ts基础语法

2021年12月1日 星期三　　下午 10:44

0. typescript的意义

- TypeScript 的核心设计理念：在完整保留 JavaScript 运行时行为的基础上，通过引入静态类型系统来提高代码的可维护性，减少可能出现的 bug。 这使ts更适用于大项目，可维护性增强。
- JavaScript是弱类型，很多错误只有在运行时才会被发现，
  而TypeScript提供了一套静态检测机制，可以帮助我们在编译时就发现错误，所以Typescript是静态类型。
- 扩展了JavaScript： 类， 接口， 模块， 类型注解， Arrow函数
- 大部分 JavaScript 代码都只需要经过少量的修改（或者完全不用修改）就能变成 TypeScript 代码，这得益于 TypeScript 强大的「类型推论」，即使不去手动声明变量 foo 的类型，也能在变量初始化时自动推论出它是一个 number 类型。
- Typescript也是弱类型，因为和js一样支持隐式转换。

```
console.log(1 + '1');// 打印出字符串 '11'
```

第一、 ts的类型

Same as JavaScript: 8个js的内置类型

Boolean, Number, BigInt, String , Null, Undefined , Object object{ }, Symbol

Tips: number bigint 不兼容； null & undefined是所有类型的子类型 ；

其他类型:

1. Array:

```
let arr:string[] = ["1","2"];
let arr2:Array<string> = ["1","2"];
let arr:(number | string)[] = [1, 'b', 2, 'c'];//UnionTypes
```

指定对象成员的数组:

```
interface Arrobj{
    name:string,
    age:number
}
let arr3:Arrobj[]=[{name:'jimmy',age:22}]; // obj stack in
```

2. Function函数:

- 函数声明:

```
function sum(x: number, y: number) : number { //定义输入输出类型
    return x + y;
}
```

- 函数表达式:

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

- 可选参数

```
function buildName(firstName: string, lastName?: string) {
```

- 参数默认值

```
function buildName(firstName: string, lastName: string = 'Cat') {
```

- **剩余参数**

```
function push(array: any[], ...items: any[]) {
```

- **函数重载**

使用相同的名称，创建多个函数，绑定不同types

```
type Types = number | string
function add(a:number,b:number):number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;
function add(a:Types, b:Types) {
  if (typeof a === 'string' || typeof b === 'string') {
    return a.toString() + b.toString();
  }
  return a + b;
}
const result = add('Semlinker', ' Kakuqo');
result.split(' ');
```

3. Tuple元组 let x :[type1,type2,type3,……];

在 JavaScript 中是没有元组的，元组是 TypeScript 中特有的类型，其工作方式类似于数组。

元组最重要的特性是可以限制数组元素的个数和类型，它特别适合用来实现多值返回。

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

- 解构赋值

```
let [username, id] = x; // username = 'hello' , id = 10
```

- 可选元素 ？

4. Void: function func(a, b):**void** { }

Func 方法没有返回值将得到undefined，但是我们需要定义成void类型，而不是undefined类型。否则将报错.

5. Any:

在 TypeScript 中，**任何类型都可以被归为 any 类型。**这让 any 类型成为了类型系统的顶级类型.

6. Unknown:

unknown与any的最大区别是： 任何类型的值可以赋值给any，同时any类型的值也可以赋值给任何类型。**unknown 任何类型的值都可以赋值给它，但它只能赋值给unknown和any.**

7. Never: never类型表示的是那些永不存在的值的类型。

利用 never 类型的特性来实现全面性检查

CODE: **这也是一种类型缩小的方法！！**

```
type Foo = string | number;
function controlFlowAnalysisWithNever(foo: Foo) {
  if (typeof foo === "string") {
    // 这里 foo 被收窄为 string 类型
  } else if (typeof foo === "number") {
    // 这里 foo 被收窄为 number 类型
  } else {
    // foo 在这里是 never
    const check: never = foo;//如果编译通过，表示穷尽了foo的所有可选类型。
  }
}
```

Enum:

类型推断:

TypeScript 基于赋值表达式推断类型。 TypeScript 会根据上下文环境自动推断出变量的类型。

类型断言：

指定方法运行后的输出， TypeScript 类型检测无法做到绝对智能，毕竟程序不能像人一样思考。有时会碰到我们比 TypeScript 更清楚实际类型的情况，比如下面的例子：

```
const arrayNumber: number[] = [1, 2, 3, 4];
const greaterThan2: number = arrayNumber.find(num => num > 2); // 类型拓展 number | undefined
```

在 TypeScript 看来，greaterThan2 的类型既可能是数字，也可能是 undefined，所以上面的示例中提示了一个 ts(2322) 错误，此时我们不能把类型 undefined 分配给类型 number。

我们可以使用一种笃定的方式—**类型断言**（类似仅作用在类型层面的强制类型转换）告诉 TypeScript 按照我们的方式做类型检查。

方法1 as ：

```
const arrayNumber: number[] = [1, 2, 3, 4];
const greaterThan2: number = arrayNumber.find(num => num > 2) as number;
```

方法2 尖括号：

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

方法3 非空断言 ！：x! 将从 x 值域中排除 null 和 undefined

```
let mayNullOrUndefinedOrString: null | undefined | string;
mayNullOrUndefinedOrString!.toString(); // ok
```

方法4：let x!: number; 确定赋值断言，TypeScript 编译器就会知道该属性会被明确地赋值。

联合类型：

联合类型（Union Types）表示取值可以为多种类型中的一种。未赋值时联合类型上只能访问两个类型共有的属性和方法。let name: string | number;

类型别名： type Message = string | string[]; let x : Message;

字面量类型：

TypeScript 支持 3 种字面量类型：字符串字面量类型、数字字面量类型、布尔字面量类型

```
{
  let specifiedStr: 'this is string' = 'this is string';
  let specifiedNum: 1 = 1;
  let specifiedBoolean: true = true;
}
```

'this is string' 字面量类型可以给 string 类型赋值，但是 string 类型不能给 'this is string' 字面量类型赋值，

交叉类型：

交叉类型真正的用武之地就是将多个接口类型合并成一个类型，从而实现等同接口继承的效果，也就是所谓的合并接口类型，如下代码所示：

```
type IntersectionType = { id: number; name: string; } & { age: number };
  const mixed: IntersectionType = {
    id: 1,
    name: 'name',
    age: 18
  }
```

在混入多个类型时，若存在相同的成员，且成员类型为非基本数据类型，那么是可以成功合并。

第二、 接口

TypeScript 中的接口是一个非常灵活的概念，除了可用于[对类的一部分行为进行抽象]以外，也常用于对「对象的形状（Shape）」进行描述。

对象描述：

```
interface Person {
    name: string;
    age: number;     //可选 age?: number;
       [propName: string]: any; //任意属性， 绕开属性检查的方法之一，还有类型断言和鸭式辨型法
}
let tom: Person = {
    name: 'Tom',
    age: 25
};
```

一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集， number | string ∈ any

类抽象：

```
interface SetPoint {
  (x: number, y: number): void;
}
```

3．泛型

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

T代表type， T 是一个抽象类型，只有在调用的时候才确定它的值。

泛型约束： 定义一个类型，然后让 T 实现这个接口即可

```
interface Sizeable {
  size: number;
}
function trace<T extends Sizeable>(arg: T): T {
  console.log(arg.size);
  return arg;
}
```

Typeof 操作符： ts中，类型也是值，可以查看也可以赋值给别人，类型是type

keyof 操作符 可以用于获取某种类型的所有键，其返回类型是联合类型（enum类型）。

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

上述操作，只要K类型是T的子集即可。

In 操作符 遍历

```
type Keys = "a" | "b" | "c"

type Obj =  {
  [p in Keys]: any
}
```

infer: 在条件类型语句中，可以用 infer 声明一个类型变量并且对它进行使用。

extend: 添加泛型约束

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

interface PersonWithBirthDate extends Person {
  birth: Date;
}
```

等同: type PersonWithBirthDate = Person & { birth: Date };

TypeScript 还内置了一些常用的工具类型，比如 Partial、Required、Readonly、Record 和 ReturnType

简介好看的代码:

- **更简洁的类型指定**

```
type HTTPFunction = (url: string, opts: Options) => Promise<Response>;
const get: HTTPFunction = (url, opts) => { /* ... */ };
const post: HTTPFunction = (url, opts) => { /* ... */ };
```

- **更精确的类型指定:**

```
interface Album {\
  artist: string; // 艺术家
  title: string; // 专辑标题
  releaseDate: Date; // 发行日期: YYYY-MM-DD
  recordingType: "studio" | "live"; // 录制类型: "live" 或 "studio"
}
```

- **定义的类型总是表示有效的状态**

```
interface RequestPending {
  state: "pending";
}

interface RequestError {
  state: "error";
  errorMsg: string;
}

interface RequestSuccess {
  state: "ok";
  pageContent: string;
}
```

//状态分别定义，并整合

```
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: { [page: string]: RequestState };
}
```