问小白

更专业的 AI 搜索 wenxiaobai.com

详细讲解以下常用的TypeScript开发方法包括：

类型注解
接口（Interface）
泛型（Generics）
类型守卫（Type Guards）
联合类型（Union Types）
类型别名（Type Aliases）
解构赋值与类型
函数类型
类与继承
模块化

TypeScript 常用开发方法详细讲解

1. 类型注解 (Type Annotations)

基础类型注解

TypeScript

```
// 字符串类型
let username: string = ”张三”;
let greeting: string = `Hello, ${username}!`; // 模板字符串

// 数字类型
let age: number = 25;
let price: number = 99.99;
let binary: number = 0b1010; // 二进制
let octal: number = 0o744;  // 八进制
let hex: number = 0xf00d;   // 十六进制

// 布尔类型
let isActive: boolean = true;
let hasPermission: boolean = false;

// 空值类型
let undefinedVar: undefined = undefined;
let nullVar: null = null;

// 任何类型（慎用）
let anything: any = ”可以是任意类型”;
anything = 42;
anything = true;
```

数组和元组类型注解

TypeScript

```
// 数组类型
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: Array<string> = [”a”, ”b”, ”c”]; // 泛型语法

// 多维数组
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

// 元组类型 - 固定长度和类型的数组
let person: [string, number, boolean] = [”张三”, 25, true];

// 只读数组和元组
const readOnlyNumbers: ReadonlyArray<number> = [1, 2, 3];
// readOnlyNumbers.push(4); // 错误！只读数组不能修改

// 可选元素的元组
let optionalTuple: [string, number?] = [”hello”];
optionalTuple = [”world”, 42]; // 也可以有两个元素
```

对象类型注解

TypeScript

```
// 内联对象类型注解
let user: { name: string; age: number; email?: string } = {
  name: ”李四”,
  age: 30
  // email 是可选的，可以不提供
};

// 使用接口定义对象类型（后面会详细讲）
interface Point {
  x: number;
  y: number;
}

let point: Point = { x: 10, y: 20 };
```

2. 接口 (Interface)

基础接口定义

TypeScript

```
// 定义对象形状
interface User {
  readonly id: number;     // 只读属性，初始化后不能修改
  name: string;
  age: number;
  email?: string;        // 可选属性
  [key: string]: any;     // 索引签名，允许其他属性
}

// 实现接口
const user1: User = {
  id: 1,
  name: ”张三”,
  age: 25,
  // 由于有索引签名，可以添加任意额外属性
  nickname: ”小张”,
  avatar: ”url-to-image”
};

// user1.id = 2; // 错误！ id是只读的
```

接口的扩展和继承

TypeScript

```
// 基础接口
interface Person {
  name: string;
  age: number;
}

// 接口继承
interface Employee extends Person {
  employeeId: string;
  department: string;
  salary?: number;
}

// 多接口继承
interface Address {
  street: string;
  city: string;
  zipCode: string;
}

// 多重继承
interface ContactInfo extends Person, Address {
  phone: string;
  email: string;
}

const employee: Employee = {
  name: ”王五”,
  age: 35,
  employeeId: ”E001”,
  department: ”技术部”,
  street: ”人民路”,
  city: ”北京”,
  zipCode: ”100000”,
  phone: ”13800138000”,
  email: ”wangwu@example.com”
};
```

函数类型接口

TypeScript

```
// 定义函数接口
interface SearchFunc {
  (source: string, keyword: string): boolean;
}

// 实现函数接口
const mySearch: SearchFunc = function(src: string, kw: string): boolean {
  return src.includes(kw);
};

// 调用签名的接口
interface StringTransform {
  (input: string): string;
  description: string; // 还可以包含其他属性
}

const toUpper: StringTransform = (input: string) => input.toUpperCase();
toUpper.description = ”转换为大写”;
```

可索引接口

TypeScript

```
// 数组-like 接口
interface StringArray {
  [index: number]: string;
}

const myArray: StringArray = [”a”, ”b”, ”c”];
```

3. 泛型 (Generics)

泛型函数

TypeScript

```
// 基本泛型函数
function identity<T>(arg: T): T {
  return arg;
}

// 使用
let output1 = identity<string>(”myString”); // 显式指定类型
let output2 = identity(42); // 类型推断 - T 被推断为 number
```

泛型接口

TypeScript

```typescript
// 泛型接口
interface GenericResponse<T> {
 success: boolean;
   data: T;
 message?: string;
}

// 使用泛型接口
const numberResponse: GenericResponse<number> = {
 success: true,
 data: 100
};

// 多个类型参数的泛型接口
interface Pair<K, V> {
 key: K;
   value: V;
}

// 实现
const pair1: Pair<string, number> = {
 key: ”score”,
 value: 95
};
```

泛型类

TypeScript

```typescript
// 泛型类
class GenericNumber<T> {
 zeroValue: T;
 add: (x: T, y: T) => T;
}

// 使用泛型类
const numberInstance = new GenericNumber<number>();
numberInstance.zeroValue = 0;
numberInstance.add = (x, y) => x + y;
```

泛型约束

TypeScript

```
// 使用 extends 约束泛型
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // 现在我们知道 arg 有 length 属性
  return arg;
}

// 正确的使用
loggingIdentity(”hello”); // string 有 length 属性
// loggingIdentity(3); // 错误！ number 没有 length 属性
```

高级泛型约束

TypeScript

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

// 在泛型约束中使用类型参数
function getProperty2<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const obj = { a: 1, b: 2, c: 3 };
getProperty(obj, ”a”); // 正确
// getProperty(obj, ”m”); // 错误！ m 不在 obj 的键中
```

4. 类型守卫 (Type Guards)

typeof 类型守卫

TypeScript

```typescript
function padLeft(value: string, padding: string | number) {
  if (typeof padding === ”number”) {
    return Array(padding + 1).join(” ”) + value;
      }
  return padding + value;
}

// 使用
console.log(padLeft(”Hello”, 4));   // ”    Hello”
console.log(padLeft(”Hello”, ”    ”)); // ”    Hello”
```

instanceof 类型守卫

TypeScript

```typescript
class Bird {
  fly() {
    console.log(”flying”);
      }
  layEggs() {
    console.log(”laying eggs”);
      }
}

class Fish {
  swim() {
    console.log(”swimming”);
      }
  layEggs() {
    console.log(”laying fish eggs”);
      }
}

function getRandomPet(): Bird | Fish {
  return Math.random() > 0.5 ? new Bird() : new Fish();
}

function move(pet: Bird | Fish) {
  if (pet instanceof Bird) {
    pet.fly(); // TypeScript 知道这里 pet 是 Bird
  } else {
    pet.swim(); // TypeScript 知道这里 pet 是 Fish
      }
}
```

自定义类型守卫

TypeScript

```
// 使用类型谓词
function isFish(pet: Bird | Fish): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

// 使用自定义类型守卫
const pet = getRandomPet();

if (isFish(pet)) {
  pet.swim(); // TypeScript 知道 pet 是 Fish
} else {
  pet.fly(); // TypeScript 知道 pet 是 Bird
}
```

in 操作符类型守卫

TypeScript

```
interface Circle {
  kind: ”circle”;
  radius: number;
}

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

function calculateArea(shape: Circle | Square): number {
  if (”radius” in shape) {
    // TypeScript 知道这里是 Circle
    return Math.PI * shape.radius ** 2;
  }
}

function calculatePerimeter(shape: Circle | Square): number {
  if (shape.kind === ”circle”) {
    return 2 * Math.PI * shape.radius;
  } else {
    // TypeScript 知道这里是 Square
    return 4 * shape.sideLength;
}
```

5. 联合类型 (Union Types)

基础联合类型

TypeScript

```
// 简单联合类型
let id: string | number = ”ABC123”;
id = 456; // 合法

// 更复杂的联合类型
interface Car {
  type: ”car”;
  wheels: number;
  brand: string;
}

interface Bike {
  type: ”bike”;
  wheels: number;
  frame: string;
}

// 使用联合类型
function printVehicle(vehicle: Car | Bike) {
  console.log(vehicle.type);
  // 只能访问共同属性
  // console.log(shape.wheels); // 正确，两者都有 wheels
}
```

可辨识联合 (Discriminated Unions)

TypeScript

```
// 每个接口都有一个共同的 discriminant 属性
interface Square {
 kind: ”square”;
 size: number;
}

interface Rectangle {
 kind: ”rectangle”;
 width: number;
 height: number;
}

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

type Shape = Square | Rectangle | Triangle;

function getArea(shape: Shape): number {
 switch (shape.kind) {
  case ”square”:
   return shape.size * shape.size;
  case ”rectangle”:
   return shape.width * shape.height;
  case ”circle”:
   return Math.PI * shape.radius ** 2;
   }
}
```

6. 类型别名 (Type Aliases)

基础类型别名

TypeScript

```
// 为联合类型创建别名
type ID = string | number;
type Direction = ”left” | ”right” | ”up” | ”down”;

// 使用
let userId: ID = ”user_123”;
let productId: ID = 456;

// 复杂的类型别名
type Coordinates = [number, number, number?]; // 二维或三维坐标

type UserRole = ”admin” | ”user” | ”guest”;

// 对象类型别名
type UserProfile = {
 username: string;
 email: string;
  role: UserRole;
};
```

泛型类型别名

TypeScript

```
// 泛型类型别名
type Container<T> = { value: T };

// 使用
const numberContainer: Container<number> = { value: 42 };
```

7. 解构赋值与类型

对象解构与类型

TypeScript

```
// 对象解构
interface Person {
 name: string;
 age: number;
   address?: {
  city: string;
  country: string;
   };
}

// 带类型的对象解构
function printPerson({ name, age }: Person) {
 console.log(`${name} is ${age} years old”);
}

// 解构时重命名并指定类型
function processUser({ name: userName, age: userAge }: Person) {
 console.log(`User: ${userName}, Age: ${userAge});
}

// 数组解构
const numbers = [1, 2, 3, 4, 5];

// 数组解构并指定类型
const [first, second, ...rest]: number[] = numbers;
```

8. 函数类型

函数声明类型

```TypeScript
// 函数类型注解
function add(x: number, y: number): number {
 return x + y;
}

// 函数表达式类型
const multiply: (x: number, y: number) => number = function(x, y) {
 return x * y;
};
```

可选参数和默认参数

TypeScript

```
// 可选参数
function buildName(firstName: string, lastName?: string): string {
  return lastName ? `${firstName} ${lastName}` : firstName;
}

// 默认参数
function greet(name: string = ”World”): string {
  return `Hello, ${name}!`;
}

// 剩余参数
function sum(...numbers: number[]): number {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}
```

函数重载

TypeScript

```
// 函数重载签名
function reverse(x: string): string;
function reverse(x: number): number;
function reverse(x: string | number): string | number {
  if (typeof x === ”string”) {
    return x.split(””).reverse().join(””);
}

// 实现
function reverse(x: string | number): string | number {
  if (typeof x === ”string”) {
    return x.split(””).reverse().join(””);
  } else {
    return Number(x.toString().split(””).reverse().join(””));
}
```

9. 类与继承

基础类

TypeScript

```
class Animal {
 // 属性
 name: string;

 // 构造函数
 constructor(name: string) {
  this.name = name;
   }

 // 方法
 move(distance: number = 0) {
  console.log(`${this.name} moved ${distance}m.`);
   }
}
```

访问修饰符

TypeScript

```
class Person {
 // public (默认)
 public name: string;

 // private - 只能在类内部访问
 private ssn: string;

 // protected - 只能在类和子类中访问
 protected age: number;

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

 // protected 方法
 protected getDetails(): string {
  return `${this.name}, ${this.age} years old”);
 }
}
```

继承

TypeScript

```
class Employee extends Person {
 private employeeId: string;

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

 // 重写方法
 getDetails(): string {
  return `${super.getDetails()} (ID: ${this.employeeId})`;
   }
}
```

抽象类

TypeScript

```
abstract class Department {
 constructor(public name: string) {}

 // 抽象方法 - 必须在子类中实现
 abstract meeting(): void;
}

class ITDepartment extends Department {
 meeting(): void {
  console.log(”IT Department meeting”);
   }
}
```

10. 模块化

导出声明

TypeScript

```
// math.ts
export const PI = 3.14;

export function calculateCircumference(diameter: number): number {
 return diameter * PI;
}
```

导入模块

TypeScript

```
// app.ts
import { PI, calculateCircumference } from './math';

// 默认导出
export default class Calculator {
  // ...
}

// 重新导出
export { PI as CirclePI } from './math';
```

命名空间

```TypeScript
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;

  export class LettersOnlyValidator implementsValidator implements StringValidator {
    isAcceptable(s: string): boolean {
    return /^[A-Za-z]+$/.test(s);
     }
}
```

这些是 TypeScript 开发中最常用的方法和概念。掌握这些将帮助你编写类型安全、可维护的代码。

(注:文档部分内容可能由AI生成)