阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 乘风者计划 THE WIND RIDERS PROGRAM

现代TypeScript高级课程

帮助掌握试用TypeScript构建可扩展应用程序的技巧

LinWu

乘风者计划专家博主

卷首语

可能是市面上比较好的 Typescript 高级教程，适合有一定 Typescipt 基础的同学学习。

随着 TypeScript 的日益普及，它已经成为现代 Web 开发的重要工具。然而，尽管 TypeScript 初学者可以轻松上手并开始编写代码，但深入理解 TypeScript 的强大功能和高级特性却是一项更具挑战性的任务。"现代 TypeScript 高级教程"就是为了帮助您解开 TypeScript 的高级秘密而编写的。

在这本教程中，我们将深入探讨 TypeScript 的复杂特性，包括装饰器、泛型、高级类型以及元数据反射等。我们不仅会详细解释这些概念，还会展示如何在实际项目中运用这些高级特性，提供丰富的代码示例和最佳实践，帮助您更好地理解这些复杂的概念。

这本教程适合有一定 TypeScript 基础，希望进一步提升技能的开发者。每一章都设计得既可以独立阅读，也可以作为整个教程的一部分。我们深信，无论您是希望对 TypeScript 有更深入的了解，还是希望提升在大型项目中使用 TypeScript 的技巧，本教程都将为您提供极大的帮助。

关于我

笔名 linwu,一枚前端开发工程师，曾入职腾讯等多家知名互联网公司，后面我会持续分享精品课程，欢迎持续关注

获取最新版源码和笔记，请访问

www.coding-time.cn

阿里云开发者“藏经阁”

海量电子手册免费下载

目录页

➢ 基础

一、概述

引言

在 TypeScript 的发展过程中，对类型系统的持续改进一直是其核心任务。这在 2.0 版本中引入的严格的空值检查（--strictNullChecks）中体现得尤为明显，这个功能帮助开发者在编译时捕获可能的 null 或 undefined 引用错误。

TypeScript 2.1 带来了映射类型，这是一种创建新类型的方式，基于旧类型转换其属性。2.8 版本则引入了有条件的类型，使得类型系统具备了更多的表达力。

TypeScript 3.0 引入了项目引用，这是一种新的架构工具，允许大型项目更容易地组织代码和依赖项。3.7 版本中，TypeScript 支持了可选链和空值合并运算符，这是两个常用的 JavaScript 特性。

在最新的 TypeScript 版本中，提供了更丰富的语法特性和工具支持，比如更强大的类型推导，更精确的类型检查，以及更完善的 IDE 支持。

优势

TypeScript 的优势还包括它的可互操作性。由于 TypeScript 是 JavaScript 的超集，所以开发者可以轻松地将 JavaScript 代码迁移到 TypeScript。同时，开发者还可以使用来自 JavaScript 生态系统的库和工具。TypeScript 还支持最新的 ECMAScript 特性，如箭头函数、模块、解构等。

TypeScript 也为大型项目提供了必要的工具。TypeScript 的类型系统使得开发者可以明确定义对象和函数的结构，这样在大型项目中维护和理解代码就更加简单。此外，TypeScript 还有良好的工具支持，比如 TSLint 和 Prettier，这些工具可以帮助开发者编

写更一致、更可读的代码。

在性能方面，由于 TypeScript 在运行前进行编译，因此可以提前发现并修复很多可能在运行时才会出现的错误。这种预编译的方式可以大大提高应用程序的性能，因为运行时需要进行的工作量较少。

TypeScript 的类型定义文件（.d.ts）是一个独特的优点，它们为已有的 JavaScript 库提供类型信息。这使得开发者可以在使用这些库的同时享受到类型检查的好处。而且，由于有大量的开源贡献者，绝大多数流行的 JavaScript 库都有相应的类型定义文件。

总的来说，TypeScript 结合了 JavaScript 的灵活性和静态类型语言的安全性，使得它成为了现代 Web 开发的重要工具。

二、类型

TypeScript 提供了 JavaScript 的所有基本数据类型，如：***number***、***string***、***boolean*** 等。它还增加了额外的类型，比如 ***any***、***unknown***、***never***、***void*** 等。

1. number

在 TypeScript 中，所有的数字都是浮点数。这些数字的类型是 ***number***。下面是一些例子：

```
let decimal: number = 6; // 十进制
let hex: number = 0xf00d; // 十六进制
let binary: number = 0b1010; // 二进制
let octal: number = 0o744; // 八进制
```

2. string

string 类型表示文本数据。你可以使用单引号（'）或双引号（"）定义字符串，也可以使用反引号（`）定义模板字符串：

```
let color: string = "blue";
color = 'red';
let fullName: string = `Bob Bobbington`;let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }. I'll be ${ age
+ 1 } years old next month.`;
```

3. boolean

boolean 类型有两个值：***true*** 和 ***false***：

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

4. Array

在 TypeScript 中，数组类型有两种表达方式。一种是在元素类型后面加上 ***[]***，表示由此类型元素组成的一个数组。另一种方式是使用数组泛型，***Array***<元素类型>：

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

5. Tuple

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

```
let x: [string, number];
x = ['hello', 10]; // OK
```

以上是 TypeScript 的一些基本类型。在接下来的对话中，我们可以进一步讨论其他的 TypeScript 类型，比如枚举（***enum***）、***null***、***undefined***、***never***、***void*** 以及对象类型。

6. Enum

Enum 是一种特殊的类型，它可以更容易地处理一组有名字的常量。在 TypeScript 中，enum 的默认起始值是 0，然后每个成员的值都会依次增加。你也可以手动为 enum 的成员指定值：

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
// 手动指定成员的数值
enum Color {Red = 1, Green = 2, Blue = 4}let c: Color = Color.Green;
```

7. Null and Undefined

在 TypeScript 中，***undefined*** 和 ***null*** 各自有自己的类型，分别是 ***undefined*** 和 ***null***。默认情况下，它们是所有类型的子类型。这意味着你可以把 ***null*** 和 ***undefined*** 赋值给 ***number*** 类型的变量。

然而，当你指定了 ***--strictNullChecks*** 标记，***null*** 和 ***undefined*** 只能赋值给 ***void*** 和它们各自的类型：

```
let u: undefined = undefined;
let n: null = null;
```

8. Any

当你不确定一个变量应该是什么类型的时候，你可能需要用到 ***any*** 类型。***any*** 类型的变量允许你对它进行任何操作，它就像是 TypeScript 类型系统的一个逃生窗口：

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

9. Unknown

unknown 类型是 TypeScript 3.0 中引入的一种新类型，它是 any 类型对应的安全类型。***unknown*** 类型的变量只能被赋值给 ***any*** 类型和 ***unknown*** 类型本身：

```
let value: unknown;

value = true;             // OK
value = 42;               // OK
value = "Hello World";    // OK
value = [];               // OK
value = {};               // OK
```

10. Never

never 类型表示的是那些永不存在的值的类型。例如，***never*** 类型是那些总是会抛出异常或者根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型：

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

11. void

在 TypeScript 中，***void*** 类型用于表示没有返回值的函数的返回类型。在 JavaScript 中，当一个函数没有返回任何值时，它会隐式地返回 ***undefined***。***void*** 类型就是用来表达这种情况的：

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

在这个例子中，***warnUser*** 函数没有返回任何值，所以它的返回类型是 ***void***。

需要注意的是，在 TypeScript 中，你可以声明一个 ***void*** 类型的变量，但是你只能为它赋予 ***undefined*** 和 ***null***（在非严格 null 检查模式下）：

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

通常情况下，我们不会这样使用 ***void*** 类型，因为除了 ***undefined*** 和 ***null*** 之外，你不能将任何值赋给 ***void*** 类型的变量。

12. 联合类型（Union Types）

TypeScript 联合类型是一种将多种类型组合到一起的方式，表示一个值可以是多种类型之一。你可以使用管道符（|）来分隔每个类型。这可以让你在不确定一个值是什么类型的时候，为它选择多个可能的类型。

例如，假设我们有一个函数，这个函数的参数可以是 ***number*** 类型或者 ***string*** 类型：

```
function display(input: string | number) {
   console.log(input);}
```

在上面的函数中，我们声明了 ***input*** 参数可以是 ***string*** 类型或者 ***number*** 类型。你可以传递一个 ***string*** 类型或者 ***number*** 类型的值给 ***display*** 函数，而 TypeScript 编译器不会报错：

```
display(1); // OK
display("Hello"); // OK
```

你也可以将联合类型用于变量和属性。例如：

```
let variable: string | number;

variable = "Hello"; // OK
variable = 1; // OK
```

在上面的代码中，我们声明了 ***variable*** 可以是 ***string*** 类型或者 ***number*** 类型。然后我们可以安全地将一个字符串或者数字赋值给 ***variable***。

联合类型在 TypeScript 中非常常用，因为它们可以帮助你编写更灵活的代码。

13. 交叉类型（Intersection Types）

交叉类型是将多个类型合并为一个类型。这让我们可以把现有的多种类型叠加到一起获得所需的能力。它被定义为 ***Type1 & Type2 & Type3 & ... & TypeN***。它包含了所需的所有类型的成员。

```
interface Part1 {
  a: string;}
interface Part2 {
  b: number;}
```

```
type Combined = Part1 & Part2;
let obj: Combined = {
  a: 'hello',
  b: 42,};
```

14. 类型别名（Type Aliases）

类型别名是给一个类型起个新名字。类型别名有时和接口很相似，但可以作用于原始值，联合类型，交叉类型等任何我们需要手写的地方。

```
type MyBool = true | false;
type StringOrNumber = string | number;
```

15. 字符串字面量类型（String Literal Types）

字符串字面量类型允许你指定字符串必须的固定值。在实践中，这种类型常与联合类型、类型别名和类型保护结合使用。

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        // ...
    }}
let button = new UIElement();
button.animate(0, 0, "ease-in"); // OK
button.animate(0, 0, "uneasy");  // Error: "uneasy" is not allowed here
```

三、函数

TypeScript 提供了丰富的函数类型定义方式，可以对函数参数、返回值进行类型注解，从而提供了更为强大的类型检查。

1. 函数声明

在 TypeScript 中，你可以在函数声明中对函数的参数和返回值进行类型注解。以下是一个例子：

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

在这个例子中，我们定义了一个 ***add*** 函数，它接受两个参数 ***x*** 和 ***y***，这两个参数都是 ***number*** 类型，函数的返回值也是 ***number*** 类型。

如果你尝试调用这个函数并传入一个非数字类型的参数，TypeScript 编译器会报错：

```
add("Hello", 1); // Error: Argument of type '"Hello"' is not assignable
to parameter of type 'number'.
```

2. 函数表达式

在 JavaScript 中，函数不仅可以通过函数声明的方式定义，还可以通过函数表达式定义。在 TypeScript 中，函数表达式也可以使用类型注解：

```
let myAdd: (x: number, y: number) => number = function(x: number, y:
number): number {
    return x + y;
    };
```

在上面的例子中，我们首先定义了 ***myAdd*** 变量的类型为一个函数类型 ***(x: number, y:***

number) => number，然后将一个匿名函数赋值给 ***myAdd***。这个匿名函数的参数 ***x*** 和 ***y*** 的类型是 ***number***，返回值的类型也是 ***number***。

3. 可选参数和默认参数

TypeScript 支持可选参数和默认参数。你可以使用 ***?*** 来标记可选参数，或者使用 ***=*** 来指定参数的默认值：

```
   function buildName(firstName: string, lastName?: string) {
       if (lastName)
           return firstName + " " + lastName;
       else
           return firstName;}
   let result1 = buildName("Bob");  // works correctly nowlet result2
= buildName("Bob", "Adams");  // ah, just right
```

在上面的例子中，***lastName*** 是一个可选参数。你可以不传这个参数调用 ***buildName*** 函数。

```
   function buildName(firstName: string, lastName = "Smith") {
       return firstName + " " + lastName;}
   let result1 = buildName("Bob");  // returns "Bob Smith"let result2
= buildName("Bob", "Adams");  // returns "Bob Adams"
```

在上面的例子中，***lastName*** 有一个默认值 "***Smith***"。如果你不传这个参数调用 ***buildName*** 函数，***lastName*** 的值将是 "***Smith***"。

4. 剩余参数（Rest Parameters）

当你不知道要操作的函数会有多少个参数时，TypeScript 提供了剩余参数的概念。与 JavaScript 一样，你可以使用三个点 ***...*** 定义剩余参数：

```
    function buildName(firstName: string, ...restOfName: string[]) {
        return firstName + " " + restOfName.join(" ");
    }
    let employeeName = buildName("Joseph", "Samuel", "Lucas",
"MacKinzie");
```

在上面的例子中，***restOfName*** 就是剩余参数，它可以接受任意数量的参数。

5. this 和箭头函数

箭头函数可以保留函数创建时的 ***this*** 值，而不是调用时的值。在 ***TypeScript*** 中，你可以使用箭头函数来确保 ***this*** 的值：

```
    let deck = {
        suits: ["hearts", "spades", "clubs", "diamonds"],
        cards: Array(52),
        createCardPicker: function() {
            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);
```

在上面的例子中，***createCardPicker*** 函数返回一个箭头函数，这个箭头函数可以访问创建时的 ***this*** 值。

6. 重载

在 JavaScript 中，根据传入不同的参数调用同一个函数，返回不同类型的值是常见的情况。TypeScript 通过为同一个函数提供多个函数类型定义来实现这个功能：

```
   function reverse(x: number): number;function reverse(x: string): string;function reverse(x: number | string): number | string {
       if (typeof x === 'number') {
           return Number(x.toString().split('').reverse().join(''));
       } else if (typeof x === 'string') {
           return x.split('').reverse().join('');
       }}
   reverse(12345);   // returns 54321reverse('hello');   // returns 'olleh'
```

在上面的例子中，我们定义了两个重载：一个是接受 ***number*** 类型的参数，另一个是接受 ***string*** 类型的参数。然后我们在实现函数中处理了这两种情况。

四、接口和类

在 TypeScript 中，接口（Interfaces）和类（Classes）是实现面向对象编程（Object-Oriented Programming，OOP）的基础工具。这些工具提供了一种方式来定义和组织复杂的数据结构和行为。

1. 接口

接口在 TypeScript 中扮演着关键的角色，用于强类型系统的支持。接口可以描述对象的形状，使我们可以编写出预期的行为。接口可用于描述对象、函数或者类的公共部分。

以下是一个基本的接口示例：

```
interface LabelledValue {
  label: string;}
function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);}
let myObj = { size: 10, label: "Size 10 Object" };printLabel(myObj);
```

在这个例子中，***LabelledValue*** 接口就像一个名片，告诉其他代码，只要一个对象有 ***label*** 属性，并且类型为 ***string***，那么就可以认为它是 ***LabelledValue***。

接口也可以描述函数类型：

```
interface SearchFunc {
  (source: string, subString: string): boolean;}
let mySearch: SearchFunc;mySearch = function(src: string, sub: string):
boolean {
  let result = src.search(sub);
  return result > -1;}
```

此外，接口还能用于描述数组和索引类型：

```
interface StringArray {
  [index: number]: string;}
let myArray: StringArray;
myArray = ["Bob", "Fred"];
interface Dictionary {
  [index: string]: string;}
let myDict: Dictionary;
myDict = { "key": "value" };
```

2. 类

与传统的 JavaScript 一样，TypeScript 也使用类（Classes）来定义对象的行为。然而，TypeScript 的类具有一些额外的特性，如访问修饰符（Access Modifiers）、静态属性（Static Properties）、抽象类（Abstract Classes）等。

以下是一个基本的类示例：

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }}
let greeter = new Greeter("world");
```

TypeScript 还引入了访问修饰符 ***public***、***private*** 和 ***protected***。如果没有指定访问修饰符，则默认为 ***public***。

```
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }}
```

TypeScript 类还支持继承，通过 ***extends*** 关键字可以创建一个子类。子类可以访问和改变父类的属性和方法：

```
class Animal {
  name: string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }}
class Dog extends Animal {
  constructor(name: string) { super(name); }
  bark() {
    console.log('Wo
of! Woof!');
  }}
const dog = new Dog('Tom');
dog.bark();
dog.move(10);
dog.bark();
```

为了实现多态，TypeScript 提供了抽象类的概念。抽象类是不能被实例化的类，只能作为其他类的基类。抽象类中可以定义抽象方法，抽象方法必须在派生类中实现：

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log('roaming the earth...');
  }}
class Dog extends Animal {
```

```
  makeSound() {
    console.log('Woof! Woof!');
  }}
const myDog = new Dog();
myDog.makeSound();
myDog.move();
```

五、枚举和泛型

接下来我们将学习 TypeScript 中的两个重要主题：枚举（Enums）和泛型（Generics）。这两个特性能大大提高代码的可重用性和安全性。

1. 枚举

枚举是 TypeScript 中一种特殊的数据类型，允许我们为一组数值设定友好的名字。枚举的定义使用 ***enum*** 关键字。

```
enum Direction {
    Up = 1,
    Down,
    Left,
    Right,}
```

在这个例子中，我们定义了一个名为 ***Direction*** 的枚举，它有四个成员：***Up***、***Down***、***Left*** 和 ***Right***。***Up*** 的初始值为 1，其余成员的值会自动递增。

除了使用数值，我们也可以使用字符串：

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",}
```

此外，TypeScript 还支持计算的和常量成员。常量枚举通过 const enum 进行定义，TypeScript 会在编译阶段进行优化：

```
const enum Enum {
    A = 1,
    B = A * 2}
```

2. 异构枚举

TypeScript 支持数字和字符串混用的枚举，这种类型的枚举被称为异构枚举：

```
enum BooleanLikeHeterogeneousEnum {
    No = 0,
    Yes = "YES",}
```

尽管 TypeScript 支持这种用法，但我们在实际项目中应尽可能避免使用异构枚举，因为这会引入不必要的复杂性。

3. 枚举成员的类型

在某些特殊的情况下，枚举成员本身也可以作为一种类型：

```
enum ShapeKind {
    Circle,
    Square,}
interface Circle {
    kind: ShapeKind.Circle;
    radius: number;}
interface Square {
    kind: ShapeKind.Square;
    sideLength: number;}
```

4. 泛型

在 TypeScript 中，泛型（Generics）是一种强大的类型工具，它允许我们编写可重用、灵活和类型安全的代码。泛型允许我们在定义函数、类或接口时使用类型参数，这些类型参数在使用时可以被动态地指定具体的类型。

以下是泛型在 TypeScript 中的几个常见应用场景：

1. 函数泛型

函数泛型允许我们编写可适用于多种类型的函数，提高代码的重用性和灵活性。例如：

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

let result = identity<number>(42);  // result 的类型为 number
```

在上面的示例中，***identity***函数接受一个类型参数 T，表示输入和输出的类型。通过在函数调用时显式指定类型参数为 ***number***，我们可以将 ***42***传递给 ***identity***函数并推断出结果的类型为 ***number***。

2. 接口泛型

接口泛型允许我们创建可适用于不同类型的接口定义。例如：

```
interface Pair<T, U> {
  first: T;
  second: U;}
let pair: Pair<number, string> = { first: 42, second: "hello" };
```

在上面的示例中，我们定义了一个 ***Pair***接口，它接受两个类型参数 ***T***和 ***U***，表示 ***first***和 ***second***属性的类型。通过指定类型参数为 ***number***和 ***string***，我们创建了一个具体的 ***pair***对象，它的 ***first***属性类型为 ***number***，***second***属性类型为 ***string***。

3. 类泛型

类泛型允许我们创建可适用于不同类型的类定义。例如：

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

```
  constructor(value: T) {
    this.value = value;
  }

  getValue(): T {
    return this.value;
  }}
let container = new Container<number>(42);let value = container.getValue();  // value 的类型为 number
```

在上面的示例中，我们定义了一个 ***Container*** 类，它接受一个类型参数 ***T***，表示类的内部值的类型。通过在创建类的实例时显式指定类型参数为 ***number***，我们创建了一个具体的 ***container*** 对象，它的 ***value*** 属性类型为 ***number***，并可以使用 ***getValue*** 方法获取该值。

泛型还支持约束（Constraints）的概念，通过使用约束，我们可以限制泛型的类型范围，使其满足特定的条件。

泛型在 TypeScript 中广泛应用于函数、类、接口和类型别名的定义中，它提供了一种灵活、类型安全且可重用的方式来处理不同类型的数据。通过使用泛型，我们可以在编写代码时提供更强大的类型支持，从而减少错误并提高代码的可维护性和可读性。

六、命名空间和模块

1. 命名空间（Namespace）

在 TypeScript 中，命名空间是一种将代码封装在一个特定名称下的方式，以防止全局作用域污染并避免命名冲突。命名空间在 TypeScript 中非常重要，因为它们为模块化和封装提供了灵活的选项。

创建命名空间的语法如下：

```
namespace MyNamespace {
  export const myVar: number = 10;
  export function myFunction(): void {
    console.log("Hello from MyNamespace");
  }}
```

在此例中，我们创建了一个名为 ***MyNamespace*** 的命名空间，该命名空间内有一个变量 ***myVar*** 和一个函数 ***myFunction***。***export*** 关键字允许我们从命名空间外部访问这些元素。

命名空间中的元素可以通过以下方式访问：

```
console.log(MyNamespace.myVar); // 输出: 10
MyNamespace.myFunction(); // 输出: Hello from MyNamespace
```

我们也可以使用嵌套的命名空间：

```
namespace ParentNamespace {
  export namespace ChildNamespace {
    export const myVar: number = 20;
  }}console.log(ParentNamespace.ChildNamespace.myVar); // 输出: 20
```

2. 命名空间（Namespace）使用场景

在 TypeScript 的早期版本中，命名空间被广泛地使用来组织和包装一组相关的代码。然而，随着 ES6 模块系统（ES6 Modules）的出现和广泛使用，命名空间的用法变得越来越少，现在被视为一种遗留的方式来组织代码。

1）第三方库

一些第三方库仍然使用命名空间来组织自己的代码，并提供命名空间作为库的入口点。在这种情况下，我们需要使用命名空间来访问和使用库中的类型和函数。

```
namespace MyLibrary {
  export function myFunction() {
    // ...
  }}

MyLibrary.myFunction();
```

2）兼容性

在一些遗留的 JavaScript 代码或库中，命名空间仍然是一种常见的组织代码的方式。如果我们需要与这些代码进行交互，我们可能需要创建命名空间来适应它们。

```
// legacy.jsvar MyNamespace = {
  myFunction: function() {
    // ...
  }};

MyNamespace.myFunction();
```

在上面的示例中，我们演示了命名空间的几个使用场景。第一个示例展示了如何使用命名空间访问和使用第三方库的函数。第二个示例展示了如何使用命名空间来管理全局状态。第三个示例展示了如何在与遗留 JavaScript 代码进行交互时创建命名空间。

虽然在现代 TypeScript 开发中，模块是更常见和推荐的代码组织方式，但命名空间仍然在特定的情况下具有一定的用处，并且在与一些特定的库或代码进行交互时可能是必需的。

3. 模块

在 TypeScript 中，模块是另一种组织代码的方式，但它们更关注的是依赖管理。每个模块都有其自己的作用域，并且只有明确地导出的部分才能在其他模块中访问。

创建和使用模块的方式如下：

在 ***myModule***.ts 文件中：

```
export const myVar: number = 10;export function myFunction(): void {
  console.log("Hello from myModule");}
```

在另一个文件中导入和使用模块：

```
import { myVar, myFunction } from './myModule';
console.log(myVar); // 输出：10myFunction(); // 输出：Hello from myModule
```

在 TypeScript 中，我们可以使用模块解析策略（如 Node 或 Classic），以确定如何查找模块。这些策略在 ***tsconfig.json*** 文件的 ***compilerOptions*** 选项下的 ***moduleResolution*** 选项中定义。

命名空间与模块的对比

虽然命名空间和模块在某种程度上有所相似，但它们有以下几个关键区别：

- **作用域**：命名空间是在全局作用域中定义的，而模块则在自己的作用域中定义。这意味着，在模块内部定义的所有内容默认情况下在模块外部是不可见的，除非显式地导出它们。

- **文件组织**：命名空间通常用于组织在同一文件中的代码，而模块则是跨文件进行组织。
- **依赖管理**：模块关注的是如何导入和导出功能，以便管理代码之间的依赖关系。相比之下，命名空间并未对依赖管理提供明确的支持。
- **使用场景**：随着 ES6 模块语法的普及，现代 JavaScript 项目通常更倾向于使用模块来组织代码。然而，对于一些遗留项目或那些需要将多个文件合并为一个全局可用的库的场景，命名空间可能更为合适。

➢ 进阶

七、类型系统层级

TypeScript 的类型系统是强类型和静态类型的，这为开发者提供了强大的类型检查和类型安全保障，同时也增加了一定的学习复杂性。

为了更好地理解 TypeScript 的类型系统，本文将全面介绍其类型系统层级，包括顶层类型（Top Type）和底层类型（Bottom Type），以及在这个层次结构中如何处理和操作各种类型。理解 TypeScript 的类型系统层级有助于我们更好地使用和掌握 TypeScript，写出更健壮、可维护的代码。

1. 顶层类型（Top Type）

顶层类型是所有其他类型的父类型，这意味着在 TypeScript 中的任何类型都可以看作是顶层类型的子类型。TypeScript 中有两个特殊的顶层类型：***any*** 和 ***unknown***。

1） any 类型

any 类型是 TypeScript 的一个逃生窗口，它可以接受任意类型的值，并且对 ***any*** 类型的值进行的任何操作都是允许的。使用 ***any*** 类型，可以使我们绕过 TypeScript 的类型检查。下面的例子展示了 ***any*** 类型的灵活性：

```
let a: any = 123;  // OK
a = 'hello';  // OK
a = true;  // OK
a = { id: 1, name: 'Tom' }; // OK

a.foo();  // OK
```

我们可以看到，我们可以将任何类型的值赋给 ***any*** 类型的变量，甚至可以对 ***any*** 类型的值进行我们想要的任何操作，而 TypeScript 编译器并不会对此做出任何投诉。

然而，正是由于其超高的灵活性，使得 ***any*** 类型在一定程度上削弱了 TypeScript 的类型安全性，因此在我们编写 TypeScript 代码时，应尽量避免使用 ***any*** 类型。

2） unknown 类型

unknown 类型与 ***any*** 类型在接受任何类型的值这一点上是一样的，但 ***unknown*** 类型却不能像 ***any*** 类型那样对其进行任何操作。我们在对 ***unknown*** 类型的值进行操作之前，必须进行类型检查或类型断言，确保操作的安全性。

下面的例子展示了 unknown 类型的使用：

```
let u: unknown = 123;  // OK
u = 'hello';  // OK
u = true;  // OK
u = { id: 1, name: 'Tom' }; // OK
// Error: Object is of type 'unknown'.// u.foo();
if (typeof u === 'object' && u !== null) {
  // OK after type check
  console.log((u as { id: number,

 name: string }).name);}
```

在这个例子中，我们对 ***unknown*** 类型的值 ***u*** 进行了类型检查，然后通过类型断言安全地访问了其 ***name*** 属性。

2. 底层类型（Bottom Type）

与顶层类型相对，底层类型是所有类型的子类型。这意味着，在类型系统的层次结构中，任何类型都可以被看作是底层类型的超类型。在 TypeScript 中，***never*** 类型是唯一的底层类型。

never 类型用来表示永远不可能存在的值的类型。比如，一个永远抛出错误或者永远处于死循环的函数的返回类型就是 ***never***。

```
function error(message: string): never {
  throw new Error(message);}
function infiniteLoop(): never {
  while (true) {}}
```

在上面的代码中，函数 ***error*** 和 ***infiniteLoop*** 的返回类型都是 ***never***，这是因为这两个函数都永远不会有返回值。

3. 对比：顶层类型 vs 底层类型

顶层类型和底层类型是 TypeScript 类型系统的两个重要组成部分，它们各自扮演着不同的角色。

顶层类型 ***any*** 和 ***unknown*** 能够接受任何类型的值，这使得我们可以灵活地处理不确定类型的数据。然而，***any*** 类型和 ***unknown*** 类型在使用上有着重要的区别：***any*** 类型允许我们对其进行任何操作，而 ***unknown*** 类型则要求我们在操作之前进行类型检查或类型断言，以确保类型的安全性。

底层类型 ***never*** 有点特殊，它表示一个永远不会有值的类型。在实际开发中，我们可能很少直接使用 ***never*** 类型，但是它在 TypeScript 的类型推断和控制流分析中起着非常重要的作用。

理解 TypeScript 的类型系统层级有助于我们编写更健壮、可维护的 TypeScript 代码。尽管 ***any*** 类型提供了很大的灵活性，但是它的滥用可能会削弱 TypeScript 的类型安全性。因此，我们应尽量避免使用 ***any*** 类型，而优先使用 ***unknown*** 类型和类型断言、类型保护等方式来处理不确定类型的数据。同时，虽然我们可能很少直接使用 ***never*** 类型，但是理解它的含义和用法，对于我们理解 TypeScript 的类型推断和控制流分析也是非常有帮助的。

八、高级类型

1. 映射类型（Mapped Types）

映射类型（Mapped Types）是 TypeScript 中一种强大的类型操作工具，它允许我们在编译时转换已知类型的属性，并创建一个新的类型。通过映射类型，我们可以对已有类型的属性进行转换、修改或添加新的属性。这在许多情况下都非常有用，例如将属性变为只读或可选，从现有属性中选择一部分属性等。

映射类型的语法形式为：

```
type NewType = {
  [Property in keyof ExistingType]: TransformType;};
```

其中，***NewType*** 是我们要创建的新类型，***Property*** 是 ***ExistingType*** 的键，***TransformType*** 是对应属性的转换类型。

下面是一些常见的映射类型的示例：

1） Readonly

Readonly 是 TypeScript 内置的一个映射类型，它将给定类型的所有属性变为只读。

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

示例使用：

```
interface Person {
  name: string;
  age: number;}
type ReadonlyPerson = Readonly<Person>;
```

```
const person: ReadonlyPerson = {
  name: "John",
  age: 30,};

person.name = "Alice";  // Error: Cannot assign to 'name' because it is
a read-only property.
```

2） Partial

Partial是另一个内置的映射类型，它将给定类型的所有属性变为可选。

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

示例使用：

```
interface Person {
  name: string;
  age: number;}
type PartialPerson = Partial<Person>;
const person: PartialPerson = {
  name: "John",};

person.age = 30;  // Valid: age is optional
```

3） Pick

Pick是一个映射类型，它从给定类型中选择一部分属性来创建新类型。

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];};
```

示例使用：

```
interface Person {
  name: string;
  age: number;
  occupation: string;}
type PersonInfo = Pick<Person, "name" | "age">;
const info: PersonInfo = {
  name: "John",
  age: 30,};
```

4） Record

Record 是一个映射类型，它根据指定的键类型和值类型创建一个新的对象类型。

```
type Record<K extends keyof any, T> = {
  [P in K]: T;};
```

示例使用：

```
type Weekday = "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday";
type WorkingHours = Record<Weekday, string>;
const hours: WorkingHours = {
  Monday: "9am-6pm",
  Tuesday: "9am-6pm",
  Wednesday: "9am-6pm",
  Thursday: "9am-6pm",
  Friday: "9am-5pm",};
```

2. 条件类型（Conditional Types）

它允许我们根据类型的条件判断结果来选择不同的类型。条件类型的语法形式为：

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

其中，T 是待检查的类型，U 是条件类型，X 是满足条件时返回的类型，Y 是不满足条件时返回的类型。条件类型通常与泛型一起使用，以便根据不同的类型参数值进行类型推断和转换。

1）条件类型与 infer

当我们在 TypeScript 中使用条件类型时，有时候我们希望从某个类型中提取出一个部分类型并进行推断。这时就可以使用 ***infer*** 关键字。

infer 关键字用于声明一个类型变量，在条件类型中表示待推断的部分类型。它通常在条件类型的分支中使用，以便从给定类型中提取和推断出某些信息。

下面是一个示例，展示了如何使用 ***infer*** 关键字：

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
function add(a: number, b: number): number {
  return a + b;}
type AddReturnValue = ReturnType<typeof add>;  // 推断为 number 类型
```

在上面的示例中，我们定义了一个条件类型 ***ReturnType<T>***，它接受一个函数类型 ***T*** 作为输入。当 ***T*** 是一个函数类型时，我们使用 ***infer R*** 声明一个类型变量 ***R*** 来推断函数的返回类型，并将其作为结果返回。

通过调用 ***ReturnType<typeof add>***，我们将函数 ***add*** 的类型传递给 ***ReturnType<T>***，从而提取并推断出其返回类型。结果 ***AddReturnValue*** 的类型被推断为 ***number***，因为 ***add*** 函数返回一个数字。

infer 关键字的作用是告诉 TypeScript 编译器在条件类型中推断一个待定的类型，并将其赋值给声明的类型变量。这使得我们可以在条件类型中使用这个推断出的类型进行

进一步的类型操作。

需要注意的是，***infer***关键字只能在条件类型的右侧使用，用于声明一个待推断的类型变量，而不能在其他地方使用。此外，每个条件类型只能使用一次 ***infer***关键字，并且通常与泛型一起使用。

infer关键字是 TypeScript 中用于提取并推断待定类型的工具。它允许我们在条件类型中声明一个类型变量，用于在类型推断过程中捕获和使用待推断的类型，从而使类型系统更加灵活和强大。

3. 模板字面量类型（Template Literal Types）

模板字面量类型（Template Literal Types）是 TypeScript 4.1 引入的新特性，它允许我们在类型级别上操作字符串字面量类型。通过使用模板字面量类型，我们可以创建基于字符串模板的复杂类型。

下面是一个使用模板字面量类型的示例：

```
type Greeting<T extends string> = `Hello, ${T}!`;
type GreetingWorld = Greeting<'World'>;  // GreetingWorld 的类型为"Hello, World!"
```

在上面的示例中，我们定义了一个模板字面量类型 ***Greeting<T>***，它接受一个字符串类型参数 ***T***，并使用字符串模板将其包装在 ***Hello,***和***!***之间。通过使用 ***Greeting<'World'>***，我们可以将字符串字面量类型 ***'World'***传递给模板字面量类型，从而创建一个具体的类型 ***GreetingWorld***，它的类型被推断为"***Hello, World!***"。

模板字面量类型还支持模板字符串的拼接、条件语句、循环等操作，使得我们可以在类型级别上创建更加动态和复杂的类型。

```
type Pluralize<T extends string> = `${T}s`;type Message<T extends boolean> = T extends true ? 'Enabled' : 'Disabled';
type Plural = Pluralize<'apple'>;  // Plural 的类型为"apples"type EnabledMessage = Message<true>;  // EnabledMessage的类型为'Enabled'
```

在上面的示例中，我们定义了两个模板字面量类型，Pluralize<T>用于将字符串类型 T 转换为其复数形式，Message<T>用于根据布尔类型参数 T 返回不同的消息。

九、类型推断

TypeScript 通过类型推断可以自动推导出变量和表达式的类型，提高代码的可读性和可维护性。

1. 类型推断

类型推断是 TypeScript 在编译时根据上下文自动推导变量和表达式的类型。它根据变量的赋值、函数的返回值、表达式的操作等信息来确定变量或表达式的最佳类型。

1） 基础类型推断

TypeScript 根据变量的初始值来推断基础类型，包括字符串、数字、布尔值等。

```
let name = "John";  // 推断为 string 类型 let age = 30;  // 推断为 number 类型 let isStudent = true;  // 推断为 boolean 类型
```

2） 最佳公共类型推断

当我们将不同类型的值赋给一个变量或数组时，TypeScript 会根据这些值的类型推断出一个最佳公共类型。

```
let values = [1, 2, "three", true];  // 推断为 (number | string | boolean)[]
```

在上面的示例中，数组 ***values*** 包含了数字、字符串和布尔值，TypeScript 推断出这个数组的类型为 ***(number | string | boolean)[]***，即联合类型。

3） 上下文类型推断

TypeScript 会根据上下文中的预期类型推断变量的类型。这种上下文可以是函数参数、赋值语句等。

```
function greet(person: string) {
  console.log(`Hello, ${person}!`);}
greet("John");  // person 的类型推断为 string
```

在上面的示例中，函数 ***greet*** 的参数 ***person*** 的类型被推断为 ***string***，因为在函数调用时传入的实参是一个字符串。

4） 类型断言

如果我们希望手动指定一个变量或表达式的类型，可以使用类型断言（Type Assertion）来告诉 TypeScript 我们的意图。

```
let value = "Hello, TypeScript!";let length = (value as string).length;
// 类型断言为 string
```

在上面的示例中，我们使用类型断言 ***as string*** 将变量 value 的类型指定为 ***string***，以便在后面获取其长度时，TypeScript 能正确推断出类型。

5） 类型推断和泛型

在使用泛型时，TypeScript 会根据传入的参数类型推断泛型类型的具体类型。

```
function identity<T>(value: T): T {
  return value;}
let result = identity("Hello, TypeScript!");  // result 的类型推断为 string
```

在上面的示例中，泛型函数 ***identity*** 的参数 ***value*** 的类型被推断为传入的实参类型，因此返回值的类型也被推断为 ***string***。

2. 总结

类型推断是 TypeScript 中的一个重要特性，通过自动推导变量和表达式的类型，可以

提高代码的可读性和可维护性。TypeScript 根据赋值、返回值、上下文等信息进行类型推断，并在需要时允许手动指定类型。在编写现代化高级 TypeScript 代码时，深入了解和应用类型推断是非常重要的一部分。通过结合最新的 TypeScript 语法和类型推断，我们可以编写更具表达力和类型安全的代码。

十、类型守卫

1. 概述

在 TypeScript 中，类型守卫可以用于在运行时检查变量的类型，并在代码块内部将变量的类型范围缩小到更具体的类型。这种类型收窄可以让 TypeScript 编译器更好地理解我们代码的意图，从而提供更准确的类型推断和类型检查。

类型守卫通常使用类型断言、类型谓词、typeof 操作符、instanceof 操作符或自定义的谓词函数来判断变量的具体类型，并根据判断结果收窄变量的类型范围。

2. typeof 类型守卫

typeof 类型守卫允许我们使用 typeof 操作符来在代码中根据变量的类型范围进行条件判断。以下是一个示例：

```
function printValue(value: string | number) {
  if (typeof value === 'string') {
    console.log(value.toUpperCase());
  } else {
    console.log(value.toFixed(2));
  }}
printValue('hello');  // 输出: HELLOprintValue(3.1415);   // 输出: 3.14
```

在上面的示例中，我们使用 ***typeof*** 操作符在条件语句中检查变量 ***value*** 的类型。如果它的类型是 ***'string'***，则调用 ***toUpperCase*** 方法；如果是 ***'number'***，则调用 ***toFixed*** 方法。通过使用 typeof 类型守卫，我们能够根据不同的类型执行不同的代码逻辑。

3. instanceof 类型守卫

instanceof 类型守卫允许我们使用 instanceof 操作符来检查对象的类型，并在代码块内部收窄对象的类型范围。以下是一个示例：

```
class Animal {
  move() {
    console.log('Animal is moving');
  }}
class Dog extends Animal {
  bark() {
    console.log('Dog is barking');
  }}
function performAction(animal: Animal) {
  if (animal instanceof Dog) {
    animal.bark();
  } else {
    animal.move();
  }}
const animal1 = new Animal();const animal2 = new Dog();
performAction(animal1);          //     输     出     :     Animal     is
movingperformAction(animal2);  // 输出: Dog is barking
```

在上面的示例中，我们使用 instanceof 操作符在条件语句中检查变量 ***animal*** 的类型。如果它是 Dog 类的实例，则调用 ***bark*** 方法；否则调用 ***move*** 方法。通过使用 instanceof 类型守卫，我们可以根据对象的具体类型执行不同的代码逻辑。

4. 使用自定义谓词函数类型守卫

自定义谓词函数类型守卫允许我们定义自己的函数，根据特定条件判断变量的类型，并在代码块内部收窄变量的类型范围。以下是一个示例：

```
interface Circle {
  kind: 'circle';
  radius: number;}
interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;}
```

```
type Shape = Circle | Rectangle;
function calculateArea(shape: Shape) {
  if (isCircle(shape)) {
    console.log(Math.PI * shape.radius ** 2);
  } else {
    console.log(shape.width * shape.height);
  }}
function isCircle(shape: Shape): shape is Circle {
  return shape.kind === 'circle';}
const circle: Circle = { kind: 'circle', radius: 5 };const rectangle:
Rectangle = { kind: 'rectangle', width: 10, height: 20 };
calculateArea(circle);                              //      输      出      :
78.53981633974483calculateArea(rectangle);  // 输出: 200
```

在上面的示例中，我们定义了 ***Shape*** 类型，它可以是 ***Circle*** 或 ***Rectangle***。通过自定义的谓词函数 ***isCircle***，我们判断变量 ***shape*** 的类型是否是 ***Circle***，并在条件语句内部收窄变量的类型范围。通过使用自定义谓词函数类型守卫，我们能够根据特定的谓词条件执行相应的代码逻辑。

5. 联合类型守卫

类型守卫最常用于联合类型中，因为联合类型可能包含多个不同的类型选项。以下是一个更复杂的示例，展示了如何使用类型守卫和联合类型来提供更精确的类型推断和类型检查：

```
interface Car {
  type: 'car';
  brand: string;
  wheels: number;}
interface Bicycle {
  type: 'bicycle';
  color: string;}
interface Motorcycle {
  type: 'motorcycle';
  engine: number;}
```

```
type Vehicle = Car | Bicycle | Motorcycle;
function printVehicleInfo(vehicle: Vehicle) {
  switch (vehicle.type) {
    case 'car':
      console.log(`Brand:          ${vehicle.brand},          Wheels:
${vehicle.wheels}`);
      break;
    case 'bicycle':
      console.log(`Color: ${vehicle.color}`);
      break;
    case 'motorcycle':
      console.log(`Engine: ${vehicle.engine}`);
      break;
    default:
      const _exhaustiveCheck: never = vehicle;
  }}
const car: Car = { type: 'car', brand: 'Toyota', wheels: 4 };const bicycle:
Bicycle = { type: 'bicycle', color: 'red' };const motorcycle: Motorcycle
= { type: 'motorcycle', engine: 1000 };
printVehicleInfo(car);              // 输出: Brand: Toyota, Wheels:
4printVehicleInfo(bicycle);                  //    输    出    :    Color:
redprintVehicleInfo(motorcycle);  // 输出: Engine: 1000
```

在上面的示例中，我们定义了 ***Vehicle*** 类型，它是 ***Car***、***Bicycle*** 和 ***`Motor cycle*** 的联合类型。通过使用 ***switch*** 语句和根据 ***vehicle.type*** 的不同值进行类型守卫，我们可以在每个 ***case*** 分支中收窄 ***vehicle`*** 的类型范围，并执行相应的代码逻辑。通过这种方式，我们能够更准确地推断和检查联合类型的变量。

非常抱歉之前的回答没有覆盖到你提到的其他重要概念。下面我将继续解析这些概念并提供相应的代码示例。

6. 使用 *in* 操作符进行类型守卫

in 操作符可以用于在 TypeScript 中判断一个属性是否存在于对象中，从而进行类型判

断和类型收窄。以下是一个示例：

```
interface Circle {
  kind: 'circle';
  radius: number;}
interface Rectangle {
  kind: 'rectangle';
  width: number;
  height: number;}
type Shape = Circle | Rectangle;
function printArea(shape: Shape) {
  if ('radius' in shape) {
    console.log(Math.PI * shape.radius ** 2);
  } else {
    console.log(shape.width * shape.height);
  }}
const circle: Circle = { kind: 'circle', radius: 5 };const rectangle:
Rectangle = { kind: 'rectangle', width: 10, height: 20 };
printArea(circle);     // 输出: 78.53981633974483printArea(rectangle);
// 输出: 200
```

在上面的示例中，我们使用 in 操作符来检查属性 ***'radius'*** 是否存在于 ***shape*** 对象中。如果存在，则收窄 ***shape*** 的类型为 ***Circle***，并执行相应的代码逻辑。通过使用 ***in*** 操作符进行类型判断，我们可以根据属性的存在与否进行类型收窄。

7. 控制流类型守卫

在 TypeScript 中，当执行特定的操作后，编译器会智能地调整变量的类型范围，这被称为控制流类型收窄。以下是一些常见的控制流类型收窄情况：

if 语句的条件判断

```
function printValue(value: string | number) {
  if (typeof value === 'string') {
    console.log(value.toUpperCase());
```

```
  } else {
    console.log(value.toFixed(2));
  }}
```

在上面的示例中，当执行 ***typeof value === 'string'*** 的条件判断时，TypeScript 编译器会收窄 ***value*** 的类型为 ***string***，从而在代码块内部提供相应的智能提示和类型检查。

switch 语句的 case 判断

```
type Fruit = 'apple' | 'banana' | 'orange';
function getFruitColor(fruit: Fruit) {
  let color: string;
  switch (fruit) {
    case 'apple':
      color = 'red';
      break;
    case 'banana':
      color = 'yellow';
      break;
    default:
      color = 'orange';
  }
  console.log(`The color of ${fruit} is ${color}`);}
```

在上面的示例中，根据 ***switch*** 语句中的 ***case*** 判断，TypeScript 编译器会智能地收窄 ***color*** 的类型为相应的颜色字符串。

真值类型守卫

真值收窄是一种在条件表达式中进行类型收窄的机制。当条件表达式的结果是真值时，TypeScript 编译器会将变量的类型范围缩小为 ***true*** 的类型。以下是一个示例：

```
function processValue(value: string | null) {
  if (value) {
    console.log(value.toUpperCase());
```

```
  } else {
    console.log('Value is null or empty');
  }}
```

在上面的示例中，当条件表达式 value 的结果是真值（即不为 null 或空字符串）时，TypeScript 编译器会将 value 的类型范围缩小为 string。

8. 自定义类型判断式（Type Predicates）守卫

TypeScript 提供了自定义类型判断式的功能，它允许我们定义自己的谓词函数来进行类型判断和类型收窄。以下是一个示例：

```
interface Bird {
  fly(): void;}
interface Fish {
  swim(): void;}
function isBird(animal: Bird | Fish): animal is Bird {
  return (animal as Bird).fly !== undefined;}
function processAnimal(animal: Bird | Fish) {
  if (isBird(animal)) {
    animal.fly();
  } else {
    animal.swim();
  }}
```

在上面的示例中，我们定义了 ***isBird*** 谓词函数来判断参数 ***animal*** 是否属于 ***Bird*** 类型。在 ***processAnimal*** 函数中，通过使用自定义谓词函数 ***isBird***，我们能够根据 ***animal*** 的具体类型执行相应的代码逻辑，并在代码块内部收窄 ***animal*** 的类型范围。

十一、泛型和类型体操

泛型和类型体操（Type Gymnastics）是 TypeScript 中高级类型系统的重要组成部分。它们提供了强大的工具和技巧，用于处理复杂的类型操作和转换。

1. 泛型（Generics）

1） 泛型函数

泛型函数允许我们在函数定义中使用类型参数，以便在函数调用时动态指定类型。例如：

```
function identity<T>(arg: T): T {
  return arg;}
let result = identity<number>(42);  // result 的类型为 number
```

在上面的示例中，***identity*** 函数使用类型参数 ***T***，并返回与输入类型相同的值。通过显式传递泛型参数，我们可以确保在函数调用时指定了具体的类型。

2） 泛型接口

泛型接口允许我们在接口定义中使用类型参数，以便在实现该接口时指定具体的类型。例如：

```
interface Container<T> {
  value: T;}
let container: Container<number> = { value: 42 };
```

在上面的示例中，我们定义了一个泛型接口 ***Container***，它包含一个类型参数 ***T***。通过指定 ***Container<number>***，我们创建了一个具体的实现，其中的 ***value*** 属性类型为 ***number***。

3） 泛型类

泛型类允许我们在类定义中使用类型参数，以便在创建类的实例时指定具体的类型。例如：

```
class Stack<T> {
  private items: T[] = [];

  push(item: T) {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }}
let stack = new Stack<number>();
stack.push(1);
stack.push(2);let item = stack.pop();  // item 的类型为 number | undefined
```

在上面的示例中，我们定义了一个泛型类 ***Stack***，它使用类型参数 ***T*** 来表示堆栈中的元素类型。通过创建 ***Stack<number>*** 的实例，我们限制了堆栈中的元素必须为 ***number*** 类型。

2. 类型体操（Type Gymnastics）

1） 条件类型（Conditional Types）

条件类型允许我们根据输入类型的条件判断结果来选择不同的类型。条件类型的语法形式为：

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

其中，***T***是待检查的类型，***U***是条件类型，***X***是满足条件时返回的类型，***Y***是不满足条件时返回的类型。

下面是一个使用条件类型的示例：

```
type Check<T> = T extends string ? true : false;
type Result = Check<string>;  // Result 的类型为 true
```

在上面的示例中，我们定义了一个条件类型 ***Check<T>***，它接受一个类型参数 ***T***。如果 ***T*** 是 ***string*** 类型，那么 ***Check<T>*** 的类型将是 ***true***，否则为 ***false***。

2） keyof 操作符和索引访问类型

keyof 操作符用于获取类型的所有属性名，结合索引访问类型可以从一个类型中获取属性的具体类型。

```
interface Person {
  name: string;
  age: number;}
type PersonKeys = keyof Person;  // "name" | "age"type PersonNameType =
Person['name'];  // string
```

在上面的示例中，我们使用 ***keyof*** 操作符获取了 ***Person*** 接口的属性名集合，并通过索引访问类型获取了 ***Person*** 接口中 ***name*** 属性的类型。

3） infer 关键字

infer 关键字用于在条件类型中推断类型，并将其赋值给一个类型变量。

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
function add(a: number, b: number): number {
  return a + b;}
```

```
type AddReturnValue = ReturnType<typeof add>;  // 类型为 number
```

在上面的示例中，***ReturnType*** 类型接受一个类型参数 ***T***，并使用条件类型和 ***infer*** 关键字推断函数类型的返回类型。通过调用 ***ReturnType<typeof add>***，我们推断出 ***add*** 函数的返回类型为 ***number***。

当涉及到泛型时，还有一些重要的概念和内置泛型函数可以深入分析。让我们继续探讨 ***extends*** 关键字、TS 官方内置的一些泛型函数以及它们的使用。

3. extends 关键字和类型约束

在泛型中，我们可以使用 ***extends*** 关键字来对泛型类型进行约束。这样可以确保传递给泛型的类型满足特定条件。

```
function printProperty<T extends { name: string }>(obj: T): void {
  console.log(obj.name);}
printProperty({ name: 'John', age: 25 });  // 输出 'John'
```

在上面的示例中，***printProperty*** 函数接受一个泛型参数 T，该参数必须满足一个约束条件：具有 ***name*** 属性，且 ***name*** 的类型为 ***string***。通过使用 ***extends*** 关键字和类型约束，我们可以确保 ***obj*** 参数具有所需的属性和类型，从而避免出现错误。

1） 泛型函数 Util

TypeScript 提供了一些内置的泛型函数，这些函数被广泛用于处理各种类型操作。以下是一些常见的官方内置泛型函数：

a）Partial<T>

Partial<T> 是 TypeScript 中的一个内置泛型类型，它可以将给定类型 ***T*** 中的所有属性转换为可选属性。这对于创建部分完整的对象非常有用。

```
interface Person {
  name: string;
  age: number;}
type PartialPerson = Partial<Person>;
const partialPerson: PartialPerson = { name: 'John' };  // age 属性是可选的
```

在上面的示例中，***Partial<Person>***将 ***Person*** 接口中的所有属性变为可选属性，从而创建了一个部分完整的 ***PartialPerson*** 类型。

b）Required<T>

Required<T>是 TypeScript 中的另一个内置泛型类型，它可以将给定类型 T 中的所有可选属性转换为必需属性。这对于确保对象的完整性非常有用。

```
interface Person {
  name?: string;
  age?: number;}
type RequiredPerson = Required<Person>;
const requiredPerson: RequiredPerson = { name: 'John', age: 25 };  // name 和 age 属性是必需的
```

在上面的示例中，***Required<Person>***将 ***Person*** 接口中的所有可选属性变为必需属性，从而创建了一个要求完整性的 ***RequiredPerson*** 类型。

c）Pick<T, K>

Pick<T, K>是 TypeScript 中的另一个内置泛型函数，它可以从给定类型 ***T***中选择指定的属性 ***K***组成一个新的类型。

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

```
  address: string;}
type NameAndAge = Pick<Person, 'name' | 'age'>;
const person: NameAndAge = { name:

 'John', age: 25 };  // 只包含 name 和 age 属性
```

在上面的示例中，***Pick<Person, 'name' | 'age'>*** 从 Person 接口中选择了 ***'name'*** 和 ***'age'*** 属性，创建了一个新的类型 ***NameAndAge***。

我们还可以结合泛型和内置泛型函数来实现更复杂的类型操作。以下是一个示例，展示了如何使用 Pick 和泛型来创建一个函数，该函数从给定对象中选择指定属性，并返回一个新的对象。

```
function pickProperties<T, K extends keyof T>(obj: T, keys: K[]): Pick<T,
K> {
  const result: Partial<T> = {};
  for (const key of keys) {
    result[key] = obj[key];
  }
  return result as Pick<T, K>;}
interface Person {
  name: string;
  age: number;
  address: string;}
const person: Person = {
  name: 'John',
  age: 25,
  address: '123 Main St'};
const nameAndAge = pickProperties(person, ['name', 'age']);  // 只包含
name 和 age 属性 console.log(nameAndAge);  // 输出: { name: 'John', age:
25 }
```

在上面的示例中，***pickProperties*** 函数接受一个泛型参数 ***T*** 和一个属性数组 ***keys***。通过使用 ***Pick<T, K>***，我们将从给定对象 ***obj*** 中选择指定的属性 ***keys***，并创建一个新的

对象。

这个例子结合了泛型、内置泛型函数 ***Pick***、***keyof*** 操作符和 ***extends*** 关键字，展示了如何在 ***TypeScript*** 中处理复杂的类型操作和转换。

当涉及到官方内置的泛型函数时，还有一些重要的函数值得分析。让我们继续探讨一些常用的官方内置泛型函数以及它们的使用。

2） Exclude<T, U>

Exclude<T, U> 是 TypeScript 中的一个内置泛型函数，用于从类型 ***T*** 中排除类型 ***U***。它返回一个新类型，该新类型包含在 ***T*** 中存在但不在 ***U*** 中存在的成员类型。

```
type T = Exclude<"a" | "b" | "c", "a" | "b">;  // T 的类型为 "c"
```

在上面的示例中，***Exclude<"a" | "b" | "c", "a" | "b">*** 排除了类型 "***a***" 和 "***b***"，返回类型为 "***c***"。

3） Omit<T, K>

Omit<T, K> 是 TypeScript 中的另一个内置泛型函数，它返回一个新类型，该新类型排除了类型 ***T*** 中指定的属性 ***K***。

```
interface Person {
  name: string;
  age: number;
  address: string;}
type PersonWithoutAddress = Omit<Person, "address">;
```

在上面的示例中，***Omit<Person, "address">*** 返回了一个新类型 ***PersonWithoutAddress***，该类型排除了 ***Person*** 接口中的 ***address*** 属性。

4） Readonly<T>

Readonly<T> 是 TypeScript 中的另一个内置泛型函数，它将类型 ***T*** 中的所有属性转换为只读属性。

```
interface Person {
  name: string;
  age: number;}
type ReadonlyPerson = Readonly<Person>;
```

在上面的示例中，***Readonly<Person>*** 将 ***Person*** 接口中的所有属性变为只读属性，创建了一个新类型 ***ReadonlyPerson***。

4. 总结

泛型和类型体操是 TypeScript 中强大的类型系统的关键组成部分。通过使用泛型，我们可以创建可重用、灵活和类型安全的代码。内置泛型函数提供了一些常用的类型转换工具，如 Partial、Required 和 Pick，可以帮助我们更方便地处理类型操作。

通过结合泛型、extends 关键字、内置泛型函数和其他高级类型概念，我们能够在 TypeScript 中编写更复杂、类型安全的代码，并利用 TypeScript 的强大类型系统来提高代码的可读性、可维护性和可扩展性。

十二、类型兼容：结构化类型

TypeScript 是一种基于 JavaScript 的静态类型语言，它为 JavaScript 添加了类型系统，并提供了强大的类型检查和自动补全功能。

TypeScript 的类型系统有一个非常重要的特性，那就是 "鸭子类型"（Duck Typing）或 "结构化类型"（Structural Typing）（文章会以"鸭子类型"（Duck Typing）作为简称）。这种特性有时会让人感到惊讶，但它是 TypeScript 增强 JavaScript 开发体验的重要方式之一。

鸭子类型的概念来自一个古老的英语成语：“如果它走起路来像一只鸭子，叫起来也像一只鸭子，那么它就是一只鸭子。”在 TypeScript（或更一般地说，静态类型语言）的上下文中，鸭子类型意味着一个对象的类型不是由它继承或实现的具体类别决定的，而是由它具有的结构决定的。

本文将全面深入地探讨 TypeScript 中的鸭子类型，以及如何在实际的开发中应用和利用鸭子类型。

1. 鸭子类型：定义和示例

鸭子类型的概念来自一个古老的英语成语：“如果它走起路来像一只鸭子，叫起来也像一只鸭子，那么它就是一只鸭子。”在 TypeScript（或更一般地说，静态类型语言）的上下文中，鸭子类型意味着一个对象的类型不是由它继承或实现的具体类别决定的，而是由它具有的结构决定的。

这是一个简单的鸭子类型示例：

```
interface Duck {
    walk: () => void;
    quack: () => void;}
function doDuckThings(duck: Duck) {
```

```
    duck.walk();
    duck.quack();}
const myDuck = {
    walk: () => console.log('Walking like a duck'),
    quack: () => console.log('Quacking like a duck'),
    swim: () => console.log('Swimming like a duck')};
doDuckThings(myDuck); // OK
```

在这个例子中，我们定义了一个 ***Duck*** 接口和一个 ***doDuckThings*** 函数，这个函数需要一个 ***Duck*** 类型的参数。然后我们创建了一个 ***myDuck*** 对象，它有 ***walk***、***quack*** 和 ***swim*** 这三个方法。尽管 ***myDuck*** 并没有显式地声明它实现了 ***Duck*** 接口，但是由于 ***myDuck*** 的结构满足了 ***Duck*** 接口的要求（即 ***myDuck*** 有 ***walk*** 和 ***quack*** 这两个方法），我们可以将 ***myDuck*** 作为参数传递给 ***doDuckThings*** 函数。

这就是鸭子类型的基本概念：只要一个对象的结构满足了接口的要求，我们就可以把这个对象看作是这个接口的实例，而不管这个对象的实际类型是什么。

2. 鸭子类型的优点

鸭子类型有许多优点，特别是在编写更灵活和更通用的代码方面。

1） 代码的灵活性

鸭子类型增加了代码的灵活性。我们可以创建和使用满足特定接口的任何对象，而不必担心它们的具体类型。这使得我们可以更容易地编写通用的代码，因为我们的代码只依赖于对象的结构，而不是对象的具体类型。

2） 代码的复用

鸭子类型有助于代码的复用。由于我们的函数和方法只依赖于对象的结构，我们可以在不同的上下文中重用这些函数和方法，只要传入的对象满足所需的结构。
例如，我们可以写一个函数，它接受一个具有 ***toString*** 方法的任何对象，然后返回这个

对象的字符串表示。由于几乎所有的 JavaScript 对象都有 ***toString*** 方法，我们可以在许多不同的上下文中重用这个函数。

```
function toString(obj: { toString: () => string }) {
   return obj.toString();}
console.log(toString(123)); // "123"console.log(toString([1, 2, 3]));
// "1,2,3"console.log(toString({ a: 1, b: 2 })); // "[object Object]"
```

3） 与 JavaScript 的互操作性

鸭子类型提高了 TypeScript 与 JavaScript 的互操作性。由于 JavaScript 是一种动态类型语言，我们经常需要处理的对象可能没有明确的类型。鸭子类型使我们能够在 TypeScript 中安全地处理这些对象，只要它们的结构满足我们的需求。

例如，我们可能从一个 JavaScript 库获取一个对象，这个对象有一个 ***forEach*** 方法。我们不关心这个对象的具体类型，我们只关心它是否有 ***forEach*** 方法。使用鸭子类型，我们可以定义一个接口来描述这个对象的结构，然后在 TypeScript 中安全地使用这个对象。

```
interface Iterable {
   forEach: (callback: (item: any) => void) => void;}
function processItems(iterable: Iterable) {
   iterable.forEach(item => console.log(item));}
const jsArray = [1, 2, 3]; // From a JavaScript
libraryprocessItems(jsArray); // OK
```

3. 鸭子类型的局限性

尽管鸭子类型有许多优点，但它也有一些局限性。

1） 类型安全

鸭子类型可能会降低代码的类型安全性。因为 TypeScript 的类型检查器只检查对象是否满足接口的结构，而不检查对象是否真的是接口所期望的类型。如果一个对象恰好有与接口相同的属性和方法，但实际上它并不是接口所期望的类型，TypeScript 的类型检查器可能无法发现这个错误。

例如，我们可能有一个 ***Dog*** 类型和一个 ***Cat*** 类型，它们都有一个 ***bark*** 方法。我们可能会错误地将一个 `Cat ***对象传递给一个期望*** Dog` 对象的函数，而 TypeScript 的类型检查器无法发现这个错误。

```
interface Dog {
    bark: () => void;}
function letDogBark(dog: Dog) {
    dog.bark();}
const cat = {
    bark: () => console.log('Meow...'),  // Cats don't bark!
    purr: () => console.log('Purr...')};
letDogBark(cat); // No error, but it's wrong!
```

在这种情况下，我们需要更仔细地设计我们的类型和接口，以避免混淆。

2） 易读性和可维护性

鸭子类型可能会降低代码的易读性和可维护性。因为我们的代码只依赖于对象的结构，而不是对象的具体类型，这可能会使代码更难理解和维护。

为了提高易读性和可维护性，我们需要清晰地记录我们的接口和函数期望的对象结构。TypeScript 的类型注解和接口提供了一种强大的工具来实现这一点。

4. 使用鸭子类型的最佳实践

在使用鸭子类型时，有一些最佳实践可以帮助我们避免上述问题，并充分利用鸭子类型

的优点。

1） 清晰地定义接口

我们应该清晰地定义我们的接口，以描述我们的函数和方法期望的对象结构。这有助于提高代码的易读性和可维护性。

例如，如果我们有一个函数，它期望一个具有 ***name*** 和 ***age*** 属性的对象，我们应该定义一个接口来描述这个结构。

```
interface Person {
    name: string;
    age: number;}
function greet(person: Person) {
    console.log(`Hello, my name is ${person.name} and I'm ${person.age}
years old.`);}
```

2） 适度使用鸭子类型

我们应该适度地使用鸭子类型。虽然鸭子类型有许多优点，但如果过度使用，可能会导致类型安全性的问题，以及易读性和可维护性的降低。我们应该在类型安全性、易读性、可维护性和灵活性之间找到一个平衡。

在某些情况下，我们可能更希望使用类和继承，而不是鸭子类型。例如，如果我们有一组紧密相关的类型，它们有共享的行为和状态，使用类和继承可能更合适。

```
interface Named {
    name: string;}
class Person {
    name: string;

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

```
    }}
let p: Named;// OK, because of structural typing
p = new Person('mike');
```

在这个例子中，尽管 ***Person*** 类并没有显式地实现 ***Named*** 接口，但是因为 ***Person*** 类有一个 ***name*** 属性，所以我们可以把 ***Person*** 的实例赋值给 ***Named*** 类型的变量。这是由于 TypeScript 的 "鸭子类型" 或 "结构化类型" 系统导致的。

十三、类型兼容：协变和逆变

引言

在类型系统中，协变和逆变是对类型比较(***类型兼容***)一种形式化描述。在一些类型系统中，例如 Java，这些概念是显式嵌入到语言中的，例如使用 extends 关键字表示协变，使用 ***super*** 关键字表示逆变。在其他一些类型系统中，例如 TypeScript，协变和逆变的规则是隐式嵌入的，通过类型兼容性检查来实现。

协变和逆变的存在使得类型系统具有更大的灵活性。例如，如果你有一个 ***Animal*** 类型的数组，并且你有一个 ***Dog*** 类型的对象（假设 ***Dog*** 是 ***Animal*** 的子类型），那么你应该能够将 ***Dog*** 对象添加到 ***Animal*** 数组中。这就是协变。反过来，如果你有一个处理 ***Animal*** 类型对象的函数，并且你有一个 ***Dog*** 类型的对象，你应该可以使用这个函数来处理 ***Dog*** 对象。这就是逆变。

协变和逆变还可以帮助我们创建更通用的代码。例如，如果你有一个可以处理任何 Animal 的函数，那么这个函数应该能够处理任何 ***Animal*** 的子类型。这意味着，你可以编写一段只依赖于 ***Animal*** 类型的代码，然后使用这段代码处理任何 ***Animal*** 的子类型。

1. 协变（Covariance）

协变描述的是如果存在类型 A 和 B，并且 A 是 B 的子类型，那么我们就可以说由 A 组成的复合类型（例如 ***Array<A>*** 或者 ***(a: A) => void***）也是由 B 组成的相应复合类型（例如 ***Array<B>*** 或者 ***(b: B) => void)*** 的子类型。

让我们通过一个例子来理解协变。假设我们有两个类型 ***Animal*** 和 ***Dog***，其中 ***Dog*** 是 ***Animal*** 的子类型。

```
type Animal = { name: string };type Dog = Animal & { breed: string };
let dogs: Dog[] = [{ name: "Fido", breed: "Poodle" }];let animals: Animal[]
= dogs;  // OK because Dog extends Animal, Dog[] is a subtype of Animal[]
```

这里我们可以将类型为 ***Dog[]*** 的 ***dogs*** 赋值给类型为 ***Animal[]*** 的 ***animals***，因为 ***Dog[]*** 是 ***Animal[]*** 的子类型，所以数组是协变的。

2. 协变：类型的向下兼容性

协变是类型系统中的一个基本概念，它描述的是类型的“向下兼容性”。如果一个类型 A 可以被看作是另一个类型 B 的子类型（即 A 可以被安全地用在期望 B 的任何地方），那么我们就说 A 到 B 是协变的。这是类型系统中最常见和直观的一种关系，例如在面向对象编程中的继承就是协变的一种表现。

在 TypeScript 中，所有的类型都是自身的子类型（即每个类型到自身是协变的），并且 ***null*** 和 ***undefined*** 类型是所有类型的子类型。除此之外，接口和类也可以通过继承来形成协变关系。

```
class Animal {
  name: string;}
class Dog extends Animal {
  breed: string;}
let myDog: Dog = new Dog();let myAnimal: Animal = myDog;  // OK，因为 Dog 是 Animal 的子类型
```

这个例子中，我们可以将一个 ***Dog*** 对象赋值给一个 ***Animal*** 类型的变量，因为 ***Dog*** 到 ***Animal*** 是协变的。

在 TypeScript 中，泛型类型也是协变的。例如，如果类型 A 是类型 B 的子类型，那么 ***Array<A>*** 就是 ***Array<B>*** 的子类型。

```
let dogs: Array<Dog> = [new Dog()];let animals: Array<Animal> = dogs;
// OK，因为 Array<Dog>是 Array<Animal>的子类型
```

3. 逆变（Contravariance）

逆变是协变的反面。如果存在类型 A 和 B，并且 A 是 B 的子类型，那么我们就可以说由 B 组成的某些复合类型是由 A 组成的相应复合类型的子类型。

这在函数参数中最常见。让我们来看一个例子：

```
type Animal = { name: string };type Dog = Animal & { breed: string };
let dogHandler = (dog: Dog) => { console.log(dog.breed); }let
animalHandler: (animal: Animal) => void = dogHandler; // Error!
```

在这个例子中，我们不能将类型为***(dog: Dog) => void***的 ***dogHandler***赋值给类型为***(animal: Animal) => void***的 ***animalHandler***。因为如果我们传递一个 ***Animal***（并非所有的 Animal 都是 Dog）给 ***animalHandler***，那么在执行 ***dogHandler***函数的时候，就可能会引用不存在的 ***breed***属性。因此，函数的参数类型是逆变的。

4. 逆变：类型的向上兼容性

逆变描述的是类型的“向上兼容性”。如果一个类型 A 可以被看作是另一个类型 B 的超类型（即 B 可以被安全地用在期望 A 的任何地方），那么我们就说 A 到 B 是逆变的。在函数参数类型的兼容性检查中，TypeScript 使用了逆变。

```
type Handler = (arg: Animal) => void;
let animalHandler: Handler = (animal: Animal) => { /* ... */ };let
dogHandler: Handler = (dog: Dog) => { /* ... */ }; // OK, 因为 Animal
是 Dog 的超类型
```

这个例子中，我们可以将一个处理`***Dog***的函数赋值给一个处理 ***Animal***的函数类型的变量，因为 ***Animal***是 ***Dog***的超类型，所以***(dog: Dog) => void***类型是***(animal: Animal) => void***`类型的子类型。

这看起来可能有些反直觉，但实际上是为了保证类型安全。因为在执行 ***dogHandler*** 函数时，我们可以安全地传入一个 ***Animal*** 对象，而不需要担心它可能不是 ***Dog*** 类型。

5. 协变与逆变的平衡

协变和逆变在大多数情况下都可以提供合适的类型检查，但是它们并非完美无缺。在实际应用中，我们必须关注可能的边界情况，以避免运行时错误。在某些情况下，我们甚至需要主动破坏类型的协变或逆变，以获得更强的类型安全。例如，如果我们需要向一个 ***Dog[]*** 数组中添加 ***Animal*** 对象，我们可能需要将这个数组的类型声明为 ***Animal[]***，以防止添加不兼容的类型。

总的来说，协变和逆变是理解和应用 TypeScript 类型系统的重要工具，但我们必须在灵活性和类型安全之间找到合适的平衡。

十四、扩展类型定义

在 TypeScript 中，我们可以通过声明文件（.d.ts 文件）来为现有的 JavaScript 库提供类型定义，或者为现有的类型添加额外的属性和方法。这个过程通常被称为“类型声明扩展”。在这篇文章中，我们将详细探讨如何通过声明文件扩展类型定义。

1. 什么是声明文件？

在 TypeScript 中，声明文件是一种以 .d.ts 为扩展名的特殊文件，它不包含具体的实现，只包含类型声明。这些文件通常用来为已有的 JavaScript 库提供类型定义，使得我们可以在 TypeScript 代码中更安全、更方便地使用这些库。

声明文件的主要内容是类型声明，包括变量、函数、类、接口等的类型定义。这些类型声明提供了一种描述 JavaScript 代码的结构和行为的方式，使得 TypeScript 编译器能够理解和检查 JavaScript 代码。

例如，以下是一个简单的声明文件的例子：

```
// types.d.tsdeclare var foo: string;declare function bar(baz: number): boolean;
```

在上面的声明文件中，我们声明了一个全局变量 foo 和一个全局函数 bar，并分别给它们提供了类型声明。

2. declare

当我们在 TypeScript 中编写声明文件时，我们使用 ***declare*** 关键字来声明全局变量、函数、类、接口等类型。

declare 关键字用于告诉 TypeScript 编译器某个标识符的类型，而不需要实际的实现

代码。它用于在声明文件中描述 JavaScript 代码的类型。

下面是一些常见的用法：

1） 声明全局变量：

```
declare const myGlobal: string;
```

这个声明告诉 TypeScript 编译器，存在一个名为 ***myGlobal*** 的全局变量，它的类型是 ***string***。

2） 声明全局函数：

```
declare function myFunction(arg: number): string;
```

这个声明告诉 TypeScript 编译器，存在一个名为 ***myFunction*** 的全局函数，它接受一个 ***number*** 类型的参数，并返回一个 ***string*** 类型的值。

3） 声明全局类：

```
declare class MyClass {
  constructor(name: string);
  getName(): string;}
```

这个声明告诉 TypeScript 编译器，存在一个名为 MyClass 的全局类，它有一个接受 ***string*** 类型参数的构造函数，并且有一个返回 ***string*** 类型的 ***getName*** 方法。

4） 声明命名空间

```
declare namespace MyNamespace {
  export const myVariable: number;
  export function myFunction(): void;}
```

这个声明告诉 TypeScript 编译器，存在一个名为 ***MyNamespace*** 的全局模块/命名空间，它包含一个名为 ***myVariable*** 的变量和一个名为 ***myFunction*** 的函数。
通过使用 ***declare*** 关键字，我们可以在声明文件中描述出我们所需要的类型信息，以便 TypeScript 编译器进行类型检查和类型推断。

需要注意的是，***declare*** 关键字只用于类型声明，不包含具体的实现代码。在使用声明文件时，我们需要确保提供了实际的实现代码，以便程序在运行时可以访问到所声明的类型。

5） 声明模块

当我们在声明文件中使用 ***declare module*** 时，我们可以定义一个模块，并在其中声明模块内部的类型。这样，其他文件在导入该模块时，就可以按照模块的名称来引用其中的类型。

```
declare module 'my-module' {
  export const myVariable: string;
  export function myFunction(): void;}
```

在这个示例中，我们在 ***my-module*** 模块中声明了一个名为 ***myVariable*** 的变量和一个名为 ***myFunction*** 的函数，并通过 ***export*** 关键字将它们导出，使其在导入该模块时可见。

6）通过声明文件扩展类型定义

在某些情况下，我们可能需要为已有的类型添加额外的属性或方法。比如，我们可能在使用一个库时发现它缺少一些我们需要的类型定义，或者我们可能想要为一些内置类型（如 ***string*** 或 ***Array***）添加一些自定义的方法。

这时，我们可以通过在声明文件中使用“声明合并”（Declaration Merging）来扩展类型定义。声明合并是 TypeScript 的一项特性，它允许我们在多个位置声明同名的类型，然后 TypeScript 会将这些声明合并为一个类型。

例如，假设我们想要为所有的数组添加一个 ***last*** 属性，该属性返回数组的最后一个元素。我们可以在声明文件中为 ***Array*** 类型添加一个新的声明：

```
// types.d.tsinterface Array<T> {
  last: T;}
```

在上面的代码中，我们通过声明一个同名的 Array 接口来为 Array 类型添加一个新的 last 属性。这样，我们在 TypeScript 代码中使用数组时，就可以访问这个新的 last 属性：

```
let nums: number[] = [1, 2, 3];console.log(nums.last);  // 3
```

3. 注意事项

虽然通过声明文件扩展类型定义可以让我们更灵活地使用类型，但也需要注意一些事项。

首先，声明文件只提供类型信息，不包含实现。也就是说，如果我们为一个类型添加了新的属性或方法，我们还需要在实际的代码中提供这些属性或方法的实现。
其次，尽管 TypeScript 允许我们为内置类型添加自定义的属性和方法，但这并不意味着这是一个好的做法。在很多情况下，过度修改内置类型可能会导致代码难以理解和维护。因此，我们应该谨慎使用这种特性，尽可能地遵循库或语言的原始设计。

最后，当我们在一个项目中使用多个声明文件时，需要注意文件的加载顺序和作用域问题。因为声明文件中的类型声明会影响整个项目，所以我们需要确保所有的声明文件都被正确地加载，并且不会互相冲突。

4. 为第三方库创建声明文件

当我们在使用第三方库时，通常会遇到缺乏类型声明的情况。我们可以通过创建一个声明文件来为该库添加类型声明，以便在 TypeScript 代码中使用该库的时候获得类型检

查和自动完成的支持。

以下是一个实际的示例，假设我们使用的是一个名为 ***axios*** 的库，它是一个流行的用于发起 HTTP 请求的库。假设 ***axios*** 库没有提供类型声明文件，我们可以创建一个声明文件 ***axios.d.ts*** 来为它添加类型声明：

```
declare module 'axios' {
  export interface AxiosRequestConfig {
    url: string;
    method?: string;
    data?: any;
    headers?: any;
  }

  export interface AxiosResponse<T = any> {
    data: T;
    status: number;
    statusText: string;
    headers: any;
    config: AxiosRequestConfig;
  }

  export function request<T = any>(config: AxiosRequestConfig): Promise<AxiosResponse<T>>;
  export function get<T = any>(url: string, config?: AxiosRequestConfig): Promise<AxiosResponse<T>>;
  export function post<T = any>(url: string, data?: any, config?: AxiosRequestConfig): Promise<AxiosResponse<T>>;
  // ... 其他请求方法的类型声明 ...}
```

在这个声明文件中，我们使用 ***declare module*** 来声明一个名为 ***axios*** 的模块，并在其中定义了与 ***axios*** 相关的类型声明。

我们定义了 ***AxiosRequestConfig*** 接口，它描述了发起请求时的配置选项；定义

了 ***AxiosResponse*** 接口，它描述了请求返回的响应数据的结构。

然后，我们通过 ***export*** 关键字将 ***request***、***get*** 和 ***post*** 等函数导出为模块的公共 API，以便在其他文件中使用这些函数。

现在，在我们的 TypeScript 代码中，我们可以通过导入 ***axios*** 模块来使用这些类型声明，以及使用 ***axios*** 库的方法：

```
import axios, { AxiosResponse, AxiosRequestConfig } from 'axios';
const config: AxiosRequestConfig = {
  url: 'https://api.example.com',
  method: 'GET',};

axios.get(config)
  .then((response: AxiosResponse) => {
    console.log(response.data);
  })
  .catch((error) => {
    console.error(error);
  });
```

通过这种方式，我们可以为第三方库创建声明文件，并在 TypeScript 代码中使用它们来获得类型检查和自动完成的支持，提高代码的可靠性和开发效率。

十五、装饰器与反射元数据

1. 介绍

在过去的几年中，JavaScript 及其生态系统发生了快速的变化。其中，TypeScript 已成为许多开发人员的首选语言。其主要优势在于其静态类型系统，它使我们可以在编译时捕获错误，而不是在运行时。除此之外，TypeScript 还为我们提供了许多 ES6+特性以及一些其他的独有特性，例如枚举、命名空间和装饰器。

2. 装饰器简介

在 TypeScript 中，装饰器是一种特殊类型的声明，可以被附加到类声明，方法，属性，访问器或参数上。装饰器的核心思想是增强已经存在的类、方法、属性等的行为，或者添加新的行为。通过装饰器，我们可以在不改变原始类的定义的情况下，为类添加新的特性。

在 TypeScript 中，装饰器使用@***expression*** 的形式。其中，***expression*** 必须为一个返回函数的表达式，这个函数在运行时会被调用，传入相关的装饰器参数。
TypeScript 支持以下几种类型的装饰器：

- 类装饰器
- 方法装饰器
- 访问器装饰器
- 属性装饰器
- 参数装饰器

3. 类装饰器

类装饰器应用于类的构造函数，用于观察、修改或替换类定义。类装饰器在应用时，会作为函数调用，并将构造函数作为其唯一的参数。

```
function Sealed(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);}
@Sealedclass Greeter {
    constructor(public greeting: string) {}
    greet() {
        return "Hello, " + this.greeting;
    }}
```

4. 方法装饰器

方法装饰器应用于方法的属性描述符，并可以用于观察、修改或替换方法定义。当装饰器被调用时，它会接收到三个参数：当前类的原型，方法名，以及该方法的属性描述符。

```
function Log(target: Object, propertyKey: string, descriptor:
TypedPropertyDescriptor<any>) {
    let originalMethod = descriptor.value; // 保存原始函数
    descriptor.value = function (...args: any[]) {
        console.log("Arguments: ", JSON.stringify(args));
        let result = originalMethod.apply(this, args);


 console.log("Result: ", result);
        return result;
    }}
class Calculator {
    @Log
    add(x: number, y: number): number {
        return x + y;
    }}
```

5. 访问器装饰器

访问器装饰器可以应用于访问器的属性描述符，并可以用于观察、修改或替换访问器的定义。访问器装饰器和方法装饰器有相似的语法。

```
function ReadOnly(target: any, key: string, descriptor:
PropertyDescriptor) {
    descriptor.writable = false;
    return descriptor;}
class Circle {
    private _radius: number;

    constructor(radius: number) {
        this._radius = radius;
    }

    @ReadOnly
    get radius() {
        return this._radius;
    }}
```

6. 装饰器与反射元数据

为了让装饰器能够更好地工作，TypeScript 提供了反射元数据 API。 这是一个实验性的 API，它允许装饰器在声明时添加元数据。 可以使用 npm 来安装反射元数据 API:

反射元数据（Reflect Metadata）是一个实验性的 API，用于在声明装饰器时执行元数据类型注解和元数据反射。

```
npm install reflect-metadata --save
```

然后，你需要在全局范围内导入反射 API:

```
import "reflect-metadata";
```

在 TypeScript 配置文件 ***tsconfig.json*** 中，你需要开启 ***emitDecoratorMetadata*** 选项:

```
{
  "compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "ES5",
    "module": "commonjs"
  }}
```

然后你就可以在装饰器中使用反射 API 了:

```
function logType(target : any, key : string) {
  var t = Reflect.getMetadata("design:type", target, key);
  console.log(`${key} type: ${t.name}`);}
class Demo{
  @logType
  public attr1 : string;}
```

以上就是装饰器的基本原理与应用。装饰器可以使我们的代码更简洁，更易读，也使得我们的代码更易于管理和维护。但是，需要注意的是，装饰器目前还处于实验阶段，如果你决定在生产环境中使用装饰器，你需要了解使用装饰器可能带来的风险。

7. 结论

TypeScript 的装饰器为我们提供了一种强大的工具，可以在运行时改变类的行为。通过理解装饰器的工作原理，我们可以创造更加强大、灵活且易于维护的应用。

十六、解读 TSConfig

TypeScript 配置文件（tsconfig.json）是用于配置 TypeScript 项目的重要文件。它允许开发者自定义 TypeScript 编译器的行为，指定编译选项、文件包含与排除规则、输出目录等。通过合理配置 tsconfig.json，我们可以根据项目需求进行灵活的 TypeScript 编译设置。

本文将全面解读 tsconfig.json 的各个配置选项，并提供一些常见的使用场景和示例代码，以及封装定制化自己 ***tsconfig.base*** 配置

1. 创建和基本配置

要使用 TypeScript 配置文件，我们首先需要创建一个名为 ***tsconfig.json*** 的文件，并将其放置在项目的根目录下。

下面是一个基本的 tsconfig.json 配置示例：

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "dist"
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules",
    "dist"
  ]}
```

在上述示例中，我们使用 ***compilerOptions*** 配置选项指定了 TypeScript 编译器的选项。其中：

- ***"target"***: ***"es6"***指定编译的目标 JavaScript 版本为 ES6。
- ***"module"***: ***"commonjs"***指定模块的生成方式为 CommonJS。
- ***"outDir"***: ***"dist"***指定输出目录为 "dist"。

同时，我们使用 include 和 exclude 配置选项分别指定了需要编译的源文件的包含规则和排除规则。

2. compilerOptions

compilerOptions 是 tsconfig.json 中最重要的配置选项之一，它允许我们指定 TypeScript 编译器的各种行为和设置。以下是一些常用的 compilerOptions 配置选项：

1）target

target 选项指定了编译后的 JavaScript 代码所要遵循的 ECMAScript 标准。常见的选项包括 "***es5***"、"***es6***"、"***es2015***"、"***es2016***" 等。

```
"compilerOptions": {
  "target": "es6"}
```

2）module

module 选项用于指定生成的模块化代码的模块系统。常见的选项包括 "***commonjs***"、"***amd***"、"***es2015***"、"***system***" 等。

```
"compilerOptions": {
  "module": "commonjs"}
```

3）outDir

outDir 选项指定了编译输出的目录路径。

```
"compilerOptions": {
  "outDir": "dist"}
```

4）strict

strict 选项用于启用严格的类型检查和更严格的编码规范。

```
"compilerOptions": {
  "strict": true}
```

5）lib

lib 选项用于指定 TypeScript 编译器可以使用的 JavaScript 标准库的列表。默认情况下，TypeScript 编译器会根据目标版本自动选择合适的库。

```
"compilerOptions": {
  "lib": ["es6", "dom"]}
```

6）sourceMap

sourceMap 选项用于生成与源代码对应的源映射文件（.map 文件），以便在调试过程中可以将编译后的 JavaScript 代码映射回原始 TypeScript 代码。

```
"compilerOptions": {
  "sourceMap": true}
```

7）paths

paths 选项用于配置模块解析时的路径映射，可以帮助我们简化模块导入的路径。

```
"compilerOptions": {
  "paths": {
    "@/*": ["src/*"]
  }}
```

8）allowJs

allowJs 选项允许在 TypeScript 项目中引入 JavaScript 文件，使得我们可以混合使用 TypeScript 和 JavaScript。

```
"compilerOptions": {
  "allowJs": true}
```

9）esModuleInterop 和 allowSyntheticDefaultImports

esModuleInterop 属性用于提供对 ES 模块的兼容性支持。当我们在 TypeScript 项目中引入 CommonJS 模块时，可以通过设置 ***esModuleInterop*** 为 ***true*** 来避免引入时的错误。

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "lib": ["es6", "dom"],
    "outDir": "dist",
    "rootDir": "src",
    "strict": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true
  }}
```

在上面的示例中，我们设置了 ***esModuleInterop*** 和 ***allowSyntheticDefaultImports*** 属

性为 ***true***，以支持对 ES 模块的兼容性导入。

更多的 ***compilerOptions*** 配置选项可以参考 TypeScript 官方文档：Compiler Optionsopen in new window.

3. include 和 exclude

include 和 ***exclude*** 配置选项用于指定哪些文件应该包含在编译过程中，以及哪些文件应该排除在编译过程之外。

include 是一个文件或者文件夹的数组，用于指定需要编译的文件或文件夹的路径模式。

```
"include": [
  "src/**/*.ts",
  "test/**/*.ts"]
```

exclude 是一个文件或者文件夹的数组，用于指定需要排除的文件或文件夹的路径模式。

```
"exclude": [
  "node_modules",
  "dist"]
```

在上述示例中，我们将 src 文件夹和 test 文件夹下的所有 TypeScript 文件包含在编译过程中，并排除了 node_modules 文件夹和 dist 文件夹。

4. 文件引用和 composite

files 配置选项允许我们显式列出需要编译的文件路径。

```
"files": [
  "src/main.ts",
  "src/utils.ts"]
```

composite 配置选项用于启用 TypeScript 的项目引用功能，允许我们将一个 TypeScript 项目作为另一个项目的依赖。

```
"composite": true
```

5. declaration

declaration 配置选项用于生成声明文件（.d.ts 文件），它们包含了编译后 JavaScript 代码的类型信息。

```
"declaration": true
```

6. tsconfig.json 继承

TypeScript 支持通过 extends 配置选项从其他的 ***tsconfig.json*** 文件中继承配置。

```
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "outDir": "dist"
  },
  "include": [
    "src/**/*.ts"
  ]}
```

在上述示例中，我们通过 ***extends*** 指定了一个基础配置文件 ***tsconfig.base.json***，然后在当前的 ***tsconfig.json*** 中添加了额外的编译选项和文件包含规则。

7. 定制化 tsconfig.base

制化 tsconfig.base 可以让我们在多个项目中共享和复用配置，提高开发效率。下面是一些步骤来封装自己的 TSConfig 为一个库：

1）首先，我们需要创建一个新的 TypeScript 项目作为我们的库项目。

可以使用以下命令初始化一个新的项目：

```
$ mkdir my-tsconfig-lib
$ cd my-tsconfig-lib
$ npm init -y
```

2） 创建 tsconfig.json 文件

在项目根目录下创建一个名为 ***tsconfig.json*** 的文件，并将 TSConfig 的配置内容添加到其中。

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "outDir": "dist"
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]}
```

这是一个示例的 TSConfig 配置，你可以根据自己的需求进行相应的修改。

3） 创建包入口文件

为了能够在其他项目中使用我们的库，我们需要创建一个入口文件来导出我们的 TSConfig。

在项目根目录下创建一个名为 ***index.ts*** 的文件，并添加以下代码：

```
import * as tsconfig from './tsconfig.json';
export default tsconfig;
```

在上述代码中，我们将 ***tsconfig.json*** 导入为一个模块，并使用 ***export default*** 将其导出。

4） 构建和发布

现在我们可以使用 TypeScript 编译器将我们的代码构建为 JavaScript，以便在其他项目中使用。

首先，确保你已经在项目中安装了 TypeScript：

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

然后，在 ***package.json*** 中添加构建脚本：

```
{
  "scripts": {
    "build": "tsc"
  }}
```

最后，运行构建命令进行构建：

```
$ npm run build
```

构建完成后，我们的库文件将位于 ***dist*** 目录下。

5） 发布到 NPM

要将我们的 TSConfig 封装为一个库，并使其可供其他项目使用，我们可以将其发布到 NPM。

首先，创建一个 NPM 账号，并登录到 NPM：

```
$ npm login
```

然后，在项目根目录下运行以下命令发布库：

```
$ npm publish
```

发布成功后，我们的 TSConfig 库就可以在其他项目中使用了。

6） 在其他项目中使用

在其他项目中使用我们的 TSConfig 库非常简单。首先，在目标项目中安装我们的库：

```
$ npm install my-tsconfig-lib --save-dev
```

然后，在目标项目的 ***tsconfig.json*** 文件中使用我们的 TSConfig：

```
{
  "extends": "my-tsconfig-lib"}
```

通过 extends 配置选项，我们可以继承和使用我们的 TSConfig。

8. 总结

通过 tsconfig.json 文件，我们可以配置 TypeScript 编译器的行为，包括编译选项、文件包含与排除规则、输出目录等。合理配置 tsconfig.json 可以帮助我们根据项目需求进行灵活的 TypeScript 编译设置。

详细的 TypeScript 配置文件的参考信息可以在 TypeScript 官方文档中找到：tsconfig.json

➤ 实战

十七、TypeScript 封装 Fetch

1. 安装与配置 TypeScript

首先，你的电脑上安装 TypeScript。在命令行中输入以下命令：

```
npm install -g typescript
```

在你的项目根目录中，生成一个 ***tsconfig.json*** 文件来配置 TypeScript 的编译选项。在命令行中输入以下命令：

```
tsc --init
```

编辑 ***tsconfig.json*** 文件。这个文件配置了 TypeScript 的编译选项。确保以下设置已经开启：

```
{
  "compilerOptions": {
    "target": "ES2015",
    "module": "commonjs",
    "strict": true
  }}
```

2. 创建 Fetch 服务

在 ***src*** 文件夹中创建一个新的 ***FetchService.ts*** 文件。我们将在这个文件中封装 fetch API：当然，下面我们会将 put 和 delete 方法也添加到我们的 ***FetchService*** 中：

```
export class FetchService {
  async get<T>(url: string): Promise<T> {
```

```
  const response = await fetch(url);
  if (!response.ok) {
    throw new Error(response.statusText);
  }
  const data: T = await response.json();
  return data;
}

async post<T>(url: string, body: any): Promise<T> {
  const response = await fetch(url, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(body)
  });
  if (!response.ok) {
    throw new Error(response.statusText);
  }
  const data: T = await response.json();
  return data;
}

async put<T>(url: string, body: any): Promise<T> {
  const response = await fetch(url, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(body)
  });
  if (!response.ok) {
    throw new Error(response.statusText);
  }
  const data: T = await response.json();
  return data;
}
```

```
  async delete<T>(url: string): Promise<T> {
    const response = await fetch(url, {
      method: 'DELETE'
    });
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    const data: T = await response.json();
    return data;
  }}
```

这样我们就成功地创建了一个 ***FetchService*** 类，它封装了 fetch API 的 ***GET***, ***POST***, ***PUT*** 和 ***DELETE*** 方法。每个方法都返回一个 Promise，该 Promise 解析为一个泛型 ***T***，这意味着你可以指定返回数据的类型。

3. 使用 FetchService 类

```
import { FetchService } from './FetchService';
const fetchService = new FetchService();
// GET requestconst fetchData = async () => {
  try {
    const data = await fetchService.get<{ message:
string }>('https://api.example.com/data');
    console.log(data.message);
  } catch (error) {
    console.error('An error occurred:', error);
  }}
// POST requestconst sendData = async () => {
  try {
    const data = await fetchService.post<{ message:
string }>('https://api.example.com/data', {
      key: 'value'
    });
    console.log(data.message);
  } catch (error) {
```

```
    console.error('An error occurred:', error);
  }}
// PUT requestconst updateData = async () => {
  try {
    const     data     =     await     fetchService.put<{     message:
string }>('https://api.example.com/data', {
      key: 'new-value'
    });
    console.log(data.message);
  } catch (error) {
    console.error('An error occurred:', error);
  }}
// DELETE requestconst deleteData = async () => {
  try {
    const     data     =     await     fetchService.delete<{     message:
string }>('https://api.example.com/data');
    console.log(data.message);
  } catch (error) {
    console.error('An error occurred:', error);
  }}
fetchData();sendData();updateData();deleteData();
```

以上代码首先导入了我们创建的 ***FetchService*** 类，并实例化了一个新的 ***fetchService*** 对象。然后，我们定义了四个异步函数，每个函数都执行一个网络请求，并在请求成功时打印出返回数据中的 ***message*** 字段。这四个函数分别对应 ***GET***, ***POST***, ***PUT*** 和 ***DELETE*** 请求。

我们使用了 ***<{ message: string }>*** 来指定返回数据的类型，这样我们就可以得到 TypeScript 的类型检查和自动补全功能。如果你的数据类型更复杂，你可以定义一个接口来描述它，然后在这里使用那个接口。

如果请求失败，我们在 ***catch*** 块中捕获错误并打印错误消息。如果服务器返回的 HTTP 状态码不是 200-299，***fetch API*** 会认为请求成功，不会抛出错误。因此，我们在 ***FetchService*** 类的每个方法中都检查了 ***response.ok*** 属性，如果请求未成功，我们

抛出一个包含状态文本的错误。

4. 拦截器实现

在这个版本的 FetchService 中，我们把公共的请求逻辑放到了 _request 方法中。我们把方法（GET、POST、PUT、DELETE），URL 和可能的请求体传递给 _request 方法，然后它处理所有的共享逻辑，包括运行拦截器，发送请求，处理响应和解析 JSON。

```
export class FetchService {
  private requestInterceptors: Array<(url: string, options: RequestInit)
=> void> = [];
  private responseInterceptors: Array<(response: Response) => void> = [];

  async get<T>(url: string): Promise<T> {
    return this._request('GET', url);
  }

  async post<T>(url: string, body: any): Promise<T> {
    return this._request('POST', url, body);
  }

  async put<T>(url: string, body: any): Promise<T> {
    return this._request('PUT', url, body);
  }

  async delete<T>(url: string): Promise<T> {
    return this._request('DELETE', url);
  }

  addRequestInterceptor(interceptor: (url: string, options: RequestInit)
=> void) {
    this.requestInterceptors.push(interceptor);
  }

  addResponseInterceptor(interceptor: (response: Response) => void) {
    this.responseInterceptors.push(interceptor);
```

```
  }

  private async _request<T>(method: string, url: string, body?: any):
Promise<T> {
    let options: RequestInit = {
      method: method,
      headers: {
        'Content-Type': 'application/json'
      }
    };
    if (body) {
      options.body = JSON.stringify(body);
    }
    this.runRequestInterceptors(url, options);
    const response = await fetch(url, options);
    this.runResponseInterceptors(response);
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    const data: T = await response.json();
    return data;
  }

  private runRequestInterceptors(url: string, options: RequestInit) {
    this.requestInterceptors.forEach(interceptor  =>  interceptor(url,
options));
  }

  private runResponseInterceptors(response: Response) {
    this.responseInterceptors.forEach(interceptor                    =>
interceptor(response));
  }}
```

示例:

```
const fetchService = new FetchService();
```

```
// 添加一个请求拦截器
fetchService.addRequestInterceptor((url, options) => {
  options.headers = {
    ...options.headers,
    'Authorization': 'Bearer ' + localStorage.getItem('token')
  };});
// 添加一个响应拦截器
fetchService.addResponseInterceptor(response => {
  if (response.status === 401) {
    console.error('Unauthorized!');
  }});
```

5. 总结

这是一个基础的实现，其它可以根据你的需求来进行修改或扩展。

十八、TS 实战之扑克牌排序

1. 类型和转换

在线运行

我们用 ***ts 实现扑克牌排序问题***，首先，我们将定义所需的数据类型，然后专注于模式查找算法，该算法有几个有趣的要点。

1. 类型和转换

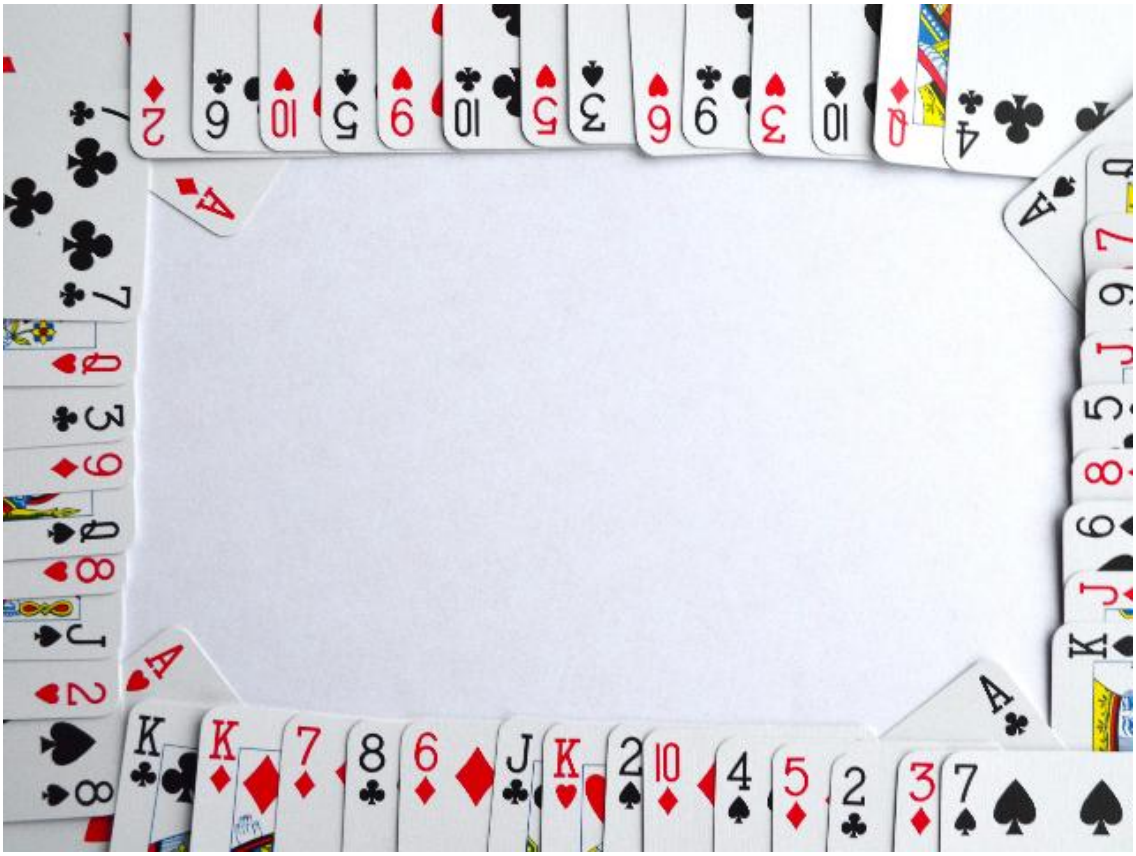
定义一些我们需要的类型。***Rank*** 和 ***Suit*** 是明显的联合类型。

```
type Rank =
 | 'A' | '2' | '3'  | '4' | '5' | '6' | '7'
 | '8' | '9' | '10' | 'J' | 'Q' | 'K'
type Suit = '♥' | '♦' | '♠' | '♣';
```

我们将使用 ***Card*** 对象进行处理，将 rank 和 suit 转换为数字。卡片将用从 1（Ace）到 13(King)的值表示，花色从 1(红心)到 4(梅花)。***rankToNumber()*** 和 ***suitToNumber()*** 函数处理从 ***Rank*** 和 ***Suit*** 值到数字的转换。

```
type Card = { rank: number; suit: number };
const rankToNumber = (rank: Rank): number =>
 rank === 'A'  ? 1
   : rank === 'J' ? 11
   : rank === 'Q' ? 12
   : rank === 'K' ? 13
   : Number(rank);
const suitToNumber = (suit: Suit): number =>
 suit === '♥' ? 1
   : suit === '♦' ? 2
   : suit === '♠' ? 3
```

```
  : /* suit === "♣" */ 4;
```



这些类型用于内部工作；我们还必须定义手牌检测算法的结果类型。我们需要一个枚举类型来表示手牌的可能值。这些值按照从最低（"高牌"）到最高（"皇家同花顺"）的顺序排列。

```
enum Hand {
 HighCard, // 高牌
 OnePair, //  一对
 TwoPairs, // 两对
 ThreeOfAKind, // 三条
 Straight, // 顺子
 Flush, // 同花
 FullHouse, //  葫芦
 FourOfAKind, // 四条
 StraightFlush, //  同花顺
 RoyalFlush //皇家同花顺}
```

2. 我们有什么手牌？

让我们首先定义我们将要构建的 ***handRank()***函数。我们的函数将接收一个包含五张牌的元组，并返回一个 ***Hand*** 结果。

```
export function handRank(
 cardStrings: [string, string, string, string, string]): Hand {
 .
 .
 .}
```

由于处理字符串比我们需要的要困难，我们将把牌字符串转换为具有数字 ***rank*** 和 ***suit*** 值的 Card 对象，以便更容易编写。

```
const cards: Card[] = cardStrings.map((str: string) => ({
  rank: rankToNumber(
    str.substring(0, str.length - 1) as Rank
  ),
  suit: suitToNumber(str.at(-1) as Suit)
}));
.
.
.
// 继续...
```

确定玩家手牌的价值的关键在于知道每个等级的牌有多少张，以及我们有多少计数。例如，如果我们有三张 J 和两张 K，J 的计数为 3，K 的计数为 2。然后，知道我们有一个计数为三和一个计数为两的计数，我们可以确定我们有一个葫芦。另一个例子：如果我们有两个 Q，两个 A 和一个 5，我们会得到两个计数为两和一个计数为一；我们有两对。

生成计数很简单。我们希望 A 的计数在 ***countByRank[1]***处，因此我们不会使用 ***countByRank*** 数组的初始位置。类似地，花色的计数将位于 ***countBySuit[1]***到 ***countBySuit[4]***之间，因此我们也不会使用该数组的初始位置。

```
// ...继续
.
.
.
const countBySuit = new Array(5).fill(0);
const countByRank = new Array(15).fill(0);
const countBySet = new Array(5).fill(0);

cards.forEach((card: Card) => {
  countByRank[card.rank]++;
  countBySuit[card.suit]++;
});
countByRank.forEach(
  (count: number) => count && countBySet[count]++
```

```
);
.
.
.
// 继续...
```

我们不要忘记 A 可能位于顺子的开头（A-2-3-4-5）或结尾（10-J-Q-K-A）。我们可以通过在 K 之后复制 Aces 计数来处理这个问题。

```
// ...继续
.
.
.
countByRank[14] = countByRank[1];
.
.
.
// 继续...
```

现在我们可以开始识别手牌了。我们只需要查看按等级计数即可识别几种手牌：

```
// ...继续
.
.
.
if (count

BySet[4] === 1 && countBySet[1] === 1)
  return Hand.FourOfAKind;
else if (countBySet[3] && countBySet[2] === 1)
  return Hand.FullHouse;
else if (countBySet[3] && countBySet[1] === 2)
  return Hand.ThreeOfAKind;
else if (countBySet[2] === 2 && countBySet[1] === 1)
  return Hand.TwoPairs;
```

```
else if (countBySet[2] === 1 && countBySet[1] === 3)
  return Hand.OnePair;
.
.
.
// 继续...
```

例如，如果有四张相同等级的牌，我们知道玩家将获得“四条”。可能会问：如果 ***countBySet[4] === 1***，为什么还要测试 ***countBySet[1] === 1?*** 如果四张牌的等级相同，应该只有一张其他牌，对吗？答案是“防御性编程”——在开发代码时，有时会出现错误，通过在测试中更加具体，有助于排查错误。

上面的情况包括了所有某个等级出现多次的可能性。我们必须处理其他情况，包括顺子、同花和“高牌”。

```
// ...继续
.
.
.
else if (countBySet[1] === 5) {
  if (countByRank.join('').includes('11111'))
    return !countBySuit.includes(5)
      ? Hand.Straight
      : countByRank.slice(10).join('') === '11111'
      ? Hand.RoyalFlush
      : Hand.StraightFlush;
  else {
    return countBySuit.includes(5)
      ? Hand.Flush
      : Hand.HighCard;
  }
} else {
  throw new Error(
    'Unknown hand! This cannot happen! Bad logic!'
  );
```

```
}
```

这里我们再次进行防御性编程；即使我们知道我们有五个不同的等级，我们也确保逻辑工作良好，甚至在出现问题时抛出一个 ***throw***。

我们如何测试顺子？我们应该有五个连续的等级。如果我们查看 ***countByRank*** 数组，它应该有五个连续的 1，所以通过执行 ***countByRank.join()*** 并检查生成的字符串是否包含 ***11111***，我们可以确定是顺子。

我们必须区分几种情况：

- 如果没有五张相同花色的牌，那么它是一个普通的顺子
- 如果所有牌都是相同花色，如果顺子以一张 A 结束，则为皇家同花顺
- 如果所有牌都是相同花色，但我们不以 A 结束，那么我们有一个同花顺

如果我们没有顺子，只有两种可能性：

- 如果所有牌都是相同花色，我们有一个同花
- 如果不是所有牌都是相同花色，我们有一个“高牌”

完整的函数如下所示：

```
export function handRank(
 cardStrings: [string, string, string, string, string]): Hand {
 const cards: Card[] = cardStrings.map((str: string) => ({
   rank: rankToNumber(
     str.substring(0, str.length - 1) as Rank
   ),
   suit: suitToNumber(str.at(-1) as Suit)
 }));

 // We won't use the [0] place in the following arrays
 const countBySuit = new Array(5).fill(0);
 const countByRank = new Array(15).fill(0);
 const countBySet = new Array(5).fill(0);

 cards.forEach((card: Card) => {
   countByRank[card.rank]++;
   countBySuit[card.suit]++;
 });
 countByRank.forEach(
   (count: number) => count && countBySet[count]++
 );

 // count the A also as a 14, for straights
 countByRank[14] = countByRank[1];

 if (countBySet[4] === 1 && countBySet[1] === 1)
   return Hand.FourOfAKind;
 else if (countBySet[3] && countBySet[2] === 1)
   return Hand.FullHouse;
 else if (countBySet[3] && countBySet[1] === 2)
   return Hand.ThreeOfAKind;
 else if (countBySet[2] === 2 && countBySet[1] === 1)
   return Hand.TwoPairs;
```

```
else if (countBySet[2] === 1 && countBySet[1] === 3)
  return Hand.OnePair;
else if (countBySet[1] === 5) {
  if (countByRank.join('').includes('11111'))
    return !countBySuit.includes(5)
      ? Hand.Straight
      : countByRank.slice(10).join('') === '11111'
      ? Hand.RoyalFlush
      : Hand.StraightFlush;
  else {
    /* !countByRank.join("").includes("11111") */
    return countBySuit.includes(5)
      ? Hand.Flush
      : Hand.HighCard;
  }
} else {
  throw new Error(
    'Unknown hand! This cannot happen! Bad logic!'
  );
}}
```

测试代码

```
console.log(handRank(['3♥',   '5♦',   '8♣',   'A♥',   '6♠'])); //
0console.log(handRank(['3♥',   '5♦',   '8♣',   'A♥',   '5♠'])); //
1console.log(handRank(['3♥',   '5♦',   '3♣',   'A♥',   '5♠'])); //
2console.log(handRank(['3♥',   '5♦',   '8♣',   '5♥',   '5♠'])); //
3console.log(handRank(['3♥',   '2♦',   'A♣',   '5♥',   '4♠'])); //
4console.log(handRank(['J♥',   '10♦',   'A♣',   'Q♥',   'K♠'])); //
4console.log(handRank(['3♥',   '4♦',   '7♣',   '5♥',   '6♠'])); //
4console.log(handRank(['3♥',   '4♥',   '9♥',   '5♥',   '6♥'])); //
5console.log(handRank(['3♥',   '5♦',   '3♣',   '5♥',   '3♠'])); //
6console.log(handRank(['3♥',   '3♦',   '3♣',   '5♥',   '3♠'])); //
7console.log(handRank(['3♥',   '4♥',   '7♥',   '5♥',   '6♥'])); //
8console.log(handRank(['K♥', 'Q♥', 'A♥', '10♥', 'J♥'])); // 9
```

在线运行