TypeScript随堂笔记

TypeScript

TypeScript是什么

TypeScript 是一种由微软开发的自由开源的编程语言，他是JavaScript的一个超集，扩展了JavaScript的语法，主要提供了类型系统和对 ES6 的支持。

TypeScript 设计目标是开发大型应用，它可以编译成纯 JavaScript，编译出来的 JavaScript 可以运行在任何浏览器上。

JavaScript 与 TypeScript 的区别

TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法，因此现有的 JavaScript 代码可与 TypeScript 一起工作无需任何修改，TypeScript 通过类型注解提供编译时的静态类型检查。

TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译。

TypeScript 的优势

强大的IDE支持：体现在三个特性上，1.类型检查，在TS中允许你为变量指定类型。2.语法提示。3.重构。
Angular2、vue3的开发语言

TypeScript 的缺点

有一定的学习成本，需要理解接口（Interfaces）、泛型（Generics）、类（Classes）、枚举类型（Enums）等前端开发可能不是很熟悉的知识点

编辑器

TypeScript 最大的优势之一便是增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等。
主流的编辑器都支持 TypeScript，推荐使用 Visual Studio Code。
获取其他编辑器或 IDE 对 TypeScript 的支持：
Sublime Text
Atom
WebStorm
Vim
Emacs
Eclipse
Visual Studio 2015
Visual Studio 2013

TypeScript安装

TypeScript 的命令行工具安装方法如下：
npm install -g typescript

以上命令会在全局环境下安装 tsc 命令，安装完成之后，我们就可以在任何地方执行 tsc 命令了。

查看版本

```
tsc -v
```

编译一个 TypeScript 文件

编译一个 TypeScript 文件很简单，

```
tsc hello.ts
```

使用 TypeScript 编写的文件以 .ts 为后缀

TypeScript 用法

使用 : 指定变量的类型，: 的前后有没有空格都可以

```
let num:number = 15;
num(变量名）:number(类型） = 15（具体值）
表示定义一个变量num，指定类型为number;
let str:string = 'abc';
表示定义一个变量str，指定类型为string;
```

为什么要用到TS

```
定义一个函数计算二个数据的合计
    function sum(x,y){
        if(typeof x != 'number')  { //对于形参的类型要添加转换
            x = parseInt(x);
        }
        return x+y
    };
    sum('1',2);

   TS的方式，直接约束了类型
    function sum2(x:number,y:number){
        return x+y
    };
```

各类型定义

1、类型定义

```
let flag:boolean = false;  //布尔类型
let num:number = 15;  //数值类型
let str:string = 'abc'; //字符串类型
let str2:string=`hello,${str}`;
let msg:string = `hello,${str},${num}`;
let u: undefined = undefined;
let n: null = null;
```

任意值

如果是一个普通类型，在赋值过程中改变类型是不被允许的，任意值（Any）用来表示允许赋值为任意类型。

```
let a1: string = 'seven';
a1 = 7;//error

但如果是 any 类型，则允许被赋值为任意类型。
let a2: any = 'seven';
a2 = 7;

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型
let a3;
a3 = 'seven';
a3 = 7;
相当于
let a3:any;
a3 = 'seven';
a3 = 7;
```

联合类型

表示取值可以为多种类型中的一种

```
let a4: string | number;
a4 = 'seven';
a4 = 7;
a4 = true;  不行
```

```
function getLength(str: string | number): number {
        return str.length; //会报错，为什么？
}
console.log(getLength(123));

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，只能访问此联合类型的所有类型里共有的属性或方法
function getLength(str: string | number[]): number {
        return str.length;
}
console.log(getLength([1,2,3]));
```

2、数组类型的定义

在 TypeScript 中，数组类型有多种定义方式，比较灵活。

最简单的方法是使用「类型 + 方括号」来表示数组：

```
let arr: number[] = [1, 2, 3, 4, 5];

数组的项中不允许出现其他的类型
let arr: number[] = [1, '2', 3, 4, 'a'];  //不行
```

```
其它类型的数组
let arr2: any[] = [1, '1', 2, 'x', {id:1},[1,2,3],true];
let arr3: string[] = ['x', 'y', '2', '3', '5'];
```

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

泛型变量T　　T表示任何类型

```
let arr:Array<number> =[1,2,3];

定义多个类型
let arr:Array<number|string> =['1',2,3];
```

3、对象的类型—接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象

接口（Interfaces）可以用于对「对象的形状（Shape）」进行描述。

接口的定义

```
接口首字母大写
interface Person {
    name: string;
    age: number;
}
```

```
let tom: Person = {
        name: 'Tom',
        age: 18
};

上面的例子中，我们定义了一个接口 Person，接着定义了一个变量 tom，它的类型是 Person。
这样，我们就约束了 tom 的形状必须和接口 Person 一致。
```

定义的变量比接口少了一些属性是不允许的：

```
let tom2: Person = {
        name: 'Tom'
};
let tom3: Person = {
        name: 'Tom',
        age: 25,
        sex: '男'
};
```

接口能否更灵活?

可选属性：

```
interface Person2 {
    name: string;
    age?: number;
}
```

```
let tom4: Person2 = {
    name: 'Tom'
};
```

可选属性的含义是该属性可以不存在。这时仍然不允许添加未定义的属性：

```
let tom5: Person2 = {
    name: 'Tom',
    age: 25,
    sex: '男'    //还是不行
};
```

任意属性： 一个接口允许有任意的属性

```
interface Person3 {
        name: string;
        age?: number;
        [propName: string]: any;
};
```

```
let tom6: Person3 = {
        name: 'Tom',
        sex: '男'
};
```

需要注意的是，一旦定义了任意属性，那么确定属性和可选属性都必须是它的子属性

```
interface Person4 {
        name: string;
        age?: number;
        [propName: string]: any;
};
let tom7: Person4 = {
        name: 'Tom',
        sex: '男'
};
```

4、函数的类型定义

一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到

```
function sum1(x: number, y: number): number {
    return x + y;
}
```

参数默认值

```
function sum3(x: number=5, y: number): number {
    return x + y;
}
let s1 = sum3(1, 2);
```

可选参数

```
function sum4(x: number, y?: number): number[] {
    return [x,y]
}
let s2 = sum4(1);
```

类型别名

```
function myHello(person: n) {
        return 'Hello, ' + person;
};
type n = number;
let name:n = 18;
console.log(myHello(name));
```

类

传统方法中，JavaScript 通过构造函数实现类的概念，通过原型链实现继承。而在 ES6 中，我们终于迎来了 class。

类(Class)：定义了一件事物的抽象特点，包含它的属性和方法

构造函数

```
JavaScript 语言中，生成实例对象的传统方法是通过构造函数
function Cat(name,color){   //构造函数
     this.name = name;
     this.color = color;
     //this.type='动物';
     //this.eat = function(){console.log("吃老鼠")};
};

原型中
Cat.prototype.type='动物';
Cat.prototype.eat = function () {
     return console.log("吃老鼠");
};
var cat1 = new Cat("大明","黄色");
var cat2 = new Cat("小明","白色");
```

定义类

ES6 的类，完全可以看作构造函数的另一种写法。

注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。

另外，方法之间不需要逗号分隔，加了会报错。

```
class Cat2 {
    name:string;   //值类型
    color:string;  //定义值类型
    constructor(name,color) {
        this.name = name;
        this.color = color;
    }
    eat() {
```

```
        return console.log("吃老鼠");
    }
    sayName() {
        return `My name is ${this.name}`;
    }
}
var cat3 = new Cat2("小小明","黑色");
console.log(cat3.eat())
console.log(cat3.sayName())
```

类的继承

```
class Animal {
      name:string;
      constructor(name) {
            this.name = name;
      }
      eat() {
            return "吃骨头";
      }
}

继承
class Dog extends Animal {
     constructor(name) {
            super(name); // 调用父类的 constructor(name)
     }
     sayHi() {
           return this.name+',' + this.eat(); // 调用父类的
     }
}

let d = new Dog('Tom');
console.log(d.sayHi());
```

修饰符

public 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 public 的;
private 修饰的属性或方法是私有的，不能在声明它的类的外部访问;
protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的;

```
class Animal3 {
      public name;
      public constructor(name) {
             this.name = name;
      }
}
```

```
class Animal4 {
      // 使用 private 修饰的属性或方法，在子类中也是不允许访问的
      //private name;
      //使用 protected 修饰，则允许在子类中访问
      protected name;
      public constructor(name) {
            this.name = name;
      }
}
```

类的类型

```
class Animal5 {
      name: string;
      constructor(name: string) {
            this.name = name;
      }
      sayHi(): string {
            return `My name is ${this.name}`;
      }
}

let s4: Animal5 = new Animal5('Jack');
console.log(s4.sayHi());
```

类实现接口

```
interface Animal6 {   。
     name:string;
     action():string;
}

class dog2 implements Animal6 {
            name:string;
            constructor(name){
                  this.name = name;
            }
            action(){
                  return '111'
            }
}
let s6  = new dog2('Jack');
```

实现（implements）是面向对象中的一个重要概念。一般来讲，一个类只能继承自另一个类，

有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），

用 implements 关键字来实现。这个特性大大提高了面向对象的灵活性。

举例来说，门是一个类，防盗门是门的子类。如果防盗门有一个报警器的功能，

我们可以简单的给防盗门添加一个报警方法。这时候如果有另一个类，车，也有报警器的功能，

就可以考虑把报警器提取出来，作为一个接口，防盗门和车都去实现它：

```
报警器
interface Alarm {
```

```
     sing(); //一个抽象的空方法
};

定义门
class Door {  };

防盗门继承门实现报警器的功能
class SecurityDoor extends Door implements Alarm {
      sing() {
           console.log('SecurityDoor sing');
      }
};

车实现报警器的功能
class Car implements Alarm {
      sing() {
            console.log('Car sing');
      }
};
var d1 = new SecurityDoor();
var car = new Car();
```

一个类可以实现多个接口

```
interface Alarm2 {
     alert();
}

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

实现多个接口 Car 实现了 Alarm 和 Light 接口，既能报警，也能开关车灯
class Car3 implements Alarm2, Light {
       alert() {
            console.log('Car alert');
       }
       lightOn() {
            console.log('Car light on');
       }
       lightOff() {
            console.log('Car light off');
       }
}
```

泛型

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

泛型变量T ， T表示任何类型

什么时候需要定义泛型函数

要创建一个可重用的组件，其中的数据类型就必须要兼容很多的类型，那么如何兼容呢?

```
定义一个只能传入number类型的函数
function f(a:number,b:number):number[]{
        return [a,b]
}
f(1,2);

定义一个只能传入string类型的函数
function f2(a:string,b:string):string[]{
        return [a,b]
}
f2('1','3')

泛型的定义
function f3<T>(a:T,b:T):T[]{
        return [a,b]
}
f3<number>(1,2)
f3<string>('a','b')
我们把f3这个函数叫做泛型，因为它适用于所有类型，并且不会有any类型存在的问题
使用‘类型变量’ 一种特殊的变量，代表的是类型而非值
在泛型中，我们要合理正确的使用泛型变量T，要牢记T表示任何类型
```

一旦我们定义了泛型函数，有两种方法可以使用。第一种就是传入所有的参数，包括类型参数:

```
f3<string>(1,2);
f3<string | number>(1,'2');
```

第二种是最常见的。我们使用/类型推断/ 编译器会根据传入的参数自动地帮助我们确定T的类型

```
类型将会是 'string'
f3("myString");
```

泛型类型

```
函数声明
      function mySum<T>(x:T) : T{
            return x;
      }
函数表达式）
      let mySum1 = function<U> (x:U):U {
            return x;
      };
ES6 箭头函数
      let mySum2 = <T>(x:T):T=>x;
```

泛型约束

当你开始使用泛型，你可能会注意到当你创建一个类似"f4"的泛型函数，编译器会强制要求你在函数中正确的使用这些通用类型参数。

```
function f4<T>(arg: T): T {
      console.log(arg.length);  // 错误: T不存在length属性   比如数值等
}
```

使用了 extends 约束了泛型 T 必须符合接口 LengthN 的形状，也就是必须包含 length 属性。

```
interface LengthN {
     length: number;
}
function myHello<T extends LengthN>(arg: T): T {
      console.log(arg.length);
      return arg;
}
myHello<string>('123');
myHello(123);//错误，因为不符合接口要求
```

数组泛型

```
let a: Array<number> = [1, 1, 2, 3, 5];
```

这段代码编译不会报错，但是一个显而易见的缺陷是，它并没有准确的定义返回值的类型

```
function createArray(length: number, value: any): Array<any> {
            let result = [];
            for (let i = 0; i < length; i++) {
                  result[i] = value;
            }
            return result;
}
createArray(3, 'x');  //["x", "x", "x"]
```

在函数名后添加了 ，其中 T 用来指代任意输入的类型

```
function createArray2<T>(length: number, value: T): Array<T> {
      let result: T[] = [];
      for (let i = 0; i < length; i++) {
            result[i] = value;
      }
      return result;
}
createArray2<string>(3, 'x');

泛型接口
interface CreateArrayFunc {
    <T>(length: number, value: T): Array<T>;
}

var createArray5: CreateArrayFunc;
    createArray5 = function<T>(length: number, value: T): Array<T> {
        let result: T[] = [];
        for (let i = 0; i < length; i++) {
            result[i] = value;
        }
        return result;
```

```
    }
createArray5<string>(3, 'x');

也可以把泛型参数提前到接口名上
interface CreateArrayFunc2<T> {
    (length: number, value: T): Array<T>;
}

let createArray6: CreateArrayFunc2<string>;
    createArray6 = function<T>(length: number, value: T): Array<T> {
        let result: T[] = [];
        for (let i = 0; i < length; i++) {
            result[i] = value;
        }
        return result;
    }
createArray6(3, 'x');
```

泛型在类中的运用

```
class A2<T>{
    n:T;
    constructor(num:T){
        this.n = num;
    }
    add(x:T):T{
        return x ;
    }
}
var a2 = new A2<number>(1);
a2.add(3)
```

多个类型参数

```
function multi<N, S>(sum: [N, S]): [S, N] {
     return [sum[1], sum[0]];
}

multi<number,string>([1, 'one']);
```

枚举

enum类型是对JavaScript标准数据类型的一个补充

对于数组中获取数据，只能通过数组来获取，不能通过名称来获取

```
var arr = ['a','b','c'];
arr[0];//可以
arr['a'] //错误  不能通过名称来获取，只能索引
var obj = {a:'a',b:'b'};  //对象
obj.a //'a'
obj['a'] //'a'
```

枚举

```
enum Color {Red, Green, Blue};
let c = Color.Green;
```

默认情况下，从0开始为元素编号。 你也可以手动的指定成员的数值。

```
enum Color2 {Red = 1, Green, Blue};
let c2: Color2 = Color2.Green;
console.log(c2);  //2
```

也可以全部都采用手动赋值

```
enum Color3 {Red = 1, Green =5, Blue =9};
let c3: Color3 = Color3.Green;
console.log(c3);  //5
```

枚举类型提供的一个便利是你可以由枚举的值得到它的名字

例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字

```
enum Color4 {Red = 1, Green =5, Blue=2};
let c4:string = Color4[2];
console.log(c4);  //Blue
```

甚至还可以是小数

```
enum Color5 {Red = 1.6, Green, Blue};
let c5 = Color5.Green;
console.log(c5);  //2.6  步长值+1
```

任意值

```
 enum Color6 {Red = 1.6, Green, Blue=<any>"b"};
 let c6:string = Color6.Blue;
 console.log(c6);
```

如果未手动赋值的枚举项与手动赋值的重复了 后面的会覆盖前面的，尽量不要重复

```
enum Color7 {Red = 3, Green=3, Blue=<any>"b"};
let c7:string = Color7[3];
console.log(c7);
```

枚举项有两种类型：常数项和计算所得项

```
var a8='123';
enum Color8 {Red, Green=<any>a8, Blue = "blue".length};
let c8:Color8 = Color8.Blue;//4
let c8:Color8 = Color8.Green;  //123
```

如果紧接在计算所得项后面的是未手动赋值的项，那么它就会因为无法获得初始值而报错

```
var a9='123';
enum Color9 {Red=<any>a9, Green=<any>a9, Blue = "blue".length};  //可以的
//enum Color9 {Red=<any>a9, Green, Blue};  //这种错误
```

元组

数组合并了相同类型的对象，而元组（Tuple）合并了**不同类型**的对象。

我们知道数组中元素的数据类型都一般是相同的（any[] 类型的数组可以不同），如果存储的元素数据类型不同，则需要使用元组

```
var lists:number[] = [1,2,3];
//数组泛型  不允许有其它的类型
var lists2:Array<string> = ['a','b','c'];
var lists3:Array<string | number> = ['a','b',1,2];
//任意值
var lists4:any[] = ['a',1,true];
```

元组的写法：

```
let x: [number,string];
x = [5,'abc'];
```

元组它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。

1、数组中的数据类型必须和规定的类型顺序对应起来

2、当使用越界索引给数组赋值的时候，会使用联合类型（只要值是规定类型的某一种即可）。

```
 var ff =function(){
     let x1: [string, number];
     x1 = ['abc', 5];
     x1[1] = 10;
     console.log(x1); // ["abc", 10]
 }
 ff();
```

实例扩展

```
//档位
enum Gear {
    First=1, Second=3, Third=5
}
enum Color {
    White, Red
}
//接口
interface Drivable {
    //启动
    start(): void;
    //驾驶
    drive(time: number, speed: Gear): void;
    //行驶距离
    getKilometer(): number
}
```

```
abstract class Car implements Drivable {  //定义一个抽象类  来实现Drivable接口
    protected _isRunning: boolean;   //只能被子类访问的属性
    protected _distanceFromStart: number;  //只能被子类访问的属性

    constructor() {
        this._isRunning = false;
        this._distanceFromStart = 0;
    }

    public start() {    //公共启动汽车
        this._isRunning = true;
    }
    //抽象类的子类必须实现抽象类的抽象方法
    abstract drive(time: number, speed: Gear): void;
    public getKilometer(): number {    //公共行驶距离
        return this._distanceFromStart;
    }
}
// 子类
class BMW<T extends Color> extends Car {
    private color: Color;  //私有属性

    constructor(T) {
        super();
        this.color = T;
    }

    public drive(time: number, speed: Gear): void {
        if (this._isRunning) {
            this._distanceFromStart += time*speed;//行驶距离
        }
    }

    public getColor(): string {
        return Color[this.color]
    }
}

let bmw = new BMW(Color.Red);
bmw.start();
bmw.drive(10, Gear.First);
bmw.drive(60, Gear.Third);
document.body.innerHTML =
"distance:"+bmw.getKilometer()+',color:'+bmw.getColor()
```