ES6 之 class

传统的 javascript 中只有对象，没有类的概念。它是基于原型的面向对象语言。原型对象特点就是将自身的属性共享给新对象。这样的写法相对于其它传统面向对象语言来讲，很有一种独树一帜的感脚！非常容易让人困惑！

首先大家要明白，为什么要出现 class ？

首先回顾我们的 JavaScript 编程思想的发展史。

从 JS 诞生之时，刚开始做的就是面向过程的编程，把一个问题给解释清楚了，几行 js 就可以搞定。随着 js 的发展以及浏览器对于 js 执行速度越来越高。我们对于 js 实现的功能越来越多，伴随代码量也会越来越多，我们仍然使用面向过程式的编程方案，就会有问题。

我们制作打怪兽游戏：

攻击 function attack()

逃跑 function escape()

加血 function resume()

会关注这三个方法的具体实现，并且反复调用这些方法完成游戏。这是面向过程的思路。

当我们增加打怪兽的使用者时，单纯这几个方法不足以完成多人游戏。

我们需要更高级的思想面向对象,尽管 javascript 不具备面向对象的特征（继承，封装，多态）。但是我们可以采用类似的这种思想的方式去改写这样的需求

```
function BraveMan() {

}

BraveMan.prototype.attack = function () {}
```

Braveman.prototype.escape = function () {}

BraveMan.prototype.resume = function() {}

基于我们原来过程的基础之上，我们封装这样的构造函数，用于产生可以多次执行这样过程的对象。这样的话我们的打怪兽，不单纯如何打怪兽（面向过程），而是变成了 谁能打怪兽（类似面向对象的思想），这里的勇者就是我们想要的对象，可以多次实例化。这也是这种思想给我们带来的好处，模块化，可扩展等好处。

在我们的日常 codeing 中，很多大的项目当中都需要使用这种类似面向对象的思想去进行编程，在 Javascript 中不存在面向对象，我们采用的类似面向对象的过程叫，基于原型编程，下面是工作中的存在的代码（音乐播放器中的两个模块）：

```
// ControlIndex 模块
function ControlIndex(len) {
    // this 指向 Control 对象
    // Control  上挂载两个属性一个为 index  初始值为 0
    // len 为当前数据长度
    this.index = 0;
    this.len = len;
}
//  将方法挂在对象得原型上
ControlIndex.prototype = {
    prev: function () {
        return this.getIndex(-1);
    },
    next: function () {

        return this.getIndex(1);
    },
    getIndex: function (val) {
        var index = this.index;
        var len = this.len;
        var curIndex = (index + val + len) % len;
        this.index = curIndex;
        return curIndex;
    }
}
// AudioManager 模块
function AudioManager(){
```

```
    // 创建一个音频 audio 对象
    this.audio = new Audio();
    // 默认音乐状态暂停
    this.status = 'pause';
}
AudioManager.prototype = {
    play:function(){
        this.audio.play();
        this.status = 'play';
    },
    pause:function(){
        this.audio.pause();
        this.status = 'pause';
    },
    setAudioSource:function(src){
        this.audio.src = src;
        // 重新加载音频元素
        this.audio.load();
    }
}
```

在这里使用的基于原型编程的一个例子，将一个项目中的不同模块分解，每个模块使用这种方式进行编程。复用性更好，同时分工明确，不单单是 A 方法做完 B 方法做，而是统一的交给管理对象去执行这些方法。

对于 Javascript 的函数来说，我们的函数是由很多不完善的地方，首先我们通过 function 声明的函数，可以声明，普通方法，构造函数，单纯是这两个函数我们是没有犯法区分的，在之前我们默认采用大头峰式写法，表明构造函数，但是必须每个遵守才可以，而且容易出问题，当把构造函数当成普通函数来执行，会产生全局变量，还可能会报错。

基于上面的种种现象：

我们需要类似的面向对象的思想进行编程。

我们的原始的 function 声明的构造函数，约束性，安全性不够，不足以支撑这种思想。

所以在新的语法规范当中 ECMAScript6 中引入了 class，基于原有 function 的方式的语法糖，让我们使用起来，更方便，更安全，目的性更强。

而在 ES6 中引入了 Class（类）这个概念，通过 class 关键字可以定义类。该关键字的出现使得其在对象写法上更加清晰，更像是一种面向对象的语言。ES6 的写法就会是这个样子：

```
class Person{//定义了一个名字为 Person 的类
    constructor(name,age){//constructor 是一个构造方法，用来接收参数
        this.name = name;//this 代表的是实例对象
        this.age=age;
    }
    say(){//这是一个类的方法，注意千万不要加上 function
        return "我的名字叫" + this.name+"今年"+this.age+"岁了";
    }
}
var obj=new Person("duyi",1);
console.log(obj.say());//我的名字叫 duyi 今年 1 岁了
```

但是要注意的是：

```
1.在类中声明方法的时候，千万不要给该方法加上 function 关键字
2.方法之间不要用逗号分隔，否则会报错
```

通过以下代码可以看出类实质上就是一个函数。类自身指向的就是构造函数。所以可以认为 ES6 中的类其实就是构造函数的另外一种写法！

```
console.log(typeof Person);//function
console.log(Person===Person.prototype.constructor);//true
```

以下代码说明构造函数的 prototype 属性，在 ES6 的类中依然存在着。

```
console.log(Person.prototype);//输出的结果是一个对象
```

实际上类的所有方法都定义在类的 prototype 属性上。

一起来证明一下：

```
Person.prototype.say=function(){//定义与类中相同名字的方法。成功实现了覆盖！
    return "我是来证明的，你叫" + this.name+"今年"+this.age+"岁了";
}
var obj=new Person("duyi",1);
console.log(obj.say());//我是来证明的，你叫 duyi 今年 1 岁了
```

当然也可以通过 prototype 属性对类添加方法。如下：

```
Person.prototype.laodeng=function(){
    return "我是通过 prototype 新增加的方法,名字叫 laodeng";
}
var obj=new Person("duyi",1);
console.log(obj.laodeng());//我是通过 prototype 新增加的方法,名字叫 laodeng
```

还可以通过 Object.assign 方法来为对象动态增加方法

```
Object.assign(Person.prototype,{
    getName:function(){
        return this.name;
    },
    getAge:function(){
        return this.age;
    }
})
var obj=new Person("duyi",1);
console.log(obj.getName());//duyi
console.log(obj.getAge());//1
```

constructor 方法是类的构造函数的默认方法，通过 new 命令生成对象实例时，自动调用该方法。

```
class Box{
    constructor(){
        console.log("啦啦啦，今天天气好晴朗");//当实例化对象时该行代码会执行。
    }
}
var obj=new Box();
```

constructor 方法如果没有显式定义，会隐式生成一个 constructor 方法。所以即使你没有添加构造函数，构造函数也是存在的。constructor 方法默认返回实例对象 this，但是也可以指定 constructor 方法返回一个全新的对象，让返回的实例对象不是该类的实例。

```
class Desk{
    constructor(){
        this.xixi="哥是一只小小小小鸟！哦";
    }
}
class Box{
```

```
    constructor(){
      return new Desk();// 这里没有用 this，直接返回一个全新的对象
    }
}
var obj=new Box();
console.log(obj.xixi);//哥是一只小小小小鸟！哦
```

constructor 中定义的属性可以称为实例属性（即定义在 this 对象上），constructor 外声明的属性都是定义在原型上的，可以称为原型属性（即定义在 class 上)。hasOwnProperty()函数用于判断属性是否是实例属性。其结果是一个布尔值，true 说明是实例属性，false 说明不是实例属性。in 操作符会在通过对象能够访问给定属性时返回 true,无论该属性存在于实例中还是原型中。

```
class Box{
    constructor(num1,num2){
        this.num1 = num1;
        this.num2=num2;
    }
    sum(){
        return num1+num2;
    }
}
var box=new Box(12,88);
console.log(box.hasOwnProperty("num1"));//true
console.log(box.hasOwnProperty("num2"));//true
console.log(box.hasOwnProperty("sum"));//false
console.log("num1" in box);//true
console.log("num2" in box);//true
console.log("sum" in box);//true
console.log("say" in box);//false
```

类的所有实例共享一个原型对象，它们的原型都是 Person.prototype，所以 **proto** 属性是相等的

```
class Box{
    constructor(num1,num2){
        this.num1 = num1;
        this.num2=num2;
    }
    sum(){
        return num1+num2;
    }
}
//box1 与 box2 都是 Box 的实例。它们的__proto__都指向 Box 的 prototype
var box1=new Box(12,88);
```

```
var box2=new Box(40,60);
console.log(box1.__proto__===box2.__proto__);//true
```

由此，也可以通过 **proto** 来为类增加方法。使用实例的 **proto** 属性改写原型，会改变 Class 的原始定义，影响到所有实例，所以不推荐使用！

```
class Box{
    constructor(num1,num2){
        this.num1 = num1;
        this.num2=num2;
    }
    sum(){
        return num1+num2;
    }
}
var box1=new Box(12,88);
var box2=new Box(40,60);
box1.__proto__.sub=function(){
    return this.num2-this.num1;
}
console.log(box1.sub());//76
console.log(box2.sub());//20
```

class 不存在变量提升，所以需要先定义再使用。因为 ES6 不会把类的声明提升到代码头部，但是 ES5 就不一样,**ES5 存在变量提升**,可以先使用，然后再定义。

```
//ES5 可以先使用再定义,存在变量提升
new A();
function A(){

}
//ES6 不能先使用再定义,不存在变量提升 会报错
new B();//B is not defined
class B{

}
```

这是我们对 ES6 中 class（类）的概念的了解，既然提出了类，这个类又是怎么实现的呢？在这首先要了解一下类的继承：有三种属性，公有属性，私有属性， 静态属性（Es7）/静态类（Es6）

继承公有属性：

```
function Parent(){
    this.name = 'parent';
}
new Parent();//this 指向当前实例
Parent() //this 指向 window
```

```
function Child(){
    this.age = 9;
    Parent.call(this);//相当于 this.name = 'parent'   //继承父类属性
}
继承父类属性
function Parent(){
    this.name = 'parent';
}
Parent.prototype.eat = function(){
    console.log('eat')
}
function Child(){
    this.age = 9;
    Parent.call(this);//相当于 this.name = 'parent'   //继承父类属性
}
Child.prototype.smoking = function(){
    console.log('smoking')
}
Child.prototype = Parent.prototype;//这个不叫继承
//因为这样如果改变 Child.prototype 加属性，Parent.prototype 的实例也会有这个属性，，
此时这两者属于兄弟关系

Child.prototype._proto_ = Parent.prototype    // 方法一
//object.create
Child.prototype = object.create(Parent.prototype); // 常用,方法二
function create(parentPrototype,props){
    function Fn(){}
    Fn.prototype = parentPrototype;
    let fn = new Fn();
    for(let key in props){
        Object.defineProperty(fn,key,{
            ...props[key],
            enumerable:true
        });
    }
    return fn();
}
Child.prototype = create(Parent.prototype,{constructor:{value:Child}})

继承公有属性和私有属性 Child.prototype = new Parent()

之前的继承都是原型链的继承，圣杯模式
在 Class 中 的继承，有什么不一样了呢，
在 ES5 中真正的继承应该是什么样呢
```

```
function Animal (weight, name) {
        this.name = name
        this.age = 0
        this.weight = 10

    }

    Animal.prototype.eat = function () {
            console.log('animal eat')
        }
    Animal.prototype.drink = function () {
            console.log('a d')
        }

    function Person(name, weight) {
         Animal.call(this, weight, name)
    }

    Person.prototype = Object.create(Animal.prototype)
    Person.prototype.eat = function () {
        console.log('Person eat')
    }

    var p = new Person('dxm', 70)
```

这里有两点，要注意的地方，首先是 1 父类构造函数 为什么要 call

2 原型需要重写。

在 class 中继承就变的简单了许多，同比上面的例子通过 class 来实现

```
class Animal {
            constructor(name) {
                this.name = name
              }
            eat() {
                console.log('a e')
            }
            drink() {
                console.log('a d')
            }

    }

    class Person extends Animal {
            constructor(name) {
                super(name)
            }
```

```
        eat() {
            console.log('p e')
        }
    }
```

在这里实现之后，多了两个大家不认识的次， extends super
extends 后面跟的就是我们要继承的内容

super 有些注意点
子类也叫派生类，必须在 constructor 中调用 super 函数，要不无法使用 this， 准确的说子类是没有 this 的
就算是继承之后什么也不写，默认的也会填上去

```
class Person extends Animal {
     constructor(...arg) {
        super(...arg)
}
}
```

在调用 super 之前不可以使用 this 回报错
super 可以作为对象使用，指向的是 父类的原型
super 调用父类方法时，会绑定子类的 this
类的编译
1、类只能 new

```
class Parent{
    //私有属性
    constructor(){
        this.name = 'parent',
        this.age = '40'
    }
    //公有属性，原型上的方法
    eat(){
        console.log('eat')
    }
    //静态方法/属性   es6/es7
    //属于类上的方法  Child.a()
    static b(){
        return 2
    }
}
new Parent();
class Child extends Parent{ //继承父亲的私有和公有
    //私有属性
    constructor(){
        super() //  相当于 Parent.call(this)
        this.name = 'child'
    }
```

```
    //公有属性，原型上的方法
    smoking(){
        console.log('smoking')
    }
    //静态方法/属性   es6/es7
    //属于类上的方法  Child.a()
    static a(){
        return 1
    }
}
let child = new Child();
console.log(child.name,child.age,child.eat(),child.smoking,Child.b())
//类可以继承公有，私有和静态
//父类的构造函数中返回类一个引用类型，会把这个引用类型作为子类的 this
```

我们首先写一个创建类的函数

```
//检测实例是不是 new 出来的
function _classCallCheck(instance,constructor){
  if(!(instance instanceof constructor)){
    throw new Error('Class constructor Child cannot be invoked without new')
  }
}
//constructor 构造函数
//prprotoPropertys 构造函数原型
//staticPropertys 静态方法的描述
function definePropertys(target,arr){
  for(let i=0;i<arr.length;i++){
    Object.defineProperty(target,arr[i].key,{
      ...arr[i],
      configurable : true,
      enumerable : true,
      writable:true
    })
  }
}
function _createClass(constructor,protoPropertys,staticPropertys){
  if(protoPropertys.length > 0){
    definePropertys(constructor.prototype,protoPropertys)
  }
  if(staticPropertys.length > 0){
    definePropertys(constructor,staticPropertys)
  }
}
```

```
let Parent = function(){
   //写逻辑
   function P(){
      _classCallCheck(this,P)
      this.name = 'parent';
      //return {}
   }
   _createClass(P,//属性描述器
      [
         {
            key: 'eat',
            value: function () {
               console.log('吃')
            }
         }
      ],
      [
         {
            key:'b',
            value:function () {
               return 2;
            }
         }
      ]
   )
   return P;
}()
let p = new Parent();
console.log(p.eat())
```

这样我们的类就创建完成了