

class

class

Class定义类和使用基本知识

1. 声明类的方法

- 1) class关键字声明类(函数声明)
- 2) class表达式声明类(函数表达式声明)

函数声明的两种方法区别，类声明两种方法的区别？

没有声明提升和挂在全局对象window上[没有预解释，声明的变量没有挂在window对象上，不污染window对象属性和方法]

2. 在类上曾加方法

1) constructor 构造函数 new的时候运行 this.xxx=xxx; constructor必须有一个，不写得话，JS引擎默认会加一个，它相当于 <私有属性和方法>

2) 在 class {} 定义的方法都是在类的原型上定义的 类的内部定义的方法，都是不可枚举的 ES5可以 Object.keys(object.prototype) <共有方法>

3) 使用static给类定义静态方法(也就是ES5将函数当做对象使用) 类.方法()
<静态方法>

给类一次增加多个方法使用 Object.assign(Object.prototype, {}); 类似于ES5 A.prototype = {};

4) 私有的变量(#变量名)和私有方法 (暂时没有提供,待ES7增加)\

注意: Class 内部只有"静态方法，没有静态属性" 静态属性放在外面定义 A.password = 1000; ES7可以用static password = 1000;来，定义在里面

Class 的静态属性和实例属性

3. 使用类

1) 必须使用new关键字，ES5可以不使用,this问题(不适用new关键字，里面的this为window对象，相当于给window对象增加了属性和方法，加了new就是当前类的某个实例，ES6严格使用new，否则报错)

2) Class 的取值函数 (getter) 和存值函数 (setter)

注意:

1) constructor 构造函数只能有一个，默认不写得话，浏览器自动加一个

2) 类采用简写方法，所以不需要使用function关键字了，但是类本质还没有改变，是一个函数，有prototype和__proto__属性

3) 类声明方式不会被提升，这与函数定义不同，不能再上面执行，没有预解释一说了

4) 类声明中所有的代码会在严格模式下运行，并且也无法跳出严格模式

5) 类的所有方法都是不可枚举的，遍历不到，而ES5自定义属性却可以遍历到，ES5可以采用`Object.defineProperty()`才能将方法改为不可枚举的

```
for (var key in person2) {  
    if (Object.hasOwnProperty(key)) { // ==>>过滤共有的  
        console.log(key)  
    }  
}
```

6) 调用类构造器必须使用`new`关键字，否则报错 ES5可以 `this`的问题

Class constructor PersonClass cannot be invoked without 'new', 防止当做函数来使用

7) 类的方法内重写类名，是不可以的 `const`定义的，常量了 可以理解外面的类采用`let`定义，内部采用`const`定义的

```
class PersonClass {  
    constructor (id, name) {  
        this.id = id;  
        this.name = name;  
    }  
  
    sayName () {  
        PersonClass = {}; // =>Uncaught TypeError:  
        Assignment to constant variable.  
        console.log(this.name);  
    }  
  
    setName (name) {  
        this.name = name;  
    }  
}
```

```
let person1 = new PersonClass(200, 'clh');  
person1.sayName();
```

8) 立即执行类

```
let person = new class {  
    constructor(name) {  
        this.name = name;  
    }  
}('常连海');
```

```
// ==>>PersonClass2类的内部使用， 外部使用PersonClass
let PersonClass = class PersonClass2 {
  constructor(name) {
    this.name = name;
  }
}
```

9) 类的静态方法static，类的实例方法，

10) 继承 super关键字必须一个，不能多个，

不写浏览器默认加一个，类似于constructor一样

```
class Animate {
  constructor(name) {
    this.name = name;
  }
  eat(food) {
    this.eat = function () {
      console.log('他喜欢吃' + this.food);
    }
  }
  static x(name) {
    alert(name)
  }
}

class Dog extends Animate {
  constructor(name, food, color) {
    super(name); // ==>>必须是一个，不能多个，调用一次，返回父类的实例，在进行加工
    this.food = food;
    this.color = color;
  }
  getColor() {
    console.log(this.color);
  }
}

let dog1 = new Dog('哈巴狗', '苹果', '白色的');
Dog.x(100); // ==>> Dog.__proto__ === Animate  Animate.x(100)
console.log(Dog.__proto__ === Animate); // ==>> true
console.log(Dog.__proto__ === Animate.prototype); // ==>> false  ES5为true
```

Super牢记几点

1) 只能在派生类中使用super()，若在非派生的类中使用，即，没有使用extends关键字的类活函数中，直接报错

2) 在构造函数中，你必须在访问this之前调用super()，由于super负责初始化this，因此必须在子类中首先调用父类，super返回父类的一个实例，在进行加工处理（父类的静态方法也可以继承过来）原型链去查找父类的静态方法了 ES5可继承不过来

3) super三用法， 子类构造函数中调用父类， 在子类的静态方法中代表父类， 在子类的原型方法上指代父类的原型

11) ES6类只有静态方法，没有静态属性，只能在类的外面自己写

继承

```

class A {
  constructor() {
    console.log(this); // ==> B的一个实例
    console.log(this instanceof A); // ==> true
    console.log(this instanceof B); // ==> true
    console.log(this instanceof Object); // ==> true
  }
}

class B extends A {
  constructor() {
    super(); // ==> 类似于 A.call(this);
  }
}

new B();

```

B.__proto__ === A; // ==> true

---->> ES5继承:

1. 原型继承 (SonClass.prototype = new FatherClass())
2. call/apply继承 (A.call(this[param1, param2,...]))
3. 冒充对象继承(拷贝对象继承)
4. 实例继承
5. 组合继承(原型继承 + 构造函数继承)
6. 寄生式继承
7. __proto__ 继承 (arguments.__proto__ = Array.prototype)

---->> Class继承

1. 继承基本写法

class Sub extends Super {} 声明类继承

let Sub = class extends Super {} 类表达式继承

只能继承一个类，不能继承多个类

2. Object.getPrototypeOf(子类) 从子类上获取父类

Object.setPrototypeOf(object, prototype); 将一个指定的对象的原型设置为另一个对象或者null(既对象的[[Prototype]]内部属性).

```

Object.setPrototypeOf(obj, prototype) => {
  obj.__proto__ = prototype;
  return obj;
};

```

Object.create(prototype);

```

Object.create(prototype) => {
  var obj = {};
  obj.__proto__ = prototype;
}

```

```

    return obj;
  };

```

3. 必须在子类的构造函数里面调用super()函数， ---> 子类实例的构建，是基于对父类实例加工，只有super方法才能返回父类实例。

//在子类运行constructor之前，不存在this

eg:

```

let A = class {

};

let B = class extends A {
  console.log(this);
  constructor() {
    super(); //====> A.constructor.call(B); 返回B的一个实例 <构造函数继承方法>
  };
};

```

4. Super关键字使用方法

//情况一: super作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次super函数。 给子类的实例增加私有的属性，从父类的构造函数中

//-->通过super调用父类的方法时，super会绑定子类的this。

//情况二: super作为对象时，在普通方法中，指向父类的原型对象； A.prototype 使用原型上的方法

//情况三: super作为对象时，在静态方法中，指向父类。 A本身， 使用静态方法和静态属性

```

class A {
  constructor() {
    console.log(this); // ==> B的一个实例
    console.log(this instanceof A); // ==> true
    console.log(this instanceof B); // ==> true
    console.log(this instanceof Object); // ==> true
  }

  getx() {
    alert('x');
  }

  static clh() {
    alert('x1');
  }
}

class B extends A {
  constructor() {
    super(); // 1.0 调用父类，并且把B的一个实例传过去，返回来在进行包装这个实例==> 类似于 A.call(this);
  }

  a() {
    super.getx(); // 2.0 super作为父类的原型去使用 ==> super.getx() == >> A.prototype.getx();
  }

  static clh() { // 3.0 super作为父类使用，使用父类的静态方法 ==> super.getx() == >> A.prototype.getx();
    alert('clh')
  }
}

B.clh(); // ==> B类的静态方法
B.__proto__.clh(); // ==> A.clh 父类A的静态方法
alert(B.__proto__ == A); // ==> true 子类的__proto__指向父类，于ES5不同，ES5指向父类的prototype

```

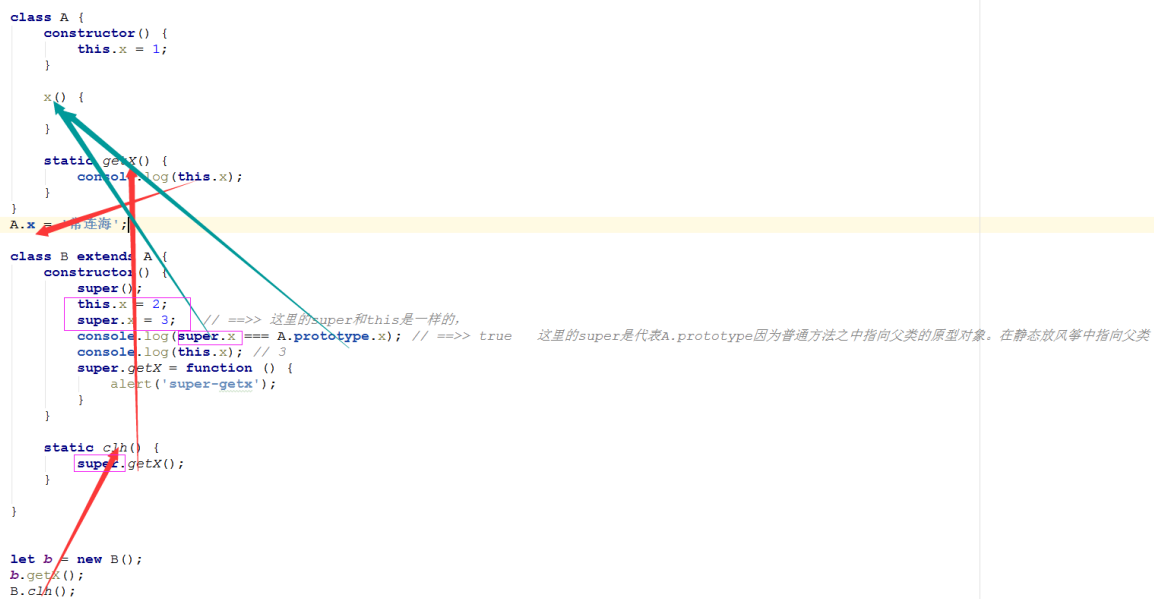
ES6 规定，通过super调用父类的方法时，方法内部的this指向子类。

// ==>> ES6 规定, 通过super调用父类的方法时, 方法内部的this指向子类。

```
class A {
  constructor() {
    this.x = 1;
  }
  print() {
    console.log(this.x);
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
  }
  m() {
    super.print(); // ==>> super.print.call(this);
  }
}

let b = new B();
b.m() // 2
```



```
class A {
  constructor() {
    this.x = 1;
  }
  x() {
  }
  static getX() {
    console.log(this.x);
  }
}
A.x = 1; 请连接';

class B extends A {
  constructor() {
    super();
    this.x = 2;
    super.x = 3; // ==>> 这里的super和this是一样的,
    console.log(super.x) === A.prototype.x; // ==>> true 这里的super是代表A.prototype因为普通方法之中指向父类的原型对象。在静态方法等中指向父类
    console.log(this.x); // 3
    super.getX = function () {
      alert('super-getx');
    }
  }
  static ch() {
    super.getX();
  }
}

let b = new B();
b.getX();
B.ch();
```

(1) 子类的 `__proto__` 属性，表示构造函数的继承，总是指向父类。

(2) 子类 `prototype` 属性的 `__proto__` 属性，表示方法的继承，总是指向父类的 `prototype` 属性。

```
class A {  
}  
  
class B extends A {  
}  
  
B.__proto__ === A // true  
B.prototype.__proto__ === A.prototype // true
```

上面代码中，子类 `B` 的 `__proto__` 属性指向父类 `A`，子类 `B` 的 `prototype` 属性的 `__proto__` 属性指向父类 `A` 的 `prototype` 属性。

```
function A() {  
}  
A.prototype.getX = function () {  
};  
  
function B() {  
}  
  
B.prototype = new A();  
B.prototype.constructor = B;  
  
console.log(B.__proto__ === A); // ==> false  
console.log(B.prototype.__proto__ === A.prototype); // ==> true
```

以上对比一下：

5. 类的 `prototype` 和 `__proto__`

ES6 继承：

(1) 子类的 `__proto__` 属性，表示构造函数的继承，总是指向父类。(***) <<就他不同>>

`B.__proto__ === A`

(2) 子类 `prototype` 属性的 `__proto__` 属性，表示方法的继承，总是指向父类的 `prototype` 属性。

`B.prototype.__proto__ === A.prototype`

(3) 子类和父类的实例都是父类的一个实例

`new A ()/ B() instanceof`

`A;`

ES5 继承：

(1) 子类的 `__proto__` 属性，指向函数类的原型

`B.__proto__ ===`

`Function.prototype`

(2) 子类 `prototype` 属性的 `__proto__` 属性，表示方法的继承，总是指向父类的 `prototype` 属性。

`B.prototype.__proto__ === A.prototype`

(3) 子类和父类的实例都是父类的一个实例

`new A ()/ B() instanceof`

`A;`

6. Extends关键字继承目标

- (1) 继承内置类的方法和属性 (内置类Array, Object,String,...)
- (2) 第二种特殊情况, 不存在任何继承。(和函数一样的prototype,__proto__)
- (3) 第三种特殊情况, 子类继承null。

//实例的__proto__

实例.__proto__.__proto__ === 父类.prototype

```
let Sub = class {};
```

```
let SuPer = class {};
```

```
let sub1 = new Sub();
```

```
let super1 = new SuPer();
```

```
sub1.__proto__.__proto__ === super1.__proto__ === SuPer.prototype; //---->> true
```

7.类的prototype和__proto__详解

//1. 每个对象都有一个属性 __proto__ (指向所属类的原型)

//2. 每个函数都有两个属性 prototype(函数原型对象) __proto__(指向跟函数原型)

函数原型对象里面有一个方法constructor() 指向当前函数本身

//在ES6中类里面

// (1) 子类的__proto__属性, 表示构造函数的继承, 总是指向父类。

//

// (2) 子类prototype属性的__proto__属性, 表示方法的继承, 总是指向父类的prototype属性。

// ES6 <<继承>>

```
class A {
```

```
}
```

```
class B extends A {
```

```
}
```



```
//      B.prototype = new A();

console.log( Object.getPrototypeOf(B) === A);
console.log(B.__proto__ === A);    //true
console.log(B.prototype.__proto__ === A.prototype);    //true    原型继承
console.log(B.__proto__ === Function.prototype);    //true    原型继承
console.log(new B() instanceof B);
console.log(new B() instanceof A);

//ES5里面    <<继承>>
let A1 = function () {

};

let B1 = function () {

};

B1.prototype = new A1();
B1.prototype.constructor = B1;
console.log(B1.__proto__ === A);    //false
console.log(B1.__proto__ === Function.prototype);    //false
console.log(B1.prototype.__proto__ === A1.prototype);
console.log(new B1() instanceof B1);
console.log(new B1() instanceof A1);
```

8. 继承案例:

```
//类怎么在他的原型上定义属性, 例如ES5中      A.prototype.password = pwd;
// ES5 的继承, 实质是先创造子类的实例对象this, 然后再将父类的方法添加到this上面
(Parent.apply(this))。    原型继承
// ES6 的继承机制完全不同, 实质是先创造父类的实例对象this (所以必须先调用super方法), 然后再用子类的构造函数修改this。    call/apply 构造函数继承
class A {
  constructor(x, y) {
    console.log(this); //--> A B
    this.x = x;
    this.y = y;
  };

  getX() {
    console.log(this.x);
  };
};
```

```

    getY() {
        console.log(this.y);
    };

    static printX() {
        console.log('A-printX');
    };
}
let a1 = new A(10, 20);

class B extends A {
    constructor(x, y) {
        //console.log(this); //----> Uncaught ReferenceError: this is
not defined    子类里面没有父类
        // console.log(super(x, y)); //---->返回B 的一个实例
        super(x, y); //---->返回B 的一个实例 调用A方法，并且A方法里面的this为
B的一个实例 ====> A.call(this, x, y) | A.apply(this, param);
        this.z = 10;
    };

    printZ() {

    };
}
let b1 = new B(1, 2);    //---->如果子类的constructor函数里面没有调用super方
法，会报错，找不到 this ----> this is not defined
b1.__proto__.__proto__ === A.prototype //-->true
b1.__proto__.__proto__.getX === A.prototype.getX //true 共享，原型链到了
//new B() instanceof B    //---> true
//new B() instanceof A    //---> true
b1.constructor === B    //---> true

console.clear();
A.printX === B.printX //--> false 不是共享静态方法，而是克隆了一份了， 所以你改
子类的静态方法不会影响父类的方法（同一个方法）
B.printX(); //讲父类的静态方法也继承过来了
B.printX = function () {
    console.log('B-printX');
};
A.printX();
B.printX();

```

9. 类的总结和案例：

```
//ES6里面的类
//1.类的声明方法  A:类方式声明  B:类表达式声明
//2.原型处理
//3.类的调用(必须使用new关键字)
```

//考虑到未来所有的代码，其实都是运行在模块之中，所以 ES6 实际上把整个语言升级到了严格模式。

//类和模块的内部，默认就是严格模式，所以不需要使用use strict指定运行模式。

//函数方法声明类

```
class Person {
    constructor(name, age) {    //构造方法,
        this.name = name;
        this.age = age;
    };

    //方法都定义在类的原型上
    toString() {
        console.log('我的名字是' + this.name + '年龄是' + this.age);
    };

    getName() {
        console.log(this.name);
    };

    getAge() {
        console.log(this.age);
    };
}

let person1 = new Person('常连海', 25);
console.dir(person1);
```

//如果添加多个方法可以使用 Object.assign方法可以很方便地一次向类添加多个方法。 类
类似于 A.prototype = {constructor: A,x: function(){}, y: function(){}}

//里面不能使用箭头函数

// Object.keys

// Object.assign

//Object.getPrototypeOf 获取对象上原型的方法，一一列出来

```
class A {
    constructor() {
```

```

        };
    }
    Object.assign(A.prototype, {
        x() {

        },
        y() {

        },
        z() {

        }
    });

    let a1 = new A();
    console.dir(a1);

```

//1.构造方法 constructor --->>constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

```

//    class B {
//        // 不写等价于下面默认增加了一个constructor
//        //定义了一个空的类B，JavaScript 引擎会自动为它添加一个空的constructor
//    }
//    class B {
//        constructor() {
//
//        }
//    }

```

//2.类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

```

class C {
    a() {

    };

    b() {

    }
}

```

```

new C();
//C(); //报错 -->> Class constructor C cannot be invoked without 'new'
console.log(Object.getPrototypeOf(new C()));

//函数声明和Class类声明对比
function A1() {

}

class A2 {

}

//函数表达式声明函数和 Class 表达式对比
let fn = function a() {
    console.log(2);
    //a在函数内部使用 a()
    // a 和 fn指向同一个内存地址
    // fn.name ---->> 'a' 并不是fn
};

fn();

let CreatePerson = class A {
    // A只在 Class 内部有定义。
    // 如果类的内部没用到的话,可以省略Me,也就是可以写成下面的形式。
    // CreatePerson.name ---->> 'A' 并不是 'CreatePerson'
};

//没有关键字后面的名称
let D1 = function () {

};

let D2 = class {

};

//函数声明和函数表达式声明的区别
//Class类声明和Class 表达式区别

//--->> 采用 Class 表达式,可以写出立即执行的 Class。
console.clear();

```

```

let card = new class Card {
  constructor(name, id) {
    this.name = name;
    this.id = id;
  };

  getCardName() {
    console.log(this.name);
  };
}('一级卡片', '11010901');
console.dir(card); //自执行函数返回一个Card实例

```

//类不存在变量提升 (hoist)

//函数声明存在 hoist 预解释

//这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

```

//      new F();//F is not defined
//      class F {
//
//      }

```

//函数声明存在 hoist 预解释

```

//      F();
//      function F() {
//      console.log('f');
//      }

```

//****注意

//类的私有方法 ， 也就是函数里面的函数

//类的私有属性， 也就是函数里的变量，函数的三角色中的普通函数 var x = 10;

```

class G {
  constructor() {
    this.x = () => {
      console.log(this); //G -->> {x: function, y: function,
z: function}
    };
    this.y = function () {
      return () => {
        console.log(this); //G -->>{x: function, y: function,
z: function}
      }
    }
  }
}

```

```
};
```

```
this.z = function () {  
    return function () {  
        console.log(this); //--> undefined    ??为什么不是window
```

对象

```
    }
```

```
};
```

```
};
```

```
//G.prototype原型上的方法
```

```
x() {
```

```
};
```

```
y() {
```

```
}
```

```
}
```

```
let g1 = new G();
```

```
console.dir(g1);
```

```
g1.x();
```

```
g1.y()();
```

```
g1.z()();
```

```
//name属性总是返回紧跟在class关键字后面的类名。
```

```
//ES5里面的东西， 类
```

```
console.clear();
```

```
var Father = function (name, age) {
```

```
    var x = 10;                //函数内私有变量
```

```
    var fn = function () {     //函数内私有函数
```

```
        // 只有普通函数运行，才可以访问到他
```

们，类是访问不到的，原型链查找机制，找不到他们

```
};
```

```
this.name = name;            //this.xx =xx私有的属性和方法 对象的私有属
```

性

```
this.age = age;
```

```
this.fn = function () {      //this.xx =function() {} 对象的私有方
```

法

```

    };

    console.log(x);
    //          return function () {
    //
    //          };    //--->>返回引用数据类型会冲掉当前这个类的实例
};

Father.prototype.x = function () {          // 类使用，原型 共有的属性和方法
    console.log(this.name);
};
Father.prototype.y = function () {

};
Father.box = {    //对象使用

};

var father1 = new Father('clh' , 25); //类使用
father1.x();

var father2 = Father();    //普通函数使用

```

```

console.clear();
let Animate = class {
    constructor() {

    }
    x() {

    }
    y() {

    }
};
Animate.aa = function () {

```

}; //这样不可以使用了，已经不是函数了，是类会报错，我们可以使用：加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```

let animate1 = new Animate();
console.log(typeof Animate);

```


//类的静态方法： 加上static关键字，就表示该方法不会被实例继承，而是<<<直接通过类来调用
>>>，这就称为“静态方法”。

//实例也不能使用静态方法，会报错 ES5也是一样

//父类的静态方法，可以被子类继承。

```
class H {
  constructor(name, age) {
    this.name = name;
    this.age = age;
    this.x = () => {
      console.log('pritive-x-method');
    };
    this.y = () => {

    };
  };
};

static x() {
  console.log('static-x-method');
};
static y() {

};

x() {
  console.log('prototype-x-method');
};
y() {

};
};

let h1 = new H();
console.dir(h1);
h1.x();
h1.__proto__.x();
H.x();
```

//父类的静态方法，可以被子类继承。----->> 在ES5中，静态方法是不能继承过来的

```
let Big = class {
  static x() {
    console.log('static-x');
  };
  static y() {
    console.log('static-y');
  };
};

class Letter extends Big {

}

Letter.x(); //--->> 'static-x'
Letter.y(); //--->> 'static-y'
```

10. 卡片类模拟：

```
class PlanClass {
  constructor(cardId, cardName, cardType, cardStatus, cardFatherId) {
    Object.assign(this, {cardId, cardName, cardType, cardStatus,
cardFatherId}); //--->> Object.assign给实例增加私有方法
  };

  //初始化页面交互和数据
  init() {
    let _this = this;
    //_this.initCard().updateCard().deleteCard(); //--->> 链式写法
    _this.initCard(); //--->>初始化卡片数据的
    _this.cardHandle(); //--->>卡片交互操作全部放在里面
  };

  initCard() {
    $.ajax({
      url: './card.json',
      type: 'get',
      dataType: 'json',
      success: function (data) {
```

```

        data = data.data;
        if (data.code === 200) {
            let initCardStr = '';
            $.each(data.cardList, function (index, item) {
                initCardStr +=
toolKit.template($('#cardInitTemplate').html()), {
                    cardId: item.cardId,
                    cardName: item.cardName,
                    cardType:
PlanClass.typeChangeText(item.cardType),
                    cardStatus:
PlanClass.statusChangeText(item.cardStatus),
                    cardFatherId: item.cardFatherId,
                });
            });
        }

        $('#.card').html(initCardStr);
    } else {
        console.log('卡片数据获取失败，稍后请重试');
    }
}

});
return this;
};

/**
 * updateCard: 更新卡片的某个属性,依据卡片的id来标识
 * @param {{string}} cardId: 卡片的id
 * @param {{string}} cardName: 卡片的名称
 * @param {{number}} cardType: 卡片的类型 1-->需求 2--> 任务
 * @param {{number}} cardStatus: 卡片的状态 1->新建 2-> 开发中 3-->验证中
4--> 已完成
 * @returns void;
 */
updateCard(cardId, cardName, cardType, cardStatus) {
    let data = {cardId, cardName, cardType, cardStatus};
    $.ajax({
        url: './test.json',
        type: 'post',
        data,
        dataType: 'json',
        success: function (data) {
            if (data.data.code === 200) {
                console.log('更新成功');
            }
        }
    });
}

```

```

        } else {
            console.log('更新失败');
        }
    }
});
return this;
};

/**
 * deleteCard: 删除某个卡片，依据卡片的id来标识
 * @param {{string}} cardId: 卡片的id
 */
deleteCard(cardId, delCardEle) {
    $.ajax({
        url: './test.json',
        type: 'post',
        data: cardId,
        dataType: 'json',
        success: function (data) {
            console.log(data);
            if (data.data.code === 200) {
                console.log('删除成功');
                delCardEle.remove();
            } else {
                console.log('删除失败');
            }
        }
    });
    return this;
};

```

//封装页面交互，在进行调用实例的原型的方法

```

cardHandle() {
    let _this = this; //类的实例

    $(document).off('click.delete').on('click.delete', '.del_card',
function (e) {
        let $that = $(this);
        let delCardId = $that.parents('.card_list').attr('data-id');
        let delCardEle = $that.parents('.card_list');
        if (confirm('确定要删除卡片吗')) {
            _this.deleteCard(delCardId, delCardEle);
        }
    }

```

```

    });
};

static statusChangeText(statusNum) {
    if (statusNum === 1) return '新建';
    if (statusNum === 2) return '开发中';
    if (statusNum === 3) return '验证中';
    if (statusNum === 4) return '已完成';
    return '';
};

static typeChangeText(typeNum) {
    if (typeNum === 1) return '需求';
    if (typeNum === 2) return '任务';
    return '';
};
}

planInstance.init();
console.dir(planInstance); //-->查看实例的详细信息
console.dir(PlanClass);    //--->查看这个类的详细信息

```

11. ES5继承分析

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：

原型链继承 、 构造函数继承(call继承)、冒充对象继承、实例继承、
组合继承、 寄生组合继承 、 __proto__ 继承法

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，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);
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
// 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__proto__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 (详细请看附录代码: 示例1)

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. 冒充对象继承 (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.__prtototo__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);
```

***再次总结Class类:

基础知识:

1. 定义类

```
function Fn(x, y, z) {
    var num = 10;
    this.x = x;
    this.y = y;
```



```
        this.z = z;
    }
    Fn.prototype = {
        constructor: Fn,
        getX: function() {
            console.log(this.x);
        },
        getY: function() {
            console.log(this.y);
        }
    };

    Fn.box.queryUrl = function() {};
    Fn.box.

    Fn.prototype.getX = function() {};
    Fn.prototype.getY = function() {};
```