class 类型

1. 简介

1.1. 属性的类型

类的属性可以在顶层声明，也可以在构造方法内部声明。 对于顶层声明的属性，可以在声明时同时给出类型。

```
class Point {
  x:number;
  y:number;
}
```

上面声明中，属性 x 和 y 的类型都是 number。如果不给出类型，TypeScript 会认为 x 和 y 的类型都是 any。

```
class Point {
  x;
  y;
}
```

上面示例中，x 和 y 的类型都是 any。

如果声明时给出初值，可以不写类型，TypeScript 会自行推断属性的类型。

```
class Point {
  x = 0;
  y = 0;
}
```

上例中，属性 x 和 y 的类型都会被推断为 number。

1.2. readonly 修饰符

属性名前面加上 readonly 修饰符，就表示该属性是只读的。实例对象不能修改这个属性。

```
class A {
  readonly id = 'foo';
}
const a = new A();
a.id = 'bar'; // Cannot assign to 'id' because it is a read-only property. 不能赋值给'id'，因为它是一个只读属性。
```

readonly 属性的初始值，可以写在顶层属性，也可以写在构造方法里面。

```
class A {
  readonly id:string;
  constructor() {
    this.id = 'bar'; // 正确
  }
  fn1() {
    this.id = 'a'; // Cannot assign to 'id' because it is a read-only property.
  }
}
```

上例中，构造方法内部设置只读属性的初值，这是可以的。但在其他地方修改就会报错。

```
class A {
  readonly id:string = 'foo';
  constructor() {
    this.id = 'bar'; // 正确
  }
  fn1() {
    this.id = 'a'; // Cannot assign to 'id' because it is a read-only property.
  }
}
```

上例中，构造方法修改只读属性的值也是可以的。或者说，如果两个地方都设置了只读属性的值，以构造方法为准。**在其他方法修改只读属性都会报错。**

1.3. 方法的类型

类的方法就是普通函数，类型声明方式与函数一致。

```
class Point {
  x:number;
  y:number;

  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }

  add(point:Point) {
    return new Point(
      this.x + point.x,
      this.y + point.y
    );
  }
}
```

上例中，构造方法 `constructor()` 和普通方法 `add()` 都注明了参数类型，但是省略了返回值类型，因为 TypeScript 可以自己推断出来。

类的方法跟普通函数一样，可以使用参数默认值，以及函数重载。

参数默认值：

```
class Point {
  x: number;
  y: number;

  constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
  }
}
```

上例中，如果新建实例时，不提供属性 `x` 和 `y` 的值，它们都等于默认值 `0`。

函数重载：

```
class Point {
  constructor(x:number, y:string);
  constructor(s:string);
  constructor(xs:number|string, y?:string) {
    // ...
  }
}
```

上例中，构造方法可以接受一个参数，也可以接受两个参数，采用函数重载进行类型声明。

构造方法不能声明返回值类型，否则报错，因为它总是返回实例对象。

```
class B {
  constructor():object { // Type annotation cannot appear on a constructor
declaration. 类型注释不能出现在构造函数声明中。
    // ...
  }
}
```

1.4. 存取器方法

存取器（accessor）是特殊的类方法，包括取值器（`getter`）和存值器（`setter`）两种方法。它们用于读写某个属性，取值器用来读取属性，存值器用来写入属性。

```
class C {
  _name = '';
  get name() {
    return this._name;
  }
  set name(value) {
    this._name = value;
  }
}
```

上例中，`get name()` 是取值器，其中 `get` 是关键词，`name` 是属性名。外部读取 `name` 属性时，实例对象会自动调用这个方法，该方法的返回值就是 `name` 属性的值。`set name()` 是存值器，其中 `set` 是关键词，`name` 是属性名。外部写入 `name` 属性时，实例对象会自动调用这个方法，并将所赋的值作为函数参数传入。

TypeScript 对存取器有以下规则。

（1）**如果某个属性只有 `get` 方法，没有set方法，那么该属性自动成为只读属性。**

```
class C {
  _name = 'foo';

  get name() {
    return this._name;
  }
}

const c = new C();
c.name = 'bar'; // Cannot assign to 'name' because it is a read-only property.
```

（2）TypeScript 5.1 版之前，set方法的参数类型，必须兼容get方法的返回值类型，否则报错。

```
// TypeScript 5.1 版之前
class C {
  _name = '';
  get name():string {  // 报错
    return this._name;
  }
  set name(value:number) {
    this._name = String(value);
  }
}
```

上面示例中，`get` 方法的返回值类型是字符串，与 `set` 方法的参数类型 `number` 不兼容，导致报错。改成下面这样，就不会报错。

```

```

```
class C {
  _name = '';
  get name():string {
    return this._name;
  }
  set name(value:number|string) {
    this._name = String(value);
  }
}
```

上例中，`set` 方法的参数类型（`number|string`）兼容 `get` 方法的返回值类型（`string`），这是允许的。

TypeScript 5.1 版做出了改变，现在两者可以不兼容。

（3）**get方法与set方法的可访问性必须一致，要么都为公开方法，要么都为私有方法。**

1.5. 属性索引

类允许定义属性索引。

```
class MyClass {
  [s:string]: boolean |
    ((s:string) => boolean);

  get(s:string) {
    return this[s] as boolean;
  }
}
```

上例中，`[s:string]` 表示所有属性名类型为字符串的属性，它们的属性值要么是布尔值，要么是返回布尔值的函数。

由于类的方法是一种特殊属性（属性值为函数的属性），所以属性索引的类型定义也涵盖了方法。如果一个对象同时定义了属性索引和方法，那么前者必须包含后者的类型。

```
class MyClass {
  [s:string]: boolean;
  f() { // Property 'f' of type '() => boolean' is not assignable to 'string'
index type 'boolean'. 属性'f'的类型'()=&gt;' Boolean '不能赋值给'string'索引类型'
Boolean '。
    return true;
  }
}
```

上例中，属性索引的类型里面不包括方法，导致后面的方法 `f()` 定义直接报错。正确的写法是：

```
class MyClass {
  [s:string]: boolean | (() => boolean);
  f() {
    return true;
  }
}
```

属性存取器视同属性。

```
class MyClass {
  [s:string]: boolean;

  get isInstance() {
    return true;
  }
}
```

上例中，属性 `inInstance` 的读取器虽然是一个函数方法，但是视同属性，所以属性索引虽然没有涉及方法类型，但是不会报错。