继承

1.ES5实现继承

大多OO语言都支持两种继承方式： 接口继承和实现继承 ， 而ECMAScript中无法实现接口继承，ECMAScript只支持实现继承，而且其实现继承主要是依靠原型链来实现，下文给大家技术js实现继承的六种方式方法总结:

原型继承，构造函数继承，call/apply继承, 冒充对象继承, 实例继承, 组合继承，__proto__继承

1) 原型继承:<SonClass.prototype = new FatherClass(param1, param2,..))>

```
var A = function (x, y) {
  var num = 10;  //函数作为普通函数执行，私有变量
  this.x = x;  //实例对象增加的私有属性
  this.y = y;
  this.getX = function() {    //实例私有的方法
    console.log(this.x);
 };
};
A.prototype.getX = function () {   //原型上共有的方法
  console.log(this.x);
};
 A.prototype.write = function() {
   console.log('write-js');
 };
A.tools = {       //函数最为对象使用的，对象.属性 = 属性名
  getName: function() {
  },
 x: 10
}
```

```
var B = function (x, y) {
  this.x = x;
  this.y = y;
};
 B.prototype = new A(10, 20);   // ==>> 这两行代码的位置不能换了，因为应用地址的问题
 B.prototype.constructor = B;  //***因为你把天生的prototype对象替换了，所以constructr你必须手动指向他本身，否则就指向了A的constructor
 B.prototype.getY = function () {
    console.log(this.y);
};



var b1 = new B(1, 2);
var b2 = new B('JS','CSS');
console.log(b1);
console.log(b1 instanceof B); //true 是子类的实例
console.log(b1 instanceof A); //true  是父类的实例
console.log(b1.hasOwnProperty('getX'));  //false 不是私有的属性(原型上)
console.log(b1.hasOwnProperty('getY'));  //false   不是私有的属性(原型上)
console.log(B.prototype.constructor === B);   //true
console.log(b1.__proto__.__proto__.getX === A.prototype.getX);  //true
console.log(new B().__proto__ instanceof A);  true
 // 实例，和实例的原型(也就是子类的原型都是父类的一个实例)
// 先找到自己类的原型,在通过自己类原型的.__proto__指向父类的原型,因为 B.prototype = new A() ， B的原型就是A的一个实例么
//b1: {x: 1, y: 2}
```

```
//   __proto__  B.prototype  {x: 10, y: 20, getY: function(){} }
//   __proto__  A.prototype {getX: function(){} }


b1.getX = function () {     //这也是在b1的内存空间增加的私有方法。和b2无关
console.log('getX-prtive');
};

b1.__proto__.getX = function () {   //这个是在b1所属类的原型上增加的方法，是共享的和，所以和b2有关系
console.log('getX-common')
};

b2.getX(); // getX-common
console.log(b1.getX === b2.getX);  //b1的私有方法getX和b2共有属性方法比较，<原型链机制查找>
console.log(b1.__proto__.getX === b2.getX); //b1和b2的getX都是共有的方法

b1.name = 'clh';  //给自己开辟的空间增加的私有属性 (***)
console.log(b2.name);

b1.__proto__.printX = function () {
console.log(this.x);
};

b2.printX();  //JS   上面那条代码在b1的类的原型上增加了一个printX方法，这里b2肯定会访问到，b2执行这个函数，this肯定是b2,所以 b2.x = JS
```

```
b1.__proto__.__proto__.init = function () {  //这是在给父类A的原型上增加一个init方法
console.log('init-function');
};

console.log(b1.init === b2.init);  //true
console.log(b1.init === A.prototype.init);  //true


b1.__proto__.__proto__.change= function() {  //在父类A的原型上增加一个方法
    alert('chagne-getX');
};

b2.change(); //弹出 'change-getX' 调用父类A的原型的方法

b1.__proto__.__proto__.write = function() {   //*** 子类从写父类原型上的方法
    console.log('write-css');
};

b2.write();  // 'write-css'   原型链查找机制，找到父类A的原型上的write方法
```

特点：

核心： 拿父类实例来充当子类原型对象, (把父类的私有属性克隆一份，放到子类的原型上，父类的共有属性通过子类的原型上的__ptoto__查找到)

1. 非常纯粹的继承关系，实例是子类的实例，也是父类的实例
2. 父类新增原型方法/原型属性，子类都能访问到

3.简单，易于实现

缺点：

要想为子类新增属性和方法，必须要在new A()这样的语句之后执行，不能放到构造器中

无法实现多继承

来自原型对象的引用属性是所有实例共享的

创建子类实例时，无法向父类构造函数传参

2) 构造函数继承(call继承)(<A.call(this[,param1, param2,...])>

```
var A1 = function (name, age) {
  var num = 10;
  this.name = name;
  this.age = age;
};
A1.prototype.getName = function () {
  console.log('A-getName');
};

var A2 = function () {
  this.x = 10;
};
 A2.prototype.getX = function(() {
   console.log(this.x);
 };

var A3 = function () {
  this.y = 20;
```

```
};
 A3.prototype.getY = function(() {
    console.log(this.y);
 };

  var B1 = function (name, age) {
  A1.call(this, name, age);  //this是，B1类的一个实例对象， 把父类的私有属性性拷贝一份，放到这个实例的私有属性上
  A1.apply(this,arguments); //apply****继承
  A2.call(this);  //继承类A2的私有方法,放到B1实例对象的私有属性中
  A3.call(this);  //继承类A3的私有方法 ,放到B1实例对象的私有属性中
};

B1.prototype.getName = function () {
  console.log(this.name);
};
B1.prototype.getAge = function () {
   console.log(this.age);
};

var b1 = new B1('clh', 25);
console.log(b1);

console.log(b1 instanceof B1); //true  是子类的实例
console.log(b1 instanceof A1); // false 不是父类的实例
console.log(b1.getName === b1.__proto__.getName); //true
console.log(b1.getName === b1.__proto__.__proto__.getName); // false 后面那个是B1类原型上的.__prtot__， 那么原型是对象，所以是Object的实例，所以指向Object原型的getName，没有返回undefiend

 特点：
```

1. 解决了1中，子类实例共享父类引用属性的问题
2. 创建子类实例时，可以向父类传递参数
3. 可以实现多继承（call多个父类对象）

缺点：

1. 实例并不是父类的实例，只是子类的实例
2. 只能继承父类的实例属性和方法，不能继承原型属性/方法
3. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能*/

3) 充对象继承(拷贝继承)

```
var A2 = function (color, fontSize) {
    this.a = 10;
    this.b = 20;
    this.name = 'clh';
    this.getName = function () {
        console.log(this.getName);
    };
};

A2.prototype.printA = function () {
  console.log(('A2-prototype.printA'));
};

var A3 = function (x, y) {
  this.x = x
  this.y = y;
};
```

```
A3.prototype.getX = function () {
   console.log(this.x);
};


var B2 = function (cont) {
var objA = new A2('#ff0', '30px');  //自己创建一个对象，把父类的私有和共有属性/方法拿过来，进行遍历，放到子类的 私有属性中
var objB = new A3(1, 2);
for (var key in objA) {      //A2类
    this[key] = objA[key];  // this.prototype[key] = obj[key]  这是放到子类的共有属性中，把父类的私有和共有属性
}

for (var key in objB) {   //A3类
    this[key] = objB[key];
}


this.write = cont; //在增加自己传进来的属性
};

B2.prototype.write = function () {
  console.log('wait-JS');
};
B2.prototype.printA = function () {
   console.log('B2-prototype.printA');
};

var b2 = new B2('CSS+DIV');
console.log(b2);
```

```
console.log(b2 instanceof B2);   // true 是子类的实例
console.log(b2 instanceof  A2);   // false 不是父类的实例
console.log(b2.hasOwnProperty('name')); //true 是私有属性
console.log(b2.hasOwnProperty('printA')); //true 是私有属性

b2.printA(); //  'A2-prototype.printA'  私有属性的printA()，也就是继承A2原型上的printA方法
b2.__proto__.printA(); //  'B2-prototype.printA'   直接找到B2原型上的printA方法

特点：

 1. 支持多继承
缺点：

1.  效率较低，内存占用高（因为要拷贝父类的属性）
2. 无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）*/
```

4) 实例继承

```
var A3 = function (name, color) {
  this.name = name;
  this.color = color;
  this.x = 1;
  this.y = [1,2,3];
  this.getY = function () {
    console.log('A-pritive-getY');
  }
```

```
};
A3.prototype.getY = function () {
   console.log('A-common-getY');
};

var B3 = function (name, color) {
  var obj = new A3(name ,color);
  obj.x = 'JS继承';
  return obj;
};

B3.prototype.getColor = function () {
  console.log('B-common-getColor');
};

var b3 = new B3('clh', 'red');
console.log(b3);
console.log(b3 instanceof B3);  //false  因为他不是B3的实例，所以访问不到B3原型上的属性和方法
console.log(b3 instanceof A3);  //true  是A3的实例
console.log(b3.getY === A3.prototype.getY); // false
console.log(b3.getY.__proto__.getY === A3.prototype.getY); //false
console.log(b3.getY === A3.prototype.getY); //false
console.log(b3.hasOwnProperty('getY')); //true
```

特点：

不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

实例是父类的实例，不是子类的实例
不支持多继承*/

5) 组合继承 < 原型链继承 + 构造函数继承(call继承)>

```
var Animate = function (name, color, age, food) {
   this.name = name;
   this.color = color;
   this. age = age;
   this.eat = function (food) {
       console.log(this.name + '喜欢吃' + food);
   };
  this.write = function() {    //私有方法 write
        console.log('wirte-js');
  };
};

Animate.prototype.write = function() {   //共有方法write
   console.log('write-css');
 }
Animate.prototype.sleep = function (sleep) {
    console.log(this.name + '喜欢在' + sleep + '睡觉');
};
Animate.prototype.running  = function (runMethod) {
   console.log(this.name + '跑的方法' + runMethod);
};
```

```
var Dog = function (name, color, age, food) {
   Animate.call(this, name, color, age, food);  //构造函数继承父类的私有属性和方法
};

Dog.prototype = new Animate(); //继承父类的私有属性和私有方法放到这个类的原型上
Dog.prototype.constructor = Dog;  //强制constructor指向自己，否则原型链会混乱了
Dog.prototype.play = function (plays) {
    console.log(this.name + '喜欢玩' + plays);
};
Dog.prototype.write = function() {
     console.log('write-html');
};

var dog1 = new Dog('小狗', 'red', 23, '骨头');
console.log(dog1);
console.log(dog1 instanceof Dog); //true  是子类的一个实例
console.log(dog1 instanceof  Animate);  //true 是父类的一个实例

console.log(dog1.sleep === Animate.prototype.sleep); //true 第一个找到自己原型上的sleep方法，也就是原型继承过来的父类的原型sleep, 第二个是父类原型上的sleep方法
console.log(dog1.__proto__.__proto__.sleep === Animate.prototype.sleep); //true    自己子类原型.__prtoto__指向父类的prototype  所以为true
dog1.sleep('爬在地上');
dog1.running('四条腿');
```

```
dog1.sleep = function() {   //修改自己类原型上的sleep方法
        alert('ok');
};

dog1.sleep(); // 'ok'

dog1.__proto__.__proto__.sleep = function() {   //修改父类原型上的sleep方法
        alert('chagne-sleep-method');
};
var  animate1 = new Animate();
animate1.sleep();   // 'chagne-sleep-method'
dog1.__proto__.__proto__.sleep(); //'chagne-sleep-method'

dog1.write ();  //  ‘  write-js' 私有属性的write方法
dog1.__proto__.write();   //’ write-html'  共有属性子类原型上的write方法, 这里因为你首先进行了原型继承，子类原型上有一个write方法，你又给这原型对象增加了一个write方法，他肯定会把继承过来的write方法给覆盖了，所以输出结果为'write-html'
dog1.__proto__.__proto__.write();   //  ‘  write-css  ‘  父类原型上的write方法
```

特点：

1. 弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法
2. 既是子类的实例，也是父类的实例
3. 不存在引用属性共享问题
4. 可传参
5. 函数可复用

6. 可以实现多继承(call)

缺点:

1. 调用了两次父类构造函数，生成了两份实例(子类实例将子类原型上的那份屏蔽了)

6) 寄生组合继承(< 原型链继承 + 构造函数继承(call继承)) (利用空对象作为中介)

```
var AClass = function (name, age) {
   this.name = name;
   this.age =age;
   this.getAge = function () {
       console.log(this.age);
   };
   this.getName = function () {

    };
};

AClass.prototype.getName =function () {
    console.log(this.name);
};


var BClass = function (name, age) {
    AClass.call(this, name, age);
   this.height = '180cm';
};
```

```
// 其实这样做的目的是原型只继承父类原型上的东西
function extend(Child, Parent) {
    var F = function(){}; //空对象
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;   //为子对象设一个uber属性，这个属性直接指向父对象的prototype属性，这等于在子对象上打开一条通道，可以直接调用父对象的方法。这一行放在这里，只是为了实现继承的完备性，纯属备用性质
}
extend(BClass, AClass);
BClass.prototype.getHeight = function () {

};

var bclass1 = new BClass('clh' ,25);
var bclass2 = new BClass('wd' ,23);
console.log(bclass1);
console.log(bclass1 instanceof BClass);  //true 是子类的实例
console.log(bclass1 instanceof AClass);  //true  是父类的实例


console.log(bclass1.getHeight === bclass2.getHeight);  // true都是子类原型上的getHeight
console.log(bclass1.getName === AClass.prototype.getName); // false 第一个是自己类上原型上的getName,  第二个是父类原型上的getName ，肯定不一样
```

console.log(bclass1.__proto__.__proto__.getName === AClass.prototype.getName); //true 都是父类原型上的getName方法

特点:

1. 堪称完美

缺点:

1. 实现较为复杂

7) __proto__继承 <arguments.__proto__ = Array.prototype>

```
function sum() {
    console.log(arguments instanceof  Array); //false, 不是数组，不可以使用数组提供的方法
    arguments.__proto__ = Array.prototype; //在中间加了一层，强制将__proto__指向了数组的原型
    console.log(arguments instanceof Array);  // true  在数组了，可以用数组的方法
    console.log(arguments.slice()); // [1,2,3,1] 克隆一份数组
}

sum(1,2,3,1);
```

// ==>>ES6 里面的继承

A1) super在子类的构造函数里面只能使用一次

B1） 刚进入构造函数，是没有this的，必须去super调用父类，进行返回子类的实例，在进行包装实例

Super关键字的使用：

1） 子类继承父类，子类的构造函数里使用super

2） 在子类的原型方法上，使用super，super指代父类的原型

3） 在子类的静态方法上，使用super，super指代父类

```
class A {
        constructor(x, y, z) {
            Object.assign(this, {x, y, z});
        }

        getValue() {
            console.log(x, y, z);
        }

        static box() {
            alert('father-static-box');
        }
}

class B extends A {
        constructor(x, y) {
            //console.log(this); // ==>> 子类里面this，当你用super调用弗雷的时候，才返回的是子类的一个实例，才可以使用this,Must call super constructor in derived class before accessing 'this' or returning from derived constructor
                // ==>>刚进入构造函数，是没有this的，必须去super调用父类
            console.log(super(x, y) instanceof A);
```

```
            //console.log(super(x, y) instanceof B);  //
==>>而且super函数只能调用一次,第二次报错  -->> Super
constructor may only be called once
            this.name = 'clh';
            this.age = 25;
        }

        getValueInfo() {
            console.log(this.name, this.age);
        }

        static  box() {
            alert('son-static-box');
        }
    }

    console.log(new B(1, 2) instanceof B);
    console.log(new B(1, 2) instanceof A);
    console.log(new B(1, 2).__proto__ instanceof A);
```