ES5中的类

在ES5声明一个函数（可以new），再将方法添加到这个方法的原型上，来创建自定义类型。

```
function Person(name) {
    this.name = name;
}
Person.prototype.sayName = function() {
    console.log(this.name);
};
let person = new Person("xunuo0x");
person.sayName(); // 输出 "xunuo0x"
console.log(person instanceof Person); // true
console.log(person instanceof Object); // true
```

ES6中类的声明

本质：**ES5实现方式的语法糖** 声明：

- 构造器：构造器内创建自有属性
- 方法：声明类实例具有的方法

```
class Person {
    // 等价于 Person 构造器
    constructor(name) {
    this.name = name;
    }
    // 等价于 Person.prototype.sayName
    sayName() {
        console.log(this.name);
    }
}
let person = new Person("xunuo0x");
person.sayName(); // 输出 "xunuo0x"
console.log(person instanceof Person); // true
console.log(person instanceof Object); // true
console.log(typeof Person); // "function"
console.log(typeof Person.prototype.sayName); // "function"
```

class和自定义类型的区别

- class的声明不会提升，与let类似
- class的声明自动运行于严格模式之下
- class声明的方法不可枚举（显著区别）
- class的内部方法没有[[construct]]属性，无法new

- 调用class的构造函数必须new
- class内部方法不能同名

用ES5重写如下： **在实现的时候，主要使用Object.defineProperty()实现class内部函数**

```
// 直接等价于 Person
let PersonType2 = (function() {
    "use strict";
    // **类的内部不能修改类名**
    const PersonType2 = function(name) {
    // 确认函数被调用时使用了 new
    if (typeof new.target === "undefined") {
        throw new Error("Constructor must be called with new.");
    }
    this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
            value: function() {
            // 确认函数被调用时没有使用 new
            if (typeof new.target !== "undefined") {
                throw new Error("Method cannot be called with new.");
            }
            console.log(this.name);
        },
         // **类的内部方法定义为不可枚举**
        enumerable: false,
        writable: true,
        configurable: true
    });
    return PersonType2;
}());
```

类表达式

- 匿名类表达式let Person = class{...}
- 具名类表达式let Person = PersonClass class{...}
- 区别仅在于class的内部实现时，const PersonClass作为内部实现的类名

class作为一级公民

> js中能当作值来使用的称为**一级公民**

用法：

- 类名作为参数传入函数
- 立即执行，实现单例模式

```
// 类名作为参数传入函数
function crateObj (ClassName){
    return new ClassName()
}
// 立即执行，实现单例模式
let person = new class {
    constructor (name) {
        this.name = name
    }
    say () {
        console.log(this.name)
    }
}('xunuo0x')
person.say() // "xunuo0x"
```

class中访问器属性

- get 关键字
- set 关键字
- 内部实现时将getter/setter变量名，通过Object.defineProperty()定义

class中静态成员

- static关键字
- 相当于ES5中Person.create() = function() {}
- 访问时直接通过类访问，不能通过实例访问

使用extends继承

> 只要一个表达式能返回具有[[constructor]]就可以使用extends继承

ES5中的继承

```
function Parent (name) {
    this.name = name
}
Parent.prototype.sayName = function () {
    console.log(this.name)
}
function Child (name) {
    Parent.call(this, name)
}
Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Child
```

在ES6中的继承

```
class Parent (name) {
    constructor (name) {
        this.name = name
    }
    sayName() {
        console.log(this.name)
    }
}
class Child extends Parent (name) {
    constructor(name) {
        super(name)
    }
    // 重写父类中的方法
    sayName () {
        console.log(`Child ${this.name}`)
    }
}
```

继承内置对象

- ES5中继承内置对象（如继承Array可能会产生问题）
- ES5中继承，this先被派生类创建
- ES6中继承，this先被基类创建，就具有了基类的方法和属性

Symbol.species属性

- extends继承时，派生类上返回的是派生类的实例
- 如果想返回基类，可以设置Symbol.species

```
class MyClass extends Array {
    static get [Symbol.species]() {
        return this; // 默认返回MyClass类型
        return Array; // 修改返回基类
    }
    constructor(value) {
        this.value = value;
    }
}
```

new.target

- 见名知意，就是new操作执行的对象
- ES6中实例化class时，必须要new，所以在constructor()中new.target不可能是undefined

小结

- ES6中class简化了ES5中的继承，但是未改变现有的继承模型。可以理解为是ES5基于原型链的语法糖
- 通过class声明一个类，constructor()作为构造函数，属性在constructor()中初始化
- class内可以定义getter/setter访问器属性
- 可以在class内定义非静态方法，静态方法绑定在构造器上
- 类的所有方法都是不可枚举的，也符合内部方法
- 实例化一个class必须要new关键字
- extends实现继承，子类中调用super()访问父类构造函数
- 因为class的实现是基于ES5类模型那一套，本质上和ES5中是一样的，如果过多使用extends可能还会降低性能