

TypeScript 2.1 现已发布。 现在下载 (#download-links)我们最新版本！

文档目录

# 类

## 介绍

传统的JavaScript程序使用函数和基于原型的继承来创建可重用的组件，但对于熟悉使用面向对象方式的程序员来讲就有些棘手。 因为他们的习惯是基类的继承并且对象理由类构建出来的。从ECMAScript 2015，也就是ECMAScript 6开始，JavaScript程序员将能够使用基于类的面向对象的方式。 使用TypeScript，我们允许开发者现在就使用这些特性，并且编译后的JavaScript可以在所有主流浏览器和平台上运行，而不需要等到下个JavaScript版本。

## 类

下面看一个使用类的例子：

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

如果你使用过C#或Java，你会对这种语法非常熟悉。 我们声明一个 Greeter 类。这个类有3个成员：一个叫做 greeting 的属性，一个构造函数和一个 greet 方法。

你会注意到，我们在引用任何一个类成员的时候都用了 this。 它表示我们访问的是类的成员。

最后一行，我们使用 new 构造了 Greeter 类的一个实例。 它会调用之前定义的构造函数，创建一个 Greeter 类型的对象，并执行构造函数初始化它。

## 继承

在TypeScript里，我们可以使用常用的面向对象模式。 当然，基于类的程序设计中最基本的模式是允许使用继承来扩展现有的类。

看下面的例子：

```
class Animal {
    name:string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`)
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

这个例子展示了TypeScript中继承的一些特征，它们与其它语言类似。 我们使用 extends 关键字来创建子类。你可以看到 Horse 和 Snake 类是基类 Animal 的子类，并且可以访问其属性和方法。

包含构造函数的派生类必须调用 super()，它会执行基类的构造方法。

这个例子演示了如何在子类里可以重写父类的方法。 Snake 类和 Horse 类都创建了 move 方法，它们重写了从 Animal 继承来的 move 方法，使得 move 方法根据不同的类而具有不同的功能。 注意，即使 tom 被声明为 Animal 类型，但因为它的值是 Horse，tom.move(34) 会调用 Horse 里的重写方法：

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

## 公共，私有与受保护的修饰符

默认为 public

在上面的例子里，我们可以自由的访问程序里定义的成员。 如果你对其它语言中的类比较了解，就会注意到我们在之前的代码里并没有使用 public 来做修饰；例如，C#要求必须明确地使用 public 指定成员是可见的。 在TypeScript里，成员都被认为 public。

你也可以明确的将一个成员标记成 public。 我们可以用下面的方式来重写上面的 Animal 类：

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }

    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`)
    }
}
```

理解 private

当成员被标记成 private 时，它就不能在声明它的类的外部访问。比如：

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }

    new Animal("Cat").name; // Error: 'name' is private;
```

TypeScript使用的是结构性类型系统。 当我们比较两种不同的类型时，并不在乎它们从何处而来，如所有成员的类型都是兼容的，我们就认为它们的类型是兼容的。

然而，当我们比较带有 private 或 protected 成员的类型的时候，情况就不同了。 如果其中一个类型里包含一个 private 成员，那么就只有当另外一个类型中也存在这样一个 private 成员，并且它们都是来自同一处声明时，我们才认为这两个类型是兼容的。 对于 protected 成员也使用这个规则。

下面来看一个例子，更好地说明了这一点：

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

class Rhino extends Animal {
    constructor() { super("Rhino"); }
}

class Employee {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: Animal and Employee are not compatible
```

这个例子中有 Animal 和 Rhino 两个类，Rhino 是 Animal 类的子类。还有一个 Employee 类，其类型看上去和 Animal 是相同的。 我们创建了几个这些类的实例，并相互赋值来看看会发生什么。 因为 Animal 和 Rhino 共享了来自 Animal 里的私有成员定义 private name: string，因此它们是兼容的。 然而，Employee 却不是这样。当把 Employee 赋值给 Animal 的时候，得到一个错误，说它们的类型不兼容。 尽管 Employee 里也有一个私有成员 name，但它明显不是一个 Animal 里面定义的那个。

理解 protected

protected 修饰符与 private 修饰符的行为很相似，但有一点不同，protected 成员在派生类中仍然可以访问。例如：

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

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

注意，我们不能在 Person 类外使用 name，但是我们仍然可以通过 Employee 类的实例方法访问，因为 Employee 是由 Person 派生而来的。

构造函数也可以被标记成 protected。 这意味着这个类不能在包含它的类外被实例化，但是能被继承。比如，

```
class Person {
    protected name: string;
    protected constructor(theName: string) { this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

## readonly 修饰符

你可以使用 readonly 关键字将属性设置为只读的。 只读属性必须在声明时或构造函数里被初始化。

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}

let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error: name is readonly.
```

## 参数属性

在上面的例子中，我们不得不定义一个受保护的成员 name 和一个构造函数参数 theName 在 Person 类里，并且立刻给 name 和 theName 赋值。 这种情况经常会遇到。 参数属性可以方便地让我们在一个地方定义并初始化一个成员。 下面的例子是对之前 Animal 类的修改版，使用了参数属性：

```
class Animal {
    constructor(private name: string) { }
    move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`)
    }
}
```

注意看我们是如何舍弃了 theName，仅在构造函数里使用 private name: string 参数来创建和初始化 name 成员。 我们把声明和赋值合并至一处。

参数属性通过给构造函数参数添加一个访问限定符来声明。 使用 private 限定一个参数属性会声明并初始化一个私有成员；对于 public 和 protected 来说也是一样。

## 存取器

TypeScript支持通过getters/setters来读取对对象成员的访问。 它能帮助你有效的控制对对象成员的访问。

下面来看如何把一个简单的类改写成使用 get 和 set。 首先，我们从一个没有使用存取器的例子开始。

```
class Employee {
    fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

我们可以随意的设置 fullName，这是非常方便的，但是这也可能会带来麻烦。

下面这个版本里，我们先检查用户名密码是否正确，然后再允许其修改员工信息。 我们对 fullName 的直接访问改成了可以检查密码的 set 方法。 我们也加了一个 get 方法，让上面的例子仍然可以工作。

```
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode !== "secret passcode") {
            this._fullName = newName;
        } else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    alert(employee.fullName);
}
```

我们可以修改一下密码，来验证一下存取器是否是工作的。当密码不对时，会提示我们没有权限去修改员工。

对于存取器有下面几点需要注意的：

首先，存取器要求你将编译器设置为输出ECMAScript 5或更高。 不支持降级到ECMAScript 3。其次，只带有 get 不带 set 的存取器会自动被推断为 readonly。这在代码生成 .d.ts 文件时是有帮助的，因为利用这个属性的用户会看到不允许修改它的值。

## 静态属性

到目前为止，我们只讨论了类的实例成员，那些仅当类被实例化的时候才会被初始化的属性。 我们使用 static 定义 origin，因为这些属性存在于类本身上面而不是类的实例上。 在这个例子里，我们使用 static 定义 origin，因为它所有网络都会用到的属性。 每个实例想要访问这个属性的时候，都要在 origin 前面加上类名。 如同在实例属性上使用 this，前缀来访问属性一样，这里我们使用 Grid. 来访问静态属性。

```
class Grid {
    static origin = { x: 0, y: 0 };
    calculateDistanceFromOrigin(point: { x: number; y: number; }) {
        let xDist = (point.x - Grid.origin.x);
        let yDist = (point.y - Grid.origin.y);
        return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
    }
    constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```

## 抽象类

抽象类做为其它派生类的基类使用。 它们一般不能直接被实例化。 不同于接口，抽象类可以包含成员的实现细节。 abstract 关键字是用于定义抽象类和在抽象类内部定义抽象方法。

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log('Roaming the earth...');
    }
}
```

抽象类中的抽象方法不包含具体实现并且必须在派生类中实现。 抽象方法的语法与接口方法相似。 两者都是定义方法签名但不包含方法体。 然而，抽象方法必须包含 abstract 关键字并且可以包含访问修饰符。

```
abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log('Department name: ' + this.name);
    }

    abstract printMeeting(): void; // 必须在派生类中实现
}

class AccountingDepartment extends Department {

    constructor() {
        super('Accounting and Auditing'); // constructors in derived classes must call super()
    }

    printMeeting(): void {
        console.log('The Accounting Department meets each Monday at 10am.');
```

高级技巧

## 构造函数

当你在TypeScript里声明了一个类的时候，实际上同时声明了很多东西。 首先就是类的 实例的类型。

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

这里，我们写了 let greeter: Greeter，意思是 Greeter 类的实例的类型是 Greeter。 这对于用过其它面向对象语言的程序员来讲已经是老习惯了。

我们也创建了一个叫做构造函数的值。 这个函数会在我们使用 new 创建类实例的时候被调用。下面我们来看看，上面的代码被编译成JavaScript后是什么样子的：

```
let Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

上面的代码里，let Greeter 将被赋值为构造函数。 当我们调用 new 并执行了这个函数后，便会得到一个类的实例。 这个构造函数也包含了类的所有静态属性。 换个角度说，我们可以认为类具有 实例部分与静态部分这两个部分。

让我们稍微改写一下这个例子，看看它们之间的区别：

```
class Greeter {
    static standardGreeting = "Hello, there!";
    greeting: string;
    greet() {
        if (this.greeting) {
            return "Hello, " + this.greeting;
        } else {
            return Greeter.standardGreeting;
        }
    }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet());

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();
console.log(greeter2.greet());
```

这个例子里，greeter1与之前看到的一样。 我们实例化 Greeter 类，并使用这个对象。 与我们之前看到的一样。

再之后，我们直接使用类。 我们创建了一个叫做 greeterMaker 的变量。 这个变量保存了这个类或者说是类构造函数。 然后我们使用 typeof Greeter，意思是取Greeter类的类型，而不是实例的类型。 或者更确切的说，“告诉我 Greeter 标识符的类型”，也就是构造函数的类型。 这个类型包含了类的所有静态成员和构造函数。 之后，就和前面一样，我们在 greeterMaker 上使用 new，创建 Greeter 的实例。

## 把类当做接口使用

如上一节里所讲的，类定义会创建两个东西：类的实例类型和一个构造函数。 因为类可以创建出类型，所以它能够在允许使用接口的地方使用类。

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

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = { x: 1, y: 2, z: 3};
```

制作 于北京 微博 @TypeScript中文网 (http://weibo.com/tslangon) 

©2012-2016 Microsoft  Microsoft

TypeScript中文网 | 帮助TypeScript语言在中国的发展

TypeScript中文网基金会保持和TypeScript官网 (http://www.typescriptlang.org/) 一致

© 京ICP备15040086号-2