TypeScript 教材文档

1. 引言

1.1 什么是TypeScript?

TypeScript 是一种由微软开发的开源编程语言，它是 JavaScript 的一个超集，这意味着任何合法的 JavaScript 代码都是合法的 TypeScript 代码。TypeScript 在 JavaScript 的基础上添加了静态类型定义，旨在帮助开发者构建更健壮、可维护的大型应用程序。通过引入类型系统，TypeScript 可以在代码运行前（即编译阶段）捕获潜在的错误，从而提高开发效率和代码质量。

1.2 为什么学习TypeScript?

学习 TypeScript 具有多方面的优势：

1. **提高代码质量和可维护性**：类型系统使得代码结构更清晰，易于理解和重构。在大型项目中，类型检查可以有效减少运行时错误，降低维护成本。
2. **增强开发效率**：IDE（集成开发环境）可以利用类型信息提供更智能的代码补全、错误提示和重构功能，显著提升开发体验。开发者可以更快地发现并修复错误，减少调试时间。
3. **更好的团队协作**：类型定义作为代码的契约，使得团队成员之间更容易理解和协作。当接口或数据结构发生变化时，类型系统能够及时发出警告，避免潜在的兼容性问题。
4. **拥抱最新JavaScript特性**：TypeScript 支持最新的 ECMAScript 标准（如 ES6/ES2015及更高版本）的特性，并将其编译为兼容不同环境的 JavaScript 代码，让开发者可以提前使用未来的 JavaScript 语法。
5. **广泛的生态系统支持**：越来越多的流行框架和库（如 React, Vue, Angular, Node.js）都提供了对 TypeScript 的良好支持，使得 TypeScript 在前端和后端开发中都得到了广泛应用。

1.3 TypeScript的优势

TypeScript 相较于纯 JavaScript 的主要优势体现在以下几个方面：

- **静态类型检查**：这是 TypeScript 最核心的优势。它允许你在编写代码时定义变量、函数参数和返回值的类型，并在编译时进行检查。这有助于在早期发现类型不匹配的错误，避免在运行时才暴露问题。
- **代码可读性与可理解性**：类型注解提供了代码的额外文档，使得其他开发者（或未来的你）更容易理解代码的意图和数据流。
- **强大的工具支持**：由于拥有类型信息，现代 IDE（如 VS Code）能够提供卓越的开发体验，包括：
    - **智能代码补全 (IntelliSense)**：根据类型信息提供准确的属性和方法建议。
    - **即时错误检测**：在保存文件前就能发现潜在的类型错误。
    - **代码重构**：安全地进行变量重命名、函数签名修改等操作。
    - **导航功能**：轻松跳转到定义、查找引用。
- **面向对象编程特性**：TypeScript 提供了类、接口、继承、抽象类、访问修饰符等面向对象编程的完整支持，使得开发者可以使用更结构化的方式组织代码。
- **更好的可伸缩性**：对于大型复杂的项目，TypeScript 的类型系统能够提供更好的结构化和模块化能力，使得项目更容易扩展和维护。
- **与JavaScript的互操作性**：TypeScript 可以与现有的 JavaScript 代码无缝集成，你可以逐步将 JavaScript 项目迁移到 TypeScript，或者在 TypeScript 项目中引用 JavaScript 库。

总之，TypeScript 通过引入类型系统和提供丰富的语言特性，极大地提升了 JavaScript 的开发体验和代码质量，是现代 Web 开发中不可或缺的工具。

2. TypeScript基础

2.1 环境搭建与第一个TypeScript程序

学习 TypeScript 的第一步是搭建开发环境。TypeScript 代码最终会被编译成 JavaScript 代码，因此我们需要安装 TypeScript 编译器。

步骤 1：安装 Node.js

TypeScript 编译器 `tsc` 是一个 Node.js 包，所以首先需要安装 Node.js。您可以访问 Node.js 官方网站 下载并安装适合您操作系统的版本。安装完成后，在命令行中运行以下命令验证安装是否成功：

```
node -v
npm -v
```

如果能正确显示版本号，则表示 Node.js 和 npm（Node.js 包管理器）已成功安装。

步骤 2：安装 TypeScript 编译器

使用 npm 全局安装 TypeScript 编译器：

```
npm install -g typescript
```

安装完成后，可以通过以下命令验证 `tsc` 是否可用：

```
tsc -v
```

步骤 3：编写第一个 TypeScript 程序

创建一个名为 `hello.ts` 的文件（`.ts` 是 TypeScript 文件的扩展名），并输入以下代码：

```
// hello.ts
function greet(person: string) {
  return "Hello, " + person;
}

let user = "TypeScript";
console.log(greet(user));
```

这段代码定义了一个 `greet` 函数，它接受一个 `string` 类型的参数 `person`，并返回一个问候字符串。我们声明了一个 `user` 变量并赋值为 "TypeScript"，然后调用 `greet` 函数并打印结果。

步骤 4：编译和运行 TypeScript 程序

在命令行中，导航到 `hello.ts` 文件所在的目录，然后运行 TypeScript 编译器：

```
tsc hello.ts
```

如果编译成功，`tsc` 命令会在同一目录下生成一个 `hello.js` 文件，这是编译后的 JavaScript 代码：

```
// hello.js
function greet(person) {
    return "Hello, " + person;
}
var user = "TypeScript";
console.log(greet(user));
```

现在，您可以使用 Node.js 运行这个编译后的 JavaScript 文件：

```
node hello.js
```

您将在控制台看到输出：`Hello, TypeScript`。

2.2 基本语法

TypeScript 的基本语法与 JavaScript 大致相同，因为它本身就是 JavaScript 的超集。这意味着所有合法的 JavaScript 语法在 TypeScript 中都是合法的。这里我们主要关注 TypeScript 引入的类型注解部分。

变量声明

在 TypeScript 中，你可以在声明变量时为其指定类型。这被称为**类型注解**。

```
let age: number = 30; // 声明一个数字类型的变量
let name: string = "Alice"; // 声明一个字符串类型的变量
let isActive: boolean = true; // 声明一个布尔类型的变量
```

如果你尝试将不匹配类型的值赋给变量，TypeScript 编译器会报错：

```
let age: number = 30;
age = "forty"; // 错误：不能将类型“string”分配给类型“number”。
```

函数

函数参数和返回值也可以进行类型注解：

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

let result = add(10, 20); // result 的类型被推断为 number
console.log(result);

// 错误示例
function multiply(a: number, b: number): number {
  return a * b;
}

// 错误：类型“string”的参数不能赋给类型“number”的参数。
// multiply("hello", 5);
```

对象

你可以使用接口（Interface）或类型别名（Type Alias）来定义对象的结构和类型：

```
// 使用接口定义对象类型
interface Person {
  name: string;
  age: number;
}

let person1: Person = {
  name: "Bob",
  age: 25
};

// 错误：缺少属性“age”，类型“{ name: string; }”的参数不能赋给类型“Person”的参数。
// let person2: Person = { name: "Charlie" };

// 使用类型别名定义对象类型
type Car = {
  brand: string;
  year: number;
};

let myCar: Car = {
  brand: "Toyota",
  year: 2020
};
```

2.3 变量声明 (let, const, var)

TypeScript 支持 JavaScript 中的所有变量声明方式：`var`、`let` 和 `const`。在 TypeScript 中，我们通常推荐使用 `let` 和 `const`，因为它们提供了更好的块级作用域和更清晰的变量生命周期。

- `var`：

- `var` 声明的变量作用域是函数作用域或全局作用域。
- 存在变量提升（hoisting）问题，可能导致意外行为。
- 可以重复声明，后声明的会覆盖前声明的。

`typescript function exampleVar() { var x = 10; if (true) { var x = 20; // 重新声明，覆盖了外部的 x console.log(x); // 输出 20 } console.log(x); // 输出 20 } exampleVar();`

- **`let`**：
  - `let` 声明的变量作用域是块级作用域（`{}` 内部）。
  - 不存在变量提升，必须先声明后使用。
  - 不能在同一作用域内重复声明。

  ```typescript function exampleLet() { let y = 10; if (true) { let y = 20; // 这是一个新的变量 y，只在 if 块内有效 console.log(y); // 输出 20 } console.log(y); // 输出 10 } exampleLet();

  // let z = 10; // let z = 20; // 错误：不能在块范围中重新声明“z”。 ```
- **`const`**：
  - `const` 声明的变量也是块级作用域。
  - 用于声明常量，一旦赋值后不能再修改其值。
  - 必须在声明时进行初始化。

  ```typescript const PI = 3.14159; // PI = 3.14; // 错误：不能为“PI”赋值，因为它是一个常量。

  const obj = { name: "Alice" }; obj.name = "Bob"; // 允许：可以修改对象内部的属性 // obj = { name: "Charlie" }; // 错误：不能为“obj”赋值，因为它是一个常量。 ```

在 TypeScript 中，推荐优先使用 `const`，如果变量需要重新赋值，则使用 `let`。尽量避免使用 `var`，以减少潜在的错误和提高代码可读性。

## 2.4 数据类型

TypeScript 提供了多种内置数据类型，以及允许你自定义复杂类型。理解这些类型是掌握 TypeScript 的关键。

### 2.4.1 原始类型 (number, string, boolean, symbol, bigint)

TypeScript 支持 JavaScript 中的所有原始数据类型：

- **`number`**：表示浮点数。除了支持十进制和十六进制字面量，TypeScript 还支持 ES6 中的二进制和八进制字面量。`typescript let decimal: number = 6; let hex: number = 0xf00d; let binary: number = 0b1010; let octal: number = 0o744;`
- **`string`**：表示文本数据。可以使用单引号、双引号或模板字符串（反引号）来定义字符串。```typescript let color: string = "blue"; color = 'red';

  let fullName: string = `Bob Smith`; let age: number = 30; let sentence: string = `Hello, my name is ${ fullName }. I'll be ${ age + 1 } years old next month.`;```
- **`boolean`**：表示逻辑值，只有 `true` 和 `false` 两个值。`typescript let isDone: boolean = false;`
- **`symbol`** (ES6 新增)：表示独一无二的值。Symbol 值通过 `Symbol()` 函数生成。`typescript let sym1: symbol = Symbol('key'); let sym2: symbol = Symbol('key'); console.log(sym1 === sym2); // false`
- **`bigint`** (ES2020 新增)：表示任意精度的整数。`BigInt` 可以表示比 `number` 类型支持的最大整数更大的整数。`typescript let bigNumber: bigint = 100n; // 使用 n 后缀表示 BigInt let anotherBigNumber: bigint = BigInt(Number.MAX_SAFE_INTEGER) + 1n; console.log(anotherBigNumber); // 9007199254740992n`

### 2.4.2 any 类型

`any` 类型是 TypeScript 类型系统中的一个特殊类型，它表示可以是任何类型。当一个变量被声明为 `any` 类型时，你可以给它赋任何类型的值，并且可以对它执行任何操作，而不会触发类型检查错误。这在以下情况下非常有用：

- **不确定类型时**：当你不知道变量的类型，或者变量的类型可能在运行时发生变化时。
- **兼容旧的 JavaScript 代码**：在将现有 JavaScript 项目逐步迁移到 TypeScript 时，可以使用 `any` 类型来处理那些暂时不想或无法进行类型注解的代码。

- **与第三方库交互**：当使用的第三方库没有提供类型声明文件时，可以使用 `any` 类型来避免类型错误。

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

notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let list: any[] = [1, true, "free"];
list[1] = 100;
```

尽管 `any` 类型提供了很大的灵活性，但过度使用它会失去 TypeScript 带来的类型检查优势，使得代码变得和纯 JavaScript 一样，容易引入运行时错误。因此，应尽量避免使用 `any`，除非确实有必要。

2.4.3 void, never, null, undefined 类型

- **`void`**：
  - `void` 表示没有任何类型。当一个函数没有返回值时，其返回类型通常被推断为 `void`。
  - 声明一个 `void` 类型的变量是没有意义的，因为你只能给它赋值 `undefined` 或 `null`（在 `--strictNullChecks` 未启用时）。

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

  let unusable: void = undefined; // unusable = null; // 启用 --strictNullChecks 时会报错 ```

- **`never`**：
  - `never` 类型表示那些永不存在的值的类型。例如，总是抛出异常的函数、或者永不返回的函数（例如无限循环的函数）的返回类型就是 `never`。
  - `never` 是任何类型的子类型，可以赋值给任何类型。然而，没有类型是 `never` 的子类型（除了 `never` 本身），这意味着 `never` 类型的变量不能被赋值为任何其他类型的值。

  ```typescript // 返回 never 的函数必须存在无法达到的终点 function error(message: string): never { throw new Error(message); }

// 推断的返回类型为 never function infiniteLoop(): never { while (true) {} }

let x: number = 1; // x = error("Something went wrong"); // 允许，因为 never 是所有类型的子类型 // let y: never = 1; // 错误：不能将类型“number”分配给类型“never”。 ```

- **`null` 和 `undefined`**：
  - 在 TypeScript 中，`null` 和 `undefined` 各自拥有自己的类型，分别为 `Null` 和 `Undefined`。
  - 默认情况下，`null` 和 `undefined` 是所有其他类型的子类型。这意味着你可以将 `null` 和 `undefined` 赋值给 `number`、`string` 等类型的变量。
  - 然而，当你在 `tsconfig.json` 中启用 `--strictNullChecks` 编译选项时，`null` 和 `undefined` 只能赋值给 `void`、`any` 或它们各自的类型（以及联合类型中包含 `null` 或 `undefined` 的情况）。这有助于更严格地检查空值引用错误。

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

let num: number = 1; // num = undefined; // 启用 --strictNullChecks 时会报错 // num = null; // 启用 --strictNullChecks 时会报错

let str: string | null = "hello"; // 联合类型，允许为 null str = null; ```

## 2.5 类型推断

TypeScript 的一个重要特性是**类型推断**。在很多情况下，即使你没有明确地为变量、函数参数或返回值指定类型，TypeScript 编译器也能够根据上下文自动推断出它们的类型。这使得 TypeScript 代码在保持类型安全的同时，可以像 JavaScript 一样简洁。

### 变量的类型推断

当你在声明变量时直接进行初始化，TypeScript 会根据初始值的类型来推断变量的类型：

```
let num = 10; // num 被推断为 number 类型
// num = "hello"; // 错误：不能将类型“string”分配给类型“number”。

let greeting = "Hello, world!"; // greeting 被推断为 string 类型
// greeting = 123; // 错误：不能将类型“number”分配给类型“string”。

let isReady = true; // isReady 被推断为 boolean 类型
```
```

如果声明变量时没有初始化，那么它会被推断为 `any` 类型。这允许你稍后给它赋任何类型的值，但同时也失去了类型检查的优势。

```
let unknownValue; // unknownValue 被推断为 any 类型
unknownValue = 10;
unknownValue = "some text";
unknownValue = {};
```

函数的类型推断

TypeScript 也能推断函数的返回类型，以及在某些情况下推断函数参数的类型（例如，在回调函数中）。

```
function add(x: number, y: number) { // 返回值被推断为 number 类型
  return x + y;
}

let sum = add(5, 3); // sum 被推断为 number 类型

// 匿名函数的类型推断
let names = ["Alice", "Bob", "Charlie"];
names.forEach(function(name) { // name 参数被推断为 string 类型
  console.log(name.toUpperCase());
});
```

对象的类型推断

当你创建对象字面量时，TypeScript 会根据对象的属性及其值的类型来推断对象的类型：

```
let point = { x: 10, y: 20 }; // point 被推断为 { x: number; y: number; } 类型
// point.x = "hello"; // 错误：不能将类型“string”分配给类型“number”。

let userProfile = {
  id: 1,
  name: "David",
  email: "david@example.com"
}; // userProfile 被推断为 { id: number; name: string; email: string; } 类型
```

何时需要明确类型注解?

尽管类型推断很强大，但在以下情况下，明确的类型注解仍然是推荐的：

1. **函数参数**：函数的参数通常需要明确的类型注解，因为编译器无法从函数体外部推断它们的类型。
2. **函数返回值**：对于复杂的函数，明确指定返回类型可以确保函数始终返回预期类型的值，这对于维护和调试非常有帮助。

3. **变量声明但未初始化**：如果你声明了一个变量但没有立即初始化，并且不希望它被推断为 `any` 类型，那么你需要明确指定其类型。
4. **复杂类型**：对于对象、数组或元组等复杂类型，明确的类型注解可以提高代码的可读性和可维护性。
5. **类型断言或类型守卫**：在需要进行类型转换或缩小类型范围时，明确的类型注解是必不可少的。

通过合理利用类型推断和明确类型注解，可以编写出既简洁又类型安全的 TypeScript 代码。

3. TypeScript类型系统

TypeScript 的核心优势在于其强大的类型系统。它允许开发者在代码中定义各种数据结构和行为的类型，从而在编译阶段捕获潜在的错误，提高代码的健壮性和可维护性。本节将详细介绍 TypeScript 中各种重要的类型。

3.1 接口 (Interface)

接口（Interface）是 TypeScript 中定义对象结构的一种方式。它定义了对象应该具有哪些属性和方法，以及它们的类型。接口只定义了“形状”，而不包含具体的实现，因此它在编译成 JavaScript 后会被完全擦除。

3.1.1 对象的形状

最常见的用法是定义一个对象必须符合的结构。例如，我们可以定义一个 `Person` 接口，要求对象必须有 `name` 和 `age` 属性：

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

let tom: Person = {
  name: 'Tom',
  age: 25
};

// 错误：类型“{ name: string; }”的参数不能赋给类型“Person”的参数。缺少属性“age”。
// let jerry: Person = {
//   name: 'Jerry'
// };

// 错误：对象文字可以只指定已知属性，并且“gender”不在类型“Person”中。
// let spike: Person = {
//   name: 'Spike',
//   age: 30,
//   gender: 'male' // 不允许出现未定义的属性
// };
```

3.1.2 可选属性与只读属性

- **可选属性**：使用 `?` 符号表示该属性是可选的，可以存在也可以不存在。

  ```typescript interface Person { name: string; age?: number; // age 属性是可选的 }

  let tom: Person = { name: 'Tom' };

  let jerry: Person = { name: 'Jerry', age: 30 }; ```

- **只读属性**：使用 `readonly` 关键字表示该属性只能在对象创建时被赋值，之后不能再修改。

  ```typescript interface Point { readonly x: number; readonly y: number; }

  let p1: Point = { x: 10, y: 20 }; // p1.x = 5; // 错误：无法为“x”赋值，因为它是只读属性。 ```

### 3.1.3 任意属性

有时我们希望一个接口可以拥有任意数量的属性，但这些属性的类型是确定的。可以使用索引签名（Index Signatures）来定义任意属性：

```ts
interface StringArray {
  [index: number]: string; // 任意数字索引的属性都是 string 类型
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];

interface StringDictionary {
  [propName: string]: string; // 任意字符串索引的属性都是 string 类型
}

let myDict: StringDictionary = {
  name: "Alice",
  city: "New York"
};
```

需要注意的是，如果接口中定义了确定属性，那么这些确定属性的类型必须与任意属性的类型兼容。

3.1.4 接口的继承

接口可以像类一样互相继承，从而扩展或组合已有的接口。一个接口可以继承多个接口。

```ts
interface Shape {
  color: string;
}

interface Square extends Shape {
  sideLength: number;
}

let square: Square = {
  color: "blue",
  sideLength: 10
};

interface Pen {
  brand: string;
}

interface DrawingTool extends Square, Pen {
  thickness: number;
}

let tool: DrawingTool = {
  color: "red",
  sideLength: 5,
  brand: "Pilot",
  thickness: 0.5
};
```

3.2 类 (Class)

TypeScript 完全支持 ES6 中的类（Class）语法，并在此基础上增加了类型注解、访问修饰符等特性，使得面向对象编程更加强大和严谨。

3.2.1 类的定义与实例化

类是创建对象的蓝图。它包含属性（数据）和方法（行为）。

```
class Greeter {
  greeting: string; // 属性

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

  greet() { // 方法
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world"); // 实例化类
console.log(greeter.greet()); // 输出: Hello, world
```

3.2.2 继承与多态

类可以通过 `extends` 关键字实现继承，子类可以继承父类的属性和方法。多态性允许不同类的对象对同一消息作出不同的响应。

```
class Animal {
  name: string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino"); // 多态：tom 是 Animal 类型，但实际是 Horse 对象

sam.move(); // Slithering... Sammy the Python moved 5m.
tom.move(34); // Galloping... Tommy the Palomino moved 34m.
```

3.2.3 抽象类

抽象类是专门用于被继承的类，它不能被直接实例化。抽象类可以包含抽象方法，抽象方法不包含具体的实现，必须在派生类中实现。

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

  printName(): void {
    console.log('Department name: ' + this.name);
  }

  abstract printMeeting(): void; // 必须在派生类中实现
}

class AccountingDepartment extends Department {
  constructor() {
    super('Accounting and Auditing'); // 在派生类的构造函数中调用基类的构造函数
  }

  printMeeting(): void {
    console.log('The Accounting Department meets each Monday at 10am.');
  }

  generateReports(): void {
    console.log('Generating accounting reports...');
  }
}

// let department = new Department(); // 错误：不能创建抽象类的实例

let department: Department; // 允许创建对抽象类型的引用
department = new AccountingDepartment();
department.printName();
department.printMeeting();
// department.generateReports(); // 错误：不能在抽象类型上调用具体方法
```

3.2.4 访问修饰符 (public, private, protected)

TypeScript 提供了三种访问修饰符来控制类成员的可见性：

- **`public`** (默认)：修饰的成员在类的内部、子类和类的外部都可以访问。
- **`private`**：修饰的成员只能在类的内部访问，不能在子类和类的外部访问。
- **`protected`**：修饰的成员可以在类的内部和子类中访问，但不能在类的外部访问。

```typescript
class Animal {
  public name: string; // public 属性
  private age: number; // private 属性
  protected species: string; // protected 属性

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

  public getAge() {
    return this.age; // 可以在类内部访问 private 属性
  }
}

class Dog extends Animal {
  constructor(name: string, age: number) {
    super(name, age, "Canine");
    console.log(this.species); // 可以在子类中访问 protected 属性
  }

  bark() {
    // console.log(this.age); // 错误：不能在子类中访问 private 属性
    console.log(`${this.name} barks!`);
  }
}

let animal = new Animal("Generic Animal", 5, "Unknown");
console.log(animal.name); // 可以在外部访问 public 属性
// console.log(animal.age); // 错误：不能在外部访问 private 属性
// console.log(animal.species); // 错误：不能在外部访问 protected 属性

let dog = new Dog("Buddy", 3);
dog.bark();
```

3.3 函数类型

在 TypeScript 中，我们可以为函数定义类型，包括参数类型和返回值类型，这有助于确保函数调用的正确性。

3.3.1 函数声明与表达式

- **函数声明**：

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

- **函数表达式**：

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

// 也可以使用接口定义函数类型 interface SearchFunc { (source: string, subString: string): boolean; }

let mySearch: SearchFunc; mySearch = function(source: string, subString: string) { return source.search(subString) !== -1; }; ```

3.3.2 可选参数与默认参数

- **可选参数**：使用 `?` 符号表示参数是可选的，可选参数必须在必选参数之后。

  ```typescript function buildName(firstName: string, lastName?: string) { if (lastName) { return firstName + " " + lastName; } else { return firstName; } }

  let result1 = buildName("Bob"); // 正确 let result2 = buildName("Bob", "Adams"); // 正确 // let result3 = buildName("Bob", "Adams", "Sr."); // 错误：参数过多 ```

- **默认参数**：为参数提供默认值，当参数未传递或为 `undefined` 时，使用默认值。默认参数可以放在必选参数之前或之后。

  ```typescript function buildName(firstName: string, lastName = "Smith") { return firstName + " " + lastName; }

  let result1 = buildName("Bob"); // Bob Smith let result2 = buildName("Bob", undefined); // Bob Smith let result3 = buildName("Bob", "Adams"); // Bob Adams ```

### 3.3.3 剩余参数

当函数需要处理不定数量的参数时，可以使用剩余参数（Rest Parameters）。剩余参数会被当作一个数组。

```
function sumNumbers(firstNumber: number, ...restOfNumbers: number[]): number {
  let total = firstNumber;
  for (let i = 0; i < restOfNumbers.length; i++) {
    total += restOfNumbers[i];
  }
  return total;
}

let sum = sumNumbers(1, 2, 3, 4, 5); // sum = 15
```

3.3.4 函数重载

函数重载允许你为同一个函数提供多个函数签名，但只有一个函数体。这在函数可以接受不同类型或数量的参数时非常有用。

```
function add(x: string, y: string): string; // 重载签名1
function add(x: number, y: number): number; // 重载签名2
function add(x: any, y: any): any { // 实际的函数实现
  return x + y;
}

console.log(add("Hello ", "TypeScript")); // Hello TypeScript
console.log(add(10, 20)); // 30
// console.log(add("Hello", 10)); // 错误：没有匹配的重载
```

注意：重载签名只用于类型检查，实际的函数实现必须兼容所有重载签名。

3.4 数组类型

TypeScript 提供了多种定义数组类型的方式。

3.4.1 类型+方括号

这是最简单和常用的方式，表示一个由特定类型元素组成的数组。

```
let numbers: number[] = [1, 2, 3];
let names: string[] = ['Alice', 'Bob', 'Charlie'];

// numbers.push('4'); // 错误：类型“string”的参数不能赋给类型“number”的参数。
```

3.4.2 数组泛型

使用泛型数组 `Array<elemType>`：

```
let numbers: Array<number> = [1, 2, 3];
let names: Array<string> = ['Alice', 'Bob', 'Charlie'];
```

3.4.3 接口表示数组

虽然不常用，但也可以使用接口来描述数组，特别是当数组需要有额外的属性时。

```ts
interface NumberArray {
  [index: number]: number;
}

let fibonacci: NumberArray = [1, 1, 2, 3, 5];
```

3.5 元组 (Tuple)

元组（Tuple）表示一个已知元素数量和类型的数组，各元素的类型可以不同。元组的元素顺序和类型必须与定义时一致。

```ts
let x: [string, number];
x = ['hello', 10]; // 正确
// x = [10, 'hello']; // 错误：类型不匹配
// x[2] = 'world'; // 错误：索引越界

console.log(x[0].substring(1)); // ello
// console.log(x[1].substring(1)); // 错误：类型“number”上不存在属性“substring”。
```

3.6 枚举 (Enum)

枚举（Enum）是 TypeScript 中特有的类型，用于定义一组命名的常量。枚举可以帮助我们提高代码的可读性和可维护性。

```
enum Color {
  Red,    // 默认值为 0
  Green,  // 默认值为 1
  Blue    // 默认值为 2
}

let c: Color = Color.Green;
console.log(c); // 输出 1

enum Color2 {
  Red = 1,
  Green,
  Blue
}

let c2: Color2 = Color2.Green;
console.log(c2); // 输出 2

enum Color3 {
  Red = 1,
  Green = 2,
  Blue = 4,
}

let c3: Color3 = Color3.Green;
console.log(c3); // 输出 2

// 字符串枚举
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}

let dir: Direction = Direction.Up;
console.log(dir); // 输出 "UP"
```

3.7 联合类型 (Union Types)

联合类型（Union Types）表示一个变量可以是多种类型中的一种。使用 `|` 符号连接不同的类型。

```ts
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven'; // 正确
myFavoriteNumber = 7; // 正确
// myFavoriteNumber = true; // 错误：不能将类型“boolean”分配给类型“string | number”。

function printId(id: number | string) {
  console.log("Your ID is: " + id);
}

printId(101); // 正确
printId("202"); // 正确
// printId(true); // 错误
```

当 TypeScript 不确定一个联合类型的变量到底是哪个类型时，我们只能访问联合类型的所有类型里共有的属性或方法。

```ts
function getLength(x: string | number): number {
  // console.log(x.length); // 错误：类型“string | number”上不存在属性“length”。
  if (typeof x === 'string') {
    return x.length; // 类型守卫，此时 x 被推断为 string
  }
  return x.toString().length; // 此时 x 被推断为 number
}
```

3.8 交叉类型 (Intersection Types)

交叉类型（Intersection Types）表示将多个类型合并为一个类型，它包含了所有类型的特性。使用 `&` 符号连接不同的类型。

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

interface Contact {
  phone: string;
  email: string;
}

type Employee = Person & Contact; // Employee 包含了 Person 和 Contact 的所有属性

let employee: Employee = {
  name: "Alice",
  age: 30,
  phone: "123-456-7890",
  email: "alice@example.com"
};

// 错误：缺少属性“email”
// let incompleteEmployee: Employee = {
//   name: "Bob",
//   age: 25,
//   phone: "987-654-3210"
// };
```

3.9 类型别名 (Type Aliases)

类型别名（Type Aliases）用于为类型创建一个新的名字。类型别名可以用于原始类型、联合类型、元组以及任何你需要重命名的类型。

```ts
type MyString = string;
let str: MyString = "hello";

type StringOrNumber = string | number;
let value: StringOrNumber = 123;
value = "abc";

type Point = [number, number];
let coordinate: Point = [10, 20];

type Callback = (data: string) => void;
function fetchData(url: string, callback: Callback) {
  // ...
}
```

类型别名和接口在很多情况下可以互换使用，但它们之间也存在一些差异：

- **扩展性**：接口可以被继承和实现，而类型别名不能。
- **合并声明**：同名的接口会自动合并，而类型别名不会。

通常，如果需要定义对象的形状并希望它能够被扩展或实现，使用接口；如果只是为现有类型创建别名或定义联合类型、交叉类型等，使用类型别名。

3.10 类型断言 (Type Assertions)

类型断言（Type Assertions）允许你手动指定一个值的类型。它告诉编译器“相信我，我知道这个变量的类型是什么”。类型断言有两种形式：

- **“尖括号”语法：** `<Type>value`
- **`as` 语法：** `value as Type` (在 JSX 中只能使用 `as` 语法)

```typescript
// 尖括号语法
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

// as 语法
let anotherValue: any = "this is another string";
let anotherStrLength: number = (anotherValue as string).length;

console.log(strLength); // 16
console.log(anotherStrLength); // 20
```

类型断言只在编译时起作用，它不会改变运行时变量的类型。它只是告诉编译器如何处理这个值。因此，在使用类型断言时需要谨慎，确保你确实了解变量的实际类型，否则可能会导致运行时错误。

3.11 类型守卫 (Type Guards)

类型守卫（Type Guards）是一种在运行时检查类型，并根据检查结果缩小变量类型范围的技术。这在处理联合类型时非常有用，可以帮助 TypeScript 编译器更智能地理解变量的类型。

常见的类型守卫包括：

- **`typeof` 运算符：** 用于判断原始类型（`number`, `string`, `boolean`, `symbol`, `undefined`, `object`, `function`, `bigint`）。

  `typescript function printId(id: number | string) { if (typeof id === "string") { console.log(id.toUpperCase()); // id 被推断为 string } else { console.log(id.toFixed(2)); // id 被推断为 number } }`

- **`instanceof` 运算符：** 用于判断一个对象是否是某个类的实例。

```typescript class Dog { bark() { console.log('Woof!'); } }

class Cat { meow() { console.log('Meow!'); } }

function animalSound(animal: Dog | Cat) { if (animal instanceof Dog) { animal.bark(); // animal 被推断为 Dog } else { animal.meow(); // animal 被推断为 Cat } }

animalSound(new Dog()); animalSound(new Cat()); ```

- **`in` 运算符**：用于判断一个对象是否包含某个属性。

  ```typescript interface Bird { fly(): void; layEggs(): void; }

  interface Fish { swim(): void; layEggs(): void; }

  function getSmallPet(): Bird | Fish { // ... return Math.random() < 0.5 ? { fly: () => {}, layEggs: () => {} } : { swim: () => {}, layEggs: () => {} }; }

  let pet = getSmallPet();

  if ("fly" in pet) { pet.fly(); // pet 被推断为 Bird } else { pet.swim(); // pet 被推断为 Fish } ```

- **自定义类型守卫**：通过定义一个返回值为 `parameterName is Type` 的函数来创建自定义类型守卫。

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

  if (isFish(pet)) { pet.swim(); // pet 被推断为 Fish } else { pet.fly(); // pet 被推断为 Bird } ```

类型守卫是 TypeScript 中非常强大的功能，它使得在处理复杂类型时能够编写出更安全、更具可读性的代码。

## 4. TypeScript面向对象

TypeScript 作为 JavaScript 的超集，提供了完整的面向对象编程（OOP）支持，包括类、接口、继承、多态、抽象类以及访问修饰符等。这使得开发者能够使用更结构化、模块化的方式来组织和管理代码，尤其适用于大型复杂项目。
  ```

4.1 类的继承与多态

继承是面向对象编程的一个核心概念，它允许一个类（子类或派生类）继承另一个类（父类或基类）的属性和方法。通过继承，子类可以复用父类的代码，并在此基础上添加新的功能或修改现有功能。

在 TypeScript 中，使用 `extends` 关键字来实现继承：

```
class Animal {
  name: string;

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

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

class Snake extends Animal {
  constructor(name: string) {
    super(name); // 调用父类的构造函数
  }

  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters); // 调用父类的 move 方法
  }
}

class Horse extends Animal {
  constructor(name: string) {
    super(name);
  }

  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move(); // 输出: Slithering... Sammy the Python moved 5m.
tom.move(34); // 输出: Galloping... Tommy the Palomino moved 34m.
```

在上面的例子中，`Snake` 和 `Horse` 都继承自 `Animal` 类。它们各自实现了自己的 `move` 方法，并通过 `super.move()` 调用了父类的 `move` 方法，实现了方法的重写和扩展。

多态是面向对象编程的另一个重要特性，它允许不同类的对象对同一消息作出不同的响应。在 TypeScript 中，多态通常通过继承和方法重写来实现。一个父类类型的引用可以指向其

子类的对象，并且在调用方法时，会根据实际对象的类型来执行相应的方法。

在上面的例子中，`let tom: Animal = new Horse("Tommy the Palomino");` 这行代码就体现了多态。`tom` 变量的类型是 `Animal`（父类），但它实际指向的是一个 `Horse`（子类）对象。当调用 `tom.move(34)` 时，执行的是 `Horse` 类中重写的 `move` 方法，而不是 `Animal` 类中的 `move` 方法。

4.2 接口与类的实现 (implements)

在 TypeScript 中，接口不仅可以用来定义对象的形状，还可以用来约束类的行为。一个类可以**实现（implements）**一个或多个接口，这意味着该类必须实现接口中定义的所有属性和方法。

使用 `implements` 关键字来表示类实现接口：

```
interface Alarm {
  alert(): void;
}

interface Light {
  lightOn(): void;
  lightOff(): void;
}

class Car implements Alarm, Light {
  alert() {
    console.log("Car alarm!");
  }

  lightOn() {
    console.log("Car lights on!");
  }

  lightOff() {
    console.log("Car lights off!");
  }
}

let car = new Car();
car.alert();
car.lightOn();
```

在这个例子中，`Car` 类实现了 `Alarm` 和 `Light` 两个接口，因此它必须提供 `alert`、`lightOn` 和 `lightOff` 方法的具体实现。如果 `Car` 类没有实现接口中定义的所有成员，TypeScript 编译器会报错。

接口实现类的好处在于：

- **强制约束**：确保类遵循特定的契约，提高了代码的一致性和可预测性。

- **代码解耦**：将类的行为抽象为接口，使得代码的耦合度降低，更易于测试和维护。
- **多重实现**：一个类可以实现多个接口，从而获得多种行为能力，弥补了单继承的不足。

4.3 抽象类与抽象方法

抽象类（Abstract Class）是不能被直接实例化的类，它主要用于定义其他类的基类。抽象类可以包含普通的方法和属性，也可以包含**抽象方法（Abstract Method）**。抽象方法只声明了方法签名，没有具体的实现，必须在派生类中实现。

使用 `abstract` 关键字来定义抽象类和抽象方法：

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

  printName(): void {
    console.log("Department name: " + this.name);
  }

  abstract printMeeting(): void; // 抽象方法，必须在派生类中实现
}

class AccountingDepartment extends Department {
  constructor() {
    super("Accounting and Auditing");
  }

  printMeeting(): void {
    console.log("The Accounting Department meets each Monday at 10am.");
  }

  generateReports(): void {
    console.log("Generating accounting reports...");
  }
}

// let department = new Department(); // 错误：不能创建抽象类的实例

let department: Department; // 允许创建对抽象类型的引用
department = new AccountingDepartment();
department.printName(); // 输出: Department name: Accounting and Auditing
department.printMeeting(); // 输出: The Accounting Department meets each Monday
at 10am.
// department.generateReports(); // 错误：不能在抽象类型上调用具体方法，因为
department 的类型是 Department
```

抽象类的特点：

- 不能被直接实例化，只能作为其他类的基类。
- 可以包含抽象方法和非抽象方法。

- 抽象方法必须在派生类中实现。
- 抽象类可以不包含抽象方法，但包含抽象方法的类必须是抽象类。

抽象类在设计模式中非常有用，它允许你定义一个通用的接口，并强制子类提供特定的实现。

5. TypeScript高级特性

TypeScript 除了提供强大的类型系统和面向对象支持外，还包含许多高级特性，这些特性使得 TypeScript 在处理复杂场景和提高代码复用性方面更加灵活和强大。

5.1 泛型 (Generics)

泛型（Generics）是 TypeScript 中一个非常重要的特性，它允许你编写出可重用、灵活且类型安全的代码。泛型的主要思想是，在定义函数、接口或类时，不预先指定具体的类型，而是在使用时再指定类型。这类似于函数参数，只不过这里参数是类型而不是值。

5.1.1 泛型函数

泛型函数允许你编写一个函数，它能够处理多种类型的数据，同时保持类型安全。

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

// 使用泛型函数
let output1 = identity<string>("myString"); // 明确指定类型参数为 string
let output2 = identity(123); // 类型推断：编译器自动推断类型参数为 number

console.log(output1); // myString
console.log(output2); // 123
```

在上面的例子中，`<T>` 表示我们定义了一个类型变量 `T`。`arg: T` 表示参数 `arg` 的类型是 `T`，`: T` 表示函数的返回类型也是 `T`。这样，无论 `arg` 是什么类型，函数的输入和输出类型都能保持一致。

5.1.2 泛型接口

泛型接口允许你定义一个接口，它能够处理多种类型的数据。

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}

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

let myIdentity: GenericIdentityFn<number> = identity;
console.log(myIdentity(456)); // 456
```

5.1.3 泛型类

泛型类允许你定义一个类，它的属性或方法可以使用泛型类型。

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

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

console.log(myGenericNumber.add(myGenericNumber.zeroValue, 10)); // 10

let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };

console.log(stringNumeric.add(stringNumeric.zeroValue, "test")); // test
```

5.1.4 泛型约束

有时你可能希望泛型类型 `T` 具有某些特定的属性。这时可以使用泛型约束。例如，你可能希望 `T` 具有 `length` 属性。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // 现在我们知道 arg 有一个 .length 属性
  return arg;
}

// loggingIdentity(3); // 错误：类型“number”的参数不能赋给类型“Lengthwise”的参数。
loggingIdentity({ length: 10, value: 3 }); // 正确
```

通过 `extends Lengthwise`，我们约束了类型变量 `T` 必须是 `Lengthwise` 接口的子类型，即必须包含 `length` 属性。

5.2 装饰器 (Decorators)

装饰器（Decorators）是一种特殊类型的声明，它能够被附加到类声明、方法、访问器、属性或参数上。装饰器使用 `@expression` 这种形式，其中 `expression` 必须是一个函数，它会在运行时被调用，并传入被装饰的声明信息。

要启用装饰器，你需要在 `tsconfig.json` 中设置 `"experimentalDecorators": true`。

5.2.1 类装饰器

类装饰器应用于类构造函数，可用于观察、修改或替换类定义。

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}

// 尝试修改 Greeter 类或其原型会报错
// delete Greeter.prototype.greet; // 错误
```

5.2.2 方法装饰器

方法装饰器应用于类的方法，可用于观察、修改或替换方法定义。

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

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

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
for (let key in greeter) {
  console.log(key); // 不会输出 greet 方法
}
```

5.2.3 属性装饰器

属性装饰器应用于类的属性，可用于观察或修改属性定义。

```
function format(formatString: string) {
  return function (target: any, propertyKey: string) {
    let value: string;
    const getter = function () {
      return formatString.replace("%s", value);
    };
    const setter = function (newVal: string) {
      value = newVal;
    };

    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true,
    });
  };
}

class Greeter {
  @format("Hello, %s")
  greeting: string;

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

let greeter = new Greeter("world");
console.log(greeter.greeting); // 输出: Hello, world
```

5.2.4 参数装饰器

参数装饰器应用于类构造函数或方法参数，可用于观察或修改参数定义。

```
function required(target: Object, propertyKey: string | symbol,
parameterIndex: number) {
  console.log(`Parameter $`{parameterIndex} of `${String(propertyKey)} is
required.`);
}

class Greeter {
  greeting: string;

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

  greet(@required name: string) {
    return `Hello, $`{name}! `${this.greeting}`;
  }
}

let greeter = new Greeter("world");
console.log(greeter.greet("TypeScript"));
```

装饰器提供了一种声明式的方式来添加元编程能力，使得代码更加简洁和可维护，尤其在框架和库的开发中非常常见。

5.3 模块 (Modules)

模块（Modules）是 TypeScript 中组织代码的重要方式，它允许你将代码分割成独立的文件，每个文件都是一个模块。模块可以导出（export）其内部的变量、函数、类、接口等，供其他模块导入（import）使用。这有助于避免全局命名空间污染，提高代码的复用性和可维护性。

5.3.1 导出与导入

- **导出（Export）**：使用 `export` 关键字来导出模块中的成员。

  ```typescript
  // math.ts
  export const PI = 3.14159;

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

  export class Calculator {
    multiply(x: number, y: number): number {
      return x * y;
    }
  }
  ```

- **导入（Import）**：使用 `import` 关键字来导入其他模块中导出的成员。

  ```typescript
  // app.ts
  import { PI, add, Calculator } from "./math";

  console.log(PI); // 3.14159
  console.log(add(1, 2)); // 3

  let calc = new Calculator();
  console.log(calc.multiply(3, 4)); // 12
  ```

5.3.2 默认导出与命名导出

- **命名导出（Named Exports）**：每个模块可以有多个命名导出，导入时需要使用 `{}` 并指定导出的名称。

  ```typescript
  // utils.ts
  export function capitalize(str: string): string {
    return str.charAt(0).toUpperCase() + str.slice(1);
  }

  export const VERSION = "1.0.0";
  ```

  ```typescript
  // main.ts
  import { capitalize, VERSION } from "./utils";

  console.log(capitalize("hello")); // Hello
  console.log(VERSION); // 1.0.0
  ```

- **默认导出（Default Exports）**：每个模块只能有一个默认导出。导入时不需要使用 `{}`，并且可以为导入的成员指定任意名称。

  `typescript // logger.ts export default class Logger { log(message: string) { console.log(`[LOG]: ${message}`); } }`

  ```typescript // app.ts import MyLogger from "./logger"; // MyLogger 可以是任意名称

  let logger = new MyLogger(); logger.log("This is a test message."); ```

模块化是现代 JavaScript 和 TypeScript 开发的基础，它使得代码结构清晰，易于管理和维护。

## 5.4 命名空间 (Namespaces)

命名空间（Namespaces）是 TypeScript 早期用于组织代码的一种方式，主要用于避免全局命名冲突。在 ES6 模块（Modules）成为标准之后，命名空间的使用场景有所减少，但对于一些旧项目或特定场景仍然有用。

命名空间使用 `namespace` 关键字定义，可以嵌套，也可以跨文件。
  ```

```
// validation.ts
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  const lettersRegexp = /^[A-Za-z]+$/;
  const numberRegexp = /^[0-9]+$/;

  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegexp.test(s);
    }
  }
}

// app.ts
/// <reference path="validation.ts" />
let strings = ["Hello", "98052", "101"];

let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

for (let s of strings) {
  for (let name in validators) {
    console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" :
"does not match" } ${ name }`);
  }
}
```

需要注意的是，如果命名空间跨越多个文件，你需要使用三斜线指令 `/// <reference path="..." />` 来引用其他文件，或者在 `tsconfig.json` 中配置 `outFile` 选项将所有文件编译到一个 JavaScript 文件中。

在现代 TypeScript 项目中，通常推荐使用 ES6 模块来组织代码，因为它更符合 JavaScript 的发展趋势，并且具有更好的工具支持。

5.5 声明文件 (.d.ts)

声明文件（Declaration Files），通常以 `.d.ts` 结尾，是 TypeScript 中非常重要的概念。它们用于描述 JavaScript 库或模块的类型信息，使得 TypeScript 编译器能够理解这些 JavaScript 代码的结构，从而提供类型检查、智能提示等功能。

5.5.1 为什么需要声明文件

JavaScript 是动态类型语言，本身没有类型信息。当你在 TypeScript 项目中使用 JavaScript 库时，如果没有 `.d.ts` 文件，TypeScript 编译器就无法知道这些库中变量、函数、类的类型，从而无法进行类型检查，也无法提供智能提示，这会大大降低开发体验。

声明文件解决了这个问题，它就像是 JavaScript 代码的“类型说明书”，告诉 TypeScript 编译器每个变量是什么类型、每个函数接受什么参数、返回什么类型等等。

5.5.2 如何编写声明文件

编写声明文件通常需要使用 `declare` 关键字。`declare` 关键字用于告诉 TypeScript 编译器，你正在描述一个已经存在于 JavaScript 运行时环境中的变量、函数、类等，而不是要创建一个新的实体。

以下是一些常见的 `declare` 用法：

- **声明变量**：

  `typescript // global.d.ts declare var jQuery: (selector: string) => any; declare const PI: number; declare let count: number;`

- **声明函数**：

  `typescript // global.d.ts declare function greet(name: string): void;`

- **声明类**：

  `typescript // global.d.ts declare class Animal { name: string; constructor(name: string); eat(): void; }`

- **声明接口**：

  `typescript // global.d.ts declare interface Person { name: string; age: number; }`

- **声明模块**：当 JavaScript 库是模块化的（例如使用 CommonJS 或 ES Modules）时，需要使用 `declare module`。

  `typescript // my-module.d.ts declare module "my-module" { export function doSomething(a: number): string; export const version:`

`string; }`

- **声明全局类型**：

  `typescript // global.d.ts declare namespace MyGlobalLib { function doSomething(): void; }`

5.5.3 第三方库的声明文件

对于大多数流行的第三方 JavaScript 库，你通常不需要手动编写声明文件，因为它们已经由社区维护在 DefinitelyTyped 项目中。你可以通过 npm 安装这些声明文件，它们通常以 `@types/` 开头。

例如，如果你想在 TypeScript 项目中使用 Lodash 库，可以安装其声明文件：

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

安装后，TypeScript 编译器会自动找到并使用这些声明文件，为你提供 Lodash 库的类型信息。

如果一个库没有提供声明文件，你可以尝试：

1. 查找社区是否有人创建了非官方的声明文件。
2. 自己编写一个简单的声明文件，或者使用 `any` 类型作为临时解决方案。
3. 向 DefinitelyTyped 贡献你的声明文件。

声明文件是 TypeScript 能够与现有 JavaScript 生态系统无缝协作的关键，它极大地提升了 TypeScript 的可用性和开发效率。

6. TypeScript实战应用

学习 TypeScript 的最终目的是将其应用于实际项目中，提升开发效率和代码质量。本节将介绍如何在常见的开发场景中使用 TypeScript，并提供一些最佳实践和配置建议。

6.1 在React/Vue项目中使用TypeScript

TypeScript 在前端框架中得到了广泛应用，特别是 React 和 Vue。它们都提供了对 TypeScript 的良好支持，使得开发者能够利用类型系统来构建更健壮的组件和应用。

在 React 项目中使用 TypeScript

创建新的 React 项目时，可以使用 Create React App 提供的 TypeScript 模板：

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

这会自动配置好 TypeScript 环境，包括 `tsconfig.json` 文件和必要的 `@types` 依赖。在 React 组件中，你可以使用类型注解来定义 props、state 和事件处理函数的类型：

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

interface GreetingProps {
  name: string;
  age?: number; // 可选属性
}

const Greeting: React.FC<GreetingProps> = ({ name, age }) => {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      {age && <p>You are {age} years old.</p>}
    </div>
  );
};

export default Greeting;
```

在 Vue 项目中使用 TypeScript

创建新的 Vue 项目时，可以使用 Vue CLI 选择 TypeScript 选项：

```
vue create my-vue-app
```

在配置过程中选择 TypeScript。Vue 3 对 TypeScript 有更好的原生支持，你可以使用 `<script setup lang="ts">` 来编写单文件组件：

```vue
<!-- src/components/HelloWorld.vue -->
<script setup lang="ts">
import { ref } from 'vue';

interface Props {
  msg: string;
}

const props = defineProps<Props>();

const count = ref(0);
</script>

<template>
  <h1>{{ props.msg }}</h1>
  <button @click="count++">count is: {{ count }}</button>
</template>
```

6.2 TypeScript与Node.js

TypeScript 也可以用于后端开发，特别是与 Node.js 结合使用。这使得你可以在整个应用栈中保持类型一致性，从而提高开发效率和代码质量。

初始化 Node.js 项目

首先，创建一个新的 Node.js 项目并初始化 `package.json`：

```
mkdir my-node-ts-app
cd my-node-ts-app
npm init -y
```

安装 TypeScript 和相关依赖

```
npm install --save-dev typescript @types/node
```

`@types/node` 提供了 Node.js 内置模块的类型声明。

配置 tsconfig.json

运行 `tsc --init` 命令生成 `tsconfig.json` 文件，然后根据需要进行配置。例如：

```jsonc
// tsconfig.json
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "outDir": "./dist", // 编译输出目录
    "rootDir": "./src", // TypeScript 源代码目录
    "strict": true, // 启用所有严格类型检查选项
    "esModuleInterop": true, // 允许从 CommonJS 模块中默认导入
    "skipLibCheck": true, // 跳过声明文件检查
    "forceConsistentCasingInFileNames": true // 强制文件名大小写一致
  },
  "include": ["src/**/*"], // 包含 src 目录下的所有文件
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

编写 TypeScript 代码

在 `src` 目录下创建你的 TypeScript 代码文件，例如 `src/app.ts`：

```typescript
// src/app.ts
import * as http from 'http';

const hostname: string = '127.0.0.1';
const port: number = 3000;

const server = http.createServer((req: http.IncomingMessage, res:
http.ServerResponse) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, TypeScript Node.js Server!\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

编译和运行

在 `package.json` 中添加编译和运行脚本：

```
// package.json
{
  "name": "my-node-ts-app",
  "version": "1.0.0",
  "description": "",
  "main": "dist/app.js",
  "scripts": {
    "build": "tsc",
    "start": "node dist/app.js",
    "dev": "tsc -w & nodemon dist/app.js" // 使用 nodemon 自动重启服务
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.x.x",
    "typescript": "^5.x.x"
  }
}
```

运行 `npm run build` 编译代码，然后 `npm start` 运行服务。

6.3 常用工具与配置 (tsconfig.json)

`tsconfig.json` 是 TypeScript 项目的配置文件，它包含了编译器的所有选项，用于控制 TypeScript 代码的编译方式。理解和配置 `tsconfig.json` 是 TypeScript 开发的关键。

一些常用的 `compilerOptions`：

- **`target`**：指定编译后的 JavaScript 版本（如 `es5`，`es2015`，`es2016`，`esnext`）。
- **`module`**：指定生成模块代码的类型（如 `commonjs`，`es2015`，`esnext`）。
- **`outDir`**：指定编译输出文件的目录。
- **`rootDir`**：指定 TypeScript 源代码的根目录。
- **`strict`**：启用所有严格类型检查选项，强烈推荐开启。
- **`esModuleInterop`**：允许 CommonJS/AMD/UMD 模块的默认导入，解决不同模块系统之间的兼容性问题。
- **`jsx`**：指定 JSX 语法处理模式（如 `react`，`preserve`）。
- **`lib`**：指定要包含在编译中的库文件（如 `es2015`，`dom`）。
- **`sourceMap`**：生成 `.map` 文件，用于调试。
- **`noImplicitAny`**：不允许隐式的 `any` 类型。

- **`strictNullChecks`**：启用严格的 null 检查，防止 `null` 或 `undefined` 赋值给非空类型。

除了 `compilerOptions`，`tsconfig.json` 还可以配置 `include`、`exclude` 和 `files` 来指定哪些文件应该被编译。

6.4 最佳实践与常见问题

最佳实践

1. **始终开启严格模式**：在 `tsconfig.json` 中设置 `"strict": true`，这会启用所有严格类型检查选项，帮助你编写更健壮的代码。
2. **明确类型，但不过度**：利用类型推断，只在必要时才明确指定类型。例如，函数参数和返回值通常需要明确类型，而简单变量的类型可以由推断得出。
3. **使用接口或类型别名定义复杂类型**：对于对象、函数签名等复杂结构，使用 `interface` 或 `type` 来定义，提高代码可读性和复用性。
4. **善用联合类型和交叉类型**：它们可以帮助你更灵活地组合类型，处理多种可能的数据结构。
5. **合理使用类型守卫**：在处理联合类型时，使用 `typeof`、`instanceof`、`in` 或自定义类型守卫来缩小类型范围，确保类型安全。
6. **为第三方库安装 `@types` 声明文件**：这能让你在 TypeScript 项目中获得第三方 JavaScript 库的类型提示和检查。
7. **模块化组织代码**：使用 ES6 模块（`import` / `export`）来组织代码，避免全局命名空间污染。
8. **编写可测试的代码**：类型系统有助于编写更易于测试的代码，因为类型错误可以在编译阶段被捕获。

常见问题

1. **类型错误 `Property 'x' does not exist on type 'Y'`**：
   - **原因**：你尝试访问一个类型上不存在的属性。
   - **解决方案**：检查类型定义是否正确，或者使用类型断言 `(value as SomeType).x`，或者使用类型守卫来缩小类型范围。
2. **类型错误 `Type 'A' is not assignable to type 'B'`**：

- **原因**：你尝试将一个类型的值赋给不兼容的另一个类型。
- **解决方案**：检查赋值操作是否符合类型定义，可能需要类型转换或调整类型定义。

3. **`any` 类型泛滥**：
   - **原因**：过度使用 `any` 类型，导致失去了 TypeScript 的类型检查优势。
   - **解决方案**：尽量避免使用 `any`，尝试为变量、函数参数和返回值添加明确的类型注解。如果确实需要处理不确定类型的数据，可以考虑使用 `unknown` 类型，它比 `any` 更安全，因为它强制你在使用前进行类型检查。

4. **模块导入/导出问题**：
   - **原因**：模块路径不正确，或者 `tsconfig.json` 中的 `module` 和 `esModuleInterop` 配置不当。
   - **解决方案**：检查模块路径是否正确，确保 `tsconfig.json` 中的 `module` 和 `esModuleInterop` 配置与你的项目需求相符。

5. **第三方库没有类型声明文件**：
   - **原因**：你使用的 JavaScript 库没有提供 `.d.ts` 文件，或者你没有安装对应的 `@types` 包。
   - **解决方案**：首先尝试安装 `@types/your-library-name`。如果找不到，可以自己创建一个简单的 `.d.ts` 文件来声明该库的类型，或者暂时使用 `declare module 'your-library-name';` 来跳过类型检查。

通过遵循最佳实践和理解常见问题的解决方案，你将能够更有效地使用 TypeScript 来构建高质量的应用程序。