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。

2. 类的 interface 接口

2.1. implements 关键字

```
interface Point {
  x: number;
  y: number;
}
class MyPoint implements Point {
  x = 1;
  y = 1;
  z:number = 1;
}
```

上例中，`MyPoint` 类实现了 `Point` 接口，但是内部还定义了一个额外的属性 `z`。

`implements` 关键字后面，不仅可以是接口，也可以是另一个类。这时，后面的类将被当作接口。

```
class Car {
  id:number = 1;
  move():void {};
}
class MyCar implements Car {
  id = 2;          // 不可省略
  move():void {}; // 不可省略
}
```

上例中，`implements` 后面是类 `Car`，这时 TypeScript 就把 `Car` 视为一个接口，要求 `MyCar` 实现 `Car` 里面的每一个属性和方法，否则就会报错。所以，这时不能因为 `Car` 类已经实现过一次，而在`MyCar` 类省略属性或方法。

`interface` 描述的是类的对外接口，也就是实例的公开属性和公开方法，不能定义私有的属性和方法。这是因为 TypeScript 设计者认为，私有属性是类的内部实现，接口作为模板，不应该涉及类的内部代码写法。

```
interface Foo {
  private member:{}; // 'private' modifier cannot appear on a type member.
“private”修饰符不能出现在类型成员上。
}
```

2.2. 实现多个接口

类可以实现多个接口（其实是接受多重限制），每个接口之间使用逗号分隔。

```
class Car implements MotorVehicle, Flyable, Swimmable { /* */ }
```

上面示例中，`Car` 类同时实现了 `MotorVehicle`、`Flyable`、`Swimmable` 三个接口。 **这意味着，它必须部署这三个接口声明的所有属性和方法，满足它们的所有条件。**

但是，同时实现多个接口并不是一个好的写法，容易使得代码难以管理，可以使用两种方法替代。

第一种方法是类的继承。

```
class Car implements MotorVehicle {}
class SecretCar extends Car implements Flyable, Swimmable {}
```

上例中，`Car` 类实现了 `MotorVehicle`，而 `SecretCar` 类继承了 `Car` 类，然后再实现 `Flyable` 和 `Swimmable` 两个接口，相当于 `SecretCar` 类同时实现了三个接口。

第二种方法是接口的继承。

```
interface A {
  a:number;
}
interface B extends A {
  b:number;
}
```

上面示例中，接口 `B` 继承了接口 `A`，类只要实现接口 `B`，就相当于实现 `A` 和 `B` 两个接口。

前一个例子可以用接口继承改写。

```
interface MotorVehicle { /* */ }
interface Flyable { /* */ }
interface Swimmable { /* */ }

interface SuperCar extends MotoVehicle,Flyable, Swimmable { /* */ }

class SecretCar implements SuperCar { /* */ }
```

上面示例中，类 `SecretCar` 通过 `SuperCar` 接口，就间接实现了多个接口。

发生多重实现时（即一个接口同时实现多个接口），不同接口不能有互相冲突的属性。

```
interface Flyable {
  foo:number;
}
interface Swimmable {
  foo:string;
}
```

上例中，属性 `foo` 在两个接口里面的类型不同，如果同时实现这两个接口，就会报错。

2.3. 类和接口的合并

如果一个类和一个接口同名，那么接口会被合并进类。

```
class A {
  x:number = 1;
}
interface A {
  y:number;
}

let a = new A();
a.y = 10;

a.x; // 1
a.y; // 10
```

上例中，类 `A` 与接口 `A` 同名，后者会被合并进前者的类型定义。

合并进类的非空属性（上例的 `y`），如果在赋值之前读取，会返回 `undefined`。

```
class A {
  x:number = 1;
}
interface A {
  y:number;
}
let a = new A();
a.y; // undefined
```

3. Class 类型

3.1. 实例类型

TypeScript 的类本身就是一种类型，但是它代表该类的实例类型，而不是 class 的自身类型。

```
class Color {
  name:string;
  constructor(name:string) {
    this.name = name;
  }
}

const green:Color = new Color('green');
```

上例中，定义了一个类 `Color`。它的类名就代表一种类型，实例对象 `green` 就属于该类型。

对于引用实例对象的变量来说，既可以声明类型为 `Class`，也可以声明类型为 `Interface`，因为两者都代表实例对象的类型。

```
interface MotorVehicle {}
class Car implements MotorVehicle {}

const c1:Car = new Car();          // 写法一
const c2:MotorVehicle = new Car(); // 写法二
```

上例中，变量的类型可以写成类 `Car`，也可以写成接口 `MotorVehicle`。它们的区别是，如果类 `Car` 有接口 `MotoVehicle` 没有的属性和方法，那么只有变量 `c1` 可以调用这些属性和方法。

作为类型使用时，类名只能表示实例的类型，不能表示类的自身类型。

```
class Point {
  x:number;
  y:number;
  constructor(x:number, y:number) {
    this.x = x;
    this.y = y;
  }
}
function createPoint(
  PointClass:Point,
  x: number,
  y: number
) {
  return new PointClass(x, y); // This expression is not constructable. Type
'Point' has no construct signatures. 这个表达式不能构造。类型“Point”没有构造签名。
}
```

上例中，函数 `createPoint()` 的第一个参数 `PointClass`，需要传入 `Point` 这个类，但是如果把参数的类型写成 `Point` 就会报错，因为 `Point` 描述的是实例类型，而不是 `Class` 的自身类型。

由于类名作为类型使用，实际上代表一个对象，因此可以把类看作为对象类型起名。事实上，TypeScript 有三种方法可以为对象类型起名：type、interface 和 class。

3.2. 类的自身类型

要获得一个类的自身类型，一个简便的方法就是使用 typeof 运算符。

```
function createPoint(
  PointClass:typeof Point,
  x:number,
  y:number
):Point {
  return new PointClass(x, y);
}
```

上例中，`createPoint()` 的第一个参数 `PointClass` 是 `Point` 类自身，要声明这个参数的类型，简便的方法就是使用 `typeof Point`。因为 `Point` 类是一个值，`typeof Point` 返回这个值的类型。注意，`createPoint()` 的返回值类型是`Point`，代表实例类型。

3.3. 结构类型原则

`Class` 也遵循"结构类型原则"。一个对象只要满足 `Class` 的实例结构，就跟该 `Class` 属于同一个类型。

```
class Foo {
  id!:number;
}
function fn(arg:Foo) { /* */ }
const bar = {
  id: 10,
  amount: 100,
};
fn(bar); // 正确
```

上例中，对象 `bar` 满足类 `Foo` 的实例结构，只是多了一个属性 `amount`。所以，它可以当作参数，传入函数 `fn()`。

如果两个类的实例结构相同，那么这两个类就是兼容的，可以用在对方的使用场合。 两个结构相同的类，被视为相同的类。

```
class Person {
  name: string = '';
}
class Customer {
  name: string = '';
}
const c:Customer = new Person();
```

上例中，Person 和 Customer 是两个结构相同的类，TypeScript 将它们视为相同类型，因此 Person 可以用在类型为 Customer 的场合。

为 Person 类添加一个属性：

```
class Person {
  name: string = '';
  age: number = '';
}
class Customer {
  name: string = '';
}
const c:Customer = new Person();
```

上例中，Person 类添加了一个属性 age，跟 Customer 类的结构不再相同。但是这种情况下，TypeScript 依然认为，Person 属于 Customer 类型。

这是因为根据"结构类型原则"，只要 Person 类具有 name 属性，就满足 Customer 类型的实例结构，所以可以代替它。反过来就不行，如果 Customer 类多出一个属性，就会报错。

```
class Person {
  name: string = '';
}
class Customer {
  name: string = '';
  age: number = 0;
}
const c:Customer = new Person(); // Property 'age' is missing in type 'Person' but
required in type 'Customer'.
```

上例中，Person 类比 Customer 类少一个属性 age，它就不满足 Customer 类型的实例结构，就报错了。因为在使用 Customer 类型的情况下，可能会用到它的 age 属性，而 Person 类就没有这个属性。只要 A 类具有 B 类的结构，哪怕还有额外的属性和方法，TypeScript 也认为 A 兼容 B 的类型。

不仅是类，**如果某个对象跟某个 class 的实例结构相同，TypeScript 也认为两者的类型相同。**

```
class Person {
  name: string = '';
}
const obj = { name: 'John' };
const p:Person = obj;
```

上例中，对象 obj 并不是 Person 的实例，但是赋值给变量 p 不会报错，TypeScript 认为 obj 也属于 Person 类型，因为它们的属性相同。

由于这种情况，运算符 instanceof 不适用于判断某个对象是否跟某个 class 属于同一类型。obj instanceof Person 结果是 false。

空类不包含任何成员，任何其他类都可以看作与空类结构相同。因此，凡是类型为空类的地方，所有类（包括对象）都可以使用。

```
class Empty {}
function fn(x:Empty) { /* */ }
fn({});
fn(window);
fn(fn);
```

上例中，函数 fn() 的参数是一个空类，这意味着任何对象都可以用作 fn() 的参数。

确定两个类的兼容关系时，只检查实例成员，不考虑静态成员和构造方法。

```
class Point {
  x: number;
  y: number;
  static t: number;
  constructor(x:number) {}
}
class Position {
  x: number;
  y: number;
  z: number;
  constructor(x:string) {}
}
const point:Point = new Position('');
```

上例中，Point 与 Position 的静态属性和构造方法都不一样，但因为 Point 的实例成员与 Position 相同，所以 Position 兼容 Point。

如果类中存在私有成员（private）或保护成员（protected），那么确定兼容关系时，TypeScript 要求私有成员和保护成员来自同一个类，这意味着两个类需要存在继承关系。

```
// 情况一
class A {
  private name = 'a';
}
class B extends A {}
const a:A = new B();

// 情况二
class A {
  protected name = 'a';
}
class B extends A {
```

```
  protected name = 'b';
}
const a:A = new B();
```

上例中，A 和 B 都有私有成员（或保护成员）name，这时只有在 B 继承 A 的情况下（class B extends A），B 才兼容 A。

4. 类的继承

类（这里又称“子类”）可以使用 extends 关键字继承另一个类（这里又称“基类”）的所有属性和方法。

```
class A {
  greet() {
    console.log('Hello, world!');
  }
}
class B extends A {}

const b = new B();
b.greet(); // "Hello, world!"
```

上例中，子类 B 继承了基类 A，因此就拥有了 greet() 方法，不需要再次在类的内部定义这个方法了。

根据结构类型原则，子类也可以用于类型为基类的场合。

```
const a:A = b;
a.greet();
```

上例中，变量 a 的类型是基类，但是可以赋值为子类的实例。

子类可以覆盖基类的同名方法。

```
class B extends A {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name}`);
    }
  }
}
```

上例中，子类 B 定义了一个方法 greet()，覆盖了基类 A 的同名方法。其中，参数 name 省略时，就调用基类 A 的 greet() 方法，这里可以写成 super.greet()，使用 super 关键字指代基类是常见做法。

子类的同名方法不能与基类的类型定义相冲突。

```
class A {
  greet() {
    console.log('Hello, world!');
  }
}

class B extends A {
  greet(name:string) { // Property 'greet' in type 'B' is not assignable to the
same property in base type 'A'.
    console.log(`Hello, ${name}`);
  }
}
```

上例中，子类 `B` 的 `greet()` 有一个 `name` 参数，跟基类 `A` 的 `greet()` 定义不兼容，因此就报错了。

如果基类包括保护成员（`protected` 修饰符），子类可以将该成员的可访问性设置为公开（`public` 修饰符），也可以保持保护成员不变，但是不能改用私有成员（`private`修饰符）。

```
class A {
  protected x: string = '';
  protected y: string = '';
  protected z: string = '';
}

class B extends A {
  public x:string = '';    // 正确
  protected y:string = ''; // 正确
  private z: string = '';  // 报错
}
```

上面示例中，子类 `B` 将基类 `A` 的受保护成员改成私有成员，就会报错。

`extends` 关键字后面不一定是类名，可以是一个表达式，只要它的类型是构造函数就可以了。

```
// 例一
class MyArray extends Array<number> {}

// 例二
class MyError extends Error {}

// 例三
class A {
  greeting() {
    return 'Hello from A';
  }
}
class B {
  greeting() {
```

```
    return 'Hello from B';
  }
}

interface Greeter {
  greeting(): string;
}

interface GreeterConstructor {
  new (): Greeter;
}

function getGreeterBase():GreeterConstructor {
  return Math.random() >= 0.5 ? A : B;
}

class Test extends getGreeterBase() {
  sayHello() {
    console.log(this.greeting());
  }
}
```

上面示例中，例一和例二的 `extends` 关键字后面都是构造函数，例三的 `extends` 关键字后面是一个表达式，执行后得到的也是一个构造函数。

5. 可访问性修饰符

类的内部成员的外部可访问性，由三个可访问性修饰符（access modifiers）控制：`public`、`private` 和 `protected`。这三个修饰符的位置，都写在属性或方法的最前面。

5.1. public

`public` 修饰符表示这是公开成员，外部可以自由访问。`public` 修饰符是默认修饰符，如果省略不写，实际上就带有该修饰符。因此，类的属性和方法默认都是外部可访问的。正常情况下，除非为了醒目和代码可读性，public都是省略不写的。

```
class Greeter {
  public greet() {
    console.log("hi!");
  }
}

const g = new Greeter();
g.greet();
```

上例中，`greet()` 方法前面的 `public` 修饰符，表示该方法可以在类的外部调用，即外部实例可以调用。

5.2. private

`private` **修饰符表示私有成员，只能用在当前类的内部，类的实例和子类都不能使用该成员。**

```
class A {
  private x:number = 0;
}

const a = new A();
a.x // Property 'x' is private and only accessible within class 'A'.

class B extends A {
  showX() {
    console.log(this.x); // Property 'x' is private and only accessible within
class 'A'.
  }
}
```

上例中，属性 `x` 前面有 `private` 修饰符，表示这是私有成员。因此，实例对象和子类使用该成员，都会报错。

子类不能定义父类私有成员的同名成员。

```
class A {
  private x = 0;
}
class B extends A {
  x = 1; // Class 'B' incorrectly extends base class 'A'. Property 'x' is private
in type 'A' but not in type 'B'. 'x' is declared but its value is never read.
类'B'错误地扩展了基类'A'。属性'x'在类型'A'中是私有的，但在类型'B'中不是。声明了'x'，但永
远不会读取它的值。
}
```

上例中，`A` 类有一个私有属性 `x`，子类 `B` 就不能定义自己的属性 `x` 了。

如果在类的内部，当前类的实例可以获取私有成员。

```
class A {
  private x = 10;
  f(obj:A) {
    console.log(obj.x);
  }
}

const a = new A();
a.f(a); // 10
```

上例中，在类 `A` 内部，`A` 的实例对象可以获取私有成员x。

严格地说，private定义的私有成员，并不是真正意义的私有成员。一方面，编译成 JavaScript 后，`private` 关键字就被剥离了，这时外部访问该成员就不会报错。另一方面，由于前一个原因，TypeScript 对于访问 `private` 成员没有严格禁止，使用方括号写法（`[]`）或者 `in` 运算符，实例对象就能访问该成员。

```
class A {
  private x = 1;
}

const a = new A();
a.x; // Property 'x' is private and only accessible within class 'A'.
a['x']; // 正确

if ('x' in a) { // 正确
  // ...
}
```

上例中，`A` 类的属性 `x` 是私有属性，但是实例使用方括号，就可以读取这个属性，或者使用 `in` 运算符检查这个属性是否存在，都可以正确执行。

```
// es2022 的写法
class A {
  #x = 1;
}

const a = new A();
a['x']; // Element implicitly has an 'any' type because expression of type '"x"'
can't be used to index type 'A'. Property 'x' does not exist on type 'A'. 元素隐式
地具有'any'类型，因为类型' x '的表达式不能用于索引类型'A'。 类型'A'上不存在属性'x'。
```

构造方法也可以是私有的，这就直接防止了使用 `new` 命令生成实例对象，只能在类的内部创建实例对象。这时一般会有一个静态方法，充当工厂函数，强制所有实例都通过该方法生成。

```
class Singleton {
  private static instance?: Singleton;
  private constructor() {}
  static getInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}
const s = Singleton.getInstance();
```

上例使用私有构造方法，实现了单例模式。想要获得 `Singleton` 的实例，不能使用 `new` 命令，只能使用 `getInstance()` 方法。

5.3. protected

`protected` 修饰符表示该成员是保护成员，只能在类的内部使用该成员，实例无法使用该成员，但是子类内部可以使用。

```
class A {
  protected x = 1;
}
class B extends A {
  getX() {
    return this.x;
  }
}

const a = new A();
const b = new B();

a.x; // Property 'x' is protected and only accessible within class 'A' and its
subclasses.
b.getX(); // 1
```

子类不仅可以拿到父类的保护成员，还可以定义同名成员。

```
class A {
  protected x = 1;
}
class B extends A {
  x = 2;
}
```

上面示例中，子类 `B` 定义了父类 `A` 的同名成员 `x`，并且父类的 `x` 是保护成员，子类将其改成了公开成员。`B` 类的 `x` 属性前面没有修饰符，等同于修饰符是 `public`，外界可以读取这个属性。

在类的外部，实例对象不能读取保护成员，但是在类的内部可以。

```
class A {
  protected x = 1;
  f(obj:A) {
    console.log(obj.x);
  }
}
const a = new A();
a.x; // Property 'x' is protected and only accessible within class 'A' and its
subclasses.
a.f(a); // 1
```

上例中，属性 `x` 是类 `A` 的保护成员，在类的外部，实例对象 `a` 拿不到这个属性。但是，实例对象 `a` 传入类 `A` 的内部，就可以从 `a` 拿到 `x`。

5.4. 实例属性的简写形式

实际开发中，很多实例属性的值，是通过构造方法传入的。

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

上例中，属性 x 和 y 的值是通过构造方法的参数传入的。

这样的写法等于对同一个属性要声明两次类型，一次在类的头部，另一次在构造方法的参数里面。这有些累赘，TypeScript 就提供了一种简写形式。

```
class Point {
  constructor(
    public x:number,
    public y:number
  ) {}
}
const p = new Point(10, 10);
p.x; // 10
p.y; // 10
```

上例中，构造方法的参数 x 前面有 public 修饰符，这时 TypeScript 就会自动声明一个公开属性 x，不必在构造方法里面写任何代码，同时还会设置 x 的值为构造方法的参数值。这里的 public 不能省略。

除了 public 修饰符，构造方法的参数名只要有 private、protected、readonly 修饰符，都会自动声明对应修饰符的实例属性。

```
class A {
  constructor(
    public a: number,
    protected b: number,
    private c: number,
    readonly d: number
  ) {}
}
```

```
"use strict";
class A {
  constructor(a, b, c, d) {
```

```
    this.a = a;
    this.b = b;
    this.c = c;
    this.d = d;
  }
}
```

`readonly` 还可以与其他三个可访问性修饰符，一起使用。

```
class A {
  constructor(
    public readonly x:number,
    protected readonly y:number,
    private readonly z:number
  ) {}
}
```

6. 静态成员

类的内部可以使用 `static` 关键字，定义静态成员。静态成员是只能通过类本身使用的成员，不能通过实例对象使用。

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }
}
MyClass.x // 0
MyClass.printX() // 0

mc.x; // Property 'x' does not exist on type 'MyClass'. Did you mean to access the
static member 'MyClass.x' instead?
mc.printX(); // Property 'printX' does not exist on type 'MyClass'. Did you mean
to access the static member 'MyClass.printX' instead?
```

上例中，`x` 是静态属性，`printX()` 是静态方法。它们都必须通过 `MyClass` 获取，而不能通过实例对象调用。

`static` 关键字前面可以使用 `public`、`private`、`protected` 修饰符。

```
class MyClass {
  private static x = 0;
}
MyClass.x // Property 'x' is private and only accessible within class 'MyClass'.
```

上例中，静态属性 `x` 前面有 `private` 修饰符，表示只能在 `MyClass` 内部使用，如果在外部调用这个属性就会报错。

静态私有属性也可以用 ES6 语法的 `#` 前缀表示：

```
class MyClass {
  static #x = 0;
}
```

`public` 和 `protected` 的静态成员可以被继承。

```
class A {
  public static x = 1;
  protected static y = 1;
}
class B extends A {
  static getY() {
    return B.y;
  }
}

B.x;      // 1
B.getY(); // 1
```

上例中，类 `A` 的静态属性 `x` 和 `y` 都被 `B` 继承，公开成员 `x` 可以在 `B` 的外部获取，保护成员 `y` 只能在 `B` 的内部获取。

7. 泛型类

类也可以写成泛型，使用类型参数。

```
class Box<Type> {
  contents: Type;
  constructor(value:Type) {
    this.contents = value;
  }
}
const b:Box<string> = new Box('hello!');
```

上例中，类 `Box` 有类型参数 `Type`，因此属于泛型类。新建实例时，变量的类型声明需要带有类型参数的值，不过本例等号左边的 `Box<string>` 可以省略不写，因为可以从等号右边推断得到。

静态成员不能使用泛型的类型参数。

```
class Box<Type> {
  static defaultContents: Type; // 报错
```

```
}
```

上例中，静态属性 `defaultContents` 的类型写成类型参数 `Type` 会报错。因为这意味着调用时必须给出类型参数（即写成 `Box<string>.defaultContents`），并且类型参数发生变化，这个属性也会跟着变，这并不是好的做法。

8.抽象类，抽象成员

TypeScript 允许在类的定义前面，加上关键字 `abstract`，表示该类不能被实例化，只能当作其他类的模板。这种类就叫做“抽象类”（abstract class）。

```
abstract class A {
  id = 1;
}
const a = new A(); // Cannot create an instance of an abstract class.
```

抽象类只能当作基类使用，用来在它的基础上定义子类。

```
abstract class A {
  id = 1;
}
class B extends A {
  amount = 100;
}
const b = new B();
b.id;     // 1
b.amount; // 100
```

上例中，`A` 是一个抽象类，`B` 是 `A` 的子类，继承了 `A` 的所有成员，并且可以定义自己的成员和实例化。

抽象类的子类也可以是抽象类，也就是说，抽象类可以继承其他抽象类。

```
abstract class A {
  foo:number;
}
abstract class B extends A {
  bar:string;
}
```

抽象类的内部可以有已经实现好的属性和方法，也可以有还未实现的属性和方法。后者就叫做“抽象成员”（`abstract member`），即属性名和方法名有 `abstract` 关键字，表示该方法需要子类实现。如果子类没有实现抽象成员，就会报错。

```

```

```
abstract class A {
  abstract foo:string;
  bar:string = '';
}
class B extends A { // Non-abstract class 'B' does not implement all abstract
members of 'A' 非抽象类'B'没有实现'A'的所有抽象成员
  foo2 = 'b';
}
```

上例中，抽象类 `A` 定义了抽象属性 `foo`，子类 `B` 必须实现这个属性，否则会报错。

如果抽象类的方法前面加上 `abstract`，就表明子类必须给出该方法的实现。

```
abstract class A {
  abstract execute():string;
}
class B extends A {
  execute() {
    return `B executed`;
  }
}
```

这里有几个注意点。

- （1）抽象成员只能存在于抽象类，不能存在于普通类。
- （2）抽象成员不能有具体实现的代码。也就是说，已经实现好的成员前面不能加 `abstract` 关键字。
- （3）抽象成员前也不能有 `private` 修饰符，否则无法在子类中实现该成员。
- （4）**一个子类最多只能继承一个抽象类。**

总之，抽象类的作用是，确保各种相关的子类都拥有跟基类相同的接口，可以看作是模板。其中的抽象成员都是必须由子类实现的成员，非抽象成员则表示基类已经实现的、由所有子类共享的成员。

9. this 问题

类的方法经常用到 `this` 关键字，它表示该方法当前所在的对象。

```
class A {
  name = 'A';
  getName() {
    return this.name;
  }
}

const a = new A();
a.getName() // 'A'
```

```
const b = {
  name: 'b',
  getName: a.getName
};
b.getName(); // 'b'
```

上例中，变量 a 和 b 的 getName() 是同一个方法，但是执行结果不一样，原因就是它们内部的 this 指向不一样的对象。如果 getName() 在变量 a 上运行，this 指向 a；如果在 b 上运行，this 指向 b。

有些场合需要给出 this 类型，但是 JavaScript 函数通常不带有 this 参数，这时 TypeScript 允许函数增加一个名为 this 的参数，放在参数列表的第一位，用来描述函数内部的 this 关键字的类型。

```
// 编译前
function fn(
  this: SomeType,
  x: number
) { /* */ }

// 编译后
function fn(x) { /* */ }
```

上例中，函数 fn() 的第一个参数是 this，用来声明函数内部的 this 的类型。编译时，TypeScript 一旦发现函数的第一个参数名为 this，则会去除这个参数，即编译结果不会带有该参数。

```
class A {
  name = 'A';
  getName(this: A) {
    return this.name;
  }
}

const a = new A();
const b = a.getName;

b(); // The 'this' context of type 'void' is not assignable to method's 'this' of
type 'A'. 类型'void'的'this'上下文不能分配给类型'A'的方法'this'。
```

上例中，类 A 的 getName() 添加了 this 参数，如果直接调用这个方法，this 的类型就会跟声明的类型不一致，从而报错。

this 参数的类型可以声明为各种对象。

```
function foo(
  this: { name: string }
) {
  this.name = 'Jack';
  this.name = 0; // Type 'number' is not assignable to type 'string'.
```

```
}

foo.call({ name: 123 }); // Type 'number' is not assignable to type 'string'.
```

上例中，参数 `this` 的类型是一个带有 `name` 属性的对象，不符合这个条件的 `this` 都会报错。

在类的内部，`this` 本身也可以当作类型使用，表示当前类的实例对象。

```
class Box {
  contents:string = '';
  set(value:string):this {
    this.contents = value;
    return this;
  }
}
```

上例中，`set()` 方法的 **返回值类型就是this，表示当前的实例对象。**

注意，this类型不允许应用于静态成员。

```
class A {
  static a:this; // A 'this' type is available only in a non-static member of a
class or interface. “this”类型仅在类或接口的非静态成员中可用。
}
```

上例中，静态属性 `a` 的返回值类型是 `this`，就报错了。原因是this类型表示实例对象，但是静态成员拿不到实例对象。

有些方法返回一个布尔值，表示当前的 `this` 是否属于某种类型。这时，这些方法的返回值类型可以写成 `this is Type` 的形式，其中用到了 `is` 运算符。

```
class FileSystemObject {
  isFile(): this is FileRep {
    return this instanceof FileRep;
  }
  isDirectory(): this is Directory {
    return this instanceof Directory;
  }
  // ...
}
```

上例中，两个方法的返回值类型都是布尔值，写成 `this is Type` 的形式，可以精确表示返回值。