# JS对象模型

JavaScript 是一种基于原型（Prototype）的面向对象语言，而不是基于类的面向对象语言。

C++、Java有类Class和实例Instance的概念，类是一类事物的抽象，而实例则是类的实体。

JS是基于原型的语言，它只有原型对象的概念。原型对象就是一个模板，新的对象从这个模板构建从而获取最初的属性。任何对象在运行时可以动态的增加属性。而且，任何一个对象都可以作为另一个对象的原型，这样后者就可以共享前者的属性。

## 定义类

### 字面式声明方式

```
var obj = {
    property_1:   value_1,   // property_# may be an identifier...
    property_2:   value_2,   // or a number...
    ...,
    "property n": value_n    // or a string
};

var obj = {
    x : 1,
    1 : 'abc',
    'y':'123'
}

for (let s in obj)
    console.log(s, typeof(s));
```

这种方法也称作字面值创建对象。
Js 1.2开始支持。

### ES6之前——构造器

1、定义一个函数（构造器）对象，函数名首字母大写
2、使用this定义属性
3、使用new和构造器创建一个新对象

```
// 定义类，构造器
function Point(x, y) {
    this.x = x;
    this.y = y;
    this.show = () => {console.log(this,this.x,this.y)};
    console.log('Point~~~~~~~~');
}

console.log(Point);
```

```
p1 = new Point(4,5);
console.log(p1);
console.log('----------------');
// 继承
function Point3D(x,y,z) {
    Point.call(this,x,y); // "继承"
    this.z = z;
    console.log('Point3D~~~~~~~~');
}

console.log(Point3D);
p2 = new Point3D(14,15,16);
console.log(p2);
p2.show();
```

new 构建一个新的通用对象，new操作符会将新对象的this值传递给Point3D构造器函数，函数为这个对象创建z属性。
从上句话知道，new后得到一个对象，使用这个对象的this来调用构造器，那么如何执行"基类"的构造器方法呢？
使用Point3D对象的this来执行Point的构造器，所以使用call方法，传入子类的this。
最终，构造完成后，将对象赋给p2。

注意：如果不使用new关键字，就是一次普通的函数调用，this不代表实例。

## ES6中的class

从ES6开始，新提供了class关键字，使得创建对象更加简单、清晰。

1、类定义使用class关键字。创建的本质上还是函数，是一个特殊的函数
2、一个类只能拥有一个名为constructor的构造器方法。如果没有显式的定义一个构造方法，则会添加一个默认的constuctor方法。
3、继承使用extends关键字
4、一个构造器可以使用super关键字来调用一个父类的构造函数
5、类没有私有属性

```
// 基类定义
class Point {
    constructor(x,y) /*构造器*/ {
        this.x = x;
        this.y = y;
    }
    show() /*方法*/ {
        console.log(this,this.x,this.y);
    }
}

let p1 = new Point(10,11)
p1.show()

// 继承
class Point3D extends Point {
    constructor (x,y,z) {
        super(x,y);
```

```
        this.z = z;
    }
}

let p2 = new Point3D(20,21,22);
p2.show()
```

## 重写方法

子类Point3D的show方法，需要重写

```
// 基类定义
class Point {
    constructor(x,y) /*构造器*/ {
        this.x = x;
        this.y = y;
    }
    show() /*方法*/ {
        console.log(this,this.x,this.y);
    }
}

let p1 = new Point(10,11)
p1.show()

// 继承
class Point3D extends Point {
    constructor (x,y,z) {
        super(x,y);
        this.z = z;
    }

    show(){ // 重写
        console.log(this,this.x,this.y, this.z);
    }
}

let p2 = new Point3D(20,21,22);
p2.show();
```

子类中直接重写父类的方法即可。
如果需要使用父类的方法，使用super.method()的 方式调用。

使用箭头函数重写上面的方法

```
// 基类定义
// 基类定义
class Point {
    constructor(x,y) /*构造器*/ {
        this.x = x;
```

```
        this.y = y;
        //this.show = function () {console.log(this,this.x,this.y)};
        this.show = () => console.log('Point');
    }
}

// 继承
class Point3D extends Point {
    constructor (x,y,z) {
        super(x,y);
        this.z = z;
        this.show = () => console.log('Point3D');
    }
}

let p2 = new Point3D(20,21,22);
p2.show(); // Point3D
```

从运行结果来看，箭头函数也支持子类的覆盖

```
// 基类定义
class Point {
    constructor(x,y) /*构造器*/ {
        this.x = x;
        this.y = y;
        this.show = () => console.log('Point');
    }
    // show() /*方法*/ {
    //     console.log(this,this.x,this.y);
    // }
}

// 继承
class Point3D extends Point {
    constructor (x,y,z) {
        super(x,y);
        this.z = z;
        //this.show = () => console.log('Point3D');
    }

    show(){ // 重写
        console.log('Point3D');
    }
}

let p2 = new Point3D(20,21,22);
p2.show(); // Point
```

上例优先使用了父类的属性show

```
// 基类定义
```

```
class Point {
    constructor(x,y) /*构造器*/ {
        this.x = x;
        this.y = y;
        //this.show = () => console.log('Point');
    }
    show() /*方法*/ {
        console.log(this,this.x,this.y);
    }
}

// 继承
class Point3D extends Point {
    constructor (x,y,z) {
        super(x,y);
        this.z = z;
        this.show = () => console.log('Point3D');
    }
}

let p2 = new Point3D(20,21,22);
p2.show(); // Point3D
```

优先使用了子类的属性。

总结
父类、子类使用同一种方式类定义属性或者方法，子类覆盖父类。
访问同名属性或方法时，优先使用属性。

## 静态属性

静态属性目前还没有得到很好的支持。

## 静态方法

在方法名前加上static，就是静态方法了。

```
class Add {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    static print(){
        console.log(this.x); // ? this是什么
    }
}

add = new Add(40, 50);
console.log(Add);
Add.print();
//add.print(); // 实例不能访问直接访问静态方法，和C++、Java一致
add.constructor.print(); // 实例可以通过constructor访问静态方法
```

静态方法中的this是Add类，而不是Add的实例

注意：
静态的概念和Python的静态不同，相当于Python中的类变量。

## this的坑

虽然Js和`C++`、Java一样有this，但是Js的表现是不同的。
原因在于，`C++`、Java是静态编译型语言，this是编译期绑定，而Js是动态语言，运行期绑定。

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return function () {
            console.log(this === global); // this是否是global对象
            return this.name;
        };
    }
};

console.log(school.getNameFunc()());

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
true
undefined
*/
```

为了分析上面的程序，先学习一些知识：
函数执行时，会开启新的执行上下文环境ExecutionContext。
创建this属性，但是this是什么就要看函数是怎么调用的了。
1、myFunction(1,2,3)，普通函数调用方式，this指向**全局对象**。全局对象是nodejs的global或者浏览器中的window。
2、myObject.myFunction(1,2,3)，对象方法的调用方式，this指向包含该方法的对象。
3、call和apply方法调用，要看第一个参数是谁。

分析上例
`magedu` 和 `{ name: 'magedu', getNameFunc: [Function: getNameFunc] }`很好理解。
第三行打印的true，是`console.log(this == global)`执行的结果，说明当前是global，因为调用这个返回的函数是直接调用的，这就是个普通函数调用，所以this是全局对象。
第四行undefined，就是因为this是global，没有name属性。

这就是函数调用的时候，调用方式不同，this对应的对象不同，它已经不是C++、Java的指向实例本身了。
this的问题，这是历史遗留问题，新版只能保留且兼容了。
而我们在使用时，有时候需要明确的让this必须是我们期望的对象，如何解决这个问题呢？

### 1 显式传入

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return function (that) {
            console.log(this == global); // this是否是global对象
            return that.name;
        };
    }
}

console.log(school.getNameFunc()(school));

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
false
magedu
*/
```

通过主动传入对象，这样就避开了this的问题

## 2 ES3（ES-262第三版）引入了apply、call方法

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return function () {
            console.log(this == global); // this是否是global对象
            return this.name;
        };
    }
}

console.log(school.getNameFunc().call(school)); // call方法显式传入this对应的对象

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
false
magedu
*/
```

apply、call方法都是函数对象的方法，第一参数都是传入对象引入的。
apply传其他参数需要使用数组
call传其他参数需要使用可变参数收集

```
function Print() {
    this.print = function(x,y) {console.log(x + y)};
}

p = new Print(1,2)
p.print(10, 20)
p.print.call(p, 10, 20);
p.print.apply(p, [10, 20]);
```

### 3 ES5 引入了bind方法

bind方法来设置函数的this值

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return function () {
            console.log(this == global); // this是否是global对象
            return this.name;
        };
    }
}

console.log(school.getNameFunc().bind(school)); // bind方法绑定

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
[Function: bound ]
*/
```

只打印了三行，说明哪里有问题，问题出在bind方法用错了。

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return function () {
            console.log(this == global); // this是否是global对象
            return this.name;
        };
    }
}

var func = school.getNameFunc();
console.log(func);

var boundfunc = func.bind(school); // bind绑定后返回新的函数
```

```
console.log(boundfunc);
console.log(boundfunc());

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
[Function]
[Function: bound ]
false
magedu
*/
```

apply、call方法，参数不同，调用时传入this。
bind方法是为函数先绑定this，调用时直接用。

## 4 ES6引入支持this的箭头函数

ES6 新技术，就不需要兼容this问题。

```
var school = {
    name : 'magedu',
    getNameFunc : function ()  {
        console.log(this.name);
        console.log(this);
        return () => {
            console.log(this == global); // this是否是global对象
            return this.name;
        };
    }
}

console.log(school.getNameFunc()());

/* 运行结果
magedu
{ name: 'magedu', getNameFunc: [Function: getNameFunc] }
false
magedu
*/
```

ES6 新的定义方式如下

```
class school{
    constructor(){
        this.name = 'magedu';
    }

    getNameFunc()  {
        console.log(this.name);
        console.log(this, typeof(this));
        return () => {
```

```
            console.log(this == global); // this是否是global对象
            return this.name;
        };
    }
}

console.log(new school().getNameFunc()());
/* 运行结果
magedu
school { name: 'magedu' } 'object'
false
magedu
*/
```

以上解决this问题的方法，bind方法最常用。

---

# 高阶对象、高阶类、或称Mixin模式

Mixin模式，混合模式。这是一种不用继承就可以复用的技术。主要还是为了解决多重继承的问题。多继承的继承路径是个问题。

JS是基于对象的，类和对象都是对象模板。
混合mixin，指的是将一个对象的全部或者部分拷贝到另一个对象上去。其实就是属性了。
可以将多个类或对象混合成一个类或对象。

## 继承实现

先看一个继承实现的例子

```
class Serialization{
    constructor(){
        console.log('Serialization constructor~~~');
        if (typeof(this.stringify) !== 'function') {
            throw new ReferenceError('should define stringify.');
        }
    }
}

class Point extends Serialization {
    constructor(x,y){
        console.log('Point Constructor~~~~');
        super(); // 调用父构造器
        this.x = x;
        this.y = y;
    }
}

//s = new Serialization(); // 构造Serialization失败
//p = new Point(4,5); // 构造子类对象时，调用父类构造器执行也会失败
```

父类构造函数中，要求具有属性是stringify的序列化函数，如果没有则抛出异常。
以下是完整继承的代码

```
class Serialization{
    constructor(){
        console.log('Serialization constructor~~~');
        if (typeof(this.stringify) !== 'function') {
            throw new ReferenceError('should define stringify.');
        }
    }
}

class Point extends Serialization {
    constructor(x,y){
        console.log('Point Constructor~~~~');
        super(); // 调用父构造器
        this.x = x;
        this.y = y;
    }

    stringify () {
        return `<Point x=${this.x}, y=${this.y}>`
    }
}

class Point3D extends Point {
    constructor(x,y,z){
        super(x,y);
        this.z = z;
    }

    stringify () {
        return `<Point x=${this.x}, y=${this.y}, z=${this.z}>`
    }
}

p = new Point(4,5);
console.log(p.stringify())
p3d = new Point3D(7,8,9);
console.log(p3d.stringify());
```

## 高阶对象实现

将类的继承构建成箭头函数。

```
// 普通的继承
class A extends Object {};
console.log(A);

// 匿名类
const A1 = class {
```

```
    constructor(x) {
        this.x = x;
    }
}
console.log(A1);
console.log(new A1(100).x);

// 匿名继承
const B = class extends Object{
    constructor(){
        super();
        console.log('B constructor');
    }
};
console.log(B);
b = new B();
console.log(b);


// 箭头函数，参数是类，返回值也是类
// 把上例中的Object看成参数
const x = (Sup) => {
    return class extends Sup {
        constructor(){
            super();
            console.log('C constructor');
        }
    };
}
// 演化成下面的形式
const C = Sup => class extends Sup {
    constructor(){
        super();
        console.log('C constructor');
    }
};

//cls = new C(Object); // 不可以new，因为是一个普通函数，它的返回值是一个带constructor的类
cls = C(A); // 调用它返回一个类，一个带constructor的class
console.log(cls);
c = new cls();
console.log(c);

// 其它写法
c1 = new (C(Object))(); // new优先级太高了，所有后面要加括号才能先调用
```

可以改造上面序列化的例子了

```
const Serialization = Sup => class extends Sup {
    constructor(...args) {
        console.log('Serialization constructor~~~');
        super(...args);
```

```
        if (typeof(this.stringify) !== 'function'){
            throw new ReferenceError('should define stringify.');
        }
    }
}

class Point {
    constructor(x,y){
        console.log('Point Constructor~~~~');
        this.x = x;
        this.y = y;
    }
}

class Point3D extends Serialization(Point) {
    constructor(x,y,z){
        super(x,y); // super是Serialization(Point)包装过的新类型
        this.z = z;
    }

    stringify () {
        return `<Point3D ${this.x}.${this.y}.>`;
    }
}

let p3d = new Point3D(70,80,90);
console.log(p3d.stringify());
```

注意：
Serialization(Point)这一步实际上是一个匿名箭头函数调用，返回了一个新的类型，Point3D继承自这个新的匿名类型，增强了功能。
React框架大量使用了这种Mixin技术。