**面向对象**

1. 基础

function Parent() {}

Parent.prototype = {

p1: function(){}

}

1、构造函数的继承：

(1) 对象冒充

function Child() {

this.temp = Parent;

this.temp()

delete this.temp;

}

(2) call

function Child () { Parent.call(this, arguments) }

2、原型链的继承

考虑一：在继承了Parent.prototype的同时，也继承了this。会执行构造函数

Child.prototype = new Parent ();

但是当Parent的构造函数需要传入参数时，由于无法通过new Child()传入，该方法不适用。如果添加call方法传入参数，则会出现两次Parent构造函数的调用

考虑二：只继承了Parent.prototype。可以通过call继承this

Child.prototype = Parent.prototype;

该方法在修改Child.prototype的时候，也会修改Parent.prototype。所以不适用。

完整继承：

1. 生成与方法一类似的原型链对象结构，但没有继承构造函数中的this。

可以通过 if(Object.\_\_proto\_\_) 判断浏览器是否支持该属性

Child.prototype = { \_\_proto\_\_: Parent.prototype }; // IE11以下均不支持。

或

Child.prototype = Object.create(Parent.prototype); // IE9以下不支持。

兼容方法：

function Cort(){}

Cort.prototype = Parent.prototype;

Child.prototype = new Cort;

1. 通过call方法将Parent的构造函数引入Child

3、其他

function Parent(){console.log(this.className)} // className在Child的定义

1. 类当中，this的优先级高于prototype。如下：

function Child(){this.className ='C1'}

Child.prototype = Object.create(Parent.prototype);  
Child.prototype.className ='C2'

// function Child(){Parent.call(this); this.className ='C1';} 时，var c = new Child返回 C2

// function Child(){this.className ='C1'; Parent.call(this);} 时，var c = new Child返回 C1

// c.className返回 C

1. 原型的赋值

// 该方式在\_\_proto\_\_中会包含 construction属性

Parent.prototype.show = function(){}

// 该方式不会包含 construction 属性，

// 需要通过Parent.prototype.constructor = Parent 显式添加

Parent.prototype = {  
 show: function(){}  
}

3、静态属性的继承

for (var p in Parent){

if(Parent.hasOwnProperty(p) && p !== 'prototype'){

Child[p] = Parent[p]

}

}

4、通过instanceof判断构造函数

var c = new Child;

在继承了父对象原型链的情况下（以上原型链继承的任意方式），

c instanceof Parent 返回 true

5、constructor 返回实例的构造函数

没有继承的情况下，c.constructor返回 function Child(){}

在继承了父对象原型链的情况下，c.constructor 返回 function Parent(){}

可通过constructor获取构造函数，调用构造函数的静态属性

1. 进阶

实现一个class类

功能：通过属性的方式实现对父类的继承

1. es6对象定义与继承