TypeScript 接口 TypeScript 对象

TypeScript 类

TypeScript 是面向对象的 JavaScript。

类描述了所创建的对象共同的属性和方法。

TypeScript 支持面向对象的所有特性，比如 类、接口等。

TypeScript 类定义方式如下：

```
class class_name {
    // 类作用域
}
```

定义类的关键字为 class，后面紧跟类名，类可以包含以下几个模块（类的数据成员）：

- **字段** – 字段是类里面声明的变量。字段表示对象的有关数据。
- **构造函数** – 类实例化时调用，可以为类的对象分配内存。
- **方法** – 方法为对象要执行的操作。

实例

创建一个 Person 类：

TypeScript

```
class Person {
}
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var Person = /** @class */ (function () {
function Person() {
}
return Person;
}());
```

创建类的数据成员

以下实例我们声明了类 Car，包含字段为 engine，构造函数在类实例化后初始化字段 engine。

this 关键字表示当前类实例化的对象。注意构造函数的参数名与字段名相同，this.engine 表示类的字段。

此外我们也在类中定义了一个方法 disp()。

TypeScript

```
class Car {
// 字段
engine:string;
// 构造函数
constructor(engine:string) {
this.engine = engine
}
// 方法
disp():void {
console.log("发动机为 : "+this.engine)
}
}
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var Car = /** @class */ (function () {
// 构造函数
function Car(engine) {
this.engine = engine;
}
// 方法
Car.prototype.disp = function () {
console.log("发动机为 : " + this.engine);
};
return Car;
}());
```

创建实例化对象

我们使用 new 关键字来实例化类的对象，语法格式如下：

```
var object_name = new class_name([ arguments ])
```

类实例化时会调用构造函数，例如：

```
var obj = new Car("Engine 1")
```

类中的字段属性和方法可以使用 `.` 号来访问：

```
// 访问属性
obj.field_name

// 访问方法
obj.function_name()
```

完整实例

类的继承：实例中创建了 Shape 类，Circle 类继承了 Shape 类，Circle 类可以直接使用 Area 属性：

TypeScript

```
class Car {
// 字段
engine:string;
// 构造函数
constructor(engine:string) {
this.engine = engine
}
// 方法
disp():void {
console.log("函数中显示发动机型号 : "+this.engine)
}
}
// 创建一个对象
var obj = new Car("XXSY1")
// 访问字段
console.log("读取发动机型号 : "+obj.engine)
// 访问方法
obj.disp()
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var Car = /** @class */ (function () {
// 构造函数
function Car(engine) {
this.engine = engine;
}
// 方法
Car.prototype.disp = function () {
console.log("函数中显示发动机型号 : " + this.engine);
};
return Car;
}());
// 创建一个对象
var obj = new Car("XXSY1");
// 访问字段
console.log("读取发动机型号 : " + obj.engine);
// 访问方法
obj.disp();
```

输出结果为：

```
读取发动机型号 :  XXSY1
函数中显示发动机型号  :  XXSY1
```

类的继承

TypeScript 支持继承类，即我们可以在创建类的时候继承一个已存在的类，这个已存在的类称为父类，继承它的类称为子类。

类继承使用关键字 `extends`，子类除了不能继承父类的私有成员(方法和属性)和构造函数，其他的都可以继承。

TypeScript 一次只能继承一个类，不支持继承多个类，但 TypeScript 支持多重继承（A 继承 B，B 继承 C）。

语法格式如下：

```
class child_class_name extends parent_class_name
```

实例

类的继承：实例中创建了 Shape 类，Circle 类继承了 Shape 类，Circle 类可以直接使用 Area 属性：

TypeScript

```
class Shape {
Area:number
constructor(a:number) {
this.Area = a
}
}
class Circle extends Shape {
disp():void {
console.log("圆的面积: "+this.Area)
}
}
var obj = new Circle(223);
obj.disp()
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var __extends = (this && this.__extends) || (function () {
var extendStatics = function (d, b) {
extendStatics = Object.setPrototypeOf ||
({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
return extendStatics(d, b);
};
return function (d, b) {
extendStatics(d, b);
function __() { this.constructor = d; }
d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
})();
var Shape = /** @class */ (function () {
function Shape(a) {
this.Area = a;
}
return Shape;
}());
var Circle = /** @class */ (function (_super) {
__extends(Circle, _super);
```

```
function Circle() {
return _super !== null && _super.apply(this, arguments) || this;
}
Circle.prototype.disp = function () {
console.log("圆的面积: " + this.Area);
};
return Circle;
}(Shape));
var obj = new Circle(223);
obj.disp();
```

输出结果为：

```
圆的面积:  223
```

需要注意的是子类只能继承一个分类，TypeScript 不支持继承多个类，但支持多重继承，如下实例：

TypeScript

```
class Root {
str:string;
}
class Child extends Root {}
class Leaf extends Child {} // 多重继承，继承了 Child 和 Root 类
var obj = new Leaf();
obj.str ="hello"
console.log(obj.str)
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var __extends = (this && this.__extends) || (function () {
var extendStatics = function (d, b) {
extendStatics = Object.setPrototypeOf ||
({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
return extendStatics(d, b);
};
return function (d, b) {
extendStatics(d, b);
function __() { this.constructor = d; }
d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
})();
var Root = /** @class */ (function () {
function Root() {
}
return Root;
}());
var Child = /** @class */ (function (_super) {
__extends(Child, _super);
function Child() {
```

```
return _super !== null && _super.apply(this, arguments) || this;
}
return Child;
}(Root));
var Leaf = /** @class */ (function (_super) {
__extends(Leaf, _super);
function Leaf() {
return _super !== null && _super.apply(this, arguments) || this;
}
return Leaf;
}(Child)); // 多重继承，继承了 Child 和 Root 类
var obj = new Leaf();
obj.str = "hello";
console.log(obj.str);
```

输出结果为：

```
hello
```

继承类的方法重写

类继承后，子类可以对父类的方法重新定义，这个过程称之为方法的重写。

其中 super 关键字是对父类的直接引用，该关键字可以引用父类的属性和方法。

TypeScript

```
class PrinterClass {
doPrint():void {
console.log("父类的 doPrint() 方法。")
}
}
class StringPrinter extends PrinterClass {
doPrint():void {
super.doPrint() // 调用父类的函数
console.log("子类的 doPrint()方法。")
}
}
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var obj = new StringPrinter()
obj.doPrint()
var __extends = (this && this.__extends) || (function () {
var extendStatics = function (d, b) {
extendStatics = Object.setPrototypeOf ||
({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
return extendStatics(d, b);
};
return function (d, b) {
```

```
extendStatics(d, b);
function __() { this.constructor = d; }
d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
};
})();
var PrinterClass = /** @class */ (function () {
function PrinterClass() {
}
PrinterClass.prototype.doPrint = function () {
console.log("父类的 doPrint() 方法。");
};
return PrinterClass;
}());
var StringPrinter = /** @class */ (function (_super) {
__extends(StringPrinter, _super);
function StringPrinter() {
return _super !== null && _super.apply(this, arguments) || this;
}
StringPrinter.prototype.doPrint = function () {
_super.prototype.doPrint.call(this); // 调用父类的函数
console.log("子类的 doPrint()方法。");
};
return StringPrinter;
}(PrinterClass));
var obj = new StringPrinter();
obj.doPrint();
```

输出结果为：

```
父类的 doPrint() 方法。
子类的 doPrint()方法。
```

static 关键字

static 关键字用于定义类的数据成员（属性和方法）为静态的，静态成员可以直接通过类名调用。

TypeScript

```
class StaticMem {
static num:number;
static disp():void {
console.log("num 值为 "+ StaticMem.num)
}
}
StaticMem.num = 12 // 初始化静态变量
StaticMem.disp() // 调用静态方法
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var StaticMem = /** @class */ (function () {
function StaticMem() {
```

```
}
StaticMem.disp = function () {
console.log("num 值为 " + StaticMem.num);
};
return StaticMem;
}());
StaticMem.num = 12; // 初始化静态变量
StaticMem.disp(); // 调用静态方法
```

输出结果为：

```
num 值为 12
```

instanceof 运算符

instanceof 运算符用于判断对象是否是指定的类型，如果是返回 true，否则返回 false。

TypeScript

```
class Person{ }
var obj = new Person()
var isPerson = obj instanceof Person;
console.log("obj 对象是 Person 类实例化来的吗？ " + isPerson);
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var Person = /** @class */ (function () {
function Person() {
}
return Person;
}());
var obj = new Person();
var isPerson = obj instanceof Person;
console.log(" obj 对象是 Person 类实例化来的吗？ " + isPerson);
```

输出结果为：

```
obj 对象是 Person 类实例化来的吗？ true
```

访问控制修饰符

TypeScript 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 3 种不同的访问权限。

- **public（默认）**：公有，可以在任何地方被访问。
- **protected**：受保护，可以被其自身以及其子类和父类访问。
- **private**：私有，只能被其定义所在的类访问。

以下实例定义了两个变量 str1 和 str2，str1 为 public，str2 为 private，实例化后可以访问 str1，如果要访问 str2 则会编译错误。

TypeScript

```
class Encapsulate {
str1:string = "hello"
private str2:string = "world"
}
var obj = new Encapsulate()
console.log(obj.str1) // 可访问
console.log(obj.str2) // 编译错误， str2 是私有的
```

类和接口

类可以实现接口，使用关键字 implements，并将 interest 字段作为类的属性使用。

以下实例红 AgriLoan 类实现了 ILoan 接口：

TypeScript

```
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)
console.log("利润为 : "+obj.interest+"，抽成为 : "+obj.rebate )
```

编译以上代码，得到以下 JavaScript 代码：

JavaScript

```
var customer = {
var AgriLoan = /** @class */ (function () {
function AgriLoan(interest, rebate) {
this.interest = interest;
this.rebate = rebate;
}
return AgriLoan;
}());
var obj = new AgriLoan(10, 1);
console.log("利润为 : " + obj.interest + "，抽成为 : " + obj.rebate);
```

输出结果为：

```
利润为 : 10，抽成为 : 1
```

TypeScript 接口 TypeScript 对象

点我分享笔记