面向对象-封装

一、POP与OOP

1、POP（面向过程编程）

面向过程编程(Procedure Oriented Programming)：以事件为中心，分析出解决问题的步骤，然后用函数将这些步骤一步步实现，使用的时候依次调用

2、OOP（面向对象编程）

面向对象编程(Object Oriented Programming)：以事物为中心，万物皆对象，由实体引发事件，更贴近现实世界，更易于扩展

- OOP达到了软件工程的三个目标：重用性、灵活性、扩展性
- OOP的三大特性：封装、继承、多态
- OOP的核心就是对象

类 / 继承描述了一种代码的组织结构形式：一种在软件中对真实世界中问题领域的建模方法。

面向对象编程强调的是数据和操作数据的行为本质上是互相关联的（当然，不同的数据有不同的行为），因此好的设计就是把数据以及和它相关的行为打包（或者说封装）起来。这在正式的计算机科学中有时被称为数据结构。

3、POP与OOP对比

	面向过程	**面向对象**
优点	性能比面向对象高，适合跟硬件联系很紧密的东西，例如单片机就采用的面向过程编程。	易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护
缺点	不易维护、不易复用、不易扩展	性能比面向过程低

二、对象

对象是由属性和方法组成的，指的是一个具体的事物

- 属性：事物的特征，在对象中用属性来表示（名词，可以用有形容）
- 方法：事物的行为，在对象中用方法来表示（常用动词，可以用能形容）

1、创建对象

在典型的OOP语言中（比如Java），都存在类的概念，类就是对象的抽象化，对象就是类的实例化。但是JavaScript中在ES6之前并没有引入类的概念。

所以，在ES6之前，对象并不是基于类创建的，而是通过以下三种方式来创建：

- new Object()
- 对象字面量
- 构造函数

1.1 Object的方式

语法：

```
var 对象名称 = new Object( );
对象.属性 = 值;
对象.方法 = function(){};
```

示例：

```
var girl = new Object();
girl.name = "喵";
girl.age = 20;
girl.say = function() {
    // 调用对象的属性和方法，在对象内部通过this调用（this在对象内部表示当前对象）
     console.log("我的名字叫" + this.name + ",年龄" + this.age);
}

// 调用对象的属性和方法，在对象外部通过对象名称调用
girl.say();
```

1.2 对象字面量方式

- 对象字面量是对象定义的一种简写形式，简化创建包含大量属性和方法的对象的过程

语法：

```
var 对象 = {
    属性: 值,
    方法: function(){}
}
```

示例：

```
var girl = {
    name: "喵",
    age: 20,
    say: function() {
        console.log("我的名字叫" + this.name + ",今年" + this.age);
    }
}

// 调用对象方法
girl.say();
```

1.3 构造函数方式

以上两种方法，当在需要创建大量属性和方法相同的对象时，就会有大量重复的代码：

```
var girl01 = {
    name: "喵",
    age: 20,
    say: function() {
        console.log("我的名字叫" + this.name + ",今年" + this.age);
    }
}
```

```
var girl02 = {
    name: "喵2",
    age: 18,
    say: function() {
        console.log("我的名字叫" + this.name + ",今年" + this.age);
    }
}

var girl03 = {
    name: "喵3",
    age: 28,
    say: function() {
        console.log("我的名字叫" + this.name + ",今年" + this.age);
    }
}
```

- 构造函数的方式，就可以对以上代码进行优化，解决代码冗余的问题，提高代码的重用性
- 构造函数名建议采用大驼峰命名法
- 构造函数是一种特殊的函数，我们可以把对象中公共的属性和方法抽取出来，封装到这个函数里面。
- 然后通过new来实例化生成对象，并且可以通过构造函数的参数给对象的属性赋值。

语法：

```
// 第一步：声明构造函数
function 函数名(参数1,参数2,...){
    this.属性1 = 参数1;
    this.属性2 = 参数2;
    this.方法 = function(){}
}

// 第二步：实例化对象
var 对象 = new 函数名(实参1，实参2);
```

举例：

```
// 声明构造函数
function Girl(name, age) {
    this.name = name;
    this.age = age;
    this.say = function() {
        console.log("我的名字叫" + this.name + ",今年" + this.age);
    }
}

// 实例化对象
var girl01 = new Girl("喵1", 20);
var girl02 = new Girl("喵2", 18);
var girl03 = new Girl("喵3", 28);

// 调用对象方法
girl01.say();
girl02.say();
girl03.say();
```

2、实例成员和静态成员

2.1 实例成员

- 实例成员就是构造函数内部通过this添加的成员
- 实例成员只能通过实例化的对象来访问

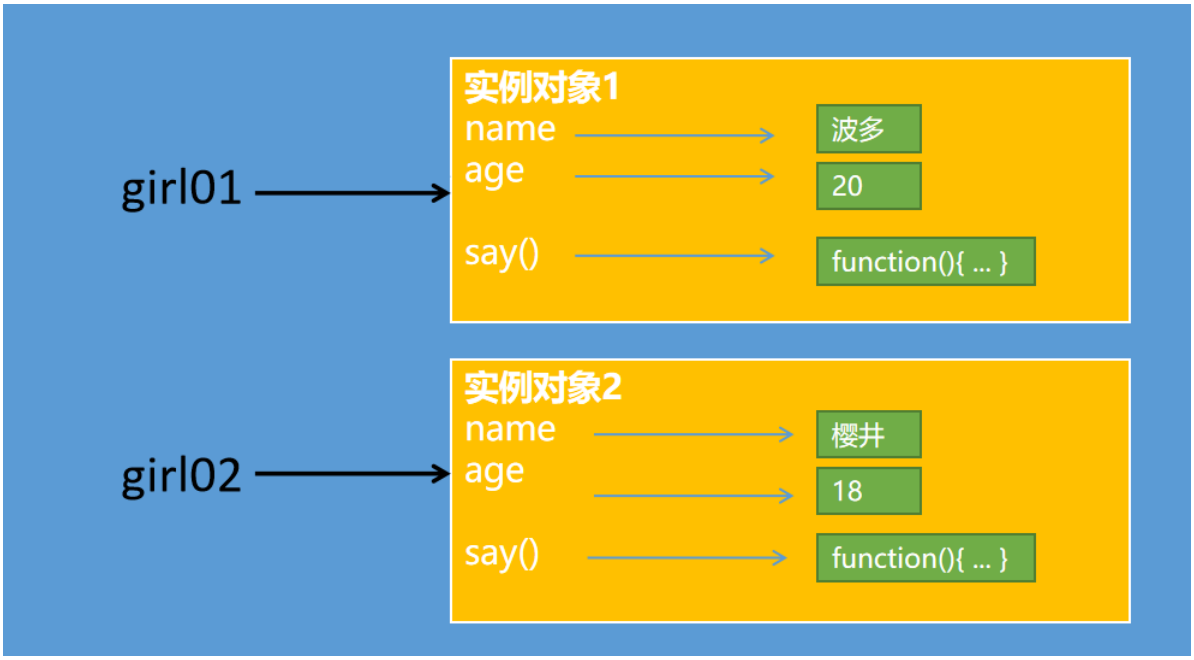
2.2 静态成员

- 静态成员就在构造函数本身上添加的成员
- 静态成员只能通过构造函数来访问，不能通过对象访问

3、原型对象

3.1 构造函数的问题

构造函数虽然很好用，但却存在内存浪费的问题。同一构造函数实例化出来的实例对象，都会给自己的成员方法分配内存空间，但实际上这些成员方法都是相同的，这样就造成了内存的浪费



3.2 prototype

JavaScript中规定，每个构造函数都有一个prototype属性指向一个对象，这个对象的所有属性和方法，都会被构造函数所拥有。

我们可以通过console.dir(函数名)来打印构造函数，查看其中的原型对象

```
▼ƒ Girl(name, age)
    arguments: null
    caller: null
    length: 2
    name: "Girl"
  ▶ prototype: {constructor: ƒ}
  ▶ __proto__: ƒ ()
    [[FunctionLocation]]: test2.html:35
  ▶ [[Scopes]]: Scopes[1]
```

- **prototype属性就是原型对象**
- **原型对象的所有属性和方法都会被构造函数的实例对象共享**

- 这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上，从而来节省内存空间

3.3 构造函数+prototype混合方式创建对象

语法：

```
// 第一步：声明构造函数
function 函数名(参数1,参数2,...){
    this.属性1 = 参数1;
    this.属性2 = 参数2;
}
// 第二步：将公共的方法定义在原型对象上
函数名.prototype.方法 = function(){}

// 第三步：实例化对象
var 对象 = new 函数名(实参1，实参2);
```

举例：

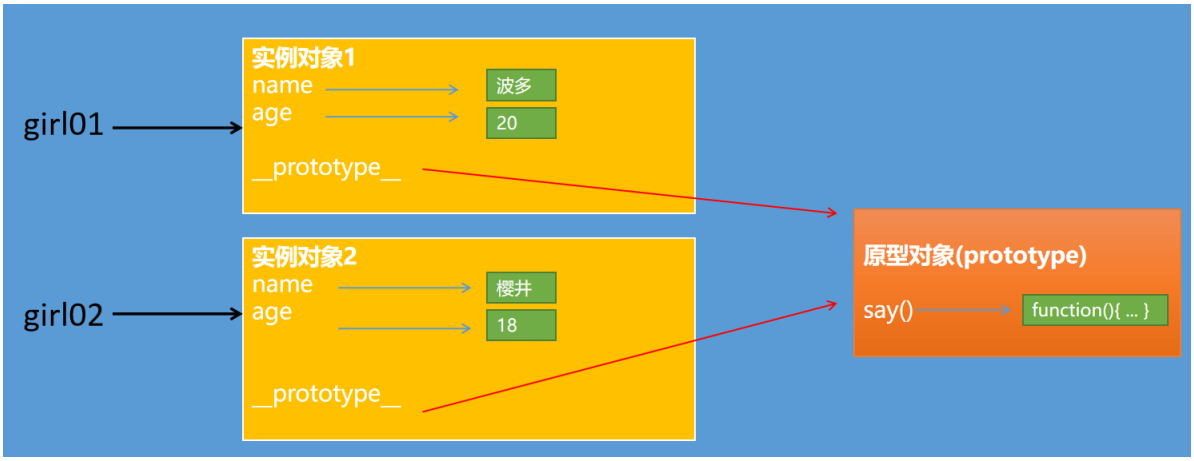
```
// 声明构造函数
function Girl(name, age) {
    this.name = name;
    this.age = age;
}

// 将公共的成员方法定义给原型对象
Girl.prototype.say = function() {
    console.log("我的名字叫" + this.name + ",今年" + this.age);
}

// 实例化对象
var girl01 = new Girl("喵1", 20);
var girl02 = new Girl("喵2", 18);
var girl03 = new Girl("喵3", 28);

// 调用对象方法
girl01.say();
girl02.say();
girl03.say();
```

> 问题：为什么每一个实例对象都可以共享原型对象中的属性和方法呢？
>
> 回答：因为每一个对象都有一个属性 proto___指向构造函数的prototype对象。



三、案例

3.1 创建Cookie对象

利用构造函数+prototype的方式创建Cookie对象，实现添加、获取、删除功能

```
// 创建Cookie对象，并将其挂载到window对象下
(function() {
     /**
      * 定义构造函数Cookie
      */
     function Cookie() {}

     /**
      * 设置cookie
      * @param {string} 名称
      * @param {mixed}  值
      * @param {number} 过期时间(单位：秒)
      */
     Cookie.prototype.setCookie = function(name, value, seconds) {
         if (seconds) {
             var date = new Date();
             date.setSeconds(date.getSeconds() + seconds);
             document.cookie = name + "=" + value + ";expires=" +
date.toUTCString();
         } else {
             document.cookie = name + "=" + value;
         }
     }

     /**
      * 根据名称获取cookie的值
      * @param {string} name 名称
      * @return {string} 值
      */
     Cookie.prototype.getCookie = function(name) {
         var cookies = document.cookie.split(";");

         for (var i = 0; i < cookies.length; i++) {
             var cookie = cookies[i].split("=");
             if (cookie[0].trim() == name) {
                 return cookie[1];
             }
         }
```

```
    }

    /**
     * 删除指定名称cookie
     * @param {string} 名称
     */
    Cookie.prototype.removeCookie = function(name) {
        var date = new Date();
        date.setSeconds(date.getSeconds() - 1);
        document.cookie = name + "=" + null + ";expires=" + date.toUTCString();
    }

    /**
     * 删除全部cookie
     */
    Cookie.prototype.clearCookie = function() {
        var cookies = document.cookie.split(";");

        for (var i = 0; i < cookies.length; i++) {
            var cookie = cookies[i].split("=");
            var date = new Date();
            date.setSeconds(date.getSeconds() - 1);
            document.cookie = cookie[0] + "=" + null + ";expires=" +
date.toUTCString();
        }
    }

    // 挂载到widow全局对象
    window.Cookie = Cookie;
})(window)

// 实例化Cookie对象
var cookie = new Cookie();

//调用Cookie对象中的方法来操作cookie
cookie.setCookie("username", "Rose2");
console.log(cookie.getCookie("username"));
```

3.2 创建Array对象

利用构造函数+prototype的方式创建Array对象，实现最大值、最小、去重功能

```
// 创建Array对象，并将其挂载到window对象下
(function() {
     /**
     * 定义构造函数Cookie
     */
    function Array(arr) {
        this.arr = arr;
    }

     /**
     * 求元素最大值
     */
    Array.prototype.max = function() {
        var max = this.arr[0];
```

```
        for (var i = 1; i < this.arr.length; i++) {
            if (max < this.arr[i]) {
                max = this.arr[i];
            }
        }
        return max;
    }

    /**
     * 求元素最小值
     */
    Array.prototype.min = function() {
        var min = this.arr[0];
        for (var i = 1; i < this.arr.length; i++) {
            if (min > this.arr[i]) {
                min = this.arr[i];
            }
        }
        return min;
    }

    /**
     * 对元素进行去重
     */
    Array.prototype.filter = function() {
        var newArr = [];
        for (var i = 1; i < this.arr.length; i++) {
            if (newArr.indexOf(this.arr[i]) == -1) {
                newArr.push(this.arr[i])
            }
        }
        return newArr;
    }

    // 挂载到window对象
    window.Array = Array;
})(window)

// 实例化对象
var array = new Array([10, 40, 50, 12, 10, 70, 12]);

// 调用对象方法
console.log(array.filter());
console.log(array.max());
console.log(array.min());
```

练习

一、以new Object()创建对象

二、以对象字面量方式创建对象

三、以构造函数方式创建对象

四、以构造函数+prototype方式创建对象

五、封装Cookie对象，实现cookie的添加、获取、删除、清除操作

六、封装Array对象，实现数组的最大值、最小值、去重、排序操作

面和对象-继承

继承是面向对象的三大特性之一。现实中有很多对象具有相似的特征，这时我们通常将这些共有的属性和方法定义在父对象中，然后子对象通过继承的特性来获取这些属性和方法。

比如：

```
// 人类
function Person() {}
Person.prototype.eys = 2;
Person.prototype.mouse = 1;
Person.prototype.walk = function() {
     console.log("我会走路")
}

// 如果我们需要定义中国人、美国人、韩国人这些对象时如何继承到人类对象的这些属性和方法呢？
```

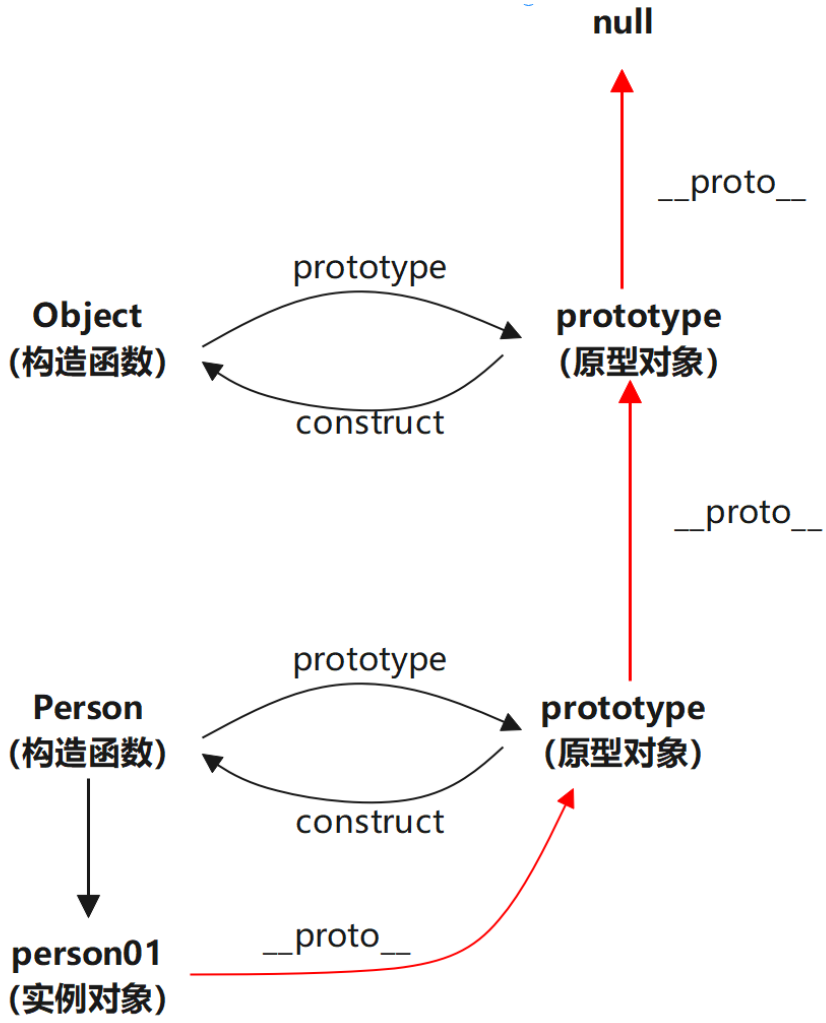
JavaScript继承的实现主要通过以下几种方式：

一、原型链继承

1、什么是原型链

在JavaScript中万物都是对象，对象和对象之间也有关系，并不是孤立存在的。对象之间的继承关系，在JavaScript中是通过prototype对象指向父类对象，直到指向Object对象为止，这样就形成了一个原型指向的链条，专业术语称之为原型链。

下图中红线所指即是原型链。



2、原型链继承（继承原型对象中的属性和方法）

ECMAScript 中将原型链作为实现继承的主要方法。其基本思想是利用原型让一个对象继承另一个对象的属性和方法

- 缺点：需要创建父对象实例

```
// 人类
function Person() {}
Person.prototype.name = "";
Person.prototype.legs = 2;
Person.prototype.mouse = 1;

Person.prototype.walk = function() {
    console.log(this.name + "有" + this.legs + "条腿，我会走路")
}

// 中国人
function Chinese(name) {
    this.name = name;
}
// 继承父对象
Chinese.prototype = new Person();
// 原型对象的构造方法指向当前构造方法
Chinese.prototype.constructor = Chinese;
```

```
// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

3、直接继承原型对象

- 优点：
  - 效率比较高（不用创建Person的实例了）
- 缺点：
  - Chinese.prototype和Person.prototype现在指向了同一个对象，任何对Chinese.prototype的修改，都会反映到Person.prototype

```
// 人类
function Person() {}
Person.prototype.name = "";
Person.prototype.legs = 2;
Person.prototype.mouse = 1;

Person.prototype.walk = function() {
    console.log(this.name + "有" + this.legs + "条腿，我会走路")
}

// 中国人
function Chinese(name) {
    this.name = name;
}
// 直接继承父级的原型对象
Chinese.prototype = Person.prototype;
// 原型对象的构造方法指向当前构造方法
Chinese.prototype.constructor = Chinese;

// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

4、利用空对象中转

- 虽然还是创建了对象实例，但由于空对象几乎不占内存，而且还解决修改原型对象属性对父对象的影响的问题

```
// 人类
function Person() {}
Person.prototype.name = "";
Person.prototype.legs = 2;
Person.prototype.mouse = 1;

Person.prototype.walk = function() {
     console.log(this.name + "有" + this.legs + "条腿，我会走路");
}
```

```
// 空对象
function NullFun() {};
NullFun.prototype = Person.prototype;

// 中国人
function Chinese(name) {
    this.name = name;
}
// 原型链继承
Chinese.prototype = new NullFun();
// 原型对象的构造方法指向当前构造方法
Chinese.prototype.constructor = Chinese;

// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

- 将继承封装成函数

```
// 封装继承函数
function extend(child, parent) {
    var fun = function() {};
    fun.prototype = parent.prototype;
    child.prototype = new fun();
    child.prototype.constructor = child;
}

// 人类
function Person() {}
Person.prototype.legs = 2;
Person.prototype.mouse = 1;

Person.prototype.walk = function() {
     console.log(this.name + "有" + this.legs + "条腿，我会走路")
}

// 中国人
function Chinese(name) {
    this.name = name;
}

// 继承
extend(Chinese, Person);
// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

5、拷贝继承

- 把父对象的所有属性和方法，拷贝进子对象
- 将父对象的prototype对象中的属性，一一拷贝给子对象的prototype对象

```
// 封装继承函数
function extend(Child, Parent) {
    var p = Parent.prototype;
    var c = Child.prototype;
    for (var i in p) {
        c[i] = p[i];
    }
}

// 人类
function Person() {};
Person.prototype.foot = 2;
Person.prototype.mouse = 1;

Person.prototype.walk = function() {
    console.log(this.name + "我有" + this.foot + "只脚，我会走路")
}

// 中国人
function Chinese(name) {
    this.name = name;
}

// 继承
extend(Chinese, Person);

// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

二、构造函数继承(继承实例对象中的属性和方法)

1、call()

语法:

```
函数.call(对象,参数1,参数2, ...)
```

- call()可以修改函数内部this的指向,使用call()的时候 参数1是修改后的this指向

```
function fn(x, y) {
    console.log(this);
    console.log(x + y);
}
var obj = {
    name: 'andy'
};
fn.call(obj, 1, 2); //调用了函数此时的this指向了对象obj,
```

2、apply()

语法：

```
函数.apply(对象,[参数1,参数2, ...])
```

- apply和call作用相同，就是传参方式不同

```
function fn(x, y) {
    console.log(this);
    console.log(x + y);
}
var obj = {
    name: 'andy'
};
fn.apply(obj, [1, 2]); //调用了函数此时的this指向了对象obj,
```

3、bind()

语法：

```
函数.bind(对象,参数1,参数2, ...)
```

- bind() 方法不会调用函数,但是能改变函数内部this 指向,返回的是原函数改变this之后产生的新函数
- 如果只是想改变 this 指向，并且不想调用这个函数的时候，可以使用bind

```
function fn(x, y) {
    console.log(this);
    console.log(x + y);
}
var obj = {
    name: 'andy'
};
fn.bind(obj, 1, 2); //没有调用函数此时的this指向了对象obj,
```

4、构造函数继承

- 构造函数继承的核心思想是：通过call()或apply()方法在子构造函数的内部调用父构造函数，从而获取到父对象中的属性和方法。

```
// 人类
function Person(name) {
    this.legs = 2;
    this.mouse = 1;
    this.name = name;
    this.walk = function() {
        console.log(this.name + "有" + this.legs + "条腿，我会走路");
    }
}

// 中国人
function Chinese(name) {
```

```
    Person.call(this, name);
}

// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
```

三、组合继承

因为原型链继承只能继承原型对象中的属性和方法，而构造函数继承又只能继承实例对象中的属性和方法，所以便有了组合继承。

组合继承也称也叫伪经典继承，核心思想是：

- 将原型链继承和构造函数继承组合在一块
- 原型链实现对原型对象属性和方法的继承
- 构造函数实现对实例对象和方法属性的继承

```
// 人类
function Person(name) {
    this.legs = 2;
    this.mouse = 1;
    this.name = name;
    this.walk = function() {
        console.log(this.name + "有" + this.legs + "条腿，我会走路");
    }
}

Person.prototype.say = function() {
    console.log("我的名字叫" + this.name);
}

// 中国人
function Chinese(name) {
    Person.call(this, name);
}
Chinese.prototype = new Person();
Chinese.prototype.constructor = Chinese;

// 实例化对象
var chinese = new Chinese('张三');
// 调用对象方法
chinese.walk();
chinese.say();
```

扩展阅读：

https://blog.csdn.net/cc18868876837/article/details/81211729

练习：

一、什么是原型链

二、举例说明原型链继承的用法

三、举例说明直接继承原型对象的用法

四、举例说明利用空对象中转继承的用法

五、举例说明拷贝继承的用法

六、举例说明构造函数继承的用法

七、举例说明组合继承的用法

八、利用面向对象继承的思想实现下列功能

创建构造函数Person，添加属性姓名（name）、语文成绩（chinese）、数学成绩（math）；添加三个方法，分别返回姓名、语文和数学成绩

创建构函数Student，继承Person的属性和方法，并添加属于自己的属性年龄（age），添加属于自己的方法，返回年龄

创建Student的对象，调用对应方法分别输出对象的姓名、语文、数学成绩和年龄