

TypeScript

- TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准。给 JavaScript 添加特性的语言扩展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。

- **基础类型（变量:类型=值）**

- 布尔值：

- let isDone: boolean = false;

- 数字（支持二，八，十，十六进制）：

- let decLiteral: number = 6;//如果写错类型最后也能出来，因为ts编译成js之后，js中是没有数据类型的，只是报错而已
 - let hexLiteral: number = 0xf00d;
 - let binaryLiteral: number = 0b1010;
 - let octalLiteral: number = 0o744;

- 数组

- let arr1:Array<number>=[1,2];
 - let arr2:string[]=['1','2'];

- 对象

- let obj1:object={name:'zhangsan'};
 - let obj2:{name:string}={name:'zhangsan'};//这种定义比较限制

- 元组

- let tupleARR:[string,number,boolean]=['1',2,true]);//注意顺序一致
 - console.log(tupleARR[0].substr(1)); // OK
 - console.log(tupleARR[1].substr(1)); // Error, 'number' does not have 'substr'

- 枚举,找一个词儿代替某一个，起一个更一目了然的名字

- enum Lev{one='青铜',two='白银',three='黄金'};
 - let myLev:Lev=Lev.two;
 - console.log(myLev)//白银
 - 默认情况下，从0开始为元素编号。也可以手动的指定成员的数值。例如将上面的例子改成从1开始编号;或者全部都采用手动赋值：

- 任意any

- let a:any='any type';

- **接口**

- 属性接口

```
// 属性接口
interface Course{
  title:string,
  intro:string,
  num?:number, // 可选
  [propName:string]:any
}
let hybrid:Course={
  title:'hybrid',
  intro:'混合应用开发',
  num:130
}
```

• 函数接口

```
// 函数接口
interface MyFunc{
  (params1:string):boolean
}
let fun:MyFunc=function(pa:string){
  return true;
}
```

• 类接口

```
// 类接口
interface User{
  name:string,
  age:number,
  pwd:string
}
class User1 implements User{//implements 实现
  name='zhangsan';
  age=20;
  pwd='12345'
}
console.log(new User1);
```

- 继承（extends）：接口继承接口;接口继承类
- 类不能继承接口，类是实现接口

```
interface User2 extends User1{
  work:string
}
```

• 泛型

- 可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。这样用户就可以以自己的数据类型来使用组件（组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型）
- 不用泛型的话，这个函数可能是：（使用any类型会导致这个函数可以接收任何类型的arg参数，这样就丢失了一些信息：传入的类型与返回的类型应该是相同的。如果我们传入一个数字，我们只知道任何类型的值都有可能被返回）

```
function identity(arg: any): any {
    return arg;
}
let zhi=identity('123');
console.log(zhi);//123
```

• 泛型函数

- 需要一种方法使返回值的类型与传入参数的类型是相同的。这里使用了 类型变量，它是一种特殊的变量，只用于表示类型而不是值。
- 给identity添加了类型变量T。T帮助我们捕获用户传入的类型，之后我们就可以使用这个类型。之后我们再次使用了T当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。这允许我们跟踪函数里使用的类型的信息。

```
function identity<T>(arg: T): T {
    return arg;
}
console.log(identity<string>('params1'));//params1
console.log(identity<number>(100));//100
console.log(identity('params1'));//params1
console.log(identity(100));//100
```

- 不带<>确定类型的时候，编译器利用类型推论根据传入的参数自动地帮助我们确定T的类型
- 这个版本的identity函数叫做泛型，因为它可适用于多个类型。不同于any，它不会丢失信息
- 编译器会报错：使用了arg.length属性，但是没有地方指明arg具有这个属性（类型变量代表的是任意类型，所以使用这个函数的人可能传入的是个数字，而数字是没有.length属性的）

```
function Identity<T>(arg: T): T {
    // console.log(arg.length);//报错，T没有length属性
    return arg;
}
```

- 我们还可以使用带有调用签名的对象字面量来定义泛型函数：

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

• 泛型接口

- 把上面例子中的对象字面量拿出来做为一个接口：

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity: GenericIdentityFn = identity;
console.log(myIdentity(100)); //100
```

- 我们可能想把泛型参数当作整个接口的一个参数。这样就能清楚的知道使用的具体是哪个泛型类型（GenericIdentityFn<number>而不只是GenericIdentityFn）这样接口里的其它成员也能知道这个参数的类型了。

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity1: GenericIdentityFn<number> = identity;
let myIdentity2: GenericIdentityFn<string> = identity;
console.log(myIdentity1(100)); //100
console.log(myIdentity2('abc')); //abc
```

- 如果调用了错误的类型，就会报错

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity: GenericIdentityFn<number> = identity;
console.log(myIdentity('100'));
```

类型“”100””的参数不能赋给类型“number”的参数。
ts(2345)

• 泛型类

- 泛型类使用（<>）括起泛型类型，跟在类名后面
- 类有两部分：静态部分和实例部分。泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

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

let myGeneric1 = new Generic<number>();
myGeneric1.value = 0;
myGeneric1.add = function(x, y) { return x + y; };
console.log(myGeneric1.add(myGeneric1.value, 2)); //2

let myGeneric2 = new Generic<string>();
myGeneric2.value = "nn";
myGeneric2.add = function(x, y) { return x + y; };
console.log(myGeneric2.add(myGeneric2.value, "test")); //nntest
```

• 装饰器

- 装饰器是一种特殊类型的声明，它能够被附加到类声明，方法，访问符，属性或参数上。装饰器使用 `@expression` 这种形式，`expression` 求值后必须为一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入。`@expr` 语法其实是语法糖
- 其实就是一个函数，在函数里可以写一些新的逻辑。包裹后面修饰的内容，将新的逻辑传递到被修饰的内容中去
- 高阶组件———其实就是一个函数，就是装饰器
- 普通装饰器（无参数）

```
// 普通装饰器（无参数）
function helloWord(target: any) { // target 指代的就是后边的类
  // do something with "target"
  console.log('hello Word!'); // hello Word!
}
@helloWord
class HelloWorldClass {
}
```

```
function addUrl(target: any) { // target 指代的就是后边的类
  target.prototype.url = 'http://';
}
@addUrl
class HomeServer {
  url: any;
  getData() {
    console.log(this.url); // http://
  }
}
let home = new HomeServer();
home.getData()
```

• 装饰器工厂（带参数）

- 定制一个修饰器如何应用到一个声明上，我们得写一个装饰器工厂函数。装饰器工厂就是一个简单的函数，它返回一个表达式，以供装饰器在运行时调用

```
// 装饰器工厂（带参数）
function helloWord(p: string) { // 这是一个装饰器工厂
  return function (target: any) { // 这才是真正装饰器
    // do something with "target" and "p"
    console.log(p)
  }
}
@helloWord('hello')
class HelloWorldClass {
}
```

• 类装饰器

- 类装饰器在类声明之前被声明（紧靠着类声明）。类装饰器应用于类构造函数

，可以用来监视，修改或替换类定义。类装饰器表达式会在运行时当作函数被调用，类的构造函数作为其唯一的参数。

- 当@sealed被执行的时候，它将密封此类的构造函数和原型。
- Object.seal()方法封闭一个对象，阻止添加新属性并将所有现有属性标记为不可配置。当前属性的值只要原来是可写的就可以改变。

```
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;
        console.log("Hello, " + this.greeting);
    }
}
let home=new Greeter('hi');
home.greet();
```

• 方法装饰器

- 方法装饰器声明在一个方法的声明之前（紧靠着方法声明）。它会被应用到方法的属性描述符上，可以用来监视，修改或者替换方法定义
- 方法装饰器表达式会在运行时当作函数被调用，传入下列3个参数：（如果方法装饰器返回一个值，它会被用作方法的属性描述符。）
 - 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
 - 成员的名字。
 - 成员的属性描述符。

```
// 方法装饰器
function enumerable(value: boolean) { // 因为修饰的是方法（实例成员）所以target是类原型对象，不是他的构造函数
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        target.name = 'mu'; // 在原型上直接加一个属性
        descriptor.enumerable = value; // 修改属性描述符的enumerable属性为false
        console.log(propertyKey) // 成员名字: greet
        console.log(descriptor)
        // 成员描述符: {"configurable": true, "enumerable": false, "value": [Function greet], "writable": true}
    };
}
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    @enumerable(false)
    greet() {
        return "Hello, " + this.greeting;
    }
}
console.log(new Greeter('world').name) // mu
```

- 这里的@enumerable(false)是一个装饰器工厂（应用于Greeter类的方法上）。当装饰器 @enumerable(false)被调用时，它会修改属性描述符的enumerable属性。

- 属性装饰器

- 属性装饰器声明在一个属性声明之前（紧靠着属性声明）属性装饰器表达式会在运行时当作函数被调用，传入下列2个参数：
 - 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
 - 成员的名字。

- 参数装饰器

- 参数装饰器声明在一个参数声明之前（紧靠着参数声明）。参数装饰器应用于类构造函数或方法声明；参数装饰器表达式会在运行时当作函数被调用，传入下列3个参数：（参数装饰器只能用来监视一个方法的参数是否被传入且参数装饰器返回值会被忽略）
 - 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
 - 成员的名字。
 - 参数在函数参数列表中的索引。

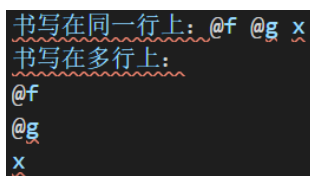
```
function DefaultValue(value: string) { //target是类的构造函数
    return function (target: any, propertyName: string) {
        console.log(propertyName); //greeting
        target[propertyName] = value;
    }
}

class Hello {
    @DefaultValue("Hello")
    greeting: string;
}

console.log(new Hello().greeting); //hello
```

- 装饰器组合

- 多个装饰器可以同时应用到一个声明上



书写在同一行上: @f @g x
书写在多行上:
@f
@g
x

- 当多个装饰器应用于一个声明上，它们求值方式与复合函数相似。在这个模型下，当复合f和g时，复合的结果(f g)(x)等同于f(g(x))
- 同样的，在TypeScript里，当多个装饰器应用在一个声明上时会进行如下步骤的操作：
 - 由上至下依次对装饰器表达式求值。
 - 求值的结果会被当作函数，由下至上依次调用。

```
function f() {  
  console.log("f(): evaluated");//1  
  return function (target:any, propertyKey: string, descriptor: PropertyDescriptor) {  
    console.log("f(): called");//4  
  }  
}  
  
function g() {  
  console.log("g(): evaluated");//2  
  return function (target:any, propertyKey: string, descriptor: PropertyDescriptor) {  
    console.log("g(): called");//3  
  }  
}  
  
class C {  
  @f()  
  @g()  
  method() {}  
}
```

```
LOG f(): evaluated  
LOG g(): evaluated  
LOG g(): called  
LOG f(): called
```