ES6(15)class (基本语法)

ES6

ES6(15)class (基本语法)

总结

1、class是用来创建实例的 不能当做函数来调用(必须使用 new调用)

2、类里边可以定义构造函数，当你创建一个类的实例的时候 就会调用构造函数

3、定义“类”的方法的时候，前面不需要加上function这个关键字

简介

ES6的class可以看作只是一个语法糖

1、ES6的constructor方法，就是构造方法，而this关键字则代表实例对象

2、注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错

ES6 的类，完全可以看作构造函数的另一种写法

1、类的数据类型是函数

2、类本身就指向构造函数，Point === Point.prototype.constructor

3、使用方法和构造函数一致，对类使用new命令(只能使用new，不能直接加括号)

4、类的所有方法都定义在类的prototype属性上

5、在类的实例上面调用方法，其实就是调用原型上的方法，b.constructor === B.prototype.constructor=== B

6、类的新方法可以添加在类的prototype对象上面（Object.assign）

7、prototype对象的constructor属性指向“类”的本身，Point.prototype.constructor === Point

8、类的内部所有定义的方法，都是不可枚举的

constructor 方法

1、constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法

2、一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加

3、constructor方法默认返回实例对象（即this）,完全可以指定返回另外一个对象（结果会导致实例对象不是类的实例）

4、类必须使用new调用，否则会报错

类的实例

1、生成类的实例的写法是使用new命令

2、如果忘记加上new，像函数那样调用Class，将会报错

3、实例的私有、公有属性

(1)实例的私有属性都是显式定义在其本身（即定义在constructor中的this对象上）

(2)实例的公有属性都是定义在原型上（即定义在class.prototype上）

4、类的所有实例共享一个原型对象

（1）可以通过实例的__proto__属性为“类”添加方法，但不建议

（2）生产环境中，可以使用Object.getPrototypeOf方法来获取实例对象的原型，然后再来为原型添加方法/属性

取值函数（getter）和存值函数（setter）

1、在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为

2、存值函数和取值函数是设置在属性的 Descriptor 对象上的

属性表达式

Class 表达式

类也可以使用表达式的形式定义const MyClass = class Me {}

立即执行的 Class，const MyClass = new class {}(‘张三’)

注意点

（1）严格模式，类和模块的内部，默认就是严格模式

（2）不存在提升，类不存在变量提升，必须保证子类在父类之后定义，这一点与 ES5 完全不同

（3）name 属性，总是返回紧跟在class关键字后面的类名

（4）Generator 方法，如果某个方法之前加上星号（*），就表示该方法是一个Generator函数

（5）this 的指向

类的方法内部如果含有this，它默认指向类的实例，但是，必须非常小心，一旦单独使用该方法(调用时不是跟在实例后面)，很可能报错，this会指向该方法运行时所在的环境

1、在构造方法中（constructor）绑定（bind）this

2、在constructor中使用箭头函数定义方法（这个方法是私有的）

3、使用Proxy，获取方法的时候，自动绑定this

静态方法

1、如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”

2、在实例上调用静态方法，会抛出一个错误，表示不存在该方法

3、静态方法中的this指的是类，而不是实例

4、静态方法可以与非静态方法重名

5、父类的静态方法，可以被子类继承

6、子类可以从super对象上调用父类的静态方法

实例属性的新写法

1、实例属性可以定义在类的最顶层（实例属性一般定义在constructor()方法里面）

静态属性

1、静态属性指的是 Class本身的属性，即Class.propName，而不是定义在实例对象（this）上的属性

2、目前，只有这种写法可行，因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性

私有方法和私有属性

1、私有方法和私有属性，是只能在类的内部访问的方法和属性，外部不能访问
2、但 ES6 不提供，只能通过变通方法模拟实现。
（1）、命名上加以区别
（2）、将私有方法移出模块
（3）、利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值

new.target属性，new命令作用于的那个构造函数
确定构造函数是怎么调用的
如果构造函数不是通过new命令或Reflect.construct()调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的

子类继承父类时，new.target会返回子类
写出不能独立使用、必须继承后才能使用的类

在函数外部，使用new.target会报错

总结

> 以前js里类和函数是一体的 现在分开了

1、`class`是用来创建实例的 不能当做函数来调用(必须使用 new调用)

2、类里边可以定义构造函数，当你创建一个类的实例的时候 就会调用构造函数

3、定义“类”的方法的时候，前面不需要加上function这个关键字

```
class Parent {
    constructor(name){
        this.name=name;      //实例的私有属性
    }
    //实例的公有属性，也就是相当于原型上的属性
    getName(){
        console.log(this.name)
    }
}
```

```
let children= new Parent('lyp')

children.getName();  //'lyp'
```

ES5实现

```
"use strict";

function _instanceof(left, right) {
    if (right != null && typeof Symbol !== "undefined" && right[Symbol.ha
sInstance]) {
        return right[Symbol.hasInstance](left);
    } else {
        return left instanceof right;
    }
}

function _classCallCheck(instance, Constructor) {
    if (!_instanceof(instance, Constructor)) {
        throw new TypeError("Cannot call a class as a function");
    }
}

function _defineProperties(target, props) {
    for (var i = 0; i < props.length; i++) {
        var descriptor = props[i];
        descriptor.enumerable = descriptor.enumerable || false; descripto
r.configurable = true;
        if ("value" in descriptor) descriptor.writable = true;
        Object.defineProperty(target, descriptor.key, descriptor);
    }
}

function _createClass(Constructor, protoProps, staticProps) {
    if (protoProps) _defineProperties(Constructor.prototype, protoProps);
    if (staticProps) _defineProperties(Constructor, staticProps);
    return Constructor;
}

var Parent =
    /*#__PURE__*/
    function () {
        function Parent(name) {
            _classCallCheck(this, Parent);

            this.name = name; //实例的私有属性
        }
        //实例的公有属性，也就是相当于原型上的属性
        _createClass(Parent, [{
            key: "getName",
            value: function getName() {
```

```
                console.log(this.name);
            }
        }]);

        return Parent;
    }();

var children = new Parent('lyp');
children.getName(); //'lyp'
```

简介

ES6的class可以看作只是一个语法糖

它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

```
//es5
function Point(x, y) {
    //实例的私有属性
  this.x = x;
  this.y = y;
}
//实例的公有属性，也就是相当于原型上的属性
Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};

var p = new Point(1, 2);


//es6
class Point {
  constructor(x, y) {
    //实例的私有属性
    this.x = x;
    this.y = y;
  }
  //实例的公有属性，也就是相当于原型上的属性
  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

上面代码：

1、`ES6的constructor`方法，`就是构造方法`，而`this`关键字则`代表实例对象`

> 也就是说，ES5 的构造函数`Point`，`对应`ES6 的`Point类的构造方法`。
>
> `Point`类除了构造方法，还定义了一个`toString方法`。

2、注意，`定义“类”的方法`的时候，前面`不需要加上function`这个关键字，直接把函数定义放进去了就可以了。另外，`方法之间不需要逗号`分隔，`加了会报错`

ES6 的类，完全可以看作`构造函数的另一种写法`

1、类的数据类型是函数

2、`类本身就指向构造函数，Point === Point.prototype.constructor`

```
class Point {
  // ...
}

typeof Point // "function"
Point === Point.prototype.constructor // true
```

3、使用方法和构造函数一致，对类使用`new`命令(只能使用new，不能直接加括号)

```
class Bar {
  doStuff() {
    console.log('stuff');
  }
}

var b = new Bar();
b.doStuff() // "stuff"
```

4、类的所有方法都定义在类的prototype属性上

> 构造函数的`prototype`属性，在`ES6`的“类”上面`继续存在`。事实上，`类的所有方法都定义在类的prototype属性上面`。

```
class Point {
  constructor() {
    // ...
  }

  toString() {
    // ...
  }

  toValue() {
    // ...
  }
}

// 等同于

Point.prototype = {
  constructor() {},
  toString() {},
  toValue() {},
};
```

5、在类的实例上面调用方法，其实就是调用原型上的方法，`b.constructor === B.prototype.constructor=== B`

```
class B {}
let b = new B();

b.constructor === B.prototype.constructor // true
```

6、类的新方法可以添加在类的prototype对象上面（Object.assign）

> 由于类的方法都定义在`prototype`对象上面，所以类的新方法可以添加在`prototype`对象上面。
>
> `Object.assign`方法可以很方便地一次向类添加多个方法。

```
class Point {
  constructor(){
    // ...
  }
```

```
}

Object.assign(Point.prototype, {
  toString(){},
  toValue(){}
});
```

7、prototype对象的constructor属性指向“类”的本身，Point.prototype.constructor === Point

这与 `ES5` 的行为是 `一致` 的

```
class Point{

}
Point.prototype.constructor === Point // true
```

8、类的内部所有定义的方法，都是不可枚举的

这一点与 `ES5` 的行为 `不一致`

```
//ES6   不可枚举
class Point {
  constructor(x, y) {
    // ...
  }

  toString() {
    // ...
  }
}

Object.keys(Point.prototype)
// []
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```

```
//ES5   可枚举
var Point = function (x, y) {
  // ...
};

Point.prototype.toString = function() {
  // ...
};
```

```
Object.keys(Point.prototype)
// ["toString"]
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```

constructor 方法

1、`constructor`方法是类的`默认方法`，通过`new命令`生成对象实例时，自动`调用该方法`

2、一个类`必须有constructor`方法，如果`没有显式定义`，一个`空的constructor`方法会被`默认添加`

```
class Point {
}

// 等同于
class Point {
  constructor() {}
}
```

3、constructor方法默认返回实例对象（即this）,完全可以指定返回另外一个对象（结果会导致实例对象不是类的实例）

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}

new Foo() instanceof Foo
// false
```

上面代码中，`constructor`函数`返回`一个全`新的对象`，结果导致实例对象不是`Foo`类的实例。

4、类必须使用new调用，否则会报错

```
class Foo {
  constructor() {
    return Object.create(null);
  }
```

```
}

Foo()
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

类的实例

1、生成类的实例的写法是使用`new`命令

> 生成类的实例的写法，与 `ES5` 完全一样，也是使用 `new` 命令。

2、如果忘记加上`new`，像函数那样调用`Class`，将会报错

```
class Point {
  // ...
}

// 报错
var point = Point(2, 3);

// 正确
var point = new Point(2, 3);
```

3、实例的私有、公有属性

与`ES5`的行为保持`一致`

(1)实例的`私有属性`都是显式定义在其本身（即定义在constructor中的`this`对象上）

(2)实例的`公有属性`都是定义在原型上（即定义在`class.prototype`上）

```
//定义类
class Point {

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

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }

}
```

```
var point = new Point(2, 3);

point.toString() // (2, 3)

point.hasOwnProperty('x') // true
point.hasOwnProperty('y') // true
point.hasOwnProperty('toString') // false
point.__proto__.hasOwnProperty('toString') // true
```

`x`和`y`都是实例对象`point`自身的属性（因为定义在`this`变量上），所以`hasOwnProperty`方法返回`true`
而`toString`是原型对象的属性（因为定义在`Point`类上），所以`hasOwnProperty`方法返回`false`

4、类的所有实例共享一个原型对象

与ES5 一样

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);

p1.__proto__ === p2.__proto__
//true    原型都是Point.prototype，所以__proto__属性是相等的
```

（1）可以通过实例的`__proto__`属性为“类”添加方法，但不建议

但`不建议`在生产中使用`__proto__`属性，避免对环境产生依赖

（2）生产环境中，可以使用`Object.getPrototypeOf`方法来`获取实例对象的原型`，然后再来为原型添加方法/属性

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);

p1.__proto__.printName = function () { return 'Oops' };

p1.printName() // "Oops"
p2.printName() // "Oops"

var p3 = new Point(4,2);
p3.printName() // "Oops"
```

取值函数（getter）和存值函数（setter）

1、在“类”的内部可以使用`get`和`set`关键字，`对某个属性`设置存值函数和取值函数，`拦截`该属性的`存取行为`

```
class MyClass {
  constructor() {
    // ...
  }
  get prop() {
    return 'getter';
  }
  set prop(value) {
    console.log('setter: '+value);
  }
}

let inst = new MyClass();

inst.prop = 123;
// setter: 123

inst.prop
// 'getter'
```

2、存值函数和取值函数是设置在属性的 Descriptor 对象上的

```
class CustomHTMLElement {
  constructor(element) {
    this.element = element;
  }

  get html() {
    return this.element.innerHTML;
  }

  set html(value) {
    this.element.innerHTML = value;
  }
}

var descriptor = Object.getOwnPropertyDescriptor(
  CustomHTMLElement.prototype, "html"
);

"get" in descriptor  // true
"set" in descriptor  // true
```

属性表达式

类的属性名，可以采用表达式

```
//Square类的方法名getArea，是从表达式得到的s
let methodName = 'getArea';

class Square {
  constructor(length) {
    // ...
  }

  [methodName]() {
    // ...
  }
}
```

Class 表达式

类也可以使用表达式的形式定义const MyClass = class Me {}

```
const MyClass = class Me {
  getClassName() {
    return Me.name;
  }
};
//类的名字是Me，但是Me只在 Class 的内部可用，指代当前类。在 Class 外部，这个类只能用
MyClass引用

let inst = new MyClass();
inst.getClassName() // Me
Me.name // ReferenceError: Me is not defined

//如果类的内部没用到的话，可以省略Me
const MyClass = class { /* ... */ };
```

立即执行的 Class，const MyClass = new class {}('张三')

```
let person = new class {
  constructor(name) {
    this.name = name;
  }

  sayName() {
```

```
    console.log(this.name);
  }
}('张三');

person.sayName(); // "张三"
```

注意点

（1）严格模式，类和模块的内部，默认就是严格模式

考虑到未来所有的代码，其实都是运行在模块之中，所以 ES6 实际上把整个语言升级到了严格模式。

（2）不存在提升，类不存在变量提升，必须保证子类在父类之后定义，这一点与 ES5 完全不同

```
new Foo(); // ReferenceError   报错
class Foo {}
```

（3）name 属性，总是返回紧跟在class关键字后面的类名

ES6 的类只是 ES5 的构造函数的一层包装，所以函数的许多特性都被 Class 继承，包括 name 属性。

> name 属性总是返回紧跟在 class 关键字后面的类名

```
class Point {}
Point.name // "Point"
```

（4）Generator 方法，如果某个方法之前加上星号（*），就表示该方法是一个 Generator 函数

```
class Foo {
  constructor(...args) {
    this.args = args;
  }
  * [Symbol.iterator]() {
    for (let arg of this.args) {
      yield arg;
    }
  }
```

```
}

for (let x of new Foo('hello', 'world')) {
  console.log(x);
}
// hello
// world
```

上面代码中，`Foo`类的`Symbol.iterator`方法前有一个星号，表示该方法是一个`Generator`函数。
`Symbol.iterator`方法返回一个`Foo`类的默认遍历器，`for...of`循环会`自动调用`这个`遍历器`。

（5）this 的指向

`类的方法`内部如果含有`this`，它`默认指向类的实例`，但是，必须非常小心，一旦`单独使用该方法(调用时不是跟在实例后面)`，很可能报错，`this`会`指向该方法运行时所在的环境`

```
class Logger {
  printName(name = 'there') {
    this.print(`Hello ${name}`);
  }

  print(text) {
    console.log(text);
  }
}

const logger = new Logger();
const { printName } = logger;
printName(); // TypeError: Cannot read property 'print' of undefined
```

上面代码中，`printName`方法中的`this`，默认指向`Logger类的实例`。

但是，如果将这个方法提`取出来单独使用`，`this`会`指向该方法运行时所在的环境`（由于`class`内部是`严格模式`，所以`this`实际`指向的是undefined`），从而导致找不到`print`方法而报错。

> 解决办法

1、在构造方法中（constructor）绑定（bind）this

```
class Logger {
  constructor() {
    this.printName = this.printName.bind(this);
  }

  // ...
```

```
}
```

2、在constructor中使用箭头函数定义方法（这个方法是私有的）

```
class Obj {
  constructor() {
    this.getThis = () => this;
  }
}

const myObj = new Obj();
myObj.getThis() === myObj // true
```

`箭头函数`内部的`this`总是指向`定义时所在的对象`。

上面代码中，`箭头函数`位于`构造函数内`部，它的定义生效的时候，是在构造函数执行的时候。这时，箭头函数所在的运行环境，肯定是实例对象，所以`this`会总是指向实例对象。

3、使用Proxy，获取方法的时候，自动绑定this

```
class Logger {
  printName(name = 'there') {
    this.print(`Hello ${name}`);
  }

  print(text) {
    console.log(text);
  }
}

function selfish (target) {
  const cache = new WeakMap();
  const handler = {
    get (target, key) {
      const value = Reflect.get(target, key);
      if (typeof value !== 'function') {
        return value;
      }
      if (!cache.has(value)) {
        cache.set(value, value.bind(target));
      }
      return cache.get(value);
    }
  };
  const proxy = new Proxy(target, handler);
  return proxy;
}

const logger = selfish(new Logger());
```

静态方法

> 类 相当于 实例的原型，所有在 类中定义的方法，都会 被实例继承。

1、如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”

2、在实例上调用静态方法，会抛出一个错误，表示不存在该方法

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
```

3、静态方法中的this指的是类，而不是实例

4、静态方法可以与非静态方法重名

```
class Foo {
  static bar() {
    this.baz();
  }
  static baz() {
    console.log('hello');
  }
  baz() {
```

```
    console.log('world');
  }
}

Foo.bar() // hello
```

上面代码中，静态方法`bar`调用了`this.baz`，这里的`this`指的是`Foo`类，而不是`Foo`的实例，等同于调用`Foo.baz`。另外，从这个例子还可以看出，`静态方法可以与非静态方法重名`。

5、父类的静态方法，可以被子类继承

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
}

Bar.classMethod() // 'hello'
```

6、子类可以从super对象上调用父类的静态方法

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
  static classMethod() {
    return super.classMethod() + ', too';
  }
}

Bar.classMethod() // "hello, too"
```

实例属性的新写法

1、实例属性可以定义在类的最顶层（实例属性一般定义在constructor()方法里面）

```
class IncreasingCounter {
  constructor() {
    this._count = 0;
  }
  get value() {
    console.log('Getting the current value!');
    return this._count;
  }
  increment() {
    this._count++;
  }
}
```

> 属性也可以定义在类的最顶层，其他都不变

```
class IncreasingCounter {
  _count = 0;
  get value() {
    console.log('Getting the current value!');
    return this._count;
  }
  increment() {
    this._count++;
  }
}
```

静态属性

1、静态属性指的是 `Class` 本身的属性，即 `Class.propName`，而不是定义在实例对象（`this`）上的属性

```
class Foo {
}
//为Foo类定义了一个静态属性prop
Foo.prop = 1;
```

```
Foo.prop // 1
```

2、目前，只有这种写法可行，因为 ES6 明确规定，Class 内部只有静态方法，没有静态属性

私有方法和私有属性

1、私有方法和私有属性，是只能在类的内部访问的方法和属性，外部不能访问

2、但 ES6 不提供，只能通过变通方法模拟实现。

（1）、命名上加以区别

```
class Widget {

  // 公有方法
  foo (baz) {
    this._bar(baz);
  }

  // 私有方法
  _bar(baz) {
    return this.snaf = baz;
  }

  // ...
}
```

_bar 方法前面的下划线，表示这是一个只限于内部使用的私有方法。但是，这种命名是不保险的，在类的外部，还是可以调用到这个方法。

（2）、将私有方法移出模块

```
class Widget {
  foo (baz) {
    bar.call(this, baz);
  }
```

```
  // ...
}

function bar(baz) {
  return this.snaf = baz;
}
```

`foo`是公开方法，内部调用了`bar.call(this, baz)`。这使得`bar`实际上成为了当前模块的私有方法。

（3）、利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值

```
const bar = Symbol('bar');
const snaf = Symbol('snaf');

export default class myClass{

  // 公有方法
  foo(baz) {
    this[bar](baz);
  }

  // 私有方法
  [bar](baz) {
    return this[snaf] = baz;
  }

  // ...
};
```

`bar`和`snaf`都是`Symbol`值，一般情况下无法获取到它们，因此达到了私有方法和私有属性的效果。但是也不是绝对不行，`Reflect.ownKeys()`依然可以拿到它们。

new.target属性，new命令作用于的那个构造函数

> 一般用在`构造函数之中`，返回`new`命令作用于的那个构造函数

确定构造函数是怎么调用的

如果构造函数不是通过`new`命令或`Reflect.construct()`调用的，`new.target`会返回`undefined`，因此这个属性可以用来`确定构造函数是怎么调用的`

```
function Person(name) {
  if (new.target !== undefined) {
    this.name = name;
  } else {
    throw new Error('必须使用 new 命令生成实例');
  }
}

// 另一种写法
function Person(name) {
  if (new.target === Person) {
    this.name = name;
  } else {
    throw new Error('必须使用 new 命令生成实例');
  }
}

var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三');  // 报错
```

子类继承父类时，new.target会返回子类

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    // ...
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, width);
  }
}
//new.target会返回子类
var obj = new Square(3); // 输出 false
```

写出不能独立使用、必须继承后才能使用的类

```
class Shape {
```

```
    constructor() {
      if (new.target === Shape) {
        throw new Error('本类不能实例化');
      }
    }
  }

  class Rectangle extends Shape {
    constructor(length, width) {
      super();
      // ...
    }
  }

  var x = new Shape();  // 报错
  var y = new Rectangle(3, 4);  // 正确
```

`Shape`类不能被实例化，只能用于继承

在函数外部，使用new.target会报错