JavaScript提高班之面向对象(六)

✍JavaScript帝国之行🔥

内容	地址
JavaScript基础大总结(一) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119249534
JavaScript基础之函数与作用域(二) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250991
JavaScript基础之对象与内置对象(三) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250137
JavaScript进阶之DOM技术(四) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115416921
JavaScript进阶之BOM技术(五) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115406408
JavaScript提高之面向对象(六) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115219073
JavaScript提高之ES6(七) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115344398

1、面向对象

面向过程：分析出解决问题所需要的步骤，然后用函数把这些步骤一步步实现，使用的时候再一个个依次调用（即按照分析好的步骤解决问题）
优点：性能比面向对象高，适合和硬件联系很紧密的东西。
缺点：没有面向对象易维护、易复用、易拓展

面向对象更贴近我们的实际生活, 可以使用面向对象描述现实世界事物. 但是事物分为具体的事物和抽象的事物

面向对象的思维特点:

1. 抽取（抽象）对象共用的属性和行为组织(封装)成一个类(模板)
2. 对类进行实例化, 获取类的对象

面向对象：以对象功能来划分问题，而不是步骤，非常适合大型项目
特性：
1.封装：比如对代码的封装。
2.继承：儿子会继承父亲的一些属性和方法。
3.多态：同一个对象在不同时候可以体现出不同的状态。
优点：易维护、易复用、易拓展，由于其特性，可以设计出低耦合的系统。
缺点：性能比面向过程低。

面向过程：蛋炒饭
面向对象：盖浇饭

1.1、对象

在 JavaScript 中，**对象是一组无序的相关属性和方法的集合**，所有的事物都是对象，例如字符串、数值、数组、函数等。

对象是由**属性**和**方法**组成的

- 属性：事物的**特征，在对象中用属性**来表示
- 方法：事物的**行为，在对象中用方法**来表示

1.2、类

在 ES6 中新增加了类的概念，可以使用 class 关键字声明一个类，之后以这个类来实例化对象。

- **类**抽象了对象的公共部分，它泛指某一大类（class）
- **对象**特指某一个，通过类实例化一个具体的对象

对象：类的实例化

1.2.1、创建类

```
class name {
    // class body
}
```

- 创建实例

```
var XX = new name();
```

注意：类必须使用 new 实例化对象

1.2.2、构造函数

constructor() 方法是类的构造函数(默认方法)，**用于传递参数,返回实例对象**，通过 new 命令生成对象实例时，**自动**调用该方法。如果没有显示定义, 类内部会自动给我们创建一个constructor()

```
<script>
    // 1. 创建类 class  创建一个 明星类
    class Star {
        // constructor 构造器或者构造函数
        constructor(uname, age) {
```

```
            this.uname = uname;
            this.age = age;
        }
    }
                                    this指向的是对象，比如ldh、zxy
    // 2. 利用类创建对象 new
    var ldh = new Star('刘德华', 18);
    var zxy = new Star('张学友', 20);
    console.log(ldh);
    console.log(zxy);
</script>
```

- 通过 class 关键字创建类，类名我们还是习惯性**定义首字母大写**
- 类里面有个 `constructor` 函数，可以接收传递过来的参数，同时返回实例对象
- `constructor` 函数只要 new 生成实例时，就会自动调用这个函数，如果我们不写这个函数，类也会自动生成这个函数
- 最后注意语法规范
  - 创建类→类名后面不要加小括号
  - 生成实例→类名后面加小括号
  - 构造函数不需要加 function 关键字

1.2.3、类中添加方法

语法：

```
class Person {
  constructor(name,age) {
      // constructor 称为构造器或者构造函数
      this.name = name;
      this.age = age;
    }
   say() {
      console.log(this.name + '你好');
   }
}
var ldh = new Person('刘德华', 18);
ldh.say()
```

注意：方法之间不能加逗号分隔，同时方法不需要添加 function 关键字。

```
<script>
    // 1. 创建类 class  创建一个 明星类
    class Star {
        // 类的共有属性放到 constructor 里面
        constructor(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        sing(song) {
            console.log(this.uname + song);
        }
    }

    // 2. 利用类创建对象 new
    var ldh = new Star('刘德华', 18);
    var zxy = new Star('张学友', 20);
    console.log(ldh);
    console.log(zxy);
    // (1) 我们类里面所有的函数不需要写function
    // (2) 多个函数方法之间不需要添加逗号分隔
    ldh.sing('冰雨');
    zxy.sing('李香兰');
</script>
```

- 类的共有属性放到 `constructor` 里面
- 类里面的函数都不需要写 `function` 关键字

1.3 、类的继承

现实中的继承：子承父业，比如我们都继承了父亲的姓。

程序中的继承：子类可以继承父类的一些属性和方法。

语法：

```
// 父类
class Father {

}
// 子类继承父类
class Son extends Father {

}
```

看一个实例：

```
<script>
    // 父类有加法方法
    class Father {
        constructor(x, y) {
            this.x = x;
            this.y = y;
        }
        sum() {
            console.log(this.x + this.y);
        }
    }
    // 子类继承父类加法方法 同时 扩展减法方法
    class Son extends Father {
        constructor(x, y) {
            // 利用super 调用父类的构造函数
            // super 必须在子类this之前调用
            super(x, y);
            this.x = x;
            this.y = y;
        }
        subtract() {
            console.log(this.x - this.y);
        }
    }
    var son = new Son(5, 3);
    son.subtract();
    son.sum();
</script>
```

1.4、super关键字

- super 关键字用于访问和调用对象父类上的函数，可以调用父类的构造函数，也可以调用父类的普通函数

1.4.1、调用父类的构造函数

语法：

```
// 父类
class Person {
    constructor(surname){
        this.surname = surname;
    }
}
// 子类继承父类
class Student entends Person {
    constructor(surname,firstname) {
        super(surname);                 //调用父类的 constructor(surname)
        this.firstname = firstname;     //定义子类独有的属性
    }
}
```

注意：**子类在构造函数中使用super,必须放到this前面（必须先调用父类的构造方法，在使用子类构造方法）**

必须先调用父类函数，才能让子类的this完成赋值操作

案例：

```
// 父类
class Father {
    constructor(surname){
        this.surname = surname;
```

```
    }
    saySurname() {
        console.log('我的姓是' + this.surname);
    }
}
// 子类继承父类
class Son entends Father {
    constructor(surname,firstname) {
        super(surname);                 //调用父类的 constructor(surname)
        this.firstname = firstname;     //定义子类独有的属性
    }
    sayFirstname() {
        console.log('我的名字是:' + this.firstname);
    }
}

var damao = new Son('刘','德华');
damao.saySurname();
damao.sayFirstname();
```

1.4.2、调用父类的普通函数

语法：

```
class Father {
    say() {
        return '我是爸爸';
    }
}
class Son extends Father {
    say(){
        // super.say() super调用父类的方法
        return super.say() + '的儿子';
    }
}

var damao = new Son();
console.log(damao.say());
```

继承中，如果实例化子类输出一个方法，先看子类有没有这个方法，如果有就先执行子类的。
如果子类没有，就去查找父类有没有这个方法，如果有就执行父类的这个方法（即就近原则）

- 多个方法之间不需要添加逗号分隔
- 继承中属性和方法的查找原则：就近原则，先看子类，再看父类

1.4、三个注意点

1. 在ES6中类没有变量提升，所以必须先定义类，才能通过类实例化对象 先有类才能实例化，不然会报错
2. 类里面的共有属性和方法一定要加 this 使用
3. 类里面的 this 指向：
   - constructor 里面的 this 指向实例对象
   - 方法里面的 this 指向这个方法的调用者

```
<body>
    <button>点击</button>
    <script>
        var that;
        var _that;
        class Star {
            constructor(uname, age) {
                // constructor 里面的this 指向的是 创建的实例对象
                that = this;
                this.uname = uname;
                this.age = age;
                // this.sing();
                this.btn = document.querySelector('button');
                this.btn.onclick = this.sing;这里注意别加（），加了就是立即调用了
            }
            sing() {
            // 这个sing方法里面的this 指向的是 btn 这个按钮,因为这个按钮调用了这个函数
                console.log(that.uname);
                // that里面存储的是constructor里面的this
            }
```

```
            dance() {
                // 这个dance里面的this 指向的是实例对象 ldh 因为ldh 调用了这个函数
                _that = this;
                console.log(this);
            }
        }
        var ldh = new Star('刘德华');
        console.log(that === ldh);
        ldh.dance();
        console.log(_that === ldh);

        // 1. 在 ES6 中类没有变量提升，所以必须先定义类，才能通过类实例化对象

        // 2. 类里面的共有的属性和方法一定要加this使用.
    </script>
</body>
```

2、构造函数和原型

2.1、概述

在典型的 OOP 的语言中（如 Java），都存在类的概念，类就是对象的模板，对象就是类的实例，但在 ES6之前， JS 中并没用引入类的概念。

ES6， 全称 ECMAScript 6.0 ，2015.06 发版。但是目前浏览器的 JavaScript 是 ES5 版本，大多数高版本的浏览器也支持 ES6，不过只实现了 ES6 的部分特性和功能。

在 ES6之前 ， 对象不是基于类创建的，而是用一种称为构建函数的特殊函数来定义对象和它们的特征。

- 创建对象有三种方式
  - **对象字面量**
  - **new Object()**
  - **自定义构造函数**

```
// 1. 利用 new Object() 创建对象
var obj1 = new Object();

// 2. 利用对象字面量创建对象
var obj2 = {};

// 3.利用构造函数创建对象
function Star(uname,age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
    }
}
var ldh = new Star('刘德华',18);
```

注意：

1. 构造函数用于创建某一类对象，其首字母要大写
2. 构造函数要和 new 一起使用才有意义

2.2、构造函数

- 构造函数是一种特殊的函数，主要用来初始化对象(为对象成员变量赋初始值)，它总与 new 一起使用
- 我们可以把对象中的一些公共的属性和方法抽取出来，然后封装到这个函数里面

new 在执行时会做四件事

1. 在内存中创建一个新的空对象。
2. 让 this 指向这个新的对象。
3. 执行构造函数里面的代码，给这个新对象添加属性和方法。
4. 返回这个新对象（所以构造函数里面不需要 return ）。

2.2.1、静态成员和实例成员 面试

JavaScript 的构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的 `this` 上添加。通过这两种方式添加的成员，就分别称为**静态成员**和**实例成员**。

- 静态成员: 在构造函数本身上添加的成员为静态成员，只能由构造函数本身来访问
- 实例成员: 在构造函数内部创建的对象成员称为实例成员，只能由实例化的对象来访问

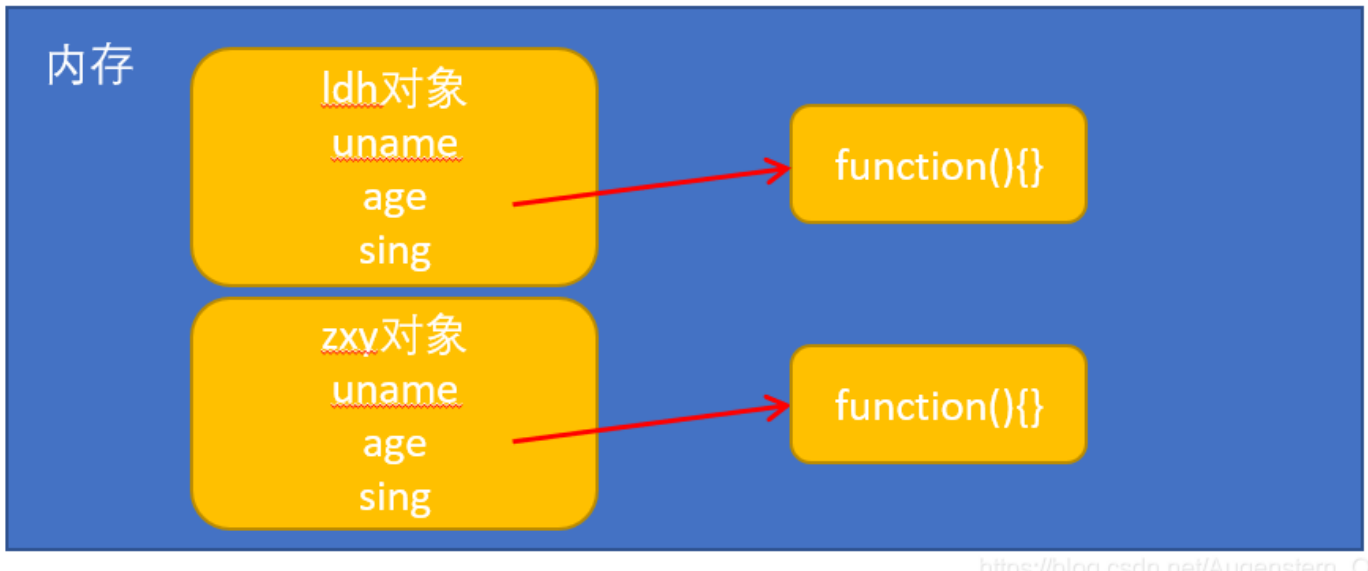
实例成员就是构造函数内部通过this添加的成员，uname age sing就是实例成员，
不可以通过构造函数来访问实例成员【比如console log (star.uname) 会显示undefined，因为this指向的是对象，没有指向star】

```
// 构造函数中的属性和方法我们称为成员，成员可以添加
function Star(uname,age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
    }
}
var ldh = new Star('刘德华',18);

// 实例成员就是构造函数内部通过this添加的成员  uname age sing  就是实例成员
// 实例成员只能通过实例化的对象来访问
ldh.sing();
Star.uname; // undefined      不可以通过构造函数来访问实例成员

// 静态成员就是在构造函数本身上添加的成员 sex 就是静态成员
// 静态成员只能通过构造函数来访问
Star.sex = '男';
Star.sex;
ldh.sex; // undefined  不能通过对象来访问
```

2.2.2、构造函数的问题

构造函数方法很好用，但是存在浪费内存的问题。

```
function Star(uname, age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
}
}
var ldh = new Star('刘德华', 18);
var zxy = new Star('张学友', 19);
```



- **我们希望所有的对象使用同一个函数，这样就比较节省内存**

2.3、构造函数原型 prototype

- 构造函数通过原型分配的函数是所有对象所共享的,这样就解决了内存浪费问题
- JavaScript 规定，每一个构造函数都有一个 `prototype` 属性，指向另一个对象，注意这个 `prototype` 就是一个对象，这个对象的所有属性和方法，都会被构造函数所拥有
- 我们可以把那些不变的方法，直接定义在 `prototype` 对象上，这样所有对象的实例就可以共享这些方法

```
<body>
    <script>
        // 1. 构造函数的问题.
        function Star(uname, age) {
            //公共属性定义到构造函数里面
            this.uname = uname;
            this.age = age;
            // this.sing = function() {
            //     console.log('我会唱歌');
            // }
        }
        //公共的方法我们放到原型对象身上
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
```

```
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
        console.log(ldh.sing === zxy.sing);
        ldh.sing();
        zxy.sing();
        // 2. 一般情况下,我们的公共属性定义到构造函数里面, 公共的方法我们放到原型对象身上
    </script>
</body>
```

- 一般情况下,我们的公共属性定义到构造函数里面, 公共的方法我们放到原型对象身上

问答：原型是什么?

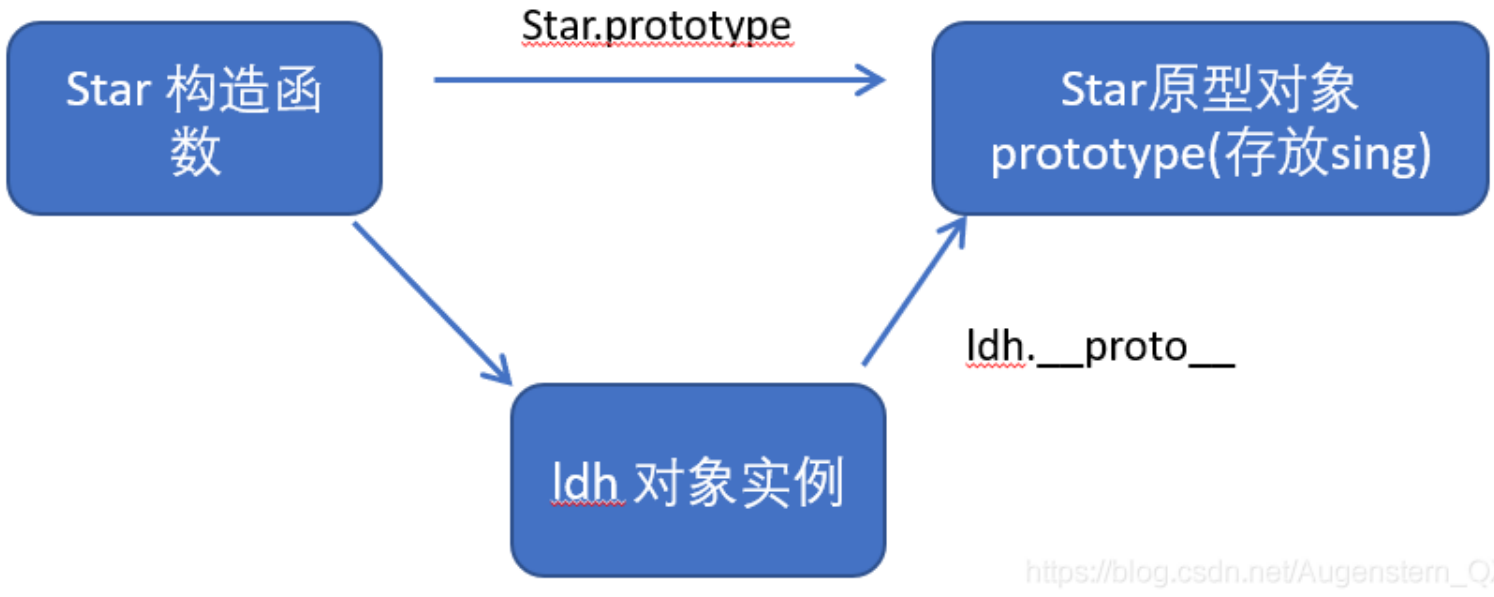
- 一个对象，我们也称为 `prototype` 为原型对象

问答：原型的作用是什么?

- 共享方法　　节省内存资源

2.4、对象原型 __ proto __

- 对象都会有一个属性 `_proto_` 指向构造函数的 `prototype` 原型对象，之所以我们对象可以使用构造函数 `prototype` 原型对象的属性和方法，就是因为对象有 `_proto_` 原型的存在。　为什么创建的对象可以调用原型
- `_proto_` 对象原型和原型对象 `prototype` 是等价的
- `_proto_` 对象原型的意义就在于为对象的查找机制提供一个方向，或者说一条路线，但是它是一个非标准属性，因此实际开发中，不可以使用这个属性，它只是内部指向原型对象 `prototype`



- `Star.prototype` 和 `ldh._proto_` 指向相同

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
        ldh.sing();
        console.log(ldh);
        // 对象身上系统自己添加一个 __proto__ 指向我们构造函数的原型对象 prototype
        console.log(ldh.__proto__ === Star.prototype);
        // 方法的查找规则: 首先先看ldh 对象身上是否有 sing 方法,如果有就执行这个对象上的sing
        // 如果没有sing 这个方法,因为有 __proto__ 的存在,就去构造函数原型对象prototype身上去查找sing这个方法
    </script>
</body>
```

2.5、constructor 构造函数

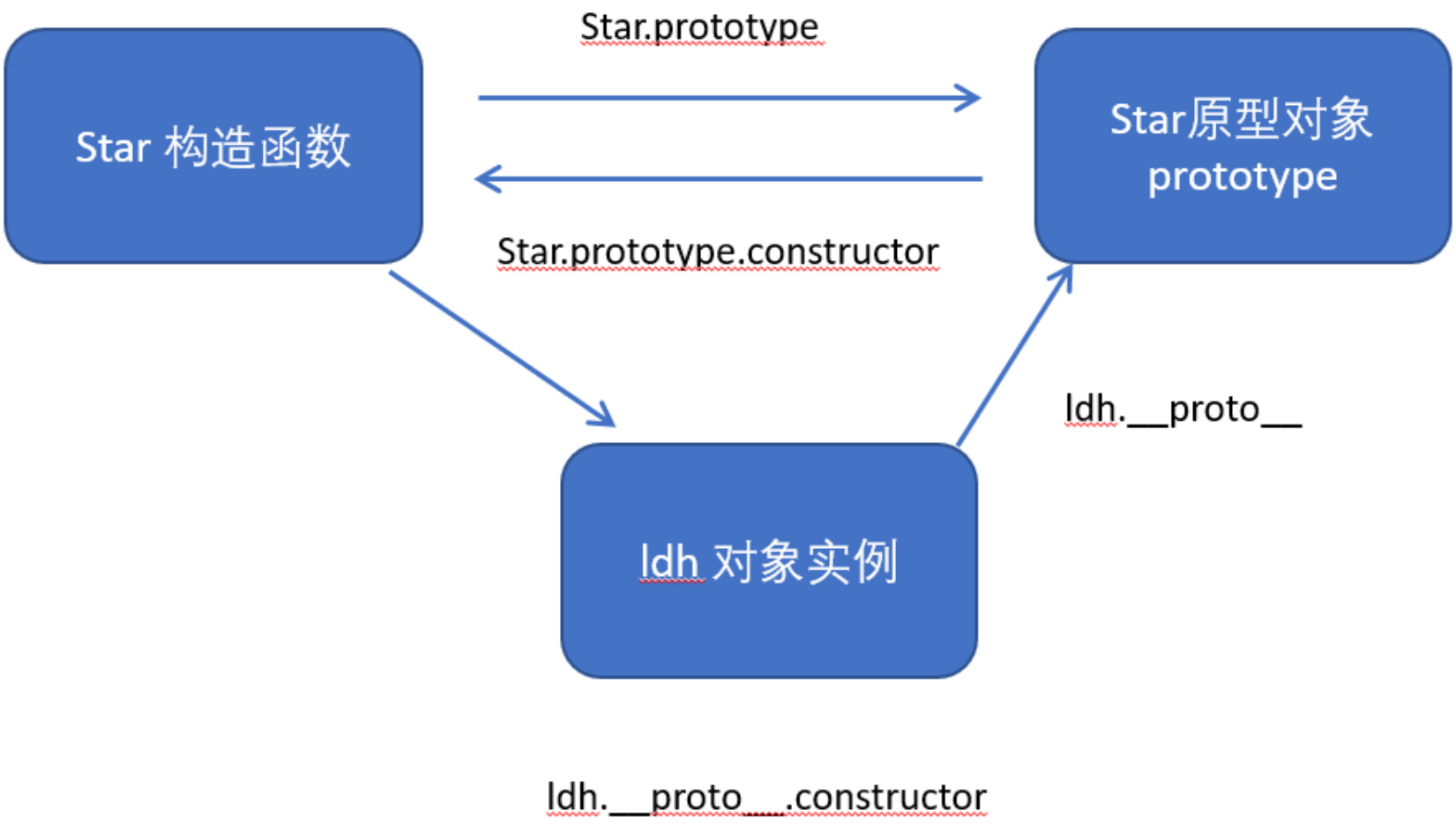
- 对象原型(__ proto __) 和构造函数(prototype)原型对象 里面都有一个属性 constructor 属性， constructor 我们称为构造函数，因为它指回构造函数本身。
- `constructor` 主要用于记录该对象引用于哪个构造函数，它可以让原型对象重新指向原来的构造函数

- **一般情况下，对象的方法都在构造函数(prototype)的原型对象中设置**
- 如果有多个对象的方法，我们可以给原型对象 `prototype` 采取对象形式赋值，但是这样会覆盖构造函数原型对象原来的内容，这样修改后的原型对象 `constructor` 就不再指向当前构造函数了。此时，我们可以在修改后的原型对象中，添加一个 `constructor` 指向原来的构造函数

具体请看实例配合理解

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        // 很多情况下,我们需要手动的利用constructor 这个属性指回 原来的构造函数
        // Star.prototype.sing = function() {
        //     console.log('我会唱歌');
        // };
        // Star.prototype.movie = function() {
        //     console.log('我会演电影');
        // }
        Star.prototype = {
            // 如果我们修改了原来的原型对象,给原型对象赋值的是一个对象,则必须手动的利用constructor指回原来的构造函数
            constructor: Star,
            sing: function() {
                console.log('我会唱歌');
            },
            movie: function() {
                console.log('我会演电影');
            }
        }
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
    </script>
</body>
```

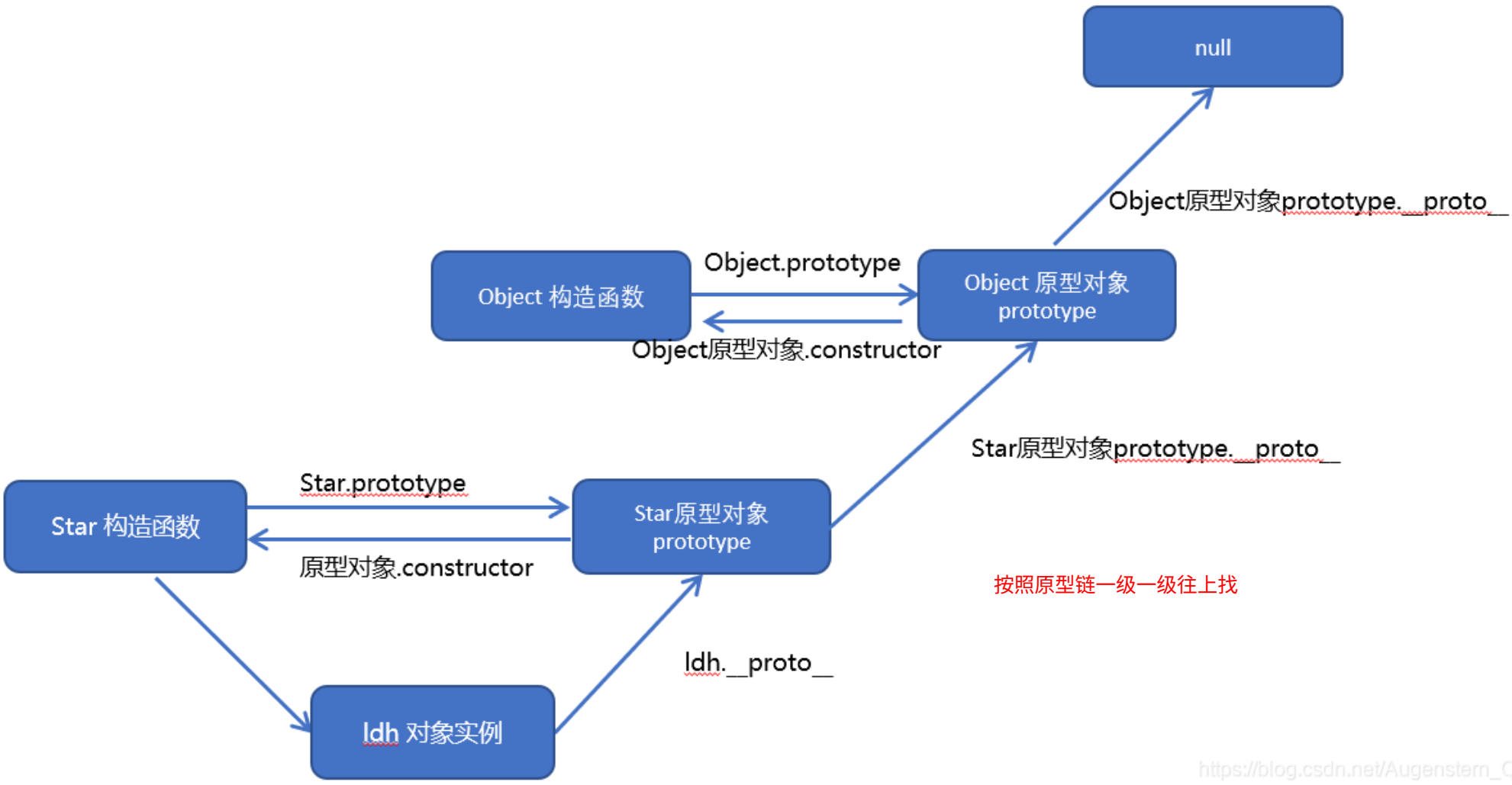
2.6、构造函数、实例、原型对象三者关系



2.7、原型链查找规则

1. 当访问一个对象的属性(包括方法)时，首先查找这个**对象自身**有没有该属性
2. 如果没有就查找它的原型(也就是 `_proto_` 指向的 `prototype原型对象`)
3. 如果还没有就查找原型对象的原型(**Object的原型对象**)
4. 依次类推一直找到Object为止(null)
5. __ proto __对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线。

1.8 原型链




```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
        var ldh = new Star('刘德华', 18);
        // 1. 只要是对象就有__proto__ 原型, 指向原型对象
        console.log(Star.prototype);
        console.log(Star.prototype.__proto__ === Object.prototype);
        // 2.我们Star原型对象里面的__proto__原型指向的是 Object.prototype
        console.log(Object.prototype.__proto__);
        // 3. 我们Object.prototype原型对象里面的__proto__原型  指向为 null
    </script>
</body>
```

2.8、原型对象this指向 全部指向实例对象

- 构造函数中的 `this` 指向我们的实例对象
- 原型对象里面放的是方法，这个方法里面的 `this` 指向的是这个方法的调用者，也就是这个实例对象

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        var that;
        Star.prototype.sing = function() {
            console.log('我会唱歌');
            that = this;
        }
        var ldh = new Star('刘德华', 18);
        // 1. 在构造函数中,里面this指向的是对象实例 ldh
        ldh.sing();
        console.log(that === ldh);

        // 2.原型对象函数里面的this 指向的是 实例对象 ldh
    </script>
</body>
```

2.9、扩展内置对象

- 可以通过原型对象，对原来的内置对象进行扩展自定义的方法
- 比如给数组增加自定义求偶数和的功能

```
<body>
    <script>
        // 原型对象的应用 扩展内置对象方法

        Array.prototype.sum = function() {
            var sum = 0;
            for (var i = 0; i < this.length; i++) {
                sum += this[i];
            }
            return sum;
        };
        // Array.prototype = {
        //     sum: function() {
        //         var sum = 0;
        //         for (var i = 0; i < this.length; i++) {
        //             sum += this[i];
        //         }
        //         return sum;
        //     }

        // }
        var arr = [1, 2, 3];
        console.log(arr.sum());
        console.log(Array.prototype);
        var arr1 = new Array(11, 22, 33);
        console.log(arr1.sum());
    </script>
</body>
```

注意：

- 数组和字符串内置对象不能给原型对象覆盖操作 `Array.prototype = {}` ，只能是 `Array.prototype.xxx = function(){}` 的方式

3、继承

ES6 之前并没有给我们提供 `extends` 继承

- 我们可以通过**构造函数+原型对象**模拟实现继承，被称为**组合继承**

3.1、call()

调用这个函数，并且修改函数运行时的 this 指向

```
fun.call(thisArg,arg1,arg2,.....)
        （指向，参数，参数.....）
```

- `thisArg`： 当前调用函数 this 的指向对象
- `arg1,arg2`： 传递的其他参数

示例

```
<body>
    <script>
        // call 方法
        function fn(x, y) {
            console.log('我希望我的希望有希望');
            console.log(this);      // Object{...}
            console.log(x + y);     // 3
        }

        var o = {
            name: 'andy'
        };
        // fn();
        // 1. call() 可以调用函数
        // fn.call();
        // 2. call() 可以改变这个函数的this指向 此时这个函数的this 就指向了o这个对象
        fn.call(o, 1, 2);
    </script>
</body>
```

3.2、借用构造函数继承父类型属性

- 核心原理: 通过 `call()` 把父类型的 this 指向子类型的 this，这样就可以实现子类型继承父类型的属性

```
<body>
    <script>
        // 借用父构造函数继承属性
        // 1. 父构造函数
        function Father(uname, age) {
            // this 指向父构造函数的对象实例
            this.uname = uname;
            this.age = age;
        }
        // 2 .子构造函数
        function Son(uname, age, score) {
            // this 指向子构造函数的对象实例
            Father.call(this, uname, age);
            this.score = score;
        }
        var son = new Son('刘德华', 18, 100);
        console.log(son);
    </script>
</body>
```

3.3、借用原型对象继承父类型方法

- 一般情况下，对象的方法都在构造函数的原型对象中设置，通过构造函数无法继承父类方法

核心原理：

1. 将子类所共享的方法提取出来，让子类的 `prototype 原型对象 = new 父类()`
2. 本质： 子类原型对象等于是实例化父类，因为父类实例化之后另外开辟空间，就不会影响原来父类原型对象
3. 将子类的 `constructor` 重新指向子类的构造函数

```
<body>
    <script>
        // 借用父构造函数继承属性
        // 1. 父构造函数
        function Father(uname, age) {
            // this 指向父构造函数的对象实例
            this.uname = uname;
            this.age = age;
        }
        Father.prototype.money = function() {
            console.log(100000);

        };
        // 2 .子构造函数
        function Son(uname, age, score) {
            // this 指向子构造函数的对象实例
            Father.call(this, uname, age);
            this.score = score;
        }
        // Son.prototype = Father.prototype;  这样直接赋值会有问题,如果修改了子原型对象,父原型对象也会跟着一起变化
        Son.prototype = new Father();
        // 如果利用对象的形式修改了原型对象,别忘了利用constructor 指回原来的构造函数
        Son.prototype.constructor = Son;
        // 这个是子构造函数专门的方法
        Son.prototype.exam = function() {
            console.log('孩子要考试');

        }
        var son = new Son('刘德华', 18, 100);
        console.log(son);
        console.log(Father.prototype);
        console.log(Son.prototype.constructor);
    </script>
</body>
```

3.3 类的本质

1. class 本质还是 function
2. 类的所有方法都定义在类的 `prototype` 属性上
3. 类创建的实例，里面也有 `_proto_` 指向类的 `prototype` 原型对象
4. 所以 ES6 的类它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。
5. 所以 ES6 的类其实就是语法糖
6. 语法糖：语法糖就是一种便捷写法，简单理解

4、ES5新增方法

ES5 给我们新增了一些方法，可以很方便的操作数组或者字符串

- 数组方法 重要，面试可能会问
- 字符串方法
- 对象方法

4.1、数组方法

- 迭代(遍历)方法：foreach() ， map()，filter()，some() ， every() ;

4.1.1、forEach()

map就是比forEach多个返回值，会创建一个数组返回
every和some比较像

```
array.forEach(function(currentValue,index,arr))
```

这里的function是个回调函数

foreach没全部遍历完遇到return true不会结束，some中遇到return true会终结
filter遇到return也不会终止；所以在数组中查询唯一元素，用some比较合适

- currentValue : 数组当前项的值
- index: 数组当前项的索引
- arr: 数组对象本身

```
<body>
    <script>
        // forEach 迭代(遍历) 数组
        var arr = [1, 2, 3];
        var sum = 0;
        arr.forEach(function(value, index, array) {
            console.log('每个数组元素' + value);
            console.log('每个数组元素的索引号' + index);
            console.log('数组本身' + array);
            sum += value;
        })
        console.log(sum);
    </script>
</body>
```

4.1.2、filter()筛选数组

```
array.filter(function(currentValue,index,arr))
```

function（数组的元素，数组元素的索引号，数组对象【只要是用不到的都可省略】）

- `filter()` 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素，主要用于筛选数组
- 注意它直接返回一个新数组

```
<body>
    <script>
        // filter 筛选数组
        var arr = [12, 66, 4, 88, 3, 7];
        var newArr = arr.filter(function(value, index) {
            // return value >= 20;
            return value % 2 === 0;
        });
        console.log(newArr);
    </script>
</body>
```

4.1.3、some() 查找数组中是否有满足条件的元素

- `some()` 方法用于检测数组中的元素是否满足指定条件（查找数组中是否有满足条件的元素）

- 注意它返回的是布尔值，如果查找到这个元素，就返回true，如果查找不到就返回false
- 如果找到第一个满足条件的元素，则终止循环，不再继续查找

```
<body>
    <script>
        // some 查找数组中是否有满足条件的元素
        var arr1 = ['red', 'pink', 'blue'];
        var flag1 = arr1.some(function(value) {
            return value == 'pink';
        });
        console.log(flag1);
        // 1. filter 也是查找满足条件的元素 返回的是一个数组 而且是把所有满足条件的元素返回回来
        // 2. some 也是查找满足条件的元素是否存在  返回的是一个布尔值 如果查找到第一个满足条件的元素就终止循环
    </script>
</body>
```

4.2、字符串方法

- `trim()` 方法会从一个字符串的两端删除空白字符
- `trim()` 方法并不影响原字符串本身，它返回的是一个新的字符串

```
<body>
    <input type="text"> <button>点击</button>
    <div></div>
    <script>
        // trim 方法去除字符串两侧空格
        var str = '   an  dy   ';
        console.log(str);
        var str1 = str.trim();
        console.log(str1);
        var input = document.querySelector('input');
        var btn = document.querySelector('button');
        var div = document.querySelector('div');
        btn.onclick = function() {
            var str = input.value.trim();
            if (str === '') {
                alert('请输入内容');
            } else {
                console.log(str);
                console.log(str.length);
                div.innerHTML = str;
            }
        }
    </script>
</body>
```

4.3、对象方法

4.3.1、Object.keys()

1. `Object.keys()` 用于获取对象自身所有的属性
2. 效果类似 `for...in`
3. 返回一个由属性名组成的数组

```
<body>
    <script>
        // 用于获取对象自身所有的属性
        var obj = {
            id: 1,
            pname: '小米',
            price: 1999,
            num: 2000
        };
        var arr = Object.keys(obj);
        console.log(arr);
        arr.forEach(function(value) {
            console.log(value);
            // id
            // pname
            // price
            // num
```

```
        })
    </script>
</body>
```

4.3.2、Object.defineProperty()

有则修改，无则添加

重要！！Vue有涉及

- `Object.defineProperty()` 定义对象中新属性或修改原有的属性[redacted]

```
Object.defineProperty(obj,prop,descriptor)
```

- obj : 目标对象
- prop : 需定义或修改的属性的名字
- descriptor : 目标属性所拥有的特性

```
<body>
    <script>
        // Object.defineProperty() 定义新属性或修改原有的属性
        var obj = {
            id: 1,
            pname: '小米',
            price: 1999
        };
        // 1. 以前的对象添加和修改属性的方式
        // obj.num = 1000;
        // obj.price = 99;
        // console.log(obj);
        // 2. Object.defineProperty() 定义新属性或修改原有的属性
        Object.defineProperty(obj, 'num', {
            value: 1000,
            enumerable: true
        });
        console.log(obj);
        Object.defineProperty(obj, 'price', {
            value: 9.9
        });
        console.log(obj);
        Object.defineProperty(obj, 'id', {
            // 如果值为false 不允许修改这个属性值 默认值也是false
            writable: false,
        });
        obj.id = 2;
        console.log(obj);
        Object.defineProperty(obj, 'address', {
            value: '中国山东蓝翔技校xx单元',
            // 如果只为false 不允许修改这个属性值 默认值也是false
            writable: false,
            // enumerable 如果值为false 则不允许遍历, 默认的值是 false
            enumerable: false,
            // configurable 如果为false 则不允许删除这个属性 不允许在修改第三个参数里面的特性 默认为false
            configurable: false
        });
        console.log(obj);
        console.log(Object.keys(obj));
        delete obj.address;
        console.log(obj);
        delete obj.pname;
        console.log(obj);
        Object.defineProperty(obj, 'address', {
            value: '中国山东蓝翔技校xx单元',
            // 如果值为false 不允许修改这个属性值 默认值也是false
            writable: true,
            // enumerable 如果值为false 则不允许遍历, 默认的值是 false
            enumerable: true,
            // configurable 如果为false 则不允许删除这个属性 默认为false
            configurable: true
        });
        console.log(obj.address);
    </script>
</body>
```

- 第三个参数 descriptor 说明：以对象形式{ }书写

- value：设置属性的值，默认为undefined
- writeable: 值是否可以重写 true | false 默认为false
- enumerable: 目标属性是否可以被枚举 true | false 默认为false
- configurable: 目标属性是否可以被删除或是否可以再次修改特性 true | false 默认为false

5、函数进阶

5.1、函数的定义方式

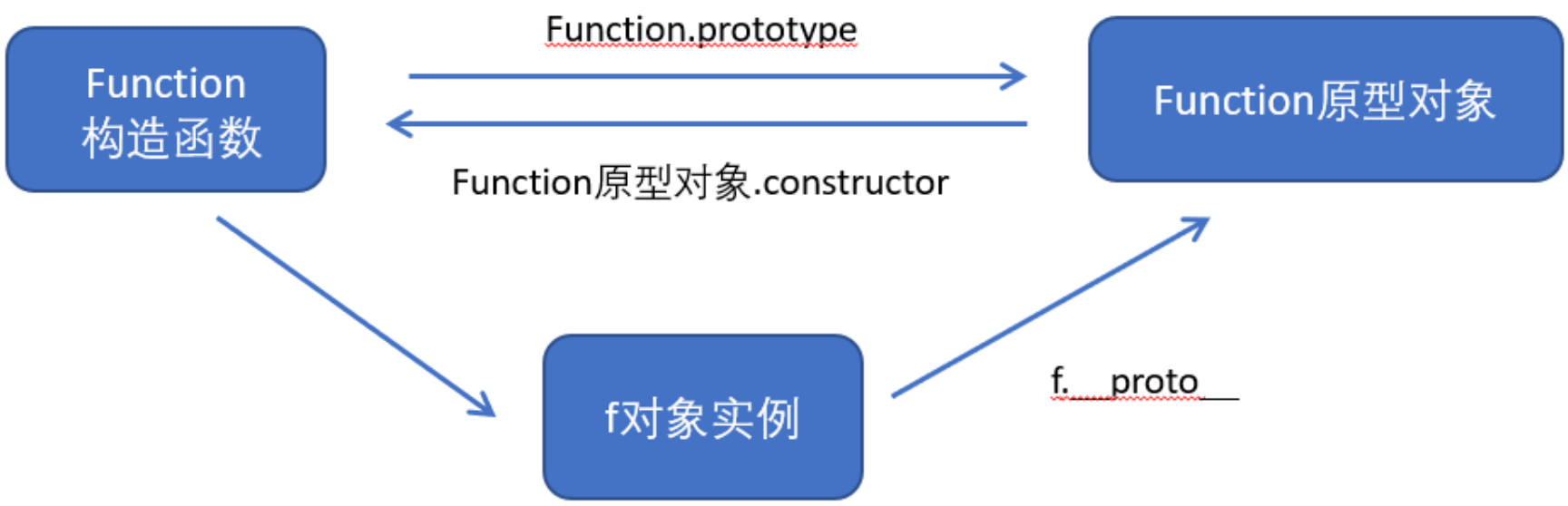
1. 函数声明方式 function 关键字(命名函数)

2. 函数表达式(匿名函数)

3. new Function()

```
var fn = new Function('参数1','参数2',.....,'函数体');
```

- Function 里面参数都必须是字符串格式
- 第三种方式执行效率低，也不方便书写，因此较少使用
- 所有函数都是 Function 的实例(对象)
- 函数也属于对象




```
<body>
    <script>
        //  函数的定义方式

        // 1. 自定义函数(命名函数)

        function fn() {};

        // 2. 函数表达式 (匿名函数)

        var fun = function() {};


        // 3. 利用 new Function('参数1','参数2', '函数体');
        //              Function 里面参数都必须是字符串格式，执行效率低，较少写

        var f = new Function('a', 'b', 'console.log(a + b)');
        f(1, 2);
        // 4. 所有函数都是 Function 的实例(对象)
        console.dir(f);
        // 5. 函数也属于对象
        console.log(f instanceof Object);
    </script>
</body>
```

5.2、函数的调用方式

1. 普通函数

2. 对象的方法

3. 构造函数

4. 绑定事件函数

5. 定时器函数

6. 立即执行函数

```
<body>
    <script>
        // 函数的调用方式

        // 1. 普通函数
        function fn() {
            console.log('人生的巅峰');

        }
        // fn();   fn.call()
        // 2. 对象的方法
        var o = {
            sayHi: function() {
                console.log('人生的巅峰');

            }
        }
        o.sayHi();
        // 3. 构造函数
        function Star() {};
        new Star();
        // 4. 绑定事件函数
        // btn.onclick = function() {};   // 点击了按钮就可以调用这个函数
        // 5. 定时器函数
        // setInterval(function() {}, 1000);  这个函数是定时器自动1秒钟调用一次
        // 6. 立即执行函数
        (function() {
            console.log('人生的巅峰');
        })();
        // 立即执行函数是自动调用
    </script>
</body>
```

5.3、函数内this的指向

- `this` 指向，是当我们调用函数的时候确定的，调用方式的不同决定了 `this` 的指向不同，一般我们指向我们的调用者

调用方式	this指向
普通函数调用	window
构造函数调用	实例对象，原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件对象
定时器函数	window
立即执行函数	window

```
<body>
    <button>点击</button>
    <script>
        // 函数的不同调用方式决定了this 的指向不同
        // 1. 普通函数 this 指向window
        function fn() {
            console.log('普通函数的this' + this);
        }
        window.fn();
        // 2. 对象的方法 this指向的是对象 o
        var o = {
            sayHi: function() {
                console.log('对象方法的this:' + this);
            }
```

```
        }
        o.sayHi();
        // 3. 构造函数 this 指向 ldh 这个实例对象 原型对象里面的this 指向的也是 ldh这个实例对象
        function Star() {};
        Star.prototype.sing = function() {

        }
        var ldh = new Star();
        // 4. 绑定事件函数 this 指向的是函数的调用者 btn这个按钮对象
        var btn = document.querySelector('button');
        btn.onclick = function() {
            console.log('绑定时间函数的this:' + this);
        };
        // 5. 定时器函数 this 指向的也是window
        window.setTimeout(function() {
            console.log('定时器的this:' + this);

        }, 1000);
        // 6. 立即执行函数 this还是指向window
        (function() {
            console.log('立即执行函数的this' + this);
        })();
    </script>
</body>
```

5.4、改变函数内部this指向

- JavaScript 为我们专门提供了一些函数方法来帮我们处理函数内部 this 的指向问题，常用的有 `bind(),call(),apply()` 三种方法

5.4.1、call() 方法

- `call()` 方法调用一个对象，简单理解为调用函数的方式，但是它可以改变函数的 `this` 指向
- `fun.call(thisArg,arg1,arg2,.....)`
- `thisArg` : 在 fun 函数运行时指定的 this 值
- `arg1,arg2` : 传递的其他参数
- 返回值就是函数的返回值，因为它就是调用函数
- 因此当我们想改变 this 指向，同时想调用这个函数的时候，可以使用 call，比如继承

```
<body>
    <script>
        // 改变函数内this指向  js提供了三种方法  call()  apply()  bind()

        // 1. call()
        var o = {
            name: 'andy'
        }

        function fn(a, b) {
            console.log(this);
            console.log(a + b);

        };
        fn.call(o, 1, 2);
        // call 第一个可以调用函数 第二个可以改变函数内的this 指向
        // call 的主要作用可以实现继承
        function Father(uname, age, sex) {
            this.uname = uname;
            this.age = age;
            this.sex = sex;
        }

        function Son(uname, age, sex) {
            Father.call(this, uname, age, sex);
        }
        var son = new Son('刘德华', 18, '男');
        console.log(son);
    </script>
</body>
```

5.4.2、apply()方法

- `apply()` 方法调用一个函数，简单理解为调用函数的方式，但是它可以改变函数的 `this` 指向
- `fun.apply(thisArg,[argsArray])`
- thisArg: 在 fun 函数运行时指定的 this 值
- argsArray : 传递的值，必须包含在数组里面
- 返回值就是函数的返回值，因为它就是调用函数
- 因此 apply 主要跟数组有关系，比如使用 Math.max() 求数组的最大值

```
<body>
    <script>
        // 改变函数内this指向  js提供了三种方法  call()  apply()  bind()

        // 2. apply()  应用 运用的意思
        var o = {
            name: 'andy'
        };

        function fn(arr) {
            console.log(this);
            console.log(arr); // 'pink'

        };
        fn.apply(o, ['pink']);
        // 1. 也是调用函数 第二个可以改变函数内部的this指向
        // 2. 但是他的参数必须是数组(伪数组)
        // 3. apply 的主要应用 比如说我们可以利用 apply 借助于数学内置对象求数组最大值
        // Math.max();
        var arr = [1, 66, 3, 99, 4];
        var arr1 = ['red', 'pink'];
        // var max = Math.max.apply(null, arr);
        var max = Math.max.apply(Math, arr);
        var min = Math.min.apply(Math, arr);
        console.log(max, min);
    </script>
</body>
```

5.4.3、bind()方法

- `bind()` 方法不会调用函数。但是能改变函数内部 `this` 指向
- `fun.bind(thisArg,arg1,arg2,....)`
- 返回由指定的 `this` 值和初始化参数改造的 原函数拷贝
- 因此当我们只是想改变 this 指向，并且不想调用这个函数的时候，可以使用bind

```
<body>
    <button>点击</button>
    <button>点击</button>
    <button>点击</button>
    <script>
        // 改变函数内this指向  js提供了三种方法  call()  apply()  bind()

        // 3. bind()  绑定 捆绑的意思
        var o = {
            name: 'andy'
        };

        function fn(a, b) {
            console.log(this);
            console.log(a + b);


        };
        var f = fn.bind(o, 1, 2);
        f();
        // 1. 不会调用原来的函数    可以改变原来函数内部的this 指向
        // 2. 返回的是原函数改变this之后产生的新函数
        // 3. 如果有的函数我们不需要立即调用,但是又想改变这个函数内部的this指向此时用bind
        // 4. 我们有一个按钮,当我们点击了之后,就禁用这个按钮,3秒钟之后开启这个按钮
        // var btn1 = document.querySelector('button');
        // btn1.onclick = function() {
        //     this.disabled = true; // 这个this 指向的是 btn 这个按钮
        //     // var that = this;
```

```
        //     setTimeout(function() {
        //         // that.disabled = false; // 定时器函数里面的this 指向的是window
        //         this.disabled = false; // 此时定时器函数里面的this 指向的是btn
        //     }.bind(this), 3000); // 这个this 指向的是btn 这个对象
        // }
        var btns = document.querySelectorAll('button');
        for (var i = 0; i < btns.length; i++) {
            btns[i].onclick = function() {
                this.disabled = true;
                setTimeout(function() {
                    this.disabled = false;
                }.bind(this), 2000);
            }
        }
    </script>
</body>
```

5.4.4、总结

call apply bind 总结：

相同点：

- `call` 和 `apply` 会调用函数，并且改变函数内部的 `this` 指向

区别点：

- `call` 和 `apply` 会调用函数，并且改变函数内部的 `this` 指向
- `call` 和 `apply` 传递的参数不一样，call 传递参数，apply 必须数组形式
- `bind` 不会调用函数，可以改变函数内部 `this` 指向

主要应用场景

1. `call` 经常做继承
2. `apply` 经常跟数组有关系，比如借助于数学对线实现数组最大值与最小值
3. `bind` 不调用函数，但是还想改变this指向，比如改变定时器内部的this指向