JavaScript 进阶

一、JS 数据渲染机制

当浏览器加载页面、JS的时候，首先会形成一个供代码自上而下执行的全局作用域 window

```
var a = 100;
/* 值类型说明:
 * 1.先声明一个变量a，没有赋值（默认值是undefined）
 * 2.在当前作用域中开辟一个位置存储100这个值
 * 3.让变量a和100关联在一起（定义：赋值）
 */

var b = a;
/* （a 的值为数值型：直接操作）把 a 存储的值放到一个新的位置上，让新位置上的值和 b 保持关联，此时的 b 和 a 没有关系 */

var b = 200;
/* 让 b 和一个新的值 200 进行关联，取代原有关联，故 b 的值为 200，而 a 的值还是 100
 */

/*上述也是值类型的特点*/

var arr1 = [100, 200];
/* 引用类型说明
 * 1、在全局作用域外新开辟一段内存空间
 * 2、新的内存空间中，0 : 100  1 : 200  length : 2
 * 3、新开辟的空间有一个地址值，通过这个地址值赋给 arr1，从而使得 arr1 与 新的内存空间进行关联
 */

var arr2 = arr1;
/* arr2 指向 arr1 所关联的内存空间 */

arr2.push(300);
/* 在 arr2 所指向的内存空间中的末尾追加一个值 300，arr1 和 arr2 所指向的均为同一内存空间，故引用类型的相关操作可改变原有空间 */

/*上述也是引用类型的特点*/
```

【 问题：函数也是引用类型 】

> [例子：实现任意数求和]
>
> ```
> function sum () {
>   var total = null;
>   for (var i = 0; i < arguments.length; i++) {
> ```

```
        var item = arguments[i];
        item = parseFloat(item);
        !isNaN ? total += item : null;
      }
      return total;
    }
    /*
     * sum = 【开辟新空间的地址值】
     * 新空间：将函数体内的代码以 字符串 的形式存储到新开辟的内存空间中
     */

    sum(100, 200, '14', 'abc');
    /* 函数的执行：均会形成一个私有的作用域
     * 其目的：使得之前存储到新地址空间中的 字符串 自上而下执行
     */
```

【问题：变量提升】

```
【例子】
console.log(a);
/* 浏览器并没有报错，而是输出 undefined
 * 那么说明：变量 a 已经存在了，即：在 js 代码执行之前，还做了 变量提升
 */
var a = 100;
```

[变量提升]
=> 当栈内存(作用域)形成，JS代码自上而下执行之前，浏览器首先会把所有带关键词（var / function）的进行提前 “声明” 或者 “定义”，这种预先处理机制称之为 “变量提升”

[变量提升阶段]
=>带“VAR”的只声明未定义
=>带“FUNCTION”的声明和赋值都完成了

[变量提升细节问题]

```
sum();
fn();   // Uncaught TypeError: fn is not a function

/*
 * 变量提升:
 *   var fn;   =>只对等号左边进行变量提升
 *   sum = AAAFFF111;
 */

//=>匿名函数之函数表达式
var fn = function () {
  console.log(1);
```

```
};  //=>代码执行到此处会把函数值赋值给fn
fn();
//=>普通的函数
function sum() {
  console.log(2);
}
----------------------------------------------------------------
/* 机制：
 * 在当前作用域下，不管条件是否成立都要进行变量提升
 *   =>带VAR的还是只声明
 *   =>带FUNCTION的在老版本浏览器渲染机制下，声明和定义都处理，但是为了迎合ES6中的块级作用域，新版浏览器对于函数（在条件判断中的函数），不管条件是否成立，都只是先声明，没有定义，类似于VAR
 */
/*
 * 变量提升：
 *   var a;
 * 在全局作用域下声明的全局变量也相当于给WIN设置了一个属性window.a=undefined
 */
console.log(a);  //=>undefined
if ('a' in window) {
  var a = 100;
}
console.log(a);  //=>100
----------------------------------------------------------------
/*
 * 变量提升：
 *   function fn;
 */
console.log(fn);   //=>undefined
if (1 === 1) {
  console.log(fn);
  // 函数本身：当条件成立，进入到判断体中
  // 在ES6中它是一个块级作用域（大括号内）
  // 第一件事并不是代码执行，而是类似于变量提升一样，先把FN声明和定义了
  // 也就是判断体中代码执行之前，FN就已经赋值了
  function fn() {
    console.log('ok');
  }
}
console.log(fn);   // 函数本身，前提上述if条件成立，进入判断体中
// console.log(fn);   // undefined，前提上述if条件不成立，没进入判断体中
```

[变量提升重名问题的处理]

```
[ 例1 ]
var fn = 12;
function fn(){}
/* 带VAR和FUNCTION关键字声明相同的名字，这种也算是重名了（其实是一个FN，只是存储值的类型不一样） */
```

如果名字重复了，不会重新的声明，但是会重新的定义（重新赋值）[不管是变量提升还是代码执行阶段皆是如此]

```
[ 例2 ]
fn();    //=>4
function fn() {console.log(1);}
fn();    //=>4
function fn() {console.log(2);}
fn();    //=>4
var fn=100;
/* 带VAR的在提升阶段只把声明处理了,赋值操作没有处理,所以在代码执行的时候需要完成赋值 FN=100 */
fn();  //=>100()  Uncaught TypeError: fn is not a function
function fn() {console.log(3);}
fn();
function fn() {console.log(4);}
fn();
/*
 * 变量提升:
 *   fn = ...（1）
 *       下面均不需要声明 fn
 *      = ...（2）
 *      = ...（3）
 *      = ...（4）
 */
```

[注意]
=>变量提升只发生在当前作用域（例如：开始加载页面的时候只对全局作用域下的进行提升，因为此时函数中存储的都是字符串）
=>在全局作用域下声明的函数或者变量是“全局变量”，同理，在私有作用域下声明的变量是“私有变量” [带VAR/FUNCTION的才是声明]
=>在ES6中基于LET/CONST等方式创建变量或者函数,不存在变量提升机制

【 带 var 和不带 var 的区别 】

带 var

```
console.log(a);                    //=>undefined
// 由变量提升机制导致
console.log(window.a);             //=>undefined
// 注意：win中有a值是undefined，没有也是undefined
// 这里是因为WINDOW中有这个属性，只不过变量提升机制导致这个属性未赋值

var a = 12;
console.log(a);                    //=>全局变量a  12
console.log(window.a);     //=>WINDOW的一个属性名a  12
```

```
/* 在全局作用域下声明一个变量，也相当于给WINDOW全局对象设置了一个属性，变量的值就是属性值（私有作用域中声明的私有变量和WINDOW没啥关系） */
/* 全局变量值修改，WIN的属性值也跟着修改 */
// 全局变量和WIN中的属性存在“映射机制”

console.log('a' in window);       //=>TRUE

/* 在变量提升阶段，在全局作用域中声明了一个变量A，此时就已经把A当做属性赋值给WINDOW了，只不过此时还没有给A赋值，默认值UNDEFINED */
// in：检测某个属性是否隶属于这个对象
```

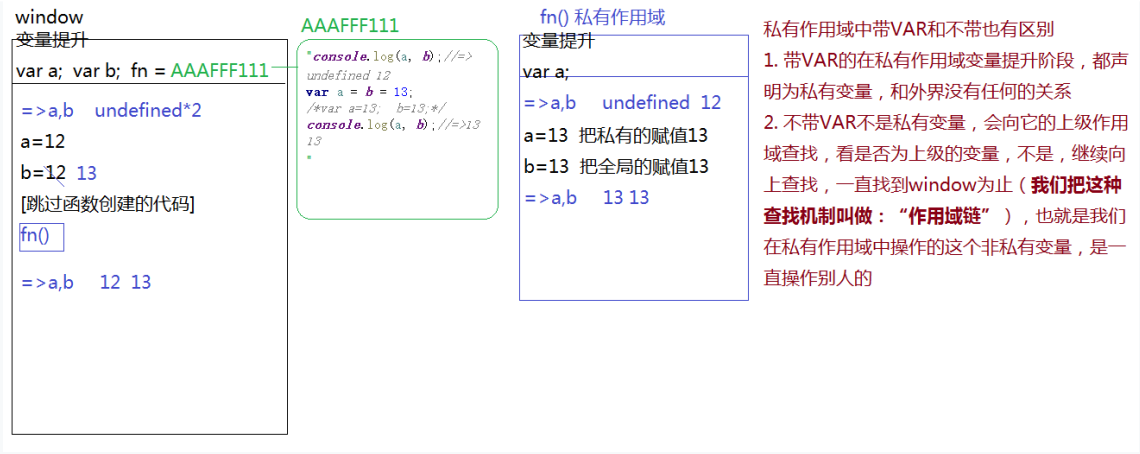
不带 var

```
// console.log(a);   //=>Uncaught ReferenceError: a is not defined
console.log(window.a);            //=>undefined
// 注意：win中有a值是undefined，没有也是undefined
// 这里是因为WINDOW中没有这个属性
console.log('a' in window);       //=>FALSE

a = 12;
console.log(a);                   //=>a  12
console.log(window.a);    //=>12
// => 这里是因为：代码简写 window.a=12
```

总结：不加VAR的本质是WIN的属性

[例题]

```
/*var a = 12,
      b = 13;   //=>这样写B是带VAR的*/
/*var a = b = 12;  //=>这样写B是不带VAR的*/

console.log(a, b);    //=>undefined undefined
var a = 12,
    b = 12;
function fn() {
  console.log(a, b);  //=>undefined 12
  var a = b = 13;
  console.log(a, b);  //=>13 13
}
fn();
console.log(a, b);  //=>12 13
```



在作用域链查找的过程中，如果找到WIN也没有这个变量，相当于给WIN设置了一个属性B (window.b=13)

【 问题：查找上级作用域 】

[全局变量和私有变量]

- 全局作用域下声明的变量为全局变量
- 私有作用域下声明的变量为私有变量

  注意：

  在私有作用域中，只有以下两种情况是私有变量

  A:声明过的变量(带VAR/FUNCTION)

  B:形参也是私有变量

  剩下的都不是自己私有的变量，都需要基于作用域链的机制向上查找

当前函数执行，形成一个私有作用域A，A的上级作用域是谁，和他在哪执行的没有关系，和他在哪创建（定义）的有关系，在哪创建的，它的上级作用域就是谁

```
var a = 12;
function fn(){
  // 形成一个私有作用域A
  console.log(a);
  // 和他在哪创建（定义）的有关系，在哪创建的，它的上级作用域就是谁
  // 这里是 window
}
function sum(){
  var a = 120;
  fn();
  // A的上级作用域是谁，和他在哪执行的没有关系
}
sum();     //=> 12
```

【 闭包 】

概念两大说法：

- 函数执形成一个私有的作用域，保护里面的私有变量不受外界的干扰，这种保护机制称之为“闭包”
- 市面上的开发者认为的闭包是：形成一个不销毁的私有作用域（私有栈内存）才是闭包

两大闭包化函数思想：

```
[ 柯理化函数 ]
function fn(){
  return function(){

  }
}
var f = fn();

[ 惰性函数 ]
var utils = (function () {
  return {

  }
})();
```

闭包的两大作用：

- 闭包具有“保护”作用：保护私有变量不受外界的干扰
- 闭包具有“保存”作用：形成不销毁的栈内存，把一些值保存下来，方便后面的调取使用

真实项目中为了保证JS的性能（堆栈内存的性能优化），应该尽可能的减少闭包的使用（不销毁的堆栈内存是耗性能的）

基于“保护作用下”，闭包的两大应用：

- 避免全局变量冲突，将这一部分内容封装到一个闭包中，让全局变量转换为私有变量

```
(function () {
  var n = 12;
  function fn() {

  }
  // ...
})();
```

- 封装类库插件的时候，把自己的程序都存放到闭包中保护起来，防止和用户的程序冲突

二、面向对象编程【OOP】

【 单例设计模式（仅仅是一种设计思想）】

表现形式：一个对象

```
var obj = {
  xxx:xxx,
  ...
}
/* 在单例设计模型中,OBJ不仅仅是对象名,它被称为“命名空间[NameSpace]”，把描述事务的属性存放到命名空间中，多个命名空间是独立分开的，互不冲突 */
```

作用：把描述同一件事务的属性和特征进行“分组、归类”(存储在同一个堆内存空间中)，因此避免了全局变量之间的冲突和污染

【 高级单例模式 】

在给命名空间赋值的时候，不是直接赋值一个对象，而是先执行匿名函数，形成一个私有作用域AA（不销毁的栈内存），在AA中创建一个堆内存，把堆内存地址赋值给命名空间

```
var nameSpace = (function () {
  var n = 12;
  function fn() {
    //...
  }
  function sum() {

  }
  return {
    fn: fn,
    sum: sum
  }
})();
```

【 面向对象编程 】

“对象、类、实例” 的概念

- 对象：万物皆对象
- 类：对象的具体细分
- 实例：类中具体的一个事物

[基于构造函数创建自定义类]
在普通函数执行的基础上“new xxx()”，这样就不是普通函数执行了，而是构造函数执行，当前的函数名称之为“类名”，接收的返回结果是当前类的一个实例

- 自定义类名，最好第一个单词首字母大写

- 这种构造函数设计模式执行，主要用于组件、类库、插件、框架等的封装，平时编写业务逻辑一般不这样处理

[JS中创建值的两种方式]

- 字面量表达式：var obj = {};
- 构造函数模式：var obj = new Object();
  不管是哪一种方式创造出来的都是Object类的实例，而实例之间是独立分开的，所以var xxx={} 这种模式就是JS中的单例模式

基本数据类型基于两种不同的模式创建出来的值是不一样的

- 基于字面量方式创建出来的值是基本类型值
- 基于构造函数创建出来的值是引用类型

```
var num1 = 12;
var num2 = new Number(12);

console.log(typeof num1);   //=>"number"
console.log(typeof num2);   //=>"object"

/* NUM2是数字类的实例，NUM1也是数字类的实例，它只是JS表达数字的方式之一，
都可以使用数字类提供的属性和方法 */
```

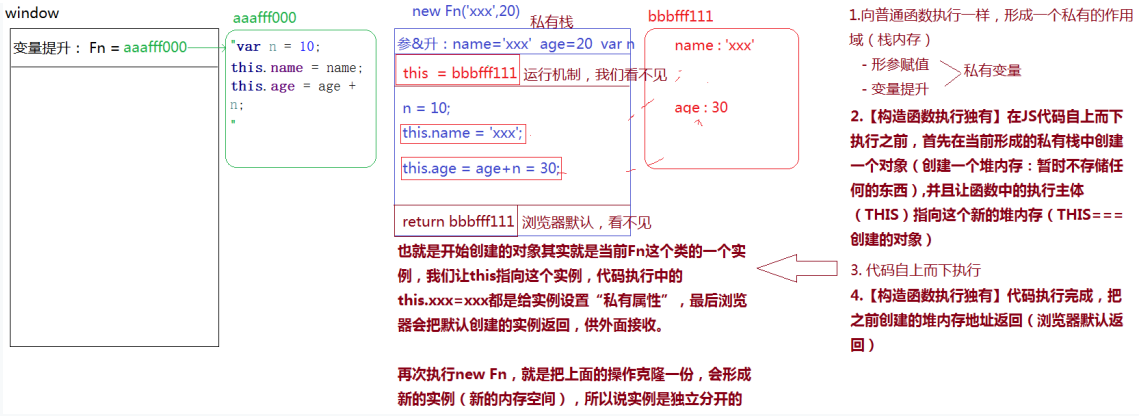
[构造函数执行机制]

普通函数执行

- 形成一个私有的作用域
- 形参赋值
- 变量提升
- 代码执行
- 栈内存释放问题

构造函数执行

```
function Fn(name, age) {
  var n = 10;
  this.name = name;
  this.age = age + n;
}
// 构造函数执行
var f1 = new Fn('xxx', 20);
var f2 = new Fn('aaa', 30);
```




```
console.log(f1 === f2);
// false：两个不同的实例（两个不同的堆内存地址）
console.log(f1.age);  //=>30
console.log(f2.name);   //=>'aaa'
console.log("name" in f1);
// true name&age在两个不同的实例都有存储，但是都是每个实例自己私有的属性
console.log(f1.n);
// undefined
// 只有this.xxx=xxx的才和实例有关系,n是私有作用域中的一个私有变量而已（this是当前类的实例）
```

构造函数执行，不写RETURN，浏览器会默认返回创建的实例，但是如果我们自己写了RETURN？

- return是的一个基本值，返回的结果依然是类的实例，没有受到影响
- 如果返回的是引用值，则会把默认返回的实例覆盖，此时接收到的结果就不在是当前类的实例了

  构造函数执行的时候，尽量减少RETURN的使用，防止覆盖实例

[三大检测]

```
function Fn() {
  var n = 10;
  this.m = n;
  // return;
  //=>这样RETURN是结束代码执行的作用，并且不会覆盖返回的实例
  // console.log(1);
}
var f = new Fn();
//=> new Fn;
// 在构造函数执行的时候，如果Fn不需要传递实参，我们可以省略小括号，意思还是创建实例（和加小括号没有区别）
console.log(f);
```

- instanceof：检测某一个实例是否隶属于这个类

```
console.log(f instanceof Fn);  //=>TRUE
console.log(f instanceof Array);   //=>FALSE
```

```
console.log(f instanceof Object);  //=>TRUE （万物皆对象：所有的对象，包含创建的实例都是Object的实例）
```

- in：检测当前对象是否存在某个属性（不管当前这个属性是对象的私有属性还是公有属性，只要有结果就是TRUE）

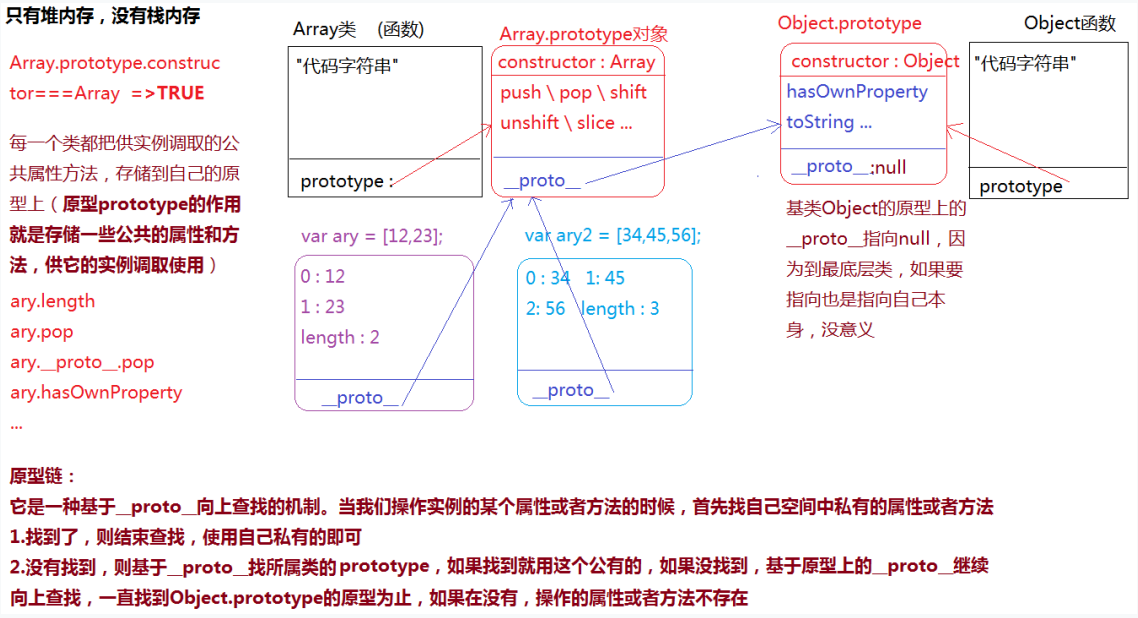
```
console.log('m' in f);  //=>TRUE
console.log('n' in f);  //=>FALSE
console.log('toString' in f);  //=>TRUE toString是它的公有属性
```

- hasOwnProperty：检测当前属性是否为对象的私有属性（不仅要有这个属性，而且必须还是私有的才可以）

```
console.log(f.hasOwnProperty('m'));  //=>TRUE
console.log(f.hasOwnProperty('n'));  //=>FALSE 连这个属性都没有
console.log(f.hasOwnProperty('toString'));  //=>FALSE 虽然有这个属性但是不是私有的属性
```

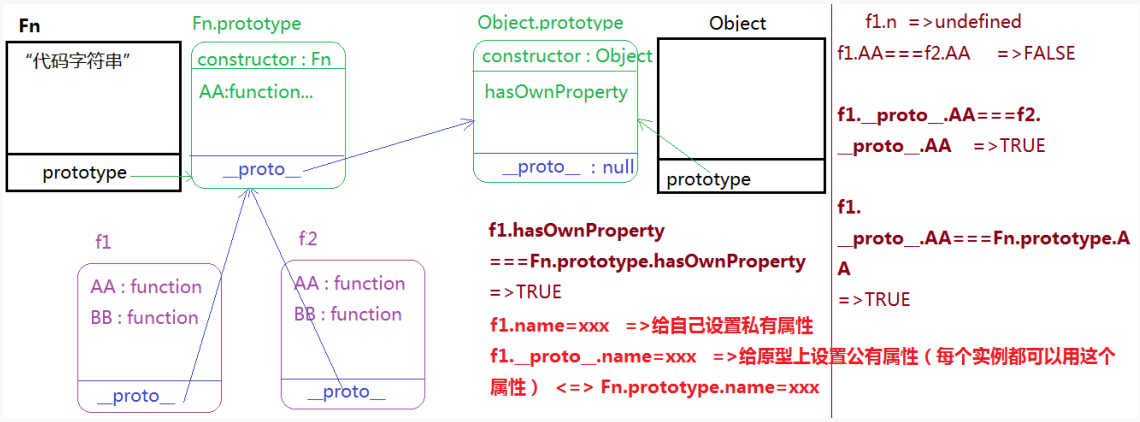
[原型（prototype）、原型链（***__proto__***）]

- 所有的函数数据类型都天生自带一个属性：prototype（原型），这个属性的值是一个对象，浏览器会默认给它开辟一个堆内存
- 在浏览器给prototype开辟的堆内存中有一个天生自带的属性：constructor，这个属性存储的值是当前函数本身
- 每一个对象都有一个***__proto__***的属性，这个属性指向当前实例所属类的prototype（如果不能确定它是谁的实例，都是Object的实例）




```
[ 例题1 ]
function Fn() {
  var n = 100;
  this.AA = function () {
    console.log(`AA[私]`);
  };
```

```
    this.BB = function () {
      console.log(`BB[私]`);
    };
  }
  Fn.prototype.AA = function () {
    console.log(`AA[公]`);
  };

  var f1 = new Fn;
  var f2 = new Fn;

  console.log(f1.n);
```



[函数的三大角色]

1、普通函数

- 堆栈内存释放
- 作用域链

2、类

- prototype：原型
- ***__proto__***：原型链
- 实例

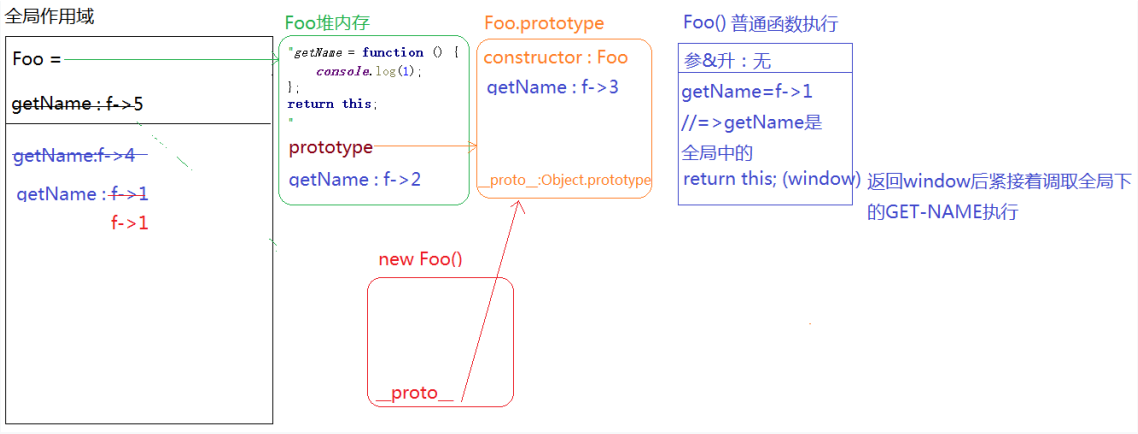
3、普通对象

- 和普通的一个OBJ没啥区别,就是对键值对的增删改查

三种角色间没有什么必然关系

```
  function Fn() {
    var n = 10;
    this.m = 100;
  }
  Fn.prototype.aa = function () {
    console.log('aa');
  };
  Fn.bb = function () {
    console.log('bb');
```

```
};
// 普通函数执行
Fn();  //=>this:window  有一个私有变量n  和原型以及属性bb没有关系

构造函数执行
var f = new Fn;  //=>this:f
console.log(f.n);  //=>undefined：n是私有变量和实例没有关系
console.log(f.m);  //=>100 实例的私有属性
f.aa();  //=>实例通过__proto__找到Fn.prototype上的方法
console.log(f.bb);  //=>undefined：bb是把Fn当做一个普通的对象设置的属性而已，和实例等没有半毛钱关系

普通对象
Fn.bb();



[ 例题（阿里经典面试题） ]
function Foo() {
    getName = function () {
        console.log(1);
    };
    return this;
}
Foo.getName = function BBB() {
    console.log(2);
};
Foo.prototype.getName = function AAA() {
    console.log(3);
};
var getName = function () {
    console.log(4);
};
function getName() {
    console.log(5);
}
Foo.getName();//=>2 把Foo当做一个对象，找Foo的私有方法执行
getName();//=>4 执行全局下的GET-NAME
Foo().getName();//=>1 先把FOO当做普通函数执行,执行返回的结果在调取GET-NAME执行
getName();//=>1 执行的依然是全局下的GET-NAME
console.log(new Foo.getName());;//=>A:(Foo.getName) =>new A()  =>2
new Foo().getName();//=>B:new Foo() =>B.getName() =>3
console.log(new new Foo().getName());
//=>C:new Foo() =>new C[Foo的实例].getName() =>D:C.getName =>new D(); =>3  (先计算new Foo()创建一个实例f，然后new f.getName()，先找到f.getName，在把这个函数new一下，最后其实相当于把f.getName当做一个类，返回这个类的一个实例)
```



[改变某一个函数中THIS关键字指向]

- call

```
window.name = 'Web前端';
let fn = function () {
    console.log(this.name);
};
let obj = {
    name: "OBJ",
    fn: fn
};
let oo = {name: "OO"};

fn();  //=> this:window "Web前端"
obj.fn();  //=> this:obj "OBJ"

/*
 *call原理：
 *[fn].call([this],[param]...)
 *fn.call：当前实例(函数FN)通过原型链的查找机制，找到Function.prototype上的call方法  =>function call(){[native code]}
 *fn.call()：把找到的call方法执行
 *
 *当call方法执行的时候，内部处理了一些事情
 *=>首先把要操作函数中的THIS关键字变为CALL方法第一个传递的实参值
 *=>把CALL方法第二个及第二个以后的实参获取到
 *=>把要操作的函数执行，并且把第二个以后的传递进来的实参传给函数
 */
fn.call(oo);  //=> this:oo
fn.call(obj,10,20,30);  //=>this:obj  改变fn的this指向，并把10,20,30作为参数传递给fn函数

/*
 *call细节：
 *非严格模式下，如果参数不传，或者第一个传递的是null/undefined，THIS都指向WINDOW
 *在严格模式下，第一个参数是谁，THIS就指向谁（包括null/undefined），不传THIS是undefined
 */
```

```
 /*
 *apply：和call基本上一模一样，唯一区别在于传参方式
 *非严格模式下，如果参数不传，或者第一个传递的是null/undefined，THIS都指向WINDOW
 *在严格模式下，第一个参数是谁，THIS就指向谁（包括null/undefined），不传THIS是undefined
 */
```

- apply

```
// 和call基本上一模一样，唯一区别在于传参方式
fn.call(obj,10,20)
fn.apply(obj,[10,20])
// APPLY把需要传递给FN的参数放到一个数组（或者类数组）中传递进去，虽然写的是一个数组，但是也相当于给FN一个个的传递
```

- bind

```
// 语法和call一模一样，唯一的区别在于立即执行还是等待执行
fn.call(obj,10,20)
// 改变FN中的THIS,并且把FN立即执行
fn.bind(obj,10,20)
// 改变FN中的THIS,此时的FN并没有执行（不兼容IE6~8）
```

三、正则（对象类型）

正则：是一个用来处理字符串的规则

- 正则只能用来处理字符串
- 处理一般包含两方面：
  - 验证当前字符串是否符合某个规则 “正则匹配”
  - 把一个字符串中符合规则的字符获取到 “正则捕获”

组成：修饰“元字符”、“符”两部分组成

创建正则的两种方式：

- 字面量方式

```
let reg1 = /^\d+$/g;
```

- 构造函数方式

```
let reg2 = new RegExp("^\\d+$", "g");
```

说明：正则两个斜杠之间包起来的都是“元字符”，斜杠后面出现的都是“修饰符”

常用的修饰符：

- i：ignoreCase 忽略大写小匹配
- m：multiline 多行匹配
- g：global 全局匹配

常用的元字符：

1、特殊元字符

- \d 0~9之间的一个数字
- \D 非0~9之间的任意字符
- \w “数字、字母、下划线”中的任意一个 =>/[0-9a-zA-Z_]/等价于\w
- \s 匹配任意一个空白字符（包括\t制表符[TAB键四个空格]）
- \b 匹配边界符 ‘zhu’(z左边和u右边就是边界) ‘zhu-feng’(z左边、u右边、f左边、g右边是边界)
- \n 匹配一个换行符
- \ 转义字符(把一个普通字符转义为特殊的字符,例如:\d，把有特殊含义的转换为普通意思，例如：. 此处的点就不是任意字符，而是一个小数点)
- . 不仅仅是小数点，代表除了\n以外的任意字符
- ^ 以某个元字符开头
- $ 以某个元字符结尾
- x|y x或者y中的任意一个(a|z…)
- [xyz] x或者y或者z中的任意一个
- [^xyz] 除了x\y\z以外的任意字符
- [a-z] 获取a-z中的任意一个字符([0-9] 等价于\d …)
- [^a-z] 除了a-z的任意字符
- () 正则分组
- (?:) 当前分组只匹配不捕获
- (?=) 正向预查
- (?!) 负向预查
- …

2、量词元字符：让其左边的元字符出现多少次

- `*` 出现零到多次
- ? 出现零到一次
- `+` 出现一到多次
- {n} 出现N次
- {n,} 出现N到多次
- {n,m} 出现N到M次

3、普通元字符

- 只要在正则中出现的元字符（在基于字面方式创建），除了特殊和有量词意义的以外，其余的都是普通元字符

中括号的一些细节：

- [xyz]，[^xyz]，[a-z]，[^a-z]
- 中括号中出现的元字符一般都是代表本身含义的
- 中括号中出现的两位数，不是两位数，而是两个数字中的任意一个

```
let reg = /^.+$/;
//=>一个正则设置了^和$，那么代表的含义其实就是只能是xxx
console.log(reg.test('n'));//=>true
console.log(reg.test('1'));//=>true
console.log(reg.test('nn'));//=>true
console.log(reg.test('\n'));//=>false
```

```
let reg = /^[.]+$/;
console.log(reg.test('n'));//=>false
console.log(reg.test('1'));//=>false
console.log(reg.test('nn'));//=>false
console.log(reg.test('\n'));//=>false
console.log(reg.test('...'));//=>true
```

```
let reg = /^[\d]+$/;
//=>\d在这里依然是0~9中的一个数字
console.log(reg.test('0'));//=>true
console.log(reg.test('d'));//=>false
```

```
let reg = /^[18]$/;
//=>不加^和$代表字符串中只要包含xxx即可
console.log(reg.test('18'));//=>false
console.log(reg.test('1'));//=>true
console.log(reg.test('8'));//=>true
```

```
let reg = /^[12-65]$/;
console.log(reg.test('13'));//=>false 不是12~65
console.log(reg.test('7'));//=>false  这个正则的意思是 1或者2~6或者5
console.log(reg.test('3.5'));//=>false
```

[例1：年龄：18~65之间]

```
let reg = /^((1[89])|([2-5]\d)|(6[0-5]))$/;
```

[例2：匹配 “[object AAA]”]

```
let reg = /^\[object .+\]$/;
```

分组的作用：

1、改变默认的优先级

```
let reg = /^18|19$/;
console.log(reg.test('18'));//=>true
console.log(reg.test('19'));//=>true
console.log(reg.test('1819'));//=>true
console.log(reg.test('189'));//=>true
console.log(reg.test('181'));//=>true
console.log(reg.test('819'));//=>true
console.log(reg.test('119'));//=>true

reg = /^(18|19)$/;
console.log(reg.test('18'));//=>true
console.log(reg.test('19'));//=>true
console.log(reg.test('1819'));//=>false
console.log(reg.test('189'));//=>false
console.log(reg.test('181'));//=>false
console.log(reg.test('819'));//=>false
console.log(reg.test('119'));//=>false
```

2、分组捕获

```
[ 例子：编写一个正则匹配身份证号码 ]
let reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/;
console.log(reg.exec('130828199012040617'));
/* EXEC实现的是正则捕获，获取的结果是一个数组，如果不匹配获取的结果是null，捕获的时候不仅把大正则匹配的信息捕获到，而且每一个小分组中的内容也捕获到了(分组捕获) : ["130828199012040617", "130828", "1990", "12", "04", "1", index: 0, input: "130828199012040617"] */

/*
 * 基于EXEC可以实现正则的捕获
 * 1.如果当前正则和字符串不匹配，捕获的结果是NULL
 * 2.如果匹配，捕获的结果是一个数组
 *   => 0:大正则捕获的内容
 *   => index:正则捕获的起始索引
 *   => input:原始操作的字符串
 *   => ...
 * 3.如果我们只在匹配的时候，想要获取大正则中部分信息，我们可以把这部分使用小括号包起来，形成一个分组，这样在捕获的时候，不仅可以把大正匹配的信息捕获到，而且还单独的把小分组匹配的部分信息也捕获到了(分组捕获)
 * 3.有时候写小分组不是为了捕获信息，只是为了改变优先级或者进行分组引用，此时我们可以在分组的前面加上“?:”，代表只去匹配，但是不把这个分组内容捕获
 * 4.“正则的捕获有懒惰性”：只能捕获到第一个匹配的内容，剩余的默认捕获不到
 *   => 解决正则捕获的懒惰性，我们需要加全局修饰符G（这个是唯一的方案，而且不加G不管用什么办法捕获，也都不能把全部匹配的捕获到）
 * let str = 'woaixuexi1314woaixuexi1315';
 * let reg = /\d+/g;
```

```
 * 5.正则捕获还具备贪婪性：每一次匹配捕获的时候，总是捕获到和正则匹配中最长的内容，例如：'2' 符合 \d+  '2018' 也符合 \d+，但是捕获的是最长的内容 '2018'...
 * let str = 'woaixuexi1314woaixuexi1315';
 * let reg = /\d+?/g;
 *  => 把问号放到量词元字符后面，代表的就不是出现零次或者一次了，而且取消捕获的贪婪性
 */

/*
* ?在正则中的作用
*   1.量词元字符：出现零次或者一次
*     /-?/ 让减号出现一次或者不出现
*
*   2.取消贪婪性
*     /\d+?/ 捕获的时候只捕获最短匹配的内容
*
*   3.?: 只匹配不捕获
*
*   4.?= 正向预查
*
*   5.?! 负向预查
*/

/*
* replace：实现正则捕获的方法（本身是字符串替换）
*/
let str = 'woaixuexi1314woaixuexi1315';
str = str.replace(/woaixuexi/g, 'woaixuexihaha');
console.log(str);//=>'woaixuexihaha1314woaixuexihaha'
```

3、分组引用

```
let reg = /^([a-z])([a-z])\2\1$/;
//=>正则中出现的\1代表和第一个分组出现一模一样的内容...
console.log(reg.test('oppo'));
console.log(reg.test('poop'));
```

常用正则表达式：

- 有效数字

```
let reg = /^[+-]?(\d|([1-9]\d+))(\.\d+)?$/;
```

- 电话(手机)号码

```
let reg = /^1\d{10}$/;
```

- 中文姓名

```
let reg = /^[\u4E00-\u9FA5]{2,}(·[\u4E00-\u9FA5]{2,})?$/;
```

- 邮箱：xxxx@xxx.xx.xx

```
let reg = /^\w+([-.]\w+)*@[A-Za-z0-9]+([-.][A-Za-z0-9]+)*(\.[A-Za-z0-9]+)
$/;
```

- 身份证号码

```
let reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/;
```

四、JS盒模型

JS盒子模型属性

> 在JS中通过相关的属性可以获取(设置)元素的样式信息,这些属性就是盒子模型属性（基本上都是有关于样式的）

1 、clientTop / Left / Width / Height

- clientWidth & clientHeight

  > 获取当前元素可视区域的宽高（内容的宽高+左右/上下padding）
  > 和内容是否有溢出无关（和是否设置了overflow:hidden也无关），就是我们自己设定的内容的宽高+padding
  > 获取当前页面一屏幕(可视区域)的宽度和高度

  ```
  document.documentElement.clientWidth || document.body.clientWid
  th
  document.documentElement.clientHeight || document.body.clientHe
  ight
  ```

- clientTop & clientLeft

  > 获取(上/左)边框的宽度

2 、offsetTop / Left / Width / Height / parent

- offsetWidth & offsetHeight

  > 在client的基础上加上border（和内容是否溢出也没有关系）

- offsetParent

> 当前盒子的父级参照物

- offsetTop / offsetLeft

> 获取当前盒子距离其父级参照物的偏移量(上偏移/左偏移) 当前盒子的外边框开始~父级参照物的内边框

> “参照物”：同一个平面中，元素的父级参照物和结构没有必然联系，默认他们的父级参照物都是BODY（当前平面最外层的盒子） BODY的父级参照物是NULL
> “参照物可以改变”：构建出不同的平面即可（使用z-Index，但是这个属性只对定位有作用），所以改变元素的定位(position:relative/absolute/fixed)可以改变其父级参照物

3 、scrollTop / Left / Width / Height

- scrollWidth & scrollHeight

> 真实内容的宽高（不一定是自己设定的值，因为可能会存在内容溢出，有内容溢出的情况下，需要把溢出的内容也算上）+ 左/上padding，而且是一个约等于的值 （没有内容溢出和client一样）
> 在不同浏览器中，或者是否设置了overflow:hidden都会对最后的结果产生影响，所以这个值仅仅做参考，属于约等于的值
> 获取当前页面的真实宽高（包含溢出的部分）

```
document.documentElement.scrollWidth || document.body.scrollWid
th
document.documentElement.scrollHeight || document.body.scrollHe
ight
```

- scrollTop / scrollLeft

> 滚动条卷去的宽度或者高度
> 最小卷去值：0
> 最大卷去值：真实页面的高度 - 一屏幕的高度
> document.documentElement.scrollHeight - document.documentElement.clientHeight

通过JS盒模型属性获取值的特点：

- 获取的都是数字不带单位
- 获取的都是整数，不会出现小数（一般都会四舍五入，尤其是获取的 偏移量）
- 获取的结果都是复合样式值（好几个元素的样式组合在一起的值）

获取元素具体的某个样式值

1、[元素].style.xxx 操作获取（不常用，因为样式基本都是写在样式表内）

> 弊端：只能获取所有写在元素行内上的样式(不写在行内上,不管写没写都获取不到,真实项目中很少会把样式写在行内上)

2、获取当前元素所有经过浏览器计算的样式

> 经过计算的样式：只要当前元素可以在页面中呈现（或者浏览器渲染它了），那么它的样式都是被计算过的

五、定时器、异步、动画库

定时器

> 设定一个定时器，并且设定了等到的时间，当到达执定的时间，浏览器会把对应的方法执行
>
> 常用定时器:

```
setTimeout([function],[interval])
setInterval([function],[interval])
// [function]：到达时间后执行的方法（设置定时器的时候方法没有执行，到时间浏览器帮我们执行）
// [interval]：时间因子（需要等到的时间 MS）
// setTimeout是执行一次的定时器，setInterval是可执行多次的定时器
```

> 清除定时器

```
clearTimeout / clearInterval
// 这两个方法中的任何一个都可以清除用任何方法创建的定时器
// 设置定时器会有一个返回值，这个值是一个数字，属于定时器的编号，代表当前是第几个定时器（不管是基于setTimeout还是setInterval创建定时器，这个编号会累加）
// clearTimeout([序号])/clearInterval([序号])：根据序号清除浏览器中设定的定时器
let count = 0;
let timer = setInterval(() => {
     count++;
     console.log(count);
     if (count === 5) {
         //=>清除定时器
         clearTimeout(timer);
     }
 }, 1000);
```

JS中的同步编程和异步编程：

同步编程：任务是按照顺序依次处理，当前这件事没有彻底做完，下一件事是执行不了的；
异步编程：当前这件事没有彻底做完，需要等待一段时间才能继续处理，此时我们不等，继续执行下面的任务，当后面的任务完成后，再去把没有彻底完成的事情完成

JS中的异步编程:

- 所有的事件绑定都是异步编程 xxx.onclick=function(){}
- 所有的定时器都是异步编程 setTimeout(function(){},1000)
- AJAX中一般都使用异步编程处理
- 回调函数也算是异步编程

…

动画（原则：能用CSS3解决的动画绝不用JS）

CSS3 动画

- transition过渡动画
- animation帧动画
- transform是变形不是动画

JS动画

- 定时器
- requestAnimationFrame(JS中的帧动画)
- canvas动画

FLASH动画（ActionScript）

六、DOM事件

事件就是一件事情或者一个行为（对于元素来说，它的很多事件都是天生自带的），只要我们去操作这个元素，就会触发这些行为

【事件绑定】

给元素天生自带的事件行为绑定方法，当事件触发，会把对应的方法执行
DOM0级事件绑定：[element].onxxx=function () {};
DOM2级事件绑定：[element].addEventListener('xxx', function(){}, false);
目的：给当前元素的某个事件绑定方法（不管是基于DOM0还是DOM2），都是为了触发元素的相关行为的时候，能做点事情（也就是把绑定的方法执行）；“不仅把方法执行了，而且浏览器还给方法传递了一个实参信息值 ==> 这个值就是事件对象”

```
box.onclick = function (e) {
    //=>定义一个形参 e 用来接收方法执行的时候，浏览器传递的信息值（事件对象：MouseEvent鼠标事件对象、KeyboardEvent键盘事件对象、Event普通事件对象...）
    console.log(e);
```

```
    //=>事件对象中记录了很多属性名和属性值，这些信息中包含了当前操作的基础信息，例如：鼠标点击位置的X/Y轴坐标，鼠标点击的是谁（事件源）等信息
    如：
    [MouseEvent]
    // e.target：事件源（操作的是哪个元素）
    // e.clientX / ev.clientY ：当前鼠标触发点距离当前窗口左上角的X/Y轴坐标
    // e.pageX / ev.pageY：当前鼠标触发点距离BODY(第一屏幕)左上角的X/Y轴坐标
    // e.preventDefault()：阻止默认行为
    // e.stopPropagation()：阻止事件的冒泡传播
    // e.type：当前事件类型
}
```

【元素天生自带的事件】

1、鼠标事件

- click：点击 (PC端是点击，移动端的click代表单击[移动端使用click会有300MS延迟的问题])
- dblclick：双击
- mouseover：鼠标经过
- mouseout：鼠标移出
- mouseenter：鼠标进入
- mouseleave：鼠标离开
- mousemove：鼠标移动
- mousedown：鼠标按下（鼠标左右键都起作用，它是按下即触发，click是按下抬起才会触发，而且是先把down和up触发，才会触发click）
- mouseup：鼠标抬起
- mousewheel：鼠标滚轮滚动
- …

2、键盘事件

- keydown：键盘按下
- keyup：键盘抬起
- keypress：和keydown类似，只不过keydown返回的是键盘码，keypress返回的是ASCII码值
- input：由于PC端有实体物理键盘，可以监听到键盘的按下和抬起，但是移动端是虚拟的键盘，所以keydown和keyup在大部分手机上都没有，我们使用input事件统一代替他们（内容改变事件）
- …

3、表单元素常用的事件

- focus：获取焦点
- blur：失去焦点
- change：内容改变

- …

4、其它常用事件

- load：加载完成
- unload
- beforeunload
- scroll：滚动条滚动事件
- resize：大小改变事件 window.onresize=function(){} 当浏览器窗口大小发生改变，会触发这个事件，执行对应的事情
- …

5、移动端手指事件

- 分单手指/多手指（具体看文档）

6、H5中的AUDIO/VIDEO音视频事件

- canplay：可以播放（播放过程中可能出现由于资源没有加载完成，导致的卡顿）
- canplaythrough：资源加载完成，可以正常无障碍播放

【事件的默认行为】

事件本身就是天生就有的，某些事件触发，即使你没有绑定方法，也会存在一些效果，这些默认的效果就是“事件的默认行为”

- A标签的点击操作
- INPUT标签
- SUBMIT按钮
- …

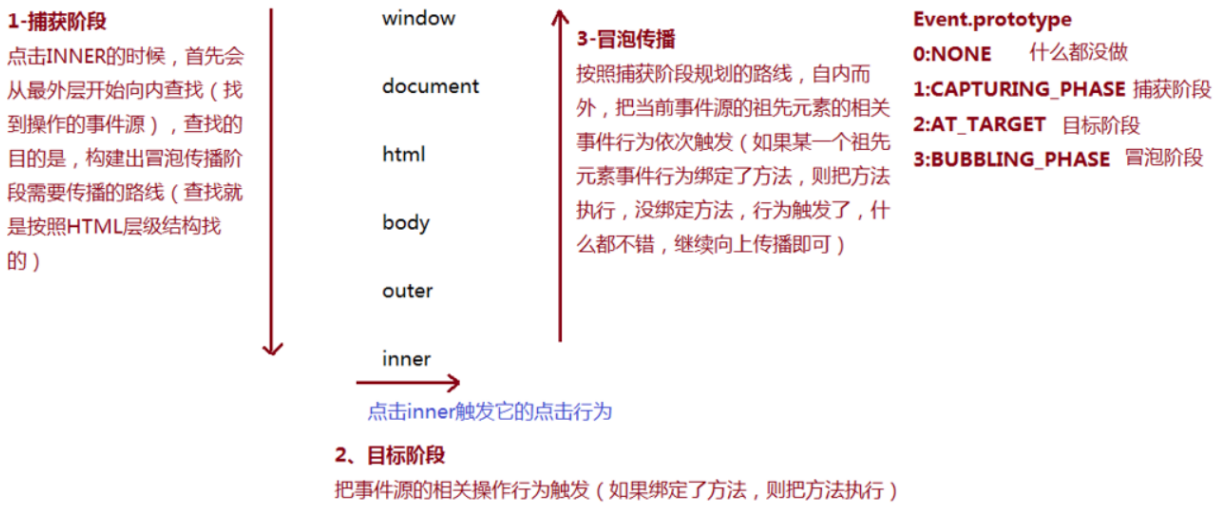
如何阻止默认行为（a标签）

```
// => 在结构中阻止
<a href="javascript:;">阻止a标签默认行为</a>
// => javascript:void 0/undefined/null...;
// => 在JS中阻止
<a id="link">阻止a标签默认行为</a>
// => 给其CLICK事件绑定方法，当我们点击A标签的时候，先触发CLICK事件，其次才会执行自己的默认行为
link.onclick = function (e) {
    e = e || window.event;
    return false;
};
```

【事件的冒泡传播机制】

> 触发当前元素的某一个事件（点击事件）行为，不仅当前元素事件行为触发，而且其祖先元素的相关事件行为也会依次被触发

```
window.onclick = function () {
    console.log('window');
};

document.onclick = function () {
    console.log('document');
};

document.documentElement.onclick = function () {
    console.log('html');
};

document.body.onclick = function () {
    console.log('body');
};

outer.onclick = function () {
    console.log('outer');
};

inner.onclick = function () {
    console.log('inner');
};
```



> xxx.onxxx=function(){} DOM0事件绑定，给元素的事件行为绑定方法，这些方法都是在当前元素事件行为的冒泡阶段(或者目标阶段)执行的

> xxx.addEventListener('xxx',function(){},false) 第三个参数FALSE也是控制绑定的方法在事件传播的冒泡阶段(或者目标阶段)执行；只有第三个参数为TRUE才代表让当前方法在事件传播的捕获阶段触发执行（这种捕获阶段执行没啥实际意义，项目中不用）

【不同浏览器对于最外层祖先元素的定义是不一样的】

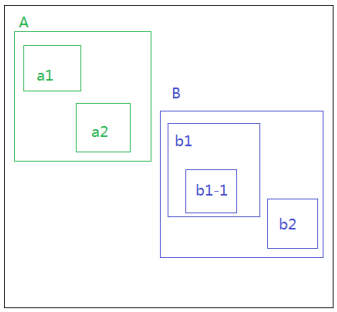
- 谷歌：window->document->html->body…
- IE高：window->html->body…
- IE低：html->body…

【事件委托（事件代理）】

> 利用事件的冒泡传播机制，如果一个容器的后代元素中，很多元素的点击行为（其它事件行为也是）都要做一些处理，此时我们不需要在像以前一样一个个获取一个个的绑定了，我们只需要给容器的CLICK绑定方法即可，这样不管点击的是哪一个后代元素，都会根据冒泡传播的传递机制，把容器的CLICK行为触发，把对应的方法执行，根据事件源，我们可以知道点击的是谁，从而做不同的事情即可

a1.onclick=function...
a2.onclick=function...
....
一个个获取元素，然后再一个个的绑定事件的方式，不仅麻烦，而且性能消耗也是相对比较大的

container.onclick = function(ev){ ... }



点击a1，不仅触发a1的点击行为，而且A以及CONTAINER的点击行为都会被依次触发
点击b1-1，也是一样的，不仅b1-1的点击行为触发，而且b1/B/CONTAINER的点击行为也都会被依次触发
...
在此过程中，只给CONTAINER的点击事件行为绑定方法即可，不管点击的是后代中的谁，绑定的方法都会执行，而且EV事件对象中记录了事件源(EV.TARGET)

```
let target=ev.target || ev.srcElement;
if(target.className==='a1'){
    ....
}else ...
```

事件委托这种处理方式比一个个的事件绑定，性能上提高50%左右，而且需要操作的元素越多，性能提高越大