# TypeScript

**安装**

npm i –g typescript

**基础语法**

Tsc常用编译参数

–-help显示帮助信息

--module 载入扩展模块

--target 设置ECMA版本

--declaration 额外生成一个.d.ts 扩展名的文件 tsc ts-hw.ts --declaration

--removeComments 删除文件的注释

--out 编译多个文件并合并到一个输出的文件

--sourcemap 生成一个sourcemap文件 存储源代码与编译代码对应位置映射的信息文件

--module nolmplicitAny 在表达式和声明上有隐含的any类型时报错

--watch 在监视模式下运行编译器。会监视输出文件，在它们改变时重新编译

Ts规则 区分大小写、分号可选（建议使用）

**基础类型**

任意类型 any

数据类型 number 双精度64位浮点型

字符串类型 string 单引号\双引来表示字符串类型，反引号定义多行文本和内嵌表达式

布尔类型 Boolean

数组类型 无（？） 例：let arr: number[] = [1, 2]; 或者let arr: Array<number> = [1, 2];

元组 无 用来表示已知元素数量和类型的数组，元素类型不用相同，但需要对应位置 例：

let x: [string, number];

x = ['Runoob', 1]; // 运行正常

x = [1, 'Runoob']; // 报错

console.log(x[0]); // 输出 Runoob

枚举 enum 用于定义数值集合

enum Color {Red, Green, Blue};

let c: Color = Color.Blue;

console.log(c); // 输出 2

void void 用于表示方法返回值的类型，表示没有返回值

function hello(): void {

alert("Hello Runoob");

}

Null 表示对象值缺失

Undefined 未定义

Never 是上述两者的子类型，代表从不会出现的值

Tip 声明变量可以用|来支持多种类型 let x: number | null | undefined;

**变量声明**

var [变量名] : [类型] = 值;

类型断言，允许手动将一个变量转变为一种类型

类型推断，在你声明变量时若未定义类型直接赋值，则自动推断为赋值的类型

变量作用域 包括全局变量，类变量（静态变量：可通过直接类名访问到；）、局部变量

**运算符**

位运算符：位操作是程序设计中对位模式按位或二进制数的一元和二元操作

& 按位与处理两个长度相同的二进制数，两个相应的二进位都为 1，该位的结果值才为 1，否则为 0

例：x=5&1 === 0101&0001 === 0001 === 1

| OR，按位或处理两个长度相同的二进制数，两个相应的二进位中只要有一个为 1，该位的结果值为 1

例：x=5|1 === 0101|0001 === 0101 === 5

~ 取反，取反是一元运算符，对一个二进制数的每一位执行逻辑反操作。使数字 1 成为 0，0 成为 1

例：x=~5 === ~0101 === 1010 === -6

^ 异或，按位异或运算，对等长二进制模式按位或二进制数的每一位执行逻辑异按位或操作。操作的结果是如果某位不同则该位为 1，否则该位为 0

例: x= 5^1 === 0101^0001 === 0100 === 4

<< 左移，把 << 左边的运算数的各二进位全部左移若干位，由 << 右边的数指定移动的位数，高位丢弃，低位补 0

例：x= 5<<1 === 0101<<1 === 1010 === 10

>>右移，把 >> 左边的运算数的各二进位全部右移若干位，>> 右边的数指定移动的位数。

例：x= 5>>1 === 0101>>1 === 0010 === 2

>>>无符号右移，与有符号右移位类似，除了左边一律使用0 补位。

例：x= 2>>>1 === 0010>>1 === 0001 === 1

**函数**

function add(x: number, y: number = 1, …nums:number[]): number { return x + y; }

定义参数类型，定义返回值类型，设置参数默认值,设置剩余参数

**接口**

接口是一系列抽象方法的声明，是一些方法特征的集合，这些方法都应该是抽象的，需要由具体的类去实现，然后第三方就可以通过这组抽象方法调用，让具体的类执行具体的方法

Interface interface\_name {a:number,fun:()=>string}

var className: interface\_name {a:1,fun:():string =>{return ‘string’}}

**类**

class class\_name {

field\_name：string;

static s\_field\_name: any; // 静态变量

protected p\_ field\_name: any

constructor(field\_name:string) {

this. field\_name = field\_name

}

Static static\_fun():void {} // 静态方法

Fun\_name ():void {

//do something

}

}

class class\_name\_2 extends class\_name {

do\_Fun\_name():void {

super.Fun\_name() // 调用父类

this.Fun\_name()

}

}

var a = new class\_name\_2(‘field\_name’)

a.show()

a instanceof class\_name\_2 // true 用于判断对象是否是指定类型

Tip静态变量、静态方法可直接调用，无需实例化

访问控制修饰符

public 公有，可以再任何地方被访问

protected ：受保护，可以被其自身以及子类和父类访问

private：私有，只能被其定义所在的类访问

类和接口：使用关键字implements，并将interest字段作为类的属性使用

interface ILoan {

interest:number

}

class AgriLoan implements ILoan {

interest:number

rebate:number

constructor(interest:number,rebate:number) {

this.interest = interest

this.rebate = rebate

}

}

var obj = new AgriLoan(10,1)

**对象**

鸭子类型（Duck Typing）

鸭子类型（英语：duck typing）是动态类型的一种风格，是多态(polymorphism)的一种形式

在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由"当前方法和属性的集合"决定。

在鸭子类型中，关注点在于对象的行为，能作什么；而不是关注对象所属的类型。例如，在不使用鸭子类型的语言中，我们可以编写一个函数，它接受一个类型为"鸭子"的对象，并调用它的"走"和"叫"方法。在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的"走"和"叫"方法。如果这些需要被调用的方法不存在，那么将引发一个运行时错误。任何拥有这样的正确的"走"和"叫"方法的对象都可被函数接受的这种行为引出了以上表述，这种决定类型的方式因此得名