}
```

类也可以实现多个接口，例如 class C implements A, B {}。

注意事项

implements 只是检查类能否被视为接口类型。如果一个类未能正确实现它，将会发出错误。

```
interface Checkable {
  check(name: string): boolean;
}

class NameChecker implements Checkable {
  check(s) {
    // Parameter 's' implicitly has an 'any' type.
    // Notice no error here
    return s.toLowerCase() === "ok";

  }
}
```

在此示例中，我们可能预期 s 的类型会受到 check 中的 name: string 参数的影响。实际并非如此 - 实现子句不会改变类体的检查方式或其类型推断。

同样，实现一个具有可选属性的接口并不会创建该属性：

```
interface A {
  x: number;
  y?: number;
}
class C implements A {
  x = 0;
}
const c = new C();
c.y = 10;
// Property 'y' does not exist on type 'C'.
```

extends

类可以从一个基类扩展。派生类具有基类的所有属性和方法，并且还可以定义额外的成员。

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  woof(times: number) {
    for (let i = 0; i < times; i++) {
      console.log("woof!");
    }
  }
}

const d = new Dog();
// Base class method
d.move();
// Derived class method
d.woof(3);
```

覆盖方法

派生类还可以覆盖基类字段或属性。你可以使用 super. 语法来访问基类方法。注意，由于JavaScript类是一个简单的查找对象，因此没有“超级字段”的概念。

TypeScript强制要求一个派生类始终是其基类的子类型。

例如，这是一个合法的覆盖方法的方式：

```
```

```
class Base {
  greet() {
    console.log("Hello, world!");
  }
}

class Derived extends Base {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name.toUpperCase()}`);
    }
  }
}

const d = new Derived();
d.greet();
d.greet("reader");
```

派生类必须遵循其基类的约束。请记住，通过基类引用派生类实例是非常常见的（并且总是合法的！）：

```
// Alias the derived instance through a base class reference
const b: Base = d;
// No problem
b.greet();
```

如果Derived没有遵循Base的约束，会怎样呢?

```
class Base {
  greet() {
    console.log("Hello, world!");
  }
}

class Derived extends Base {
  // Make this parameter required
  greet(name: string) {
    // Property 'greet' in type 'Derived' is not assignable to the same property in
base type 'Base'.
    // Type '(name: string) => void' is not assignable to type '() => void'.
    console.log(`Hello, ${name.toUpperCase()}`);
  }
}
```

如果我们尽管存在错误，但仍编译了这段代码，这个示例将会崩溃：

```
const b: Base = new Derived();
// Crashes because "name" will be undefined
b.greet();
```

仅类型字段声明

当 target >= ES2022 或 useDefineForClassFields 为 true 时，类字段在父类构造函数完成后初始化，覆盖父类设置的任何值。当你只想重新声明一个继承字段更精确的类型时，这可能会成为一个问题。为了处理这些情况，你可以写 declare 来指示TypeScript，这个字段声明不应该有运行时效果。

```
interface Animal {
  dateOfBirth: any;
}

interface Dog extends Animal {
  breed: any;
}

class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}

class DogHouse extends AnimalHouse {
  // Does not emit JavaScript code,
  // only ensures the types are correct
  declare resident: Dog;
  constructor(dog: Dog) {
    super(dog);
  }
}
```

初始化顺序

JavaScript类的初始化顺序在某些情况下可能令人惊讶。让我们考虑这段代码：

```
class Base {
  name = "base";
  constructor() {
    console.log("My name is " + this.name);
  }
}

class Derived extends Base {
  name = "derived";
}

// Prints "base", not "derived"
const d = new Derived();
```

这里发生了什么?

类初始化的顺序，按照JavaScript的定义，是:

1. 初始化基类字段
2. 运行基类构造函数
3. 初始化派生类字段
4. 运行派生类构造函数

这意味着，基类构造函数在自己的构造函数中看到的是其自己的 name 值，因为派生类字段的初始化尚未运行。

成员可见性

您可以使用 TypeScript 来控制某些方法或属性是否对类外部的代码可见。

public

类成员的默认可见性是 public。公共成员可以在任何地方被访问:

```
class Greeter {
  public greet() {
    console.log("hi!");
  }
}
const g = new Greeter();
g.greet();
```

因为 public 已经是默认的可见性修饰符，所以你不需要在类成员上写它，但可能出于样式/可读性原因选择这样做。

protected

protected 成员只对它们所在的类的子类可见。

```
class Greeter {
  public greet() {
    console.log("Hello, " + this.getName());
  }
  protected getName() {
    return "hi";
  }
}

class SpecialGreeter extends Greeter {
  public howdy() {
    // 这里可以访问受保护成员
    console.log("Howdy, " + this.getName());
  }
}
const g = new SpecialGreeter();
g.greet(); // OK
g.getName(); // 错误: 'getName' 是受保护的，只能在 'Greeter' 类和其子类中访问
```

暴露受保护成员

派生类需要遵循其基类的约束，但可能选择暴露一个具有更多能力的基类的子类型。这包括将受保护的成员公开：

```
class Base {
  protected m = 10;
}
class Derived extends Base {
  // 没有修饰符，所以默认是 'public'
  m = 15;
}
const d = new Derived();
console.log(d.m); // OK
```

注意，Derived 已经能够自由地读写 m，所以这并没有实质性地改变这个情况的“安全性”。这里需要注意的主要一点是，在派生类中，如果这种暴露不是故意的，我们需要小心地重复受保护修饰符。

private

private 类似于 protected，但不允许从子类访问成员：

```
class Base {
  private x = 0;
}
const b = new Base();
// 不能从类外部访问
console.log(b.x); // 错误: 'x' 是私有的，只能在 'Base' 类中访问
```

```
class Derived extends Base {
  showX() {
    // 不能在子类中访问
    console.log(this.x); // 错误: 'x' 是私有的，只能在 'Base' 类中访问
  }
}
```

因为私有成员对派生类不可见，所以派生类不能增加它们的可见性：

```
class Base {
  private x = 0;
}
class Derived extends Base {
  x = 1; // 错误: 'Derived' 类不正确地

扩展了基类 'Base'。'x' 在类型 'Base' 中是私有的，但在类型 'Derived' 中不是。
}
```

注意事项

像 TypeScript 的类型系统的其他方面一样，private 和 protected 只在类型检查期间被强制执行。

这意味着 JavaScript 运行时构造，比如 in 或简单的属性查找，仍然可以访问私有或受保护的成员：

```
class MySafe {
  private secretKey = 12345;
}
```

```
// 在 JavaScript 文件中...
const s = new MySafe();
// 将打印 12345
console.log(s.secretKey);
```

private 还允许在类型检查期间使用方括号记法访问。这使得 private 声明的字段对于像单元测试这样的事情更易于访问，但缺点是这些字段是软私有的，不严格执行隐私。

```
class MySafe {
  private secretKey = 12345;
}

const s = new MySafe();

// 在类型检查期间不允许
console.log(s.secretKey); // 错误: 'secretKey' 是私有的，只能在 'MySafe' 类中访问

// OK
console.log(s["secretKey"]);
```

不像 TypeScript 的 private，JavaScript 的私有字段（#）在编译后仍然是私有的，并且不提供前面提到的像使用方括号访问这样的逃生舱口，使它们成为硬私有。

```
class Dog {
  #barkAmount = 0;
  personality = "happy";

  constructor() {}
}
```

当编译为 ES2021 或更低版本时，TypeScript 将使用 WeakMaps 代替 #。

```
"use strict";
var _Dog_barkAmount;
class Dog {
    constructor() {
        _Dog_barkAmount.set(this, 0);
        this.personality = "happy";
    }
}
_Dog_barkAmount = new WeakMap();
```

如果你需要保护你的类中的值不受恶意攻击者的侵犯，你应该使用提供硬运行时隐私的机制，比如闭包、WeakMaps 或私有字段。请注意，这些在运行时增加的隐私检查可能会影响性能。

静态成员

类可能具有静态成员。这些成员不与类的特定实例关联。它们可以通过类构造器对象本身进行访问：

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }
}
console.log(MyClass.x);
MyClass.printX();
```

静态成员也可以使用相同的 public、protected 和 private 可见性修饰符：

```
class MyClass {
  private static x = 0;
}
console.log(MyClass.x);
// 错误：'x' 是私有的，只能在 'MyClass' 类中访问
```

静态成员也会被继承：

```
class Base {
  static getGreeting() {
    return "Hello world";
  }
}
class Derived extends Base {
  myGreeting = Derived.getGreeting();
}
```

特殊的静态名字

通常来说，无法覆盖 Function 原型中的属性。因为类本身就是可以用 new 调用的函数，所以某些静态名字不能被使用。Function 属性如 name、length 和 call 不可以被定义为静态成员：

```
class S {
  static name = "S!";
  // 错误：静态属性 'name' 与构造函数 'S' 的内置属性 'Function.name' 冲突
}
```

在类中的静态块

静态块允许你编写一系列具有自己作用域的语句，它们可以访问包含类中的私有字段。这意味着我们可以编写初始化代码，所有的功能都能写语句，没有变量泄漏，完全访问我们类的内部。

```
class Foo {
    static #count = 0;

    get count() {
        return Foo.#count;
    }

    static {
        try {
            const lastInstances = loadLastInstances();
            Foo.#count += lastInstances.length;
        }
        catch {}
    }
}
```

泛型类

类，就像接口一样，可以使用泛型。当使用 new 实例化一个泛型类时，它的类型参数以与函数调用相同的方式被推断：

```
class Box<Type> {
  contents: Type;
  constructor(value: Type) {
    this.contents = value;
  }
}

const b = new Box("hello!"); // const b: Box<string>
```

类可以像接口一样使用泛型和默认值。

静态成员中的类型参数

这段代码是非法的，原因可能并不明显：

```
class Box<Type> {
  static defaultValue: Type;
  // Static members cannot reference class type parameters.
}
```

记住，类型总是完全被擦除！在运行时，只有一个 `Box.defaultValue` 属性槽。这意味着，如果可能的话，设置 `Box<string>.defaultValue` 也会改变 `Box<number>.defaultValue` 这不是我们想要的。通用类的静态成员永远不能引用类的泛型参数。

在运行时类中的 `this`

重要的是要记住，TypeScript不会改变JavaScript的运行时行为，而JavaScript对于某些特殊的运行时行为很有名。

JavaScript对 `this` 的处理的确不寻常：

```
class MyClass {
  name = "MyClass";
  getName() {
    return this.name;
  }
}
const c = new MyClass();
const obj = {
  name: "obj",
  getName: c.getName,
};

// 打印 "obj"，而不是 "MyClass"
console.log(obj.getName());
```

长话短说，默认情况下，函数内部的 `this` 的值取决于函数是如何被调用的。在这个例子中，因为函数是通过 obj 引用被调用的，所以它的 `this` 的值是 obj 而不是类的实例。

这通常不是你希望发生的！TypeScript提供了一些方式来减轻或防止这种错误。

箭头函数

如果你有一个函数，经常以失去其 `this` 上下文的方式被调用，那么使用箭头函数属性而不是方法定义可能更有意义：

```
class MyClass {
  name = "MyClass";
  getName = () => {
    return this.name;
  };
}
const c = new MyClass();
const g = c.getName;
// 打印 "MyClass" 而不是崩溃
console.log(g());
```

这种做法有一些需要注意的地方：

1. 即使对于未经TypeScript检查的代码，运行时的 this 值也保证是正确的。
2. 这将使用更多的内存，因为每个类实例都会拥有以这种方式定义的每个函数的自己的副本。
3. 在派生类中，你不能使用 super.getName，因为在原型链中没有入口来获取基类方法。

`this`参数

在方法或函数定义中，一个名为 `this` 的初始参数在TypeScript中具有特殊的意义。这些参数在编译期间被擦除：

```
// TypeScript input with 'this' parameter
function fn(this: SomeType, x: number) {
  /* ... */
}
// JavaScript output
function fn(x) {
  /* ... */
}
```

TypeScript检查调用带有 `this` 参数的函数是否以正确的上下文进行。我们可以添加一个 `this` 参数到方法定义，以静态地强制该方法被正确调用：

```
class MyClass {
  name = "MyClass";
  getName(this: MyClass) {
    return this.name;
  }
}
const c = new MyClass();
// OK
c.getName();

// Error, would crash
const g = c.getName;
console.log(g());
//The 'this' context of type 'void' is not assignable to method's 'this' of type
'MyClass'.
```

和箭头函数不同：

1. JavaScript的调用者可能仍然会在不知情的情况下错误地使用类方法。
2. 每个类定义只分配一个函数，而不是每个类实例都分配一个函数。
3. 可以通过super调用基类方法的定义。

`this`类型

在类中，一种特殊的类型叫做 `this` 动态地引用当前类的类型。让我们看看这是如何有用的：

```
class Box {
  contents: string = "";
  set(value: string) {
    // (method) Box.set(value: string): this
    this.contents = value;
    return this;
  }
}
```

在这里，TypeScript推断了set的返回类型为 `this`，而不是 Box。现在让我们创建一个 Box 的子类：

```
class ClearableBox extends Box {
  clear() {
    this.contents = "";
  }
}

const a = new ClearableBox();
const b = a.set("hello");
// const b: ClearableBox
```

你也可以在参数类型注解中使用 `this` ：

```
class Box {
  content: string = "";
  sameAs(other: this) {
    return other.content === this.content;
  }
}
```

这与写 other: Box 不同 - 如果你有一个派生类，其 sameAs 方法现在只会接受那个相同派生类的其他实例：

```
class Box {
  content: string = "";
  sameAs(other: this) {
    return other.content === this.content;
  }
}

class DerivedBox extends Box {
```

```
  otherContent: string = "?";
}

const base = new Box();
const derived = new DerivedBox();
derived.sameAs(base);
Argument of type 'Box' is not assignable to parameter of type 'DerivedBox'.
  Property 'otherContent' is missing in type 'Box' but required in type
'DerivedBox'.
```

基于 `this` 的类型守卫

你可以在类和接口的方法的返回位置使用 `this is Type`。当与类型收窄（例如 if 语句）混合使用时，目标对象的类型将收窄为指定的 Type。

例如，下面的代码定义了一个 `FileSystemObject` 类，以及几个用于类型判断的方法：

```
class FileSystemObject {
  isFile(): this is FileRep {
    return this instanceof FileRep;
  }
  isDirectory(): this is Directory {
    return this instanceof Directory;
  }
  isNetworked(): this is Networked & this {
    return this.networked;
  }
  constructor(public path: string, private networked: boolean) {}
}

class FileRep extends FileSystemObject {
  constructor(path: string, public content: string) {
    super(path, false);
  }
}

class Directory extends FileSystemObject {
  children: FileSystemObject[];
}

interface Networked {
  host: string;
}

const fso: FileSystemObject = new FileRep("foo/bar.txt", "foo");

if (fso.isFile()) {
  fso.content;  // 这里 fso 被收窄为 FileRep 类型
```

```
} else if (fso.isDirectory()) {
  fso.children;  // 这里 fso 被收窄为 Directory 类型
} else if (fso.isNetworked()) {
  fso.host;  // 这里 fso 被收窄为 Networked & FileSystemObject 类型
}
```

参数属性

参数属性是一种简写，它允许你在构造函数中通过给构造函数参数添加可见性关键词（`public`、`private`、`protected`、`readonly`）来创建和初始化类成员。这可以使你的 TypeScript 代码更简洁、易于阅读。

这是使用参数属性的一个例子：

```
class MyClass {
  constructor(public myPublicProperty: string, private myPrivateProperty: number) {
  }

  printProperties() {
    console.log(`公共属性: ${this.myPublicProperty}`);
    console.log(`私有属性: ${this.myPrivateProperty}`);
  }
}

let obj = new MyClass("你好", 42);
obj.printProperties();
```

在这个例子中，`myPublicProperty` 和 `myPrivateProperty` 根据构造函数参数自动声明和初始化。`public` 关键词使 `myPublicProperty` 在类外部可见，而 `private` 关键词使 `myPrivateProperty` 仅在类内部可见。

`readonly` 关键词也可以用于参数属性，意味着一旦一个属性被赋值，它就不能被更改：

```
class MyClass {
  constructor(readonly myReadonlyProperty: string) {
  }
}

let obj = new MyClass("你好");
console.log(obj.myReadonlyProperty);  // "你好"
obj.myReadonlyProperty = "世界";  // 错误：不能赋值给只读属性
```

类表达式

类表达式是一种定义类的方式，它允许你使用表达式来定义一个类，然后再把这个类赋值给一个变量。这个变量就可以当作类使用，你可以用它来创建类的实例。类表达式与函数表达式类似，可以是匿名的，也可以是命名的。

下面是一个使用类表达式的示例：

```
let MyClass = class {
  constructor(public value: string) {}

  printValue() {
    console.log(this.value);
  }
};

let myInstance = new MyClass("Hello, world!");
myInstance.printValue(); // 输出: "Hello, world!"
```

在这个例子中，`MyClass` 是一个由类表达式定义的类，然后我们用这个类来创建了一个新的实例 `myInstance`。

类表达式也可以是命名的，这允许你在类内部引用类本身，比如在递归方法中：

```
let MyClass = class MyClass {
  constructor(public value: number) {}

  multiply(factor: number): number {
    return this.value * factor;
  }

  multiplyRecursively(factor: number, times: number): number {
    if (times <= 0) {
      return this.value;
    } else {
      return new MyClass(this.multiply(factor)).multiplyRecursively(factor, times -
1);
    }
  }
};

let myInstance = new MyClass(2);
console.log(myInstance.multiplyRecursively(2, 3)); // 输出: 16
```

这里的 `MyClass` 是类表达式的名称，它只在类内部可见。

抽象类

在 TypeScript 中，`abstract` 关键字用于定义抽象类和抽象方法。抽象类是一种特殊的类，不能直接实例化，只能作为其他类的基类。抽象方法是一种只存在于抽象类中的特殊方法，没有实现（也就是说，没有方法体）。

在派生类中，必须实现抽象类中的所有抽象方法。以下是一个使用 `abstract` 关键字的例子：

```
abstract class Animal {
  abstract makeSound(): void;

  move(): void {
    console.log("Roaming the earth...");
  }
}

class Dog extends Animal {
  makeSound() {
    console.log('Bark!');
  }
}

const myDog = new Dog();
myDog.makeSound();  // 输出 'Bark!'
myDog.move();  // 输出 'Roaming the earth...'
```

在这个例子中，`Animal` 是一个抽象类，它有一个抽象方法 `makeSound` 和一个普通方法 `move`。`Dog` 是 `Animal` 的一个子类，它实现了 `makeSound` 方法。当我们创建一个 `Dog` 的实例并调用这些方法时，就会看到它们的行为。

注意，如果一个类包含一个或多个抽象方法，那么这个类必须被声明为抽象类。如果派生类没有实现基类的抽象方法，那么 TypeScript 将会在编译时报错。

类之间的关系

在大多数情况下，TypeScript 中的类与其他类型一样，是通过结构进行比较的。

例如，这两个类可以互换使用，因为它们完全相同：

```
class Point1 {
  x = 0;
  y = 0;
}

class Point2 {
  x = 0;
  y = 0;
}

// OK
const p: Point1 = new Point2();
```

同样，即使没有显式继承，类之间也存在子类型关系：

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

class Employee {
  name: string;
  age: number;
  salary: number;
}

// OK
const p: Person = new Employee();
```

这听起来很简单，但有几种情况似乎比其他情况更奇怪。

空类没有成员。在结构类型系统中，没有成员的类型通常是其他任何类型的超类型。因此，如果你定义一个空类（不要这么做！），任何东西都可以用来替代它：

```
class Empty {}

function fn(x: Empty) {
  // 对于 'x' 无法进行任何操作，因此我不会做任何事情
}

// 全部都OK!
fn(window);
fn({});
fn(fn);
```

6. 其他

三斜线指令

在TypeScript中，三斜杠指令是一种特殊的指令，它主要用于声明文件之间的依赖关系、对内置库的引用和一些特定的编译器行为。以下是一些常见的使用场景：

1. **文件之间的依赖管理**：`/// <reference path="..." />`指令。当你的代码是全局的，不是模块化的，并且一个文件依赖于另一个文件时，这个指令就会派上用场。但是，在大多数现代的JavaScript和TypeScript项目中，这种用法已经被模块系统（如CommonJS、AMD和ES6模块）所取代。
2. **声明对类型定义包的依赖**：`/// <reference types="..." />`指令。如果你的代码依赖于某个类型定义包（如@types/node）中的类型，那么你就需要使用这个指令。这对于那些需要全局类型（例如jest，mocha等）的项目尤为有用。
3. **引用内置库**：`/// <reference lib="..." />`指令。如果你的代码需要使用TypeScript提供的内置库（如 "es2015"、"dom" 等），你就需要使用这个指令。

需要注意的是，随着TypeScript和JavaScript生态的发展，三斜杠指令的使用已经变得越来越少，大部分功能都可以通过更现代的工具和配置方式实现（例如使用模块导入/导出语法和tsconfig.json文件进行项目配置）。在某些特殊的环境和配置中，三斜杠指令可能仍然有其用武之地，但在大多数情况下，它们已经被其他技术所取代。

命名空间

在 TypeScript 中，命名空间是一个关键的概念，用于组织代码并避免命名冲突。让我们以一个实际的例子来看一下。

假设我们正在开发一个大型的电子商务应用，有许多不同的模块，例如用户(User)模块、商品(Product)模块、订单(Order)模块等。在没有使用命名空间的情况下，我们可能会遇到如下问题：

1. 命名冲突：例如，`User`、`Product`、`Order`等类可能在不同的模块中被重复定义，导致命名冲突。
2. 代码组织：如果没有适当的组织结构，难以维护和理解。

这时，命名空间可以解决上述问题。让我们看看如何使用它：

```
namespace UserModule {
  export class User {
    // User class implementation
  }
}

namespace ProductModule {
  export class Product {
    // Product class implementation
  }

  export class ProductDetail {
    // ProductDetail class implementation
```

```
  }
}

namespace OrderModule {
  export class Order {
    // Order class implementation
  }
}
```

在这个例子中，我们定义了三个命名空间：`UserModule`、`ProductModule`和`OrderModule`。每个命名空间中都定义了一些类，这样，就算类名相同，由于它们在不同的命名空间中，也不会引起命名冲突。

我们可以这样使用这些类：

```
let user = new UserModule.User();
let product = new ProductModule.Product();
let order = new OrderModule.Order();
```

在这个例子中，我们看到使用命名空间可以有效地组织代码并避免命名冲突。然而，需要注意的是，在大多数情况下，使用 ES6 的模块系统（即 `import` 和 `export` 语法）可以更好地组织代码，并且可以利用现代 JavaScript 工具链中的模块解析和捆绑功能。在一些特定情况下，例如在全局环境中定义类型或在声明合并中，命名空间可能会派上用场。

在 TypeScript 的命名空间中，你可以使用 `export` 关键字来决定一个元素是否应被导出（即，它是否应该在命名空间外部可见）。如果你没有为一个元素添加 `export` 关键字，那么它将只在命名空间内部可见。

例如，假设我们有一个名为 `MyNamespace` 的命名空间：

```
namespace MyNamespace {
    // 内部（未导出）变量
    let internalVariable = "I'm internal!";

    // 导出变量
    export let externalVariable = "I'm external!";

    // 内部（未导出）函数
    function internalFunction() {
        console.log("I'm an internal function!");
    }

    // 导出函数
    export function externalFunction() {
        console.log("I'm an external function!");
    }
}
```

在这个例子中，`internalVariable` 和 `internalFunction` 都只在 `MyNamespace` 内部可见，因为它们没有使用 `export` 关键字。相反，`externalVariable` 和 `externalFunction` 都使用了 `export` 关键字，所以它们在 `MyNamespace` 外部也可以被访问，如下所示：

```
// 访问导出的变量
console.log(MyNamespace.externalVariable);  // 输出 "I'm external!"

// 访问导出的函数
MyNamespace.externalFunction();  // 输出 "I'm an external function!"

// 访问内部变量或函数会引发错误
console.log(MyNamespace.internalVariable);  // 错误: Property 'internalVariable'
does not exist on type 'typeof MyNamespace'
MyNamespace.internalFunction();  // 错误: Property 'internalFunction' does not exist
on type 'typeof MyNamespace'
```

这样，你可以在命名空间内部保持一些实现细节的私有性，而只暴露那些需要被外部访问的部分。

但是，需要注意的是，现代的 TypeScript 和 JavaScript 项目大多数都倾向于使用 ES 模块（也就是使用 `import` 和 `export`）而不是命名空间。ES 模块提供了更好的模块作用域和更高效的代码组织方式。另外，许多现代的 JavaScript 工具（例如 Babel、Webpack、Rollup 等）都支持 ES 模块，并且可以提供例如代码拆分、懒加载等高级功能。

因此，虽然命名空间在某些情况下（例如在全局环境中或者需要声明合并时）可能仍然有用，但在大多数情况下，你应该首选 ES 模块。

枚举

枚举（Enum）是一种特殊的类型，在 TypeScript 中用于处理一组有名称的常量值。枚举的值一般为整数或字符串，但也可以有更复杂的结构。枚举为我们提供了一种友好、可读性高的方式来处理这些集合，而不是使用魔法字符串或不清晰的数字值。

以下是一个简单的枚举使用示例：

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

在这个示例中，`Direction` 枚举有四个成员：`Up`，`Down`，`Left` 和 `Right`。默认情况下，第一个成员的值为 `0`，后面的成员则自动递增。也就是说，`Direction.Up` 的值为 `0`，`Direction.Down` 的值为 `1`，以此类推。你也可以手动为枚举成员指定值：

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}
```

在这个示例中，`Direction.Up` 的值为 `1`，`Direction.Down` 的值为 `2`，以此类推。

枚举也可以被当做类型使用。例如，我们可以定义一个只接受 `Direction` 枚举值的函数：

```
function move(direction: Direction) {
  // ...
}

move(Direction.Up); // Okay
move(0); // Error
```

枚举可以被反向映射（从值到名字），如：

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}

let directionName: string = Direction[1]; // directionName is 'Up'
```

此外，TypeScript 还支持字符串枚举：

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}
```

在字符串枚举中，每个成员都必须被手动赋值为一个字符串。

枚举提供了一个便利的方式来组织和处理一组相关的常量值，它提高了代码的可读性和可维护性。

常量枚举

在TypeScript中，常量枚举是一种特殊类型的枚举，它们与普通的枚举在语义上稍有不同。当你在枚举前加上`const`关键字，TypeScript编译器会进行特殊处理。

这是一个常量枚举的例子：

```
const enum Directions {
    Up,
    Down,
    Left,
    Right
}
```

使用常量枚举的主要优点是性能和类型安全方面的优化。当你使用常量枚举时，TypeScript编译器会在生成的JavaScript代码中内联枚举的值。这意味着在运行时不会有一个真正的枚举对象，而是直接使用常量值。这可以减少在运行时的查找开销，并可以减小生成的JavaScript代码大小。

这是在TypeScript代码中使用常量枚举的例子：

```
let dir: Directions = Directions.Up;
```

编译后的JavaScript代码为：

```
let dir = 0 /* Up */;
```

注意这种情况下，`Directions.Up`被编译器替换成了它的值`0`，并且没有生成对应的JavaScript枚举对象。

然而，常量枚举也有一些限制。由于它们在运行时并不存在，所以你不能在运行时访问枚举成员，也不能使用常量枚举类型在运行时进行动态的键或值查找。而且，当`--preserveConstEnums`编译选项未开启时，常量枚举成员也不能在编译后的代码中以`.name`方式访问。

总的来说，常量枚举在需要优化性能和代码大小的场景下是很有用的，但在需要运行时枚举行为的情况下，你应该使用普通的枚举。

环境枚举

"环境枚举"（Ambient enums）是在声明文件中使用的枚举类型。它们用来描述已经存在的枚举类型的形状。

"环境枚举"在语法上类似于常规的枚举，但它们在声明的右侧没有初始化方法。例如：

```
declare enum Enum {
  A = 1,
  B,
  C = 2
}
```

你可能想知道为什么我们需要用 "环境枚举"。一个原因是它允许你表示用非TypeScript代码（例如，JavaScript或者C/C++）编写的枚举类型的值。在这些情况下，你可以使用 "环境枚举" 在 TypeScript 中表示这些外部的枚举类型。

然而，与常规的枚举相比，"环境枚举"的主要区别是，它们不会在编译生成的 JavaScript 代码中产生任何输出。这是因为它们只是用来描述已经存在的枚举类型的值，而不是创建新的枚举类型。

因此，如果你正在编写一个全新的 TypeScript 项目，并且你想在你的代码中使用枚举，那么你可能不需要使用 "环境枚举"。"环境枚举"主要用于那些需要与已存在的、非TypeScript编写的代码库进行互操作的情况。

工具类型

在 TypeScript 中，有一些内建的工具类型，它们能够帮助你处理一些常见的类型转换或操作。以下是一些常见的工具类型：

1. **Partial**: 构建一个类型，将 T 的所有属性标记为可选。这对于构建可以接受部分属性的函数或方法特别有用。
2. **Readonly**: 构建一个类型，将 T 的所有属性标记为只读。这在你想要防止对象的属性被修改时特别有用。
3. **Record<K,T>**: 构建一个类型，其属性名的类型为 K，属性值的类型为 T。
4. **Pick<T, K extends keyof T>**: 构建一个类型，只选择 T 的 K 属性。
5. **Omit<T, K extends keyof T>**: 构建一个类型，从 T 中剔除 K 属性。
6. **Exclude<T, U>**: 构建一个类型，剔除 T 中可以赋值给 U 的类型。
7. **Extract<T, U>**: 构建一个类型，从 T 中提取可以赋值给 U 的类型。
8. **NonNullable**: 构建一个类型，从 T 中剔除 null 和 undefined。
9. **ReturnType**: 获取函数类型 T 的返回类型。
10. **InstanceType**: 获取构造函数类型 T 的实例类型。

这些工具类型可以帮助你更灵活地处理类型，避免重复编写类型定义。通过组合这些工具类型，你可以构建出各种复杂的类型操作。

7. 声明文件

在大多数情况下，当我们使用 JavaScript 库时，TypeScript 并不知道这些库的具体类型。这就导致了 TypeScript 在进行类型检查时无法提供正确的错误提示和代码自动完成。为了解决这个问题，我们可以使用声明文件来告诉 TypeScript 这些库的类型信息。

声明文件的扩展名为 `.d.ts` ，它的一些主要用途包括：

1. 定义非 TypeScript 库的类型信息：当我们在 TypeScript 项目中使用 JavaScript 库时，我们可以使用 `.d.ts` 文件来提供这些库的类型定义。这样 TypeScript 就可以为这些库提供类型检查和自动完成功

能。
2. 全局变量的类型定义：在某些情况下，我们可能需要在全局范围内定义一些变量。我们可以使用 `.d.ts` 文件来为这些全局变量提供类型定义。
3. 为现有类型添加新的方法：有时我们可能需要给现有的类型（如 String、Number、Array 等）添加新的方法。我们可以使用 `.d.ts` 文件来为这些新方法提供类型定义。

因此，无论是对于一个 JavaScript 库的开发者（他们可以编写声明文件，使得 TypeScript 用户可以更容易地使用他们的库），还是对于 TypeScript 用户（他们可以使用声明文件，来获取对 JavaScript 库的类型检查和智能提示），声明文件都是非常有用的。

创建声明文件

创建一个 `.d.ts` 文件其实很简单，就像创建一个普通的 `.ts` 或 `.js` 文件一样，你只需要在你选择的目录下创建一个扩展名为 `.d.ts` 的文件。在该文件中，你可以编写类型声明。

下面是一个基本的 `.d.ts` 文件的示例。在这个例子中，我们为一个 JavaScript 模块 `someModule.js` 创建了类型声明：

```
// someModule.d.ts
declare module 'someModule' {
    export function someFunction(param: string): number;
}
```

在这个例子中，我们声明了一个名为 `someModule` 的模块，并声明了这个模块导出的一个函数 `someFunction`。这个函数接受一个字符串参数并返回一个数字。

记住，你不需要在你的 TypeScript 代码中显式地导入这个 `.d.ts` 文件。TypeScript 编译器会自动找到并使用这个文件。

创建 `.d.ts` 文件时，最重要的是确保你的类型声明是准确的。如果你的类型声明不准确，那么 TypeScript 无法正确地进行类型检查，这可能会导致难以发现的错误。因此，你应该仔细检查你的类型声明，确保它们与你的 JavaScript 代码相匹配。

TypeScript 会根据一定的规则查找和解析声明（`.d.ts`）文件。以下是这些规则的概述：

1. **包含文件：**默认情况下，TypeScript 会包含所有的 TypeScript 文件（`.ts`），tsx文件（`.tsx`），以及 JavaScript 文件（`.js`）。对于声明文件，它会包含在指定文件夹下的所有 `.d.ts` 文件。
2. **tsconfig.json：**你可以在项目的根目录下创建一个 `tsconfig.json` 文件，以更改 TypeScript 的编译选项和包含文件。在 `tsconfig.json` 文件中，你可以使用 `include` 和 `exclude` 字段来指定哪些文件应该被包含或排除。

   例如，以下的 `tsconfig.json` 配置会让 TypeScript 编译器只包含 `src` 文件夹下的所有文件：

   ```
   {
     "include": ["src"]
   }
   ```

3. **类型声明查找**：当你在代码中导入一个模块，如 `import * as foo from 'foo'`，TypeScript 会尝试在 `node_modules/@types/foo/index.d.ts` 或 `node_modules/foo/index.d.ts` 等路径下查找类型声明文件。
4. **三斜线指令**：你也可以使用三斜线指令来直接指定声明文件的位置，如 `/// <reference path="path/to/declaration.d.ts" />`。然而，这种方式现在已经不太推荐，因为它使得类型依赖关系变得难以管理。

最后需要注意的是，如果你的声明文件被正确解析，那么你不需要在你的代码中显式地导入这些声明文件。你可以直接使用声明文件中定义的类型。

最佳位置

TypeScript 声明文件（`.d.ts` 文件）通常有几种最佳的存放位置：

1. **在 node_modules/@types 文件夹中**：这是你安装的所有来自 DefinitelyTyped 的类型声明的地方。当你使用 `npm install @types/<library>` 安装类型声明时，它们会被放在这个文件夹里。你通常不会手动修改这个文件夹里的文件。
2. **项目根目录的 types 文件夹中**：如果你正在编写一个库，并希望发布你的类型声明，你可能希望把它们放在这里。然后，你可以在你的 `package.json` 文件的 `"types"` 字段中指向这个文件夹。
3. **源码同级目录**：如果你正在为你的 JavaScript 代码写类型声明，你可能希望把 `.d.ts` 文件放在你的 JavaScript 文件旁边。这样，TypeScript 就可以很容易地找到你的类型声明。

在你的 `tsconfig.json` 文件中，你可以使用 `"include"` 和 `"exclude"` 字段来控制哪些文件被 TypeScript 编译器考虑。例如，你可能希望 `"include"` 你的 `src` 和 `types` 文件夹，而 `"exclude"` 你的 `node_modules` 文件夹。

一般来说，你应该把你的类型声明放在 TypeScript 最容易找到的地方。TypeScript 编译器会自动在你的项目和 `node_modules/@types` 文件夹中查找类型声明。如果你的类型声明放在别的地方，你可能需要修改你的 `tsconfig.json` 文件来告诉 TypeScript 在哪里找到它们。

案例

全局变量

全局变量 foo 包含当前存在的部件数量。

代码

```
console.log("部件数量的一半是 " + foo / 2);
```

声明

使用 `declare var` 来声明变量。如果变量是只读的，你可以使用 `declare const`。如果变量是块作用域的，你也可以使用 `declare let`。

```
/** 当前存在的部件数量 */
declare var foo: number;
```

全局函数

你可以使用字符串调用函数 greet，向用户显示问候。

代码

```
greet("hello, world");
```

声明

使用 `declare function` 来声明函数。

```
declare function greet(greeting: string): void;
```

全局对象

全局变量 myLib 有一个函数 makeGreeting，和一个属性 numberOfGreetings。

代码

```
let result = myLib.makeGreeting("hello, world");
console.log("The computed greeting is:" + result);
let count = myLib.numberOfGreetings;
```

声明

使用 declare namespace 来描述通过点运算符访问的类型或值。

```
declare namespace myLib {
  function makeGreeting(s: string): string;
  let numberOfGreetings: number;
}
```

declare

在 TypeScript 中，`declare` 关键字用于声明变量、函数、类等的类型，而不需要实现它们。这主要在类型声明文件（`.d.ts` 文件）中使用，用来描述 JavaScript 库的类型，这样 TypeScript 就可以知道如何与 JavaScript 库交互。

例如，假设你有一个 JavaScript 库，它在全局对象 `myLib` 上提供了一个函数 `makeGreeting`，你可以在一个 `.d.ts` 文件中这样声明它：

```
declare namespace myLib {
  function makeGreeting(s: string): string;
}
```

这样，TypeScript 就知道 `myLib.makeGreeting` 是一个接受一个字符串参数并返回一个字符串的函数。

还有一个常见的用例是使用 `declare` 声明模块。例如，假设你在使用一个名为 "my-module" 的 JavaScript 库，你可以在 `.d.ts` 文件中这样声明：

```
declare module 'my-module';
```

这样，即使 "my-module" 不是用 TypeScript 写的，你也可以在你的 TypeScript 代码中 `import` 它。

在 `.d.ts` 文件中，所有的顶级声明（不在 `namespace` 或 `module` 里面的声明）都需要使用 `declare`。在 `.ts` 文件中，`declare` 用于声明全局的变量、函数、类等。

需要注意的是，`declare` 声明的只是类型，不会生成任何真正的 JavaScript 代码。这意味着，你不能在 `declare` 语句中实现函数或方法，只能声明它们的类型。

重载函数

`getWidget` 函数接受一个数字并返回一个 `Widget`，或者接受一个字符串并返回一个 `Widget` 数组。

代码

```
let x: Widget = getWidget(43);
let arr: Widget[] = getWidget("all of them");
```

声明

```
interface Widget {}
declare function getWidget(n: number): Widget;
declare function getWidget(s: string): Widget[];
```

在这个例子中，`getWidget` 函数被声明为接受一个数字或一个字符串作为参数。如果参数是数字，返回的是一个 `Widget` 对象，如果参数是字符串，返回的是一个 `Widget` 数组。通过声明函数重载，TypeScript 能够理解 `getWidget` 函数的行为，并且可以对其进行正确的类型检查。

可重用类型（接口）

当指定问候语时，你必须传递一个 `GreetingSettings` 对象。此对象具有以下属性：

1. greeting: 必须的字符串
2. duration: 可选的持续时间（以毫秒为单位）
3. color: 可选的字符串，例如 '#ff00ff'

代码

```
greet({
  greeting: "hello world",
  duration: 4000
});
```

声明

使用接口定义具有属性的类型。

```
interface GreetingSettings {
  greeting: string;
  duration?: number;
  color?: string;
}

declare function greet(setting: GreetingSettings): void;
```

在这个示例中，我们定义了一个名为 `GreetingSettings` 的接口来描述配置对象的结构。这个接口包含三个属性，其中 `greeting` 是必须的，`duration` 和 `color` 是可选的。然后，我们声明了一个函数 `greet`，它接收一个 `GreetingSettings` 类型的参数。这样，TypeScript 就可以理解 `greet` 函数接收何种结构的参数，并进行类型检查。

可重用类型（类型别名）

在任何期望得到问候语的地方，你都可以提供一个字符串、一个返回字符串的函数，或者一个 `Greeter` 实例。

代码

```
function getGreeting() {
  return "howdy";
}
class MyGreeter extends Greeter {}
greet("hello");
greet(getGreeting);
greet(new MyGreeter());
```

声明

你可以使用类型别名来为类型创建一个简称：

```
type GreetingLike = string | (() => string) | MyGreeter;
declare function greet(g: GreetingLike): void;
```

在这个示例中，我们使用 `type` 关键字创建了一个类型别名 `GreetingLike`，这个类型可以是字符串，也可以是返回字符串的函数，或者是 `MyGreeter` 类的实例。然后，我们声明了一个名为 `greet` 的函数，它接收一个 `GreetingLike` 类型的参数。这样，TypeScript 就可以理解 `greet` 函数接收何种类型的参数，并进行类型检查。

组织类型

`greeter` 对象可以记录到文件或显示一个警告。你可以向 `.log(...)` 提供 `LogOptions`，并向 `.alert(...)` 提供警告选项。

代码

```
const g = new Greeter("Hello");
g.log({ verbose: true });
g.alert({ modal: false, title: "Current Greeting" });
```

声明

使用命名空间来组织类型。

```
declare namespace GreetingLib {
  interface LogOptions {
    verbose?: boolean;
  }
  interface AlertOptions {
    modal: boolean;
    title?: string;
    color?: string;
  }
}
```

你还可以在一个声明中创建嵌套的命名空间：

```
declare namespace GreetingLib.Options {
  // 通过 GreetingLib.Options.Log 引用
  interface Log {
    verbose?: boolean;
  }
  interface Alert {
    modal: boolean;
    title?: string;
    color?: string;
  }
}
```

在这个示例中，我们使用 `declare namespace` 关键字创建了两个命名空间 `GreetingLib` 和 `GreetingLib.Options`，并在这些命名空间中定义了几个接口。这种方式可以帮助我们更好地组织代码，并使得我们的类型更容易被其他人理解和使用。

类

你可以通过实例化 `Greeter` 对象来创建一个 greeter，或者通过从它继承来创建一个定制的 greeter。

代码

```
const myGreeter = new Greeter("hello, world");
myGreeter.greeting = "howdy";
myGreeter.showGreeting();

class SpecialGreeter extends Greeter {
  constructor() {
    super("Very special greetings");
  }
}
```

声明

使用 `declare class` 来描述一个类或类似类的对象。类可以有属性和方法，以及一个构造函数。

```
declare class Greeter {
  constructor(greeting: string);
  greeting: string;
  showGreeting(): void;
}
```

在这个示例中，我们使用 `declare class` 关键字创建了一个 `Greeter` 类，它有一个构造函数，一个 `greeting` 属性和一个 `showGreeting` 方法。这样我们就可以在 TypeScript 中使用这个类了。

库描述文件

大体上来说，你的声明文件的结构取决于如何使用该库。在JavaScript中，有许多方式来提供一个库供使用，你需要编写与之相匹配的声明文件。本指南涵盖了如何识别常见的库模式，以及如何编写与该模式相对应的声明文件。

每种主要的库结构模式在模板部分都有对应的文件。你可以从这些模板开始，以帮助你更快地开始工作。

模块库

几乎每个现代的Node.js库都属于模块家族。这种类型的库只能在带有模块加载器的JS环境中工作。例如，express只能在Node.js中工作，并且必须使用CommonJS的require函数来加载。

ECMAScript 2015（也被称为ES2015、ECMAScript 6和ES6）、CommonJS和RequireJS都有类似的导入模块的概念。例如，在JavaScript CommonJS（Node.js）中，你可以这样写：

```
var fs = require("fs");
```

在TypeScript或ES6中，import关键字具有相同的功能：

```
import * as fs from "fs";
```

你通常会在模块化库的文档中看到下面这样的代码行：

```
var someLib = require("someLib");
```

或者

```
define(..., ['someLib'], function(someLib) {
});
```

像全局模块一样，你可能会在UMD模块的文档中看到这些例子，所以一定要检查代码或文档。

从代码中识别模块库

模块化库通常至少具有以下一些特征：

- 无条件调用 require 或 define
- 声明语句如 import * as a from 'b'; 或 export c;
- 分配给 exports 或 module.exports

他们很少会有：

- 分配给 window 或 global 的属性

模块模板

有四种可用于模块的模板，module.d.ts、module-class.d.ts、module-function.d.ts 和 module-plugin.d.ts。

你应首先阅读 module.d.ts 以了解它们的工作方式：

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
// Definitions by: [~你的名字~] <[~你的URL~]>
/*~ 这是模块的模板文件。你应该将其重命名为 index.d.ts
 *~ 并将其放在与模块同名的文件夹中。
 *~ 例如，如果你正在为 "super-greeter" 编写一个文件，那么
 *~ 这个文件应该是 'super-greeter/index.d.ts'
 */

/*~ 如果此模块是一个在没有模块加载器环境下
 *~ 通过全局变量 'myLib' 提供的 UMD 模块，请在此声明该全局变量。
 *~ 否则，删除此声明。
 */
export as namespace myLib;

/*~ 如果此模块导出函数，请像这样声明它们。 */
export function myFunction(a: string): string;
export function myOtherFunction(a: number): number;

/*~ 你可以声明通过导入模块可用的类型 */
export interface SomeType {
  name: string;
  length: number;
  extras?: string[];
}
```

```
/*~ 你可以使用 const、let 或 var 声明模块的属性 */
export const myField: number;
```

如果你的模块可以像函数一样被调用，使用模板 module-function.d.ts：

```
const x = require("foo");
// 注意: 调用 'x' 作为函数
const y = x(42);
```

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
// Definitions by: [~你的名字~] <[~你的URL~]>

/*~ 这是函数模块的模块模板文件。
 *~ 你应该把它重命名为 index.d.ts 并把它放在一个与模块同名的文件夹中。
 *~ 例如，如果你正在为 "super-greeter" 写一个文件，那么这个
 *~ 文件应该是 'super-greeter/index.d.ts'
 */

// 注意，ES6模块不能直接导出类对象。
// 这个文件应该使用CommonJS风格导入：
//   import x = require('[~THE MODULE~]');
//
// 或者，如果打开了 --allowSyntheticDefaultImports 或
// --esModuleInterop，这个文件也可以作为默认导入：
//   import x from '[~THE MODULE~]';
//
// 参考 TypeScript 文档
// https://www.typescriptlang.org/docs/handbook/modules.html#export--and-import--
require
// 以理解 ES6 模块的这个限制的常见解决方法。
/*~ 如果这个模块是一个 UMD 模块，当在模块加载器环境之外加载时，会暴露一个全局变量 'myFuncLib',
 *~ 在这里声明这个全局变量。否则，删除此声明。
 */
export as namespace myFuncLib;
/*~ 这个声明指定函数
 *~ 是从文件导出的对象
 */
export = Greeter;
/*~ 这个示例展示了如何为你的函数提供多个重载 */
declare function Greeter(name: string): Greeter.NamedReturnType;
declare function Greeter(length: number): Greeter.LengthReturnType;
/*~ 如果你也想从模块中暴露类型，你可以
 *~ 将它们放在这个块中。通常，你会想要描述函数的
 *~ 返回类型的形状；这个类型应该
 *~ 在这里声明，如此示例所示。
```

```
 *~
 *~ 注意，如果你决定包含此命名空间，除非打开了
 *~ --esModuleInterop，否则模块可能会被错误地作为命名空间对象导入：
 *~   import * as x from '[~THE MODULE~]'; // 错误！不要这样做！
 */
declare namespace Greeter {
  export interface LengthReturnType {
    width: number;
    height: number;
  }
  export interface NamedReturnType {
    firstName: string;
    lastName: string;
  }
  /*~ 如果模块也有属性，这里声明它们。例如，
   *~ 这个声明表示这段代码是合法的：
   *~   import f = require('super-greeter');
   *~   console.log(f.defaultName);
   */
  export const defaultName: string;
  export let defaultLength: number;
}
```

如果你的模块可以使用 new 进行构造，使用模板 module-class.d.ts：

```
const x = require("bar");
// 注意：在导入的变量上使用 'new' 运算符
const y = new x("hello");
```

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
// Definitions by: [~你的名字~] <[~你的URL~]>

/*~ 这是类模块的模块模板文件。
 *~ 你应该将其重命名为 index.d.ts，并将其放在一个与模块同名的文件夹中。
 *~ 例如，如果你正在为 "super-greeter" 编写一个文件，那么这
 *~ 文件应该是 'super-greeter/index.d.ts'
 */

// 注意，ES6模块不能直接导出类对象。
// 这个文件应该使用CommonJS风格导入：
//   import x = require('[~THE MODULE~]');
//
// 或者，如果打开了 --allowSyntheticDefaultImports 或
// --esModuleInterop，这个文件也可以作为默认导入：
//   import x from '[~THE MODULE~]';
```

```
//
// 参考 TypeScript 文档
// https://www.typescriptlang.org/docs/handbook/modules.html#export--and-import--
require
// 以理解 ES6 模块的这个限制的常见解决方法。
/*~ 如果这个模块是一个 UMD 模块，当在模块加载器环境之外加载时，会暴露一个全局变量
'myClassLib',
 *~ 在这里声明这个全局变量。否则，删除此声明。
 */
export as namespace "super-greeter";
/*~ 这个声明指定类的构造函数
 *~ 是从文件导出的对象
 */
export = Greeter;
/*~ 在这个类中编写你的模块的方法和属性 */
declare class Greeter {
  constructor(customGreeting?: string);
  greet: void;
  myMethod(opts: MyClass.MyClassMethodOptions): number;
}
/*~ 如果你也想从模块中暴露类型，你可以
 *~ 将它们放在这个块中。
 *~
 *~ 注意，如果你决定包含此命名空间，除非打开了
 *~ --esModuleInterop，否则模块可能会被错误地作为命名空间对象导入：
 *~   import * as x from '[~THE MODULE~]'; // 错误！不要这样做！
 */
declare namespace MyClass {
  export interface MyClassMethodOptions {
    width?: number;
    height?: number;
  }
}
```

如果你有一个模块，当导入时，对其他模块进行更改，使用模板 module-plugin.d.ts：

```
import { greeter } from "super-greeter";

greeter(2);
greeter("Hello world");

import "hyper-super-greeter";
greeter.hyperGreet();
```

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
```

```
// Definitions by: [~你的名字~] <[~你的URL~]>

/*~ 这是模块插件的模板文件。你应该将其重命名为 index.d.ts
 *~ 并将其放在一个与模块名称相同的文件夹中。
 *~ 例如，如果你正在为 "super-greeter" 编写一个文件，那么这个
 *~ 文件应该是 'super-greeter/index.d.ts'
 */

/*~ 在这行中，导入这个模块需要添加的模块 */
import { greeter } from "super-greeter";
/*~ 在这里，声明你上面导入的同一个模块
 *~ 然后我们扩展现有的 greeter 函数的声明
 */
export module "super-greeter" {
  export interface GreeterFunction {
    /** 更好的问候！ */
    hyperGreet(): void;
  }
}
```

全局库

全局库是可以从全局范围（即不使用任何形式的import）访问的库。许多库简单地暴露一个或多个全局变量供使用。例如，如果你使用jQuery，$变量可以通过简单地引用它来使用：

```
$(() => {
  console.log("hello!");
});
```

通常，全局库的文档中会有指南，说明如何在HTML script标签中使用库：

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

现在，大多数流行的全局可访问库实际上是以UMD库的形式编写的（见下文）。UMD库的文档与全局库的文档难以区分。在编写全局声明文件之前，确保库实际上不是UMD。

从代码中识别全局库

全局库的代码通常非常简单。全局的“Hello, world”库可能看起来像这样：

```
function createGreeting(s) {
  return "Hello, " + s;
}
```

或像这样：

```
// Web
window.createGreeting = function (s) {
  return "Hello, " + s;
};
// Node
global.createGreeting = function (s) {
  return "Hello, " + s;
};
// Potentially any runtime
globalThis.createGreeting = function (s) {
  return "Hello, " + s;
};
```

当查看全局库的代码时，你通常会看到：

- 顶级的 var 语句或函数声明
- 对 window.someName 的一次或多次赋值
- 假设 DOM 原始对象，如 document 或 window 存在

你不会看到：

- 检查或使用模块加载器，如 require 或 define
- 形式为 var fs = require("fs"); 的 CommonJS/Node.js样式导入
- 对 define(...) 的调用
- 描述如何 require 或 import 库的文档

全局库的示例

因为通常很容易将全局库转换为UMD库，所以很少有流行的库仍然以全局样式编写。然而，那些小型且需要DOM（或没有依赖项）的库可能仍然是全局的。

全局库模板

模板文件 global.d.ts 定义了一个示例库 myLib：

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
// Definitions by: [~你的名字~] <[~你的URL~]>

/*~ 如果这个库是可调用的（例如，可以被调用为 myLib(3)），
 *~ 在这里包含那些调用签名。
 *~ 否则，删除这一部分。
 */
declare function myLib(a: string): string;
declare function myLib(a: number): number;

/*~ 如果你希望这个库的名称是一个有效的类型名称，你可以在这里进行设置。
 *~
 *~ 例如，这使我们可以写成 'var x: myLib';
```

```
 *~ 确保这实际上是有意义的！如果不是，就删除这个声明，并在下面的命名空间中添加类型。
 */
interface myLib {
  name: string;
  length: number;
  extras?: string[];
}

/*~ 如果你的库在全局变量上公开了属性，将它们放在这里。
 *~ 你也应该在这里放置类型（接口和类型别名）。
 */
declare namespace myLib {
  //~ 我们可以写 'myLib.timeout = 50;'
  let timeout: number;
  //~ 我们可以访问 'myLib.version'，但不能改变它
  const version: string;
  //~ 有一些类我们可以通过 'let c = new myLib.Cat(42)' 创建
  //~ 或者引用，例如 'function f(c: myLib.Cat) { ... }
  class Cat {
    constructor(n: number);
    //~ 我们可以从 'Cat' 实例中读取 'c.age'
    readonly age: number;
    //~ 我们可以调用 'c.purr()' 从 'Cat' 实例
    purr(): void;
  }
  //~ 我们可以声明一个变量
  //~   'var s: myLib.CatSettings = { weight: 5, name: "Maru" };'
  interface CatSettings {
    weight: number;
    name: string;
    tailLength?: number;
  }

  //~ 我们可以写 'const v: myLib.VetID = 42;'
  //~  或者 'const v: myLib.VetID = "bob";'
  type VetID = string | number;
  //~ 我们可以调用 'myLib.checkCat(c)' 或 'myLib.checkCat(c, v);'
  function checkCat(c: Cat, s?: VetID);
}
```

UMD

UMD模块是既可以作为模块（通过导入）使用，也可以作为全局变量（在没有模块加载器的环境中运行时）使用的模块。许多流行的库，如 Moment.js，就是这样写的。例如，在 Node.js 或使用 RequireJS 中，你会写：

```
import moment = require("moment");
console.log(moment.format());
```

而在一个原生浏览器环境中，你会写：

```
console.log(moment.format());
```

识别一个UMD库

UMD模块会检查是否存在模块加载器环境。这是一个容易识别的模式，看起来像这样：

```
(function (root, factory) {
    if (typeof define === "function" && define.amd) {
        define(["libName"], factory);
    } else if (typeof module === "object" && module.exports) {
        module.exports = factory(require("libName"));
    } else {
        root.returnExports = factory(root.libName);
    }
}(this, function (b) {
```

如果你在库的代码中看到对 typeof define、typeof window 或 typeof module 的测试，特别是在文件的顶部，那么它几乎总是一个UMD库。

UMD库的文档也经常会演示一个“在 Node.js 中使用”示例，展示 require，以及一个“在浏览器中使用”的示例，展示使用 <script> 标签来加载脚本。

UMD库的示例

现在，大多数流行的库都可以作为UMD包使用。示例包括 jQuery、Moment.js、lodash 等等。

模板

使用 module-plugin.d.ts 模板：

```
// 类型定义 [~库的名称~] [~可选的版本号~]
// Project: [~项目名称~]
// Definitions by: [~你的名字~] <[~你的URL~]>
/*~ 这是模块插件的模板文件。你应该将其重命名为 index.d.ts
 *~ 并将其放在与模块同名的文件夹中。
 *~ 例如，如果你正在为 "super-greeter" 编写一个文件，那么
 *~ 这个文件应该是 'super-greeter/index.d.ts'
 */

/*~ 在这一行，导入此模块要添加到的模块 */
```

```ts
import { greeter } from "super-greeter";

/*~ 在这里，声明和你上面导入的模块同名的模块
 *~ 然后我们扩展 greeter 函数的现有声明
 */
export module "super-greeter" {
  export interface GreeterFunction {
    /** 更好的问候！ */
    hyperGreet(): void;
  }
}
```

在这个示例中，我们扩展了 `super-greeter` 模块的 `GreeterFunction` 接口，为其添加了一个新的方法 `hyperGreet`。注意，我们是在一个使用 `export module "super-greeter"` 声明的模块中进行扩展的，这是 TypeScript 允许我们修改现有模块的方式。

引用依赖

你的库可能有几种类型的依赖。本节展示了如何将它们导入到声明文件中。

对全局库的依赖

如果你的库依赖于全局库，使用 /// 指令：

```ts
/// <reference types="someLib" />
function getThing(): someLib.thing;
```

对模块的依赖

如果你的库依赖于模块，使用 import 语句：

```ts
import * as moment from "moment";
function getThing(): moment;
```

对UMD库的依赖

从全局库

如果你的全局库依赖于UMD模块，使用 /// <reference types 指令：

```ts
/// <reference types="moment" />
function getThing(): moment;
```

从模块或UMD库

如果你的模块或UMD库依赖于UMD库，使用 import 语句：

```
import * as someLib from "someLib";
```

不要使用 /// <reference 指令来声明对UMD库的依赖！