█js数据类型

◎JS数据类型：

基本数据类型

undefined

null

number

string

boolean

复杂数据类型

object

◎对象的种类：

Js内置的（如Number）

宿主环境（如window）

自己创建的

█

对象的私有变量，以及通过get和set来访问和修改私有变量

<script>

        var p = {

            name: 'jojo',

            work: function () {

                console.log('working...')

            },

            \_age:18, //带有下划线开头的，是对象的私有变量，不能在外部直接访问（虽然可以，但不建议在外部访问）

            // 可以使用下面的get和set方法来从外面访问和修改私有变量\_age

            get age () {

                return this.\_age

            },

            set age (val) {

                if (val < 0 || val > 150) {

                    throw new Error('invalid value')

                } else {

                    this.\_age = val

                }

            }

        }

        // 访问私有变量\_age

        console.log(p.age) //18

        // 修改私有变量\_age

        p.age = 30

        console.log(p.age) //30

    </script>

█对象的访问

通常使用点和中括号来访问对象中的属性或方法，使用中括号来访问有如下两点特点：

第一

<script>

        var a = {

            1: 'haha'

        }

        // 如果用a.1来访问会报错，因为JS不建议用数字来做键名

        // console.log(a.1)

        // 使用[]来访问则可正常运行

        console.log(a[1]) //haha

    </script>

第二，使用中括号可以以变量的形式来访问

<script>

        var a = {

            b1: '11',

            b2: '22'

        }

        var b = 'b1'

        console.log(a[b]) //11

    </script>

█使用||来访问不确定的属性

var p = {

np: {

nnp: 'haha'

}

}

// 当不确定p里面是否有某个属性时，可以用||

// 从左至右，只要有一个成立，就返回这个

var r = p.np.nnp || p.np || p

console.log(r) //haha

var r2 = p.np.nnnp || p.np || p

console.log(r2) //{ nnp: 'haha' }

<https://www.jb51.net/article/140789.htm>

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<h2>ss</h2>

<script>

function check () {

return true

}

function do1 () {

return 111

}

var b = {

a : 'aa'

}

var a = b.c || "221" //如果 obj中没有相应的属性c，a就赋值为 "221" ；

var b = check() && do1() //如果check()返回为真，就执行do()，并将结果赋值给 a;

console.log(a)

console.log(b)

</script>

</body>

</html>

█遍历对象

For in 来遍历对象是无序的

<script>

        var p = {

            a: 'aa',

            b: 'bb',

            c: 'cc'

        }

        for (x in p) {

            console.log(x + ':' + p[x]) //不能用p.因为x是变量所以要用中括号

        }

        // 输出：

        // a:aa

        // b:bb

        // c:cc

    </script>

如需有序遍历对象，可使用Object.keys方法

<script>

        var p = {

            a: 'aa',

            b: 'bb',

            c: 'cc'

        }

        // Object.keys方法会返回一个有序的数组

        var ap = Object.keys(p)

        console.log(ap) //["a", "b", "c"]

    </script>

█关于对象属性的一些操作

查看对象中是否有某个属性

删除属性

查看对象属性的特性

<script>

        var p = {

            a: 'aa',

            b: 'bb',

            c: 'cc'

        }

        console.log('a' in p) //true

        console.log(p.hasOwnProperty('a')) //true

        console.log(p.b) //bb

        delete p.b //删除P的b属性

        console.log(p.b) //undefined

        console.log(Object.getOwnPropertyDescriptor(p, 'c')) //{value: "cc", writable: true, enumerable: true, configurable: true}

    </script>

█构造器或者类的研究

<script>

        // 构造器就是构造函数或者也可以理解为类

        // 使用Object构造器生成实例a和a2，通过constructor查询它们的构造器都是Object()

        var a = {}

        console.log(a.constructor) //ƒ Object() { [native code] }

        var a2 = new Object()

        console.log(a2.constructor) //ƒ Object() { [native code] }

        // 以下会报错，因为数字1没有构造器

        //console.log(1.constructor) //Uncaught SyntaxError: Invalid or unexpected token

        // 使用Number构造器创建实例b和b2，通过constructor查询它们的构造器都是Number()

        var b = 1 //把数字1赋值给变量，变量是对象，所以有构造器

        console.log(b.constructor) //ƒ Number() { [native code] }

        var b2 = new Number(1)

        console.log(b2.constructor) //ƒ Number() { [native code] }

        // 那么Object()和Number()又是谁创建出来的呢？通过constructor查询，它们是Function()创建出来的

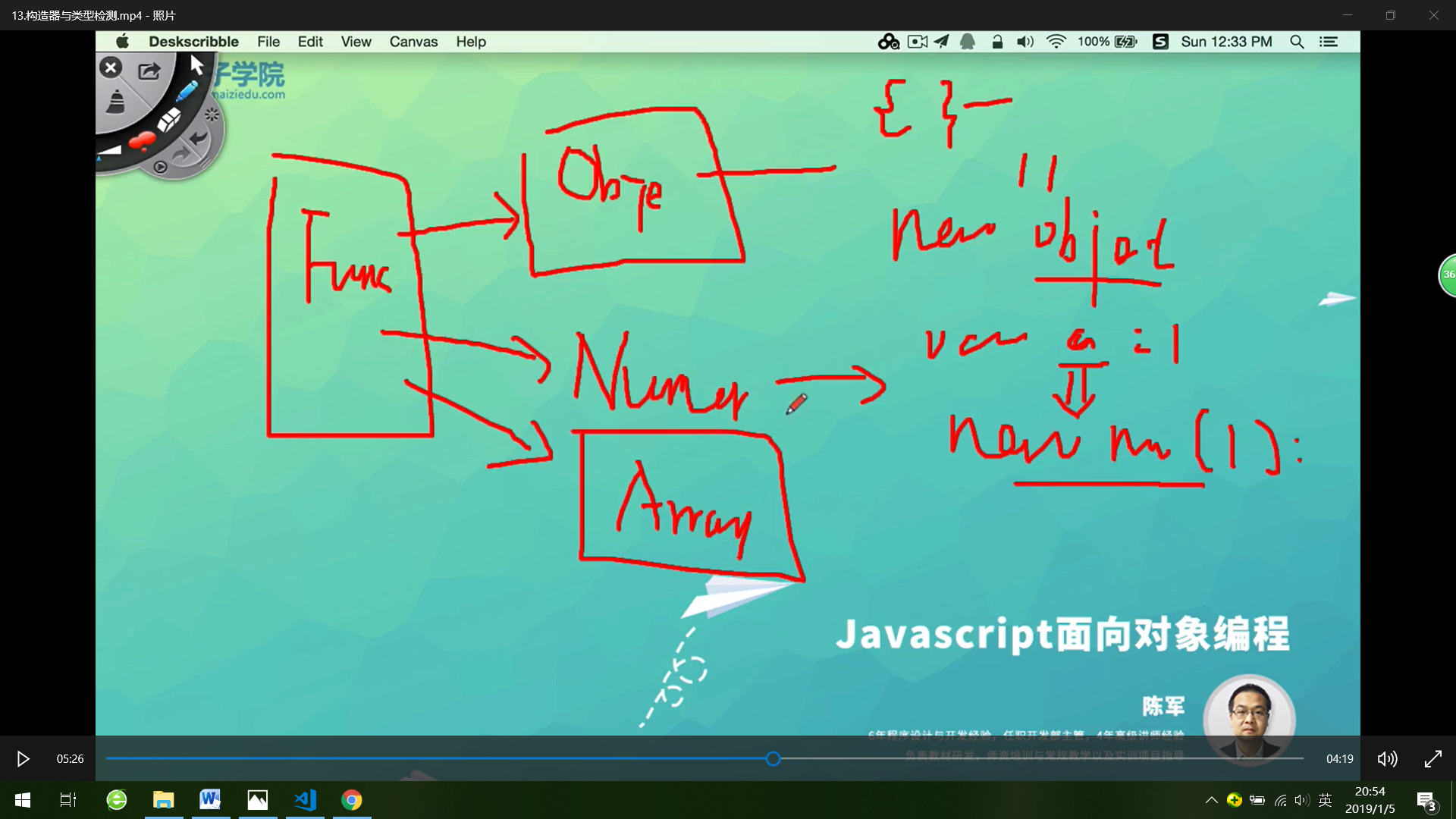
        console.log(Object.constructor) //ƒ Function() { [native code] }

        console.log(Number.constructor) //ƒ Function() { [native code] }

        // 那么Function()是谁创建的呢？通过constructor查询，它是自己创作的自己，所以它是最顶端的构造器，它创作了一切

        console.log(Function.constructor) //ƒ Function() { [native code] }

    </script>



█工厂模式

<script type="text/javascript">

            function createPerson(name, age, job){

                var o=new Object();

                o.name=name;

                o.age=age;

                o.job=job;

                o.sayName=function(){

                    alert(this.name);

                };

                return o;

            }

            var person1=createPerson("Tom", 29, "worker");

            var person2=createPerson("Jerry", 22, "teacher");

            console.log(person1.name); //Tom

            person2.sayName(); //Jerry

        </script>

以上例子还可以简写，直接return一个{} ，省略了在构造函数中创建一个对象o并return o

<script type="text/javascript">

            function createPerson(name, age, job){

                // 直接return一个{}

                return {

                    name: name,

                    age: age,

                    job: job,

                    sayName: function () {

                        alert(this.name)

                    }

                }

            }

            var person1=createPerson("Tom", 29, "worker");

            var person2=createPerson("Jerry", 22, "teacher");

            console.log(person1.name); //Tom

            person2.sayName(); //Jerry

        </script>

█构造函数模式

<script type="text/javascript">

            function Person(name, age, job){

                this.name=name;

                this.age=age;

                this.job=job;

                this.sayName=function(){

                    alert(this.name);

                };

            }

            var person1=new Person("Tom", 29, "worker");

            var person2=new Person("Jerry", 22, "teacher");

            Person("Alice", 25, "stundent"); //不使用new直接调用函数，那么this指的上一级对象，这里添加到了window

            console.log(person1.age); //29

            person2.sayName(); //Jerry

            window.sayName(); //Alice

        </script>

◎函数的调用方法有两种：

一种是直接调用方法，即，函数名加括号

另一种是构造函数调用方法，即，new加空格加函数名加括号

◎当在全局作用域中调用一个函数时，this对象总是指向Global对象（在浏览器中时window对象）。因此，上面例子中Person("Alice", 25, "stundent")将属性和方法添加给了window对象。之后可以通过window对象来调用其属性或方法。例如window.sayName

◎使用new操作符调用函数的方法叫构造函数方法。以这种方法调用的构造函数会经历以下4个步骤：

1. 创建一个新对象。
2. 将构造函数的作用域赋给新对象（因此this就指向了这个新对象）。
3. 执行构造函数中的代码（为这个新对象添加属性和方法）。
4. 返回新对象。

◎操作习惯：当创建一个函数时，如果希望创建构造函数，则函数名的首字母建议大写，这样以后便于区分哪些是普通函数，哪些是构造函数。

█原型模式

◎我们创建的每个函数都有一个prototype属性，这个属性是一个指针，指向一个对象。如果使用new操作符来把这个函数当作构造函数调用，那么prototype属性指向的对象就叫做这个构造函数创造出来的对象实例的原型对象。

◎Prototype属性指向的对象最初只有一个constructor属性，该属性也是一个指针，指向prototype属性所在的函数。

◎而被构造函数创建出来的对象实例的内在属性[[prototype]]指向其原型对象。

<script>

            //创建一个构造函数Person

            function Person(){

            }

            //定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

            Person.prototype.name="Tom";

            Person.prototype.age=29;

            Person.prototype.job="worker";

            Person.prototype.sayName=function(){

                alert(this.name);

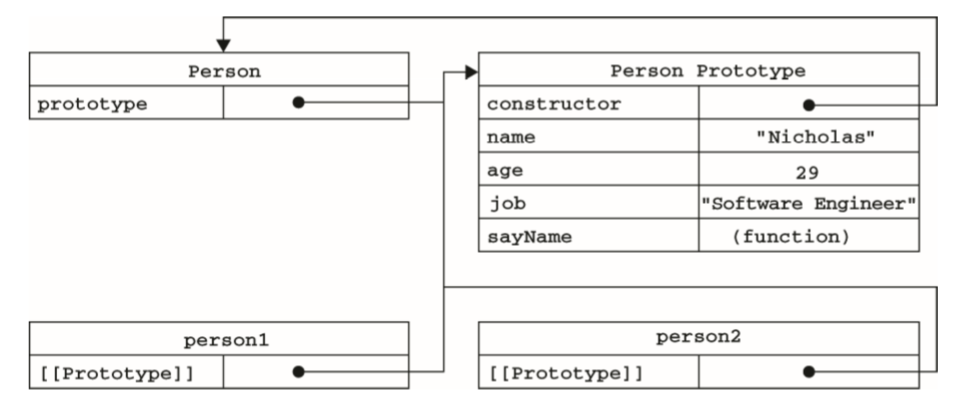
            };

            var person1=new Person();

            var person2=new Person();

            console.log(person1.job == person2.job); //会输出true，因为person1和person2引用同一个原型对象。

        </script>



总结：

原型和实例的关系：每个构造函数都有一个原型对象（构造函数的prototype属性指向这个原型对象），原型对象都包含一个指向构造函数的指针（constructor），而实例都包含一个指向原型对象的内部指针（[[prototype]]）。

上图可以用代码来验证

<script>

            function Fa () {} //建立一个构造函数Fa

            // 现在Fa有一个prototype属性，指向一个对象。这个对象叫原型对象，是将来用Fa创作出的实例对象的原型对象

            console.log(Fa.prototype) //{constructor: ƒ}

            var a = new Fa() //使用Fa构造出一个实例a

            // 现在，a里面有两个属性

            // constructor指向Fa

            // \_\_proto\_\_指向a的原型对象

            console.log(a.constructor) //ƒ Fa () {}

            console.log(a.\_\_proto\_\_) //{constructor: ƒ}

            console.log(a.\_\_proto\_\_ === Fa.prototype) //true

        </script>

◎通过简单方式来设置原型对象属性和方法，如下例

<script>

            //创建一个构造函数Person

            function Person(){

            }

            var friend=new Person();

            //简单方式定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

            Person.prototype={

                name:"Tom",

                age:29,

                job:"worker",

                sayName:function(){

                    alert(this.name);

                }

            }

            var person1=new Person();

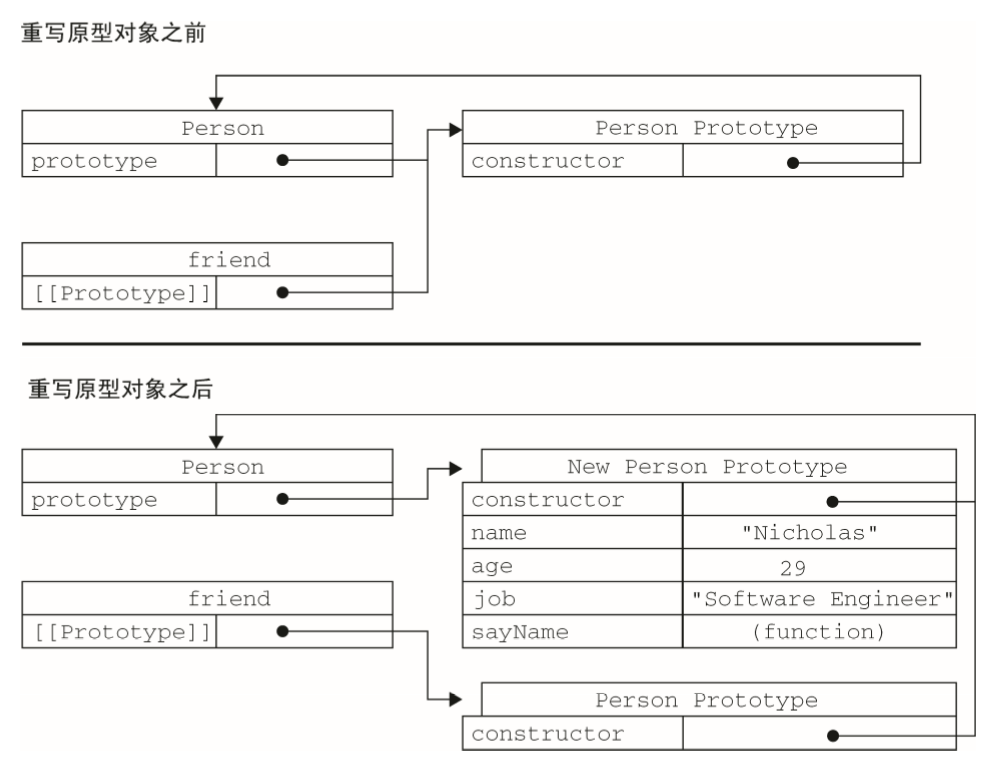
            var person2=new Person();

            console.log(person1.job == person2.job); //会输出true，因为person1和person2引用同一个原型对象。

            console.log(friend.name); //会输出undefined，因为使用简单方式重写构造函数Person的prototype后，实际是建立了一个新对象。和之前friend的原型对象不是一个对象了。

        </script>

以上例子等于是重写了原型对象，如下图描述



注意：这种方法重写的原型对象的constructor属性是可枚举的，而默认情况下，原生的constructor属性是不可枚举的。

◎原型对象的问题

因为基本类型的值是复制,引用类型的值是引用,所以在构造函数创造的实例中修改基本类型的值不会影响原型,而修改引用类型的值就会影响原型。

<script>

            //创建一个构造函数Person

            function Person(){

            }

            //简单方式定义通过Person构造函数来创建的对象实例的原型对象的属性和方法

            Person.prototype={

                name:"Tom",

                age:29,

                job:"worker",

                sayName:function(){

                    alert(this.name);

                },

                friends:["Jerry", "Alice"]

            }

            var person1=new Person();

            var person2=new Person();

            //因为基本类型的值是复制,引用类型的值是引用,所以person1中修改基本类型的值不会影响原型,而修改引用类型的值就会影响原型.

            person1.friends.push("Jon");

            person1.age=22;

            console.log(person1.friends); //输出"Jerry,Alice,Jon"

            console.log(person2.friends); //输出"Jerry,Alice,Jon"

            console.log(person1.age); //输出22

            console.log(person2.age); //输出29

        </script>

组合使用构造函数模式和原型模式来解决以上问题

<script>

            // 构造器模式

            function Person(name, age, job){

                this.name=name;

                this.age=age;

                this.job=job;

                this.friends=["Jo1", "Jo2"];

            }

            // 原型模式

            Person.prototype={

                constructor:Person,

                sayName:function(){

                    alert(this.name);

                }

            }

            var person1=new Person("Tom", 29, "worker");

            var person2=new Person("Jerry", 22, "teacher");

            person1.friends.push("Jo3");

            console.log(person1.friends); //输出"Jo1,Jo2,Jo3"

            console.log(person2.friends); //输出"Jo1,Jo2"

        </script>

█关于this

<script>

            // 情况1

            // 当点击某个DOM元素触发事件中的函数里面的this永远指向触发该事件DOM元素

            var ele = document.getElementById('dd')

            ele.addEventListener('click', function () {

                console.log(this)

            })

            // 点击div#dd时会输出<div id="dd">111</div>

            // 情况2

            // 函数内this的指向，是指向调用该函数的对象。即'.'前面的

            // this的指向是运行时决定，不是编写代码时决定

            var name = 'globalName'

            var o = {

                name: 'localName',

                print: function () {

                    console.log(this.name)

                }

            }

            // 这里指向o

            o.print() //localName

            var oo = o.print

            // 而这里指向的则是window对象，因为下面的代码也可以写成window.oo()

            oo() //globalName

        </script>

Call和apply可以改变this的指向

<script>

            function log (a, b) {

                console.log(this[a]) //不能使用this.a因为a是变量

                console.log(this[b])

            }

            var a = 'globalA'

            var b = 'globalB'

            var c = {

                a: 'localA',

                b: 'localB'

            }

            // 这里直接调用log时，this指向的是window对象

            log('a', 'b')

            // globalA

            // globalB

            // call和apply可以改变this的指向

            // 函数.call(对象（要使this指向这个对象）, 参数1, 参数2, ...)

            // 函数.apply(对象（要使this指向这个对象）, 数组（数组里是接受的参数）)

            log.call(c, 'a', 'b')

            // localA

            // localB

            log.apply(c, ['a', 'b'])

            // localA

            // localB

        </script>

Call还有隐含的用法，可以修改this指向的对象中的属性

<script>

            // 在这里，函数p的作用是给this指向的对象增加一个属性name

            function p () {

                this.name = '111'

            }

            var o = {}

            p.call(o) //这里等同于o.p()

            console.log(o.name) //原来o里没有属性name，通过p函数添加了该属性

        </script>

█封装 及 构造函数中私有和共有属性或方法

基本概念

<script>

            function Person (pname) {

                // 定义私有属性或方法，只有在函数内部才能访问

                var age = 100

                function pm () {

                    console.log('private method')

                }

                // 定义共有属性或方法

                this.name = pname

                this.test = function () {

                    console.log('public method')

                }

            }

            var p = new Person('jojo')

            console.log(p.age) //undefined

            //p.pm() //Uncaught TypeError: p.pm is not a function

            console.log(p.name) //jojo

            p.test() //public method

        </script>

函数内部的共有和私有互相引用的例子

<script>

            function Person (pname) {

                // 定义私有属性或方法，只有在函数内部才能访问

                var age = 100

                function pm () {

                    console.log(this.name)

                }

                // 定义共有属性或方法

                this.name = pname

                this.test = function () {

                    //pm() //如果这样写，则调用的是window.pm()

                    pm.call(this)

                }

            }

            var p = new Person('jojo')

            console.log(p.age) //undefined

            //p.pm() //Uncaught TypeError: p.pm is not a function

            console.log(p.name) //jojo

            p.test() //jojo

        </script>

还可以用工厂模式来实现

<script>

            function Person (pname) {

                function pm () {

                    console.log(self.name)

                }

                var self = {

                    name: pname,

                    test: function () {

                        pm()

                    }

                }

                return self

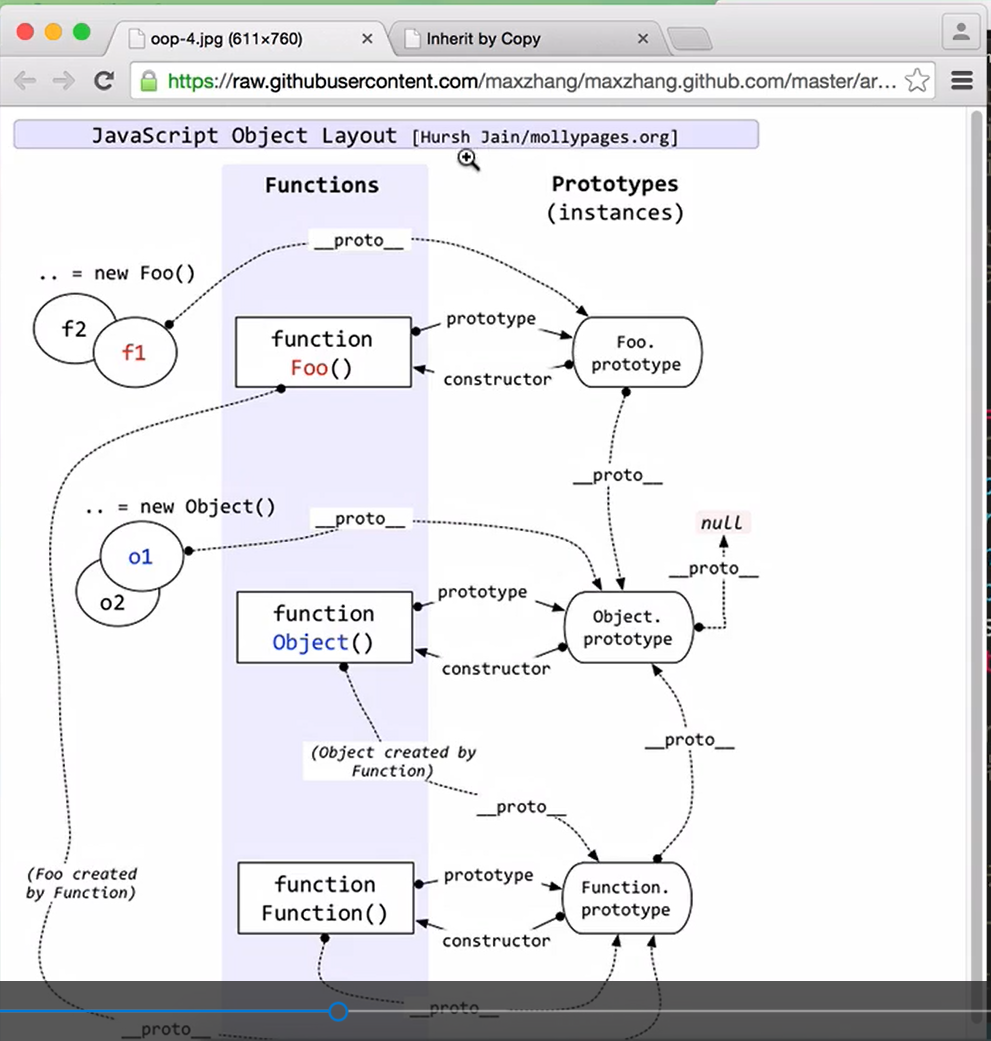
            }

            var p = Person('jojo')

            p.test() //jojo

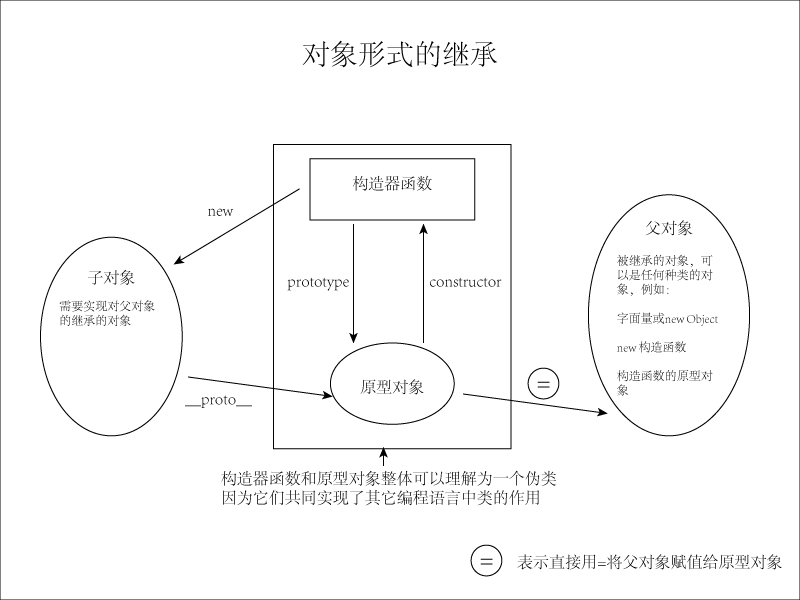
        </script>

█原型链



█继承

●对象形式的继承



以上图片的代码实现如下：

<script>

            // 父对象，将要被子对象继承

            var parentO = {

                name: 'jojo',

                age: 22

            }

            function myCreate(o) {

                // 建立一个空对象，在函数结尾return出去

                var ins = null

                // 建立一个中间的构造器函数，用来桥接

                function F () {}

                F.prototype = o

                ins = new F()

                return ins

            }

            var childO = myCreate(parentO)

            // 这个myCreate方法其实相当于es5的Object.create方法

            var childO = Object.create(parentO)

            console.log(childO.name) //jojo

            console.log(parentO.name) //jojo

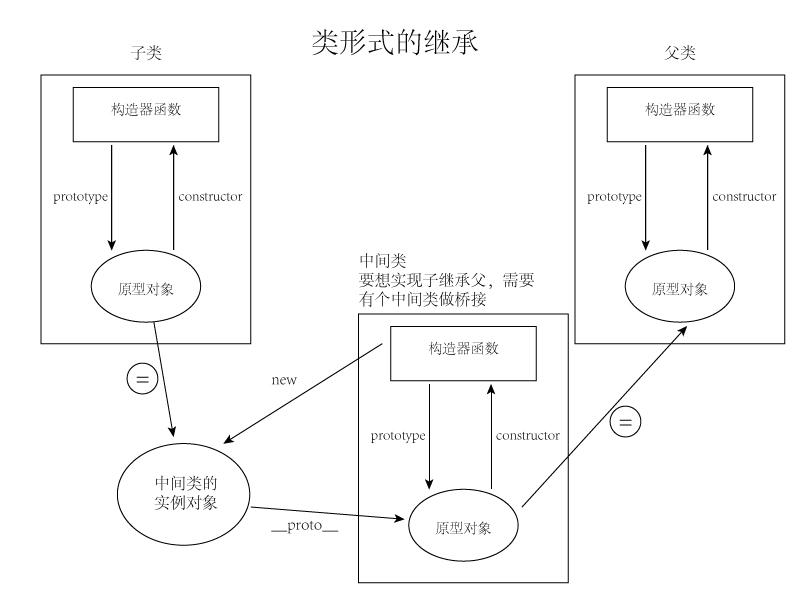
            childO.name = 'haha'

            console.log(childO.name) //haha

            console.log(parentO.name) //jojo

        </script>

●类形式的继承



问题一：中间桥接不用类，只用一个对象行吗？

答案是可以，可以把子类的原型对象指向父类的实例（new 父类）

但是这种方法存在以下缺陷

<script>

            // 创建一个父类P

            function P () {

                this.name = 'jojo'

            }

            // 创建一个子类C

            function C () {}

            // 将C的prototype指向一个对象new P()

            //由于这个对象是P的实例，所以它的\_\_proto\_\_指向P的prototype

            // 从而实现了桥接

            C.prototype = new P()

            var c1 = new C()

            console.log(c1 instanceof C) //true

            console.log(c1 instanceof P) //true

            // 但是这种使用一个实例对象桥接的形式有个缺点

            // 就是P里的一些属性也都会被C继承，因为new P()包括了这些属性

            // 而我们需要的是C只继承P.prototype里的而不要P构造函数里的属性

            console.log(c1.name) //jojo

        </script>

类形式的继承的正确实现方式如下

<script>

            // 创建一个父类P

function P () {

this.name = 'jojo'

}

// 创建一个子类C

function C () {}

// 创建一个中间类，用于桥接,并将其prototype指向父类的prototype

function F () {}

F.prototype = P.prototype

// new一个中间类的实例，然后将子类的prototype指向它

var f = new F()

C.prototype = f

            // 这里要说明一下，这样会使C的prototype的constructor会指向P，需要手动修正回来，应该指向C

            console.log(C.prototype.constructor) //指向P

            C.prototype.constructor = C

            console.log(C.prototype.constructor) //指回了C

var c1 = new C()

console.log(c1 instanceof C) //true

console.log(c1 instanceof P) //true

console.log(c1.name) //undefined 不会继承父类prototype以外的属性

        </script>

以上的实现过程还可以用es5的create来简单实现

<script>

                    // 创建一个父类P

                    function P () {

                        this.name = 'jojo'

                    }

                    // 创建一个子类C

                    function C () {}

                    C.prototype = Object.create(P.prototype)

                    // es5的create方法也需要手动修改一下C的prototype的constructor的指向

                    console.log(C.prototype.constructor) //指向P

                    C.prototype.constructor = C

                    console.log(C.prototype.constructor) //指回了C

                    var c1 = new C()

                    console.log(c1 instanceof C) //true

                    console.log(c1 instanceof P) //true

                    console.log(c1.name) //undefined 不会继承父类prototype以外的属性

                </script>

综合案例

任务1，将类形式继承的过程封装到一个函数creatEx中

任务2，实现对父类的实例属性也继承

<script>

        function P (name, age) {

            this.name = name

            this.age = age

        }

        P.prototype.headCount = 1

        P.prototype.eat = function () {

            console.log('eating...')

        }

        // 如果需要继承父类的实例属性，需要这样写

        function C (name, age, title) {

            P.apply(this, arguments)

        }

        function createEx (Child, Parent) {

            function F () {}

            F.prototype = Parent.prototype

            Child.prototype = new F()

            Child.prototype.constructor = Child

        }

        createEx(C, P)

        var c1 = new C('jojo', 22)

        console.log(c1.name) //jojo 证明P的实例属性已继承

        c1.eat() //eating... 证明P的原型方法已继承

    </script>