**1. 正则表达式(Regular Expression):**

其它简写：

要匹配一位小写字母：[a-z]

要匹配一位大写字母：[A-Z]

要匹配一位字母： [A-Za-z]

要匹配一位字母或数字：[0-9A-Za-z]

要匹配一位汉字：[\u4e00-\u9fa5]

正则表达式语法为四种最常用的字符集定义了最简化写法，称为预定义字符集。

包括：

要匹配一位数字： \d 等效于 [0-9]

要匹配一位字母、数字或\_: \w 等效于 [0-9A-Za-z\_]

要匹配一位空字符：\s 可匹配 空格、制表符Tab等空白

要匹配所有文字（通配符）： .

所以，手机号规则可进一步简写为：

1[3-9]\d\d\d\d\d\d\d\d\d

1[3-9]{9}

数量词包括两大类：

1. 有明确数量边界的数量词

字符集{n} 表示字符集必须重复n次，不能多也不能少

字符集{n,m} 表示字符集至少重复n次，最多重复m次，

比如: \d{4,6} 表示4到6位数字

字符集{n,} 表示字符集匹配的内容至少重复n次，多了不限，比如：\d{6,} 表示6位以上数字

2. 没有明确数量边界的数量词

\* 可有可无，多了不限

? 可有可无，最多一次

+ 至少一次，多了不限

如果只希望匹配特殊位置上的关键词时，就可用特殊符号表示特殊位置。

包括：

^ 表示字符串开头

$ 表示字符串结尾

\b 表示单词边界，可匹配：空格，标点符号，字符串开头和结尾等可将一个单词与其它单词分割开的符号。

屏蔽敏感词汇： ([我卧]|wo)\s\*([草艹槽操]|cao)

手机号： ((\+86|0086)\s+)?1[3-9]\d{9}

身份证号： \d{15}(\d\d[0-9x])?

开头和结尾的空格：^\s+|\s+$

**day02**

**1. String中的正则API:**

**查找:**

(1). 查找一个固定的关键词出现的位置:

var i=str.indexOf("关键词"[,starti])

在str中从starti位置开始找下一个"关键词"的位置

如果省略starti，则默认从0位置开始向后找

返回: “关键词”第一字所处位置的下标

如果找不到，返回-1

问题: 只支持查找一个固定不变的关键词

(2). 用正则表达式查找一个关键词的位置:

var i=str.search(/正则/i);

在str中查找符合"正则"要求的敏感词

返回值: 关键词的位置

如果找不到，返回-1

问题: 正则默认区分大小的！

解决: 在第2个/后加后缀i，表示ignore，意为:忽略大小写

问题: 只能获得关键词的位置，不能获得关键词的内容

解决: match

(3). match:

. 查找一个关键词的内容和位置:

var arr=str.match(/正则/i);

在str中查找符合"正则"要求的敏感词的内容和位置

返回值: 关联数组arr:[ "0":"关键词", "index": 下标i ]

如果找不到: 返回null

问题: 只能查找第一个关键词，不能查找更多关键词

. 查找所有关键词的内容:

var arr=str.match(/正则/ig) //g global 全局/全部

返回值: 保存多个关键词内容的数组

问题: 只能找到内容，无法获得位置

(4). 即获得每个关键词的内容，又获得每个关键词的位置

reg.exec()

**替换:**

将找到的关键词替换为指定的新值

2种:

. 简单替换: 将所有关键词替换为统一的新值

str=str.replace(/正则/ig,"新值")

查找str中所有符合“正则”要求的关键词，都替换为统一的新值。

问题: 所有字符串API都无权修改原字符串，只能返回新字符串。

解决: 如果希望替换原字符串，就必须自己手动赋值回原变量中。

. 高级替换: 根据每次找到的关键词不同，动态选择要替换的新值

str=str.replace(/正则/ig, function(kword){

//根据kword的不同，动态生成新值

return 新值;

})

查找str中所有符合“正则”要求的关键词

每找到一个关键词，就自动调用一次回调函数。

每次调用回调函数时自动传入本次找到的关键词给kword

回调函数中根据每次收到kword的不同，动态返回新值给replace函数

replace函数自动将回调函数返回的新值，替换到字符串原关键词位置。

|  |
| --- |
| 示例：(将首字母改为大写)  var str="you can you use";  str=str.replace(/\b[a-z]/ig,function(kword){  return kword.toUpperCase();  });  console.log(str); |

衍生操作:

删除关键字: 其实将关键词替换为""

|  |
| --- |
| 删除开头和结尾的空格：uname.trim()  等同于：function trim(str){  return str.replace(/^\s+|\s+$/g,"");  }  console.log(trim(uname)); |

**切割:**

将一个字符串，按指定的切割符，切割成多段子字符串

2种:

简单切割: 切割符是固定的

var arr=str.split("切割符")

将str按指定的"切割符"，切割为多段子字符串，保存在数组arr中

固定套路: 打散字符串为字符数组

var chars=str.split("")

排序反转：

chars.reverse(); 数组反转

str=chars.join(""); 将数组转换为字符串

复杂切割: 切割符不是固定的

var arr=str.split(/正则/i)

**2. RegExp对象:**

什么是: 保存一条正则表达式，并提供了用正则表达式执行验证和查找的API 的对象

何时: 只要在程序中保存和使用正则表达式，都要用RegExp对象

如何:

创建:

1. 用/创建(直接量): var reg=/正则/ig

何时: 如果正则表达式是固定不变的

在/…/之间是正则表达式区域，不能使用js语句。

2. 用new创建: var reg=new RegExp("正则","ig")

何时: 如果正则表达式需要在运行时才动态生成时(可以用JS函数)

API:

(1). 验证:

var bool=reg.test(str)

用正则reg，检测str是否符合正则的格式要求

问题: 正则默认只要找到部分内容与正则匹配就返回true

解决: 必须从头到尾完整匹配，才返回true

只要验证，必须前加^后加$，表示从头到尾的意思

(2). 查找:

获取每个关键词的内容和位置:

var arr=reg.exec(str)

在str中查找下一个符合reg要求的关键词的内容和位置

如果反复调用reg.exec()，只要返回值不为null，则自动会向后找下一个。

返回值: arr:[ "0":"关键词", "index": 关键词位置 ]

|  |
| --- |
| 示例：  <script>  var str="明明喜欢我，缺不告诉我，我想静静，静静是谁？明明是谁？";  var reg=/明明|静静/g;  do{  var arr=reg.exec(str);  if(arr!=null){  console.log(`在${arr["index"]}处找到${arr["0"]}`);  }  }while(arr!=null)  </script> |

**day03**

**1. Function**

什么是函数: 封装一项任务的步骤清单代码段的对象，再起一个名字。

为什么: 重用！

何时: 只要一段代码可能被反复使用，都要创建函数。

如何:

创建函数:

(1) 用声明方式创建函数:

function 函数名(形参列表){

函数体

return 返回值

}

问题: 所有声明都会被提前！打乱原有的执行顺序

复习: 声明提前(hoist): 在程序开始执行前，会先将var声明的变量和function声明的函数，提前到当前作用域的顶部集中创建。赋值留在原地。

—— 广受诟病的缺陷！

(2) 用赋值方式创建函数:

var函数名=function (形参列表){

函数体

return 返回值

}

优点: 仅函数名部分被提前，函数定义留在原地顺序执行

(3) 用new: —— 几乎不用

var 函数名=new Function("形参","形参",...,"函数体")

**重载(overload):**

什么是: 相同函数名，不同形参列表的多个函数，在调用时，可自动根据传入实参的不同，选择对应的函数调用。

为什么: 减少函数名的数量，减轻维护的负担

何时: 一件事，可能根据传入的参数不同，执行不同的操作时

如何:

问题: js语法默认不支持重载

因为js不允许多个同名函数同时存在

如果存在，最后一个函数会覆盖之前所有

解决: 使用arguments对象变通实现

什么是arguments对象: 每个函数内自动创建的，准备接受所有传入函数的实参的类数组对象。

自动创建: 不用自己创建，就可直接使用

接受所有实参: 可以根据接受实参的长度来判断执行方式。

|  |
| --- |
| function pay(){  if(arguments.length==0){ //没有传入参数  console.log("手机支付...");  }else if(arguments.length==1){ //传入1个参数  console.log(`您支付的现金是${arguments[0]}`);  }else{  console.log(`从您卡号为${arguments[0]},扣款为${arguments[1]}`);  }  } |

类数组对象: 长的像数组的对象。

vs 数组: 相同: 1. 下标 2. length 3. 遍历

不同: 类型不同，API不通用

何时: 只要不确定将来传入函数的参数个数时，就不定义形参列表，而是用arguments接住所有的实参值。

**匿名函数:**

什么是: 定义函数时，不写名的函数

何时: 如果一个函数只用一次，不会反复使用，就不起函数名

为什么: 节约内存！——使用一次后，自动释放！

如何:

2种场景:

回调函数: 自己定义的函数，自己不调用，而是传给其它对象或函数，被别人调用。

arr.sort(function(a,b){return a-b})

xhr.onreadystatechange=function(){ ... }

pool.query(sql,[参数],function(err,result){ ... })

匿名函数自调: 定义函数后，立刻自己调用自己

为什么: 避免使用全局变量，造成全局污染

何时: 所有js代码都应该封装在一个匿名函数自调中。（\*\*\*\*\*\*\*\*\*\*）

如何:

(function(){

...

})()//调用后，匿名函数内的局部变量都释放了

**作用域(scope)和作用域链(scope chain)**

什么是作用域(scope):

用法: 作用域就是一个变量的可用范围

本质: 保存变量的对象。

为什么: 防止不同范围的变量间互相干扰

包括: 2级

1. 全局作用域对象: window

专门保存所有全局变量和函数

所有全局变量都默认保存在window对象中

特点: 1. 可反复使用 2. 随处可用

2. 函数作用域对象:

专门保存函数内局部变量的对象

特点: 1. 不能反复使用 2. 仅函数内可用

一个函数的执行过程:

函数开始调用时:

为本地调用临时创建函数作用域对象

并在函数作用域对象中，临时创建局部变量。

函数执行过程中:

取出函数中每条语句执行

执行都作用域临时创建的局部变量上

函数执行后:

函数作用域对象，释放！

导致: 所有本次用过的局部变量，一同释放！

作用域链(scope chain):

什么是作用域链：由多级作用域，逐级引用，形成的链式结构，就称为作用域链。

为什么: 防止函数调用时，自己缺变量。

作用域链保存着一个函数可用的所有变量

作用域链控制着变量的使用顺序: 先局部后全局。

**闭包(closure)**

什么是闭包:

用法: 即重用变量，又保护变量的一种编程方法（3步）

本质: 外层函数留给内层函数的自己的作用域对象

为什么: 全局变量和局部变量都有不可兼得的优缺点:

全局变量: 优: 可重用！ 缺: 随处可被篡改

局部变量: 优: 不会被篡改！缺: 不可重用

何时: 如果即想重用一个变量，又想保护变量不被篡改。

如何实现闭包:

外层函数包裹要保护的变量和内层函数；

外层函数返回内层函数；

使用者调用外层函数，获得内层函数保存在变量中反复使用。

|  |
| --- |
| 示例：  //1. 用外层函数包裹内层函数和受保护变量  function 父母(){  var total=1000;  //2. 外层函数将内层函数对象return到外部  return function(x){  total-=x;  document.write(`花了${x},还剩${total}<br>`)  }  }  //3. 调用外层函数，获得内层函数对象，保存在外部变量中  var 孩子=父母();  /\*孩子=function(x){  total-=x;  document.write(`花了${x},还剩${total}<br>`)  }\*/  console.dir(孩子);//[[Scopes]]  孩子(100);//900  孩子(100);//800  total=0;//试图篡改——失败，变量被保护，但是可以重复使用  孩子(100);//700 |

**day04**

**一. OOP面向对象**

什么是对象: 程序中，描述现实中一个具体事物的程序结构。

本质：内存中储存多个数据的内存空间，再起一个名字。

为什么: 便于维护

什么是面向对象: 程序中，都是先将多个事物的属性和功能，封装为多个对象，再按需调用对象的成员

何时: 今后，几乎所有程序，都用面向对象方式实现的

如何: 封装，继承，多态——三大特点

**1.封装**：创建一个对象，集中保存一个具体事物的属性和功能。

为什么: 便于大量数据维护

何时: 只要使用面向对象编程，都要先封装对象，再按需使用对象的成员

如何: 3种方法

**(1)** 用{}: ——{}是一个动作，创建对象

var obj={

属性名: 值,

... : ... ,

方法名: function(){

... this.属性名...

}

}

访问对象的成员: 成员=属性+方法

属性其实就是保存在对象中的变量

对象.属性

方法其实就是保存在对象中的函数

对象.方法()

this:

什么是: 在函数调用时,自动指向.前的对象的关键词

为什么: 避免在对象的方法中写死对象名，形成紧耦合(如果对象变量名发生变化，如果内部不跟着改，就会出错)。

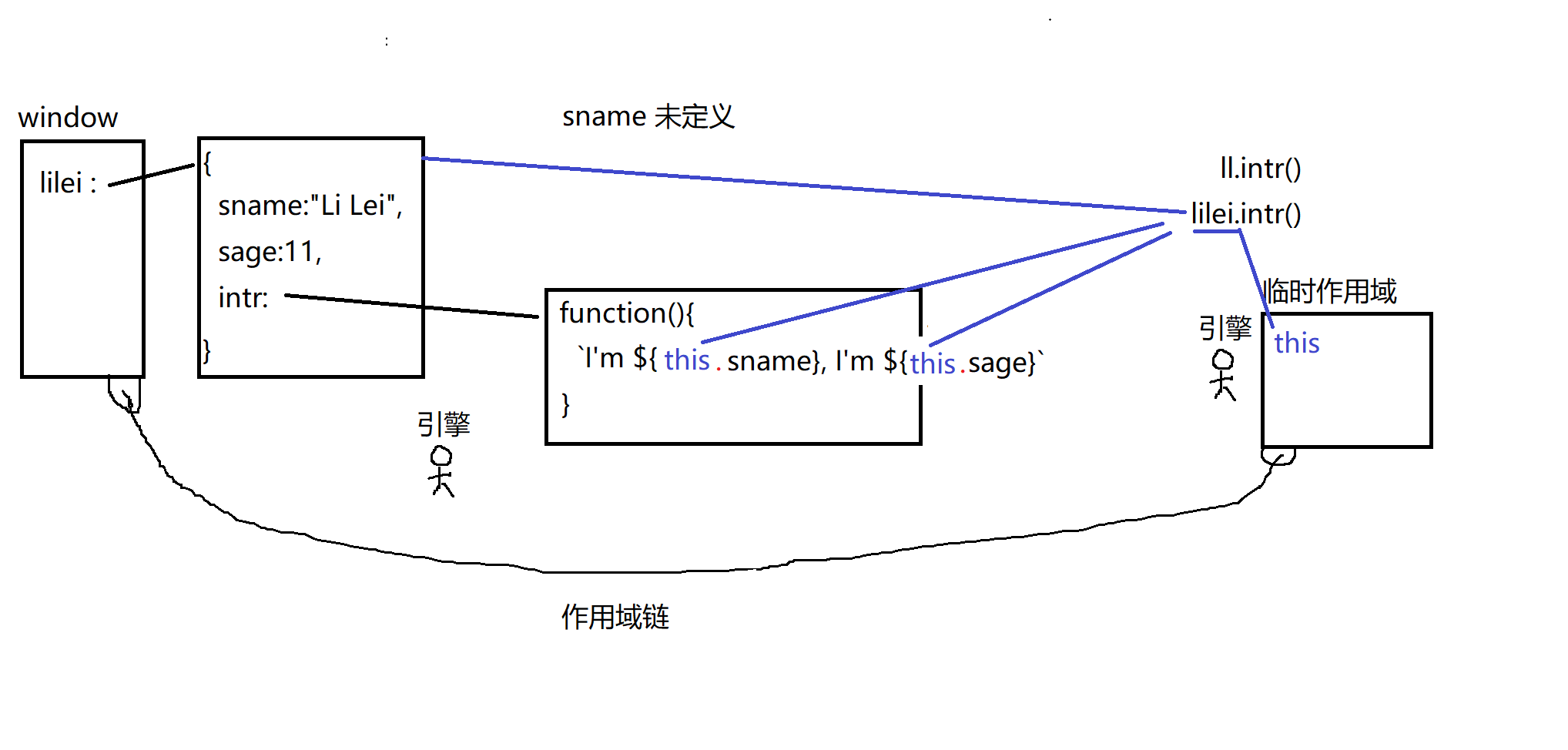
何时: 只要对象自己的方法，想访问自己的成员时，都要加this。

强调: 笔试时:

this和定义在哪儿无关，**只和调用时 点（.）前的对象有关；**

全局函数调用**,没有点（.）时, this默认指window。**

注意：要判断this，一定不要看保存在哪里，只看合适调用，以及调用时 . 前使用的是谁。



**(2)** 用new创建: 2步:

var obj=new Object(); =>obj{};

obj.属性名=值;

... = ...;

obj.方法名=function(){ ... this.属性名 ...}

揭示: js中一切对象底层，都是关联数组。

1. 访问不存在的成员，不报错，返回undefined；

2. 强行给不存在的成员赋值，不报错，而是强行赋值新成员；

3. 同样可用for in循环遍历一个对象中的所有成员；

4. 访问下标(2种):

obj["下标"] 如果属性名需要动态获得；

|| 完全等效

obj.下标 如果属性名是写死的。

|  |
| --- |
| //克隆:  function clone(obj){  var newObj={};//{}创建对象的意思  //遍历原对象lilei中每个属性  for(var key in obj){//key:"属性名"  //将lilei中的key属性，强行添加到lilei2中，并也保存为相同属性名  newObj[key]=obj[key];  //lilei2["key"]=lilei["key"]; ×不可用（变量放在双引号中变成了普通的字符串）  // ||  //lilei2.key=lilei.key; ×不可用（和上面同等效果）  }  return newObj;  }  var lilei2=clone(lilei);  console.log(lilei); //长得一样  console.log(lilei2);//长得一样  console.log(lilei==lilei2);//false =>//说明不是一个人！  // var lilei2=lilei; console.log(lilei==lilei2);//true =>//说明是同一个人！只复制了地址 |

问题: 一次只能创建一个对象，如果反复创建多个相同结构的对象时，代码会很冗余。

DRY原则=>Don't Repeat Youself（程序中不要复制自己的代码）

解决: 用构造函数创建对象:

**(3)** 构造函数（constructor）:

什么是构造函数: 描述同一类型所有对象的统一结构的函数。

何时: 反复创建多个相同结构的对象

如何:

第一步：定义构造函数:

function 类型名(形参1,形参2,...){ //类型名一般首字母大写

this.属性名=形参1;

this. ... =形参2;

this.方法名=function(){

... this.属性名

...

}

}

第二步：反复调用构造函数，创建多个相同类型的对象

var lilei=new 类型名(实参,...);

new 4件事:

创建一个新的空对象: {}

让新对象自动继承构造函数的原型对象；

用新对象调用构造函数:

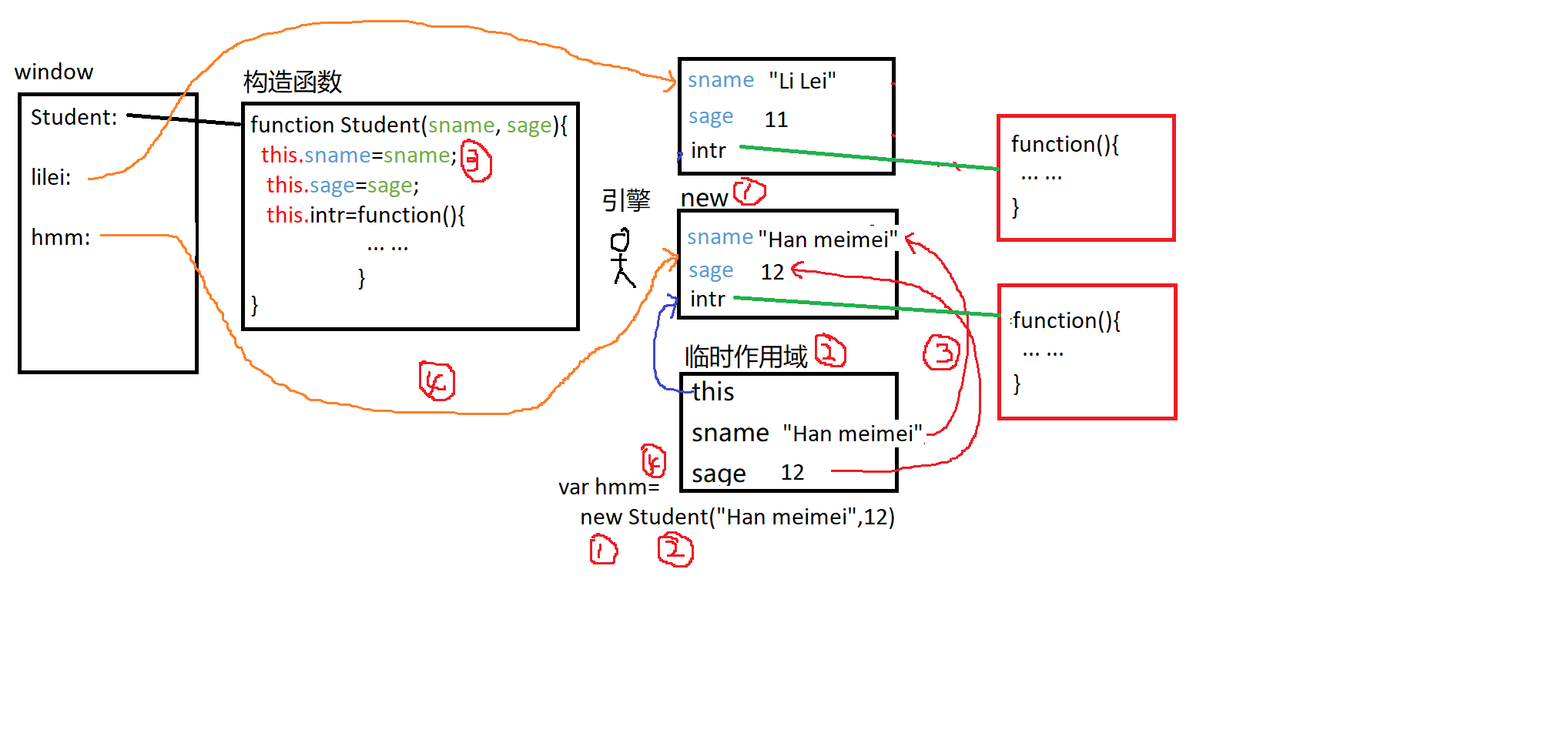
**自动将构造函数中的this->new新对象(只要有new就会吸引住this)**

通过强行赋值新属性的方式，自动添加规定的属性到新对象中，并将参数值，保存到新对象的对应属性中

new将新对象的地址返回给变量

构造函数优点: 重用结构

缺点: 无法节约内存!



**2.继承:**

什么是: 父对象的成员，子对象无需重复创建就可直接使用

为什么: 重用代码，节约内存

何时: 只要在多个子对象中都包含部分相同的成员时。

如何:

/\*\*\*\*\*\* 不用你做 \*\*\*\*\*\*\*/

js中的继承都是继承原型对象(prototype)

什么是原型对象: 集中保存所有子对象共有成员的父对象

创建原型对象: 不用自己创建！买一赠一

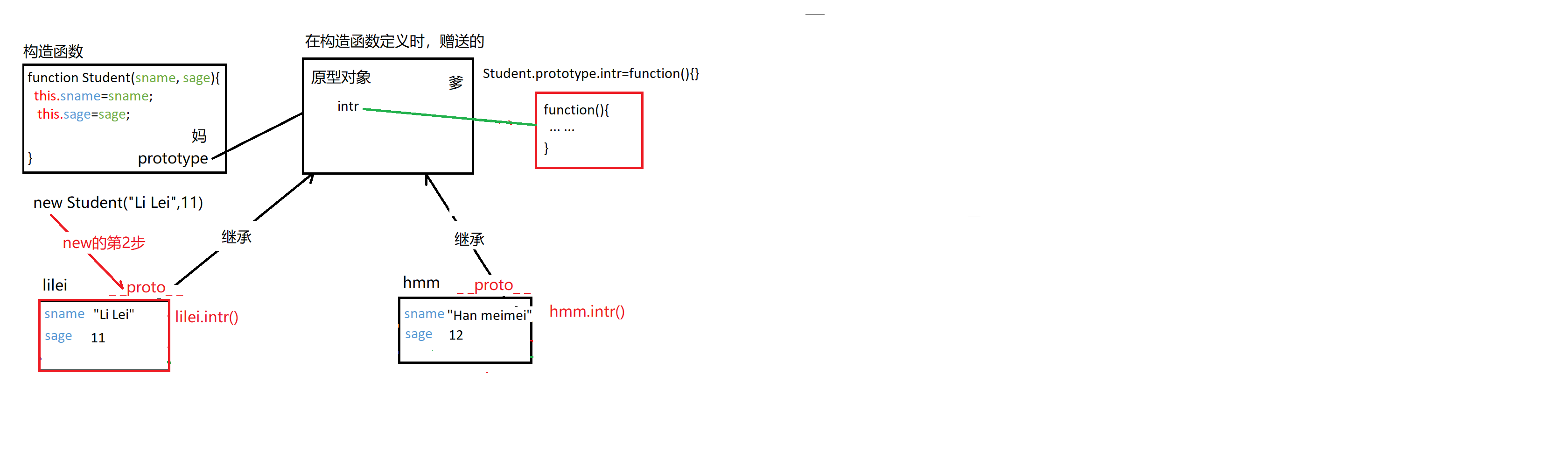
每创建一个构造函数，就附赠一个空的原型对象

构造函数.prototype={ 原型对象 }

何时继承（自动继承）:

new的第2步: 让新对象自动继承构造函数的原型对象: 新对象.\_ \_proto\_ \_==构造函数.prototype

示例：lilei.\_ \_proto\_ \_==Student.prototype



|  |
| --- |
| 示例：  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  }  Student.prototype.intr=function(){  console.log(`I'm ${this.sname},I'm ${this.sage}`);  }  var lilei=new Student("lilei",11);  lilei.intr();  var hanmeimei=new Student("hanmeimei",12);  hanmeimei.intr();  console.log(lilei. proto\_\_==Student.prototype);//true |

自有属性和共有属性:

自有属性: 保存在当前对象本地，仅归当前对象自有的属性

共有属性: 保存在原型对象中，归多个子对象共有的属性

取值/访问时: 对象.成员名

修改时: 修改自有属性: 子对象.属性=值

修改共有属性: 原型对象.共有属性=值

如果强行用子对象.共有属性=值

后果: 为子对象自己添加一个同名的自有属性，从此子对象与父对象再无关系。——错误！

内置对象的原型对象:

内置对象: ES标准中规定的，浏览器已经实现了的，咱们可以直接使用的对象

js内置对象: 11个:

String Number Boolean

Array Date RegExp Math

Error

Function Object

global

其实，每种内置对象(除Math和global外)都是一种类型:

都由两部分组成:

1. 构造函数: 创建该类型的子对象

2. 原型对象: 保存该类型共有的函数

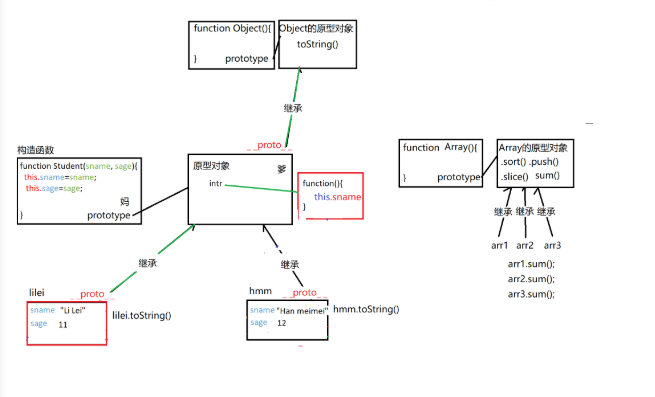
原型链（prototype chain）:

什么是: 多级原型对象逐级引用形成的链式结构

作用: 1. 保存了一个对象可用的所有成员；

2. 控制着成员的使用顺序: 先自有，再共有。

|  |
| --- |
| 示例：（给array添加方法）  Array.prototype.sum=function(){  var sum=0;  for(var i=0;i<this.length;i++){  sum+=this[i];  }  return sum;  }  var arr1=[1,2,3,4],arr2=[4,5,6,7];  console.log(arr1.sum(),arr2.sum()); |



补: for 专门遍历数字下标的数组或类数组对象。

for in 专门遍历自定义字符串下标的关联数组和对象。

in 依次获得当前对象可用的一切非隐藏的成员！

**3.多态:**

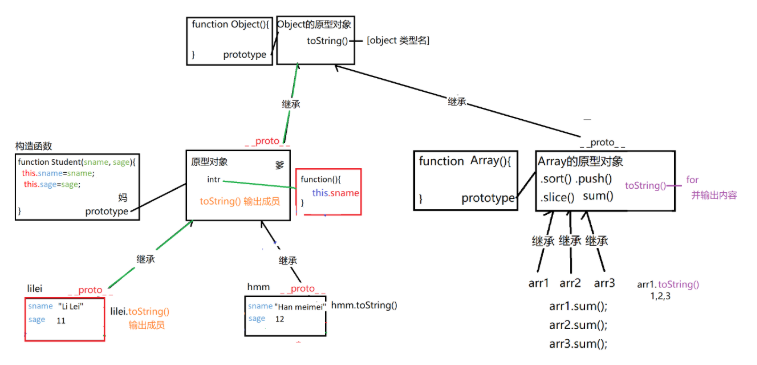
什么是: 一个函数在不同情况下表现出不同的状态

2种:

重载(overload):

重写(override): 子对象中定义了和父对象中同名的成员

何时: 只要子对象觉得父对象的成员不好用，就要定义同名的成员，来屏蔽父对象的。



|  |
| --- |
| 示例：  var arr=[1,2,3];  console.log(arr.toString()); // arr使用的toString()是它自己的父亲已经重新定义的(紫色)  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  }  Student.prototype.toString=function(){ // lilei使用的toString是我们为它父亲重新定义的(黄色)  console.log(`{sname:${this.name},sage:${this.sage} }`);  }  var lilei=new Student("lilei",11);  console.log(lilei.toString())  console.log(lilei.toString==arr.toString);//false =>toString最早是他们祖先的(黑色)，后来重新定义 |

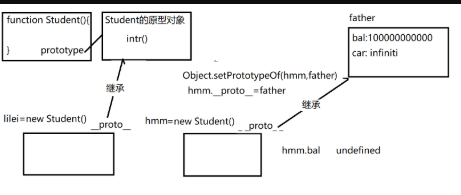
**更改继承的父对象**

（1）只更改一个对象的父对象

比如: Object.setPrototypeOf(hmm,father)

设置hmm的新爹为father(等同于：hmm.\_\_proto\_\_=father;)

只改变Student一个孩子的爹(可以使用新爹的属性)

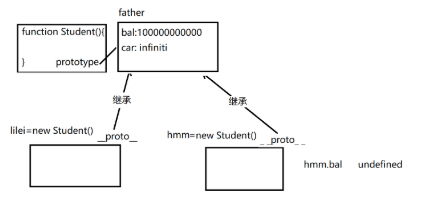


（2）更换所有孩子的父对象

比如：Student.prototype=father;

直接设置Student老公为father，以后创建的孩子都可以使用father的属性。

强调：必须在new第一个孩子之前就要更换（不然孩子会继承新创建的原生对象）。



**day05**

**ES5:**

**1.严格模式:**

什么是: 比普通js运行要求更严格的模式

为什么: js语言本身有很多广受诟病的缺陷，为了屏蔽这些缺陷。

何时: 今后所有js代码都要运行在严格模式下

如何: 在当前作用域的顶部: "use strict";

要求:

禁止给未声明的变量赋值；

将静默失败升级为错误；

静默失败: 执行不成功，也不报错！

禁用了arguments.callee；

arguments.callee: 在调用函数时，自动获得当前正在执行的函数本身。

何时: 专门用于在递归算法中，代替写死的函数名。

递归的问题: 效率极低！——重复计算量太大！

解决: 几乎所有的递归都可以用循环代替。

|  |
| --- |
| 示例：  //斐波拉契数  function fib(n){  //arguments.callee=fib  if(n<3){  return 1;  }else{//否则，当前n的值等于n-1位置的值 + n-2位置的值  //递归计算  //f(1)=1  //f(2)=1  //f(n)=f(n-1)+f(n-2)  // return arguments.callee(n-1)+arguments.callee(n-2);  //用循环解决  var f1=1,f2=1,fn;  for(var i=3;i<=n;i++){  fn=f1+f2;  //为下一轮做准备  /\* f1 f2  f3= 1 + 1  f2 f3  (f1的位置) (f2的位置)  f4= 1 + 2  \*/  f1=f2;//把f2的值给f1  f2=fn;//把fn的值给f2  }  return fn;  }  }  console.log(fib(10)); |

普通函数调用和匿名函数自调中的this不再指向window，而是undefined，防止函数内部数据泄露到全局。

原ES3.1中：ES3.1: 普通函数和匿名函数自调中的this，因为调用时，前面没有使用.，所以，this默认指window。

|  |
| --- |
| 示例：  function fun(){  console.log(this);//this->window  //ES5严格模式下this->undefined  }  //fun是全局变量，属于window  //调用时，虽然没有加window.，但是引擎会自动在全局查找该函数，并以window.fun()方式调用  fun();//等同于window.fun();  function Student(sname,sage){  this.sname=sname;  this.sage=sage;  }  var lilei=Student("lilei",11);//忘记写new  console.log(lilei);//undefined  console.log(window);//在window中添加了sname和sage，泄露到全局中，可以不通过Student属性直接打印  console.log(sname);  console.log(sage); |

**2.保护对象:**

为什么: js中的对象毫无自我保护能力

何时: 如果不希望别人擅自修改对象的属性和结构时。

如何:

**(1) 保护属性:**

将所有属性划分为2大类:

命名属性: 可用，访问到的属性

又分为2类:

数据属性: 直接存储属性值的属性；

访问器属性: 不实际存储属性值，仅负责提供对另一个数据属性的保护。

内部属性: 不能用，访问到的属性

数据属性:

ES5规定，每个数据属性，其实是一个缩微的小对象。

每个缩微的小对象其实包含四个特性: {

value: 值; 实际存储属性值,

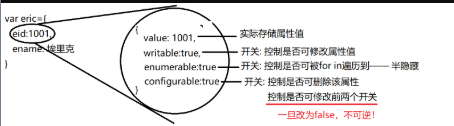
writable: 开关：控制是否可修改,

enumerable: 开关：控制是否可被for in遍历——关闭，半隐藏；但是用 . 可直接访问

configurable: 开关：控制是否可删除该属性，

控制是否可修改其它两个开关（修改其他两个开关，就要设置"双保险"configurable：false，防止再次修改，一旦改为false，不可逆！）

默认: 三个开关都是true



获得obj对象的指定"属性"的四大特性：

var attrs=Object.getOwnPropertyDescriptor(obj,"属性名")

打开和关闭开关:

Object.defineProperty(obj,"属性名",{

//重新定义obj的指定"属性"的开关

开关: true/false,

... : ...

})

问题: 一次只能修改一个属性的四大特性

解决: Object.defineProperties(obj,{

属性:{

开关:true/false,

... : ...

},

属性:{

... : ...

}

})

问题: 无法使用自定义规则保护属性

解决: **访问器属性**

什么是：不实际存储属性值，仅提供对另一个数据属性的保护——保镖

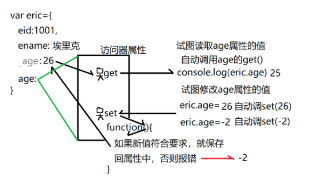
何时: 只要使用自定义规则保护一个数据属性时

如何: 有2步

前提: 必须先有一个隐藏的数据属性，实际存储属性值；

定义访问器属性: 只能用Object.defineProperties为对象添加访问器

使用访问器属性:



|  |
| --- |
| 示例：  var eric={  eid:1001,  ename:"埃里克",  \_age:24  }  Object.defineProperties(eric,{  //1.将原age属性隐姓埋名为\_age，并设置为半隐藏  \_age:{  enumerable:false,  configurable:false  },  //2.给\_age请保镖age  age:{  //其中一个保镖get，专门负责获取原始属性值  get:function(){  console.log("自动调用了age的get()");//查看自动调用  return this.\_age;  },  //另一个保镖set，专门负责修改原始属性值  set:function(value){  console.log(`自动调用了age的set(${value})`);//查看自动调用  //value可接住要赋的新值，并对新值进行验证  if(value>=18 && value<=65){  this.\_age=value;  }else{  throw Error("年龄必须介于18——65之间");  }  },  //让保镖age替\_age抛头露面  enumerable:true,  //禁止删除保镖  configurable:false  }  })  //访问器属性，用法和普通属性，完全一样  //当试图获取属性值时，就会自动调用get方法  console.log(eric.age);  //试图修改属性值时，会自动调用set方法，并将新值先交给set的value参数验证  eric.age=eric.age+1;  //先调用get()  //再调用set()  console.log(eric.age);  eric.age=-2;//报错！ |

**(2) 保护结构**: 禁止添加/删除对象中的属性

3个级别: （选其一使用）

防扩展: 禁止添加新属性

Object.preventExtensions(obj)

禁止给obj对象添加任何新属性

原理: 每个对象中都有一个内部属性: extensible:true,

而preventExtensions()就是将对象内部的属性更改为extensible:false

密封: 在兼具防扩展同时，再禁止删除现有属性——属性值可以修改

Object.seal(obj)

禁止向obj中添加新属性

禁止删除obj中现有属性

原理: 每个属性改为extensible:false、configurable:false

冻结: 在兼具密封的基础上，同时禁止修改所有属性值

Object.freeze(obj)

禁止向obj中添加新属性

禁止删除obj中现有属性

禁止修改任何属性的值

原理: 自动将每个属性改为extensible:false、configurable:false、writable:false

**day06**

一．Object.create(father)

什么是: 创建一个新对象，让新对象继承指定的父对象

何时: 在没有构造函数的情况下，也想创建子对象继承父对象时

如何:

var child=Object.create(father,{ //创建对象var child=new Object();

//Object.setPrototypeOf(child,father);

自有属性:{ //用defineProperties为child添加自有属性

value:值,

三个开关...

}

})

二. ES5

**替换this:**

1. 临时调用一次函数，并临时替换一次this: (call——调用)

call: 传入函数的每个实参必须作为一个单独的参数传入。

apply: 传入函数的每个实参必须放在一个数组中整体传入。

apply就比call多一个功能，将传入的数组打散为单独的参数传入。

|  |
| --- |
| 示例：  function calc(base,bonus1,bonus2){  console.log(`${this.ename}的总工资是：${base+bonus1+bonus2}`);  }  var lilei={ename:"lilei"}  var hmm={ename:"han meimei"}  //lilei调用calc函数  //原calc中this->window 需要换为 lilei；传入单独参数，使用call  calc.call(lilei,12000,1000,2000);  var salary=[4000,5000,6000];  //原calc中this->window 需要换为 hmm  //使用apply将数组salary打散为单个的三个值，分别传入  calc.apply(hmm,salary); |

2. 不调用函数，而是创建一个原函数的副本，永久绑定this和部分参数值: bind(bind——绑定)

何时: 回调函数被调用的次数不确定，修改回调函数的this时只能用bind。

|  |
| --- |
| 示例：(接上面示例)  //lilei绑定一个自己专属的calc  var lcalc=calc.bind(lilei,10000);  //基于原函数calc，创建一个相同的新函数，并且永久绑定this为指定的lilei，以及永久绑定base形参为10000。  /\*lcalc:function calc(base,bonus1,bonus2){  console.log(`${lilei.ename}的总工资是：${10000+bonus1+bonus2}`);  }\*/  //lilei每次使用自己专属的计算器时，不在需要重复输入自己的名字和已经绑死的参数值，只需要输入可能变化的值即可。  lcalc(2000,1000);  lcalc(1000,1000);  //lilei的专属计算器已经没有了this，所以，不会被别人抢去使用  lcalc.call(hmm,4000,5000,6000);//lilei的总工资是：19000 |

**数组API:**

1. 查找: indexOf(元素值,starti) 【等同String中的indexOf("关键词",starti)】

从下标i位置开始，找下一个"元素"的位置。

在数组中找到指定元素，返回元素下标；没找到，则返回-1。

2. 判断:

(1) 判断所有元素是否都符合要求

var bool=arr.every(function(elem, i, arr){

return 根据elem,I,arr判断结果; ——>true/false

})

every会自动拿着回调函数在每个元素上执行一次:

参数： elem 自动得到当前元素值

i 自动得到当前位置

arr 自动得到当前数组对象

返回值: 利用获得的参数，判断当前元素是否符合要求，并返回判断结果。

只有回调函数在每个元素上执行都返回true时，才整体返回true；只要碰到一个false，就返回false。

|  |
| --- |
| 示例：  var arr1=[2,4,6,8];  var bool1=arr1.every(function(elem,i,arr){  //判断arr1中元素是否都为偶数  return elem%2==0;  });  console.log(bool1);//true  //判断arr1是否升序排列  var bool2=arr1.every(function(elem,i,arr){  //当前每一个元素小于或者等于后一位元素，则为升序  //当比较到最后一位元素时，arr[i+1]已经超出数组长度，并且它已经和前一位数字做了比较，所以无需再做比较，直接返回true  return i==arr.length-1 ? true : elem<=arr[i+1];  });  console.log(bool2); |

(2) 判断数组中是否包含符合要求的元素

var bool=arr.some(function(elem, i, arr){

return 判断条件

})

只要碰到一个true，就返回true。

3. 遍历API: 对数组中每个元素执行相同的操作

(1) 遍历原数组中每个元素，执行相同的操作:

arr.forEach(function(elem, i, arr){

对原数组中的元素值，执行操作

})

补：按值传递：两个变量间赋值时，其实都是将原变量中的值复制一个副本给对方。

如果传递的是原始类型的值：number、string、bool、null、undefined

是将原始值复制一个副本给对方，修改新变量的值，不影响原变量。

如果传递的是引用类型的值，数组、日期、自定义对象

只是将地址复制给新变量，形成两个变量用相同的地址指向同一个对象，任何一方修改对象，另一个都会受到影响。

|  |
| --- |
| 示例：  var arr=[1,2,3,4,5];  arr.forEach(function(elem,i,arr){  //当前元素值\*2  //elem\*=2;不可用，因为数值是按值传递，只是更改的新变量elem的值，没有改变原来数组中的值  arr[i]\*=2;//数组是按照地址传递，更改的同一个数组  //或者arr[i]=arr[i]\*2;  });  console.log(arr); |

(2) 依次取出原数组中每个元素，执行相同的操作后，放入新数组中返回:

var arr1=arr.map(

//自动先在内部创建新数组newArr[ ]

function(elem,i,arr){

return 新值; //返回新数组中，最后给arr1

})

4. 过滤和汇总(求和):

(1) 过滤: 判断原数组中每个元素是否符合条件，并复制出符合条件的元素，组成新数组返回，原数组保持不变。

var subArr=arr.filter(function(elem,i,arr){

return 判断条件; //只有return为true，才会放在新数组中

})

(2) 汇总: 对数组中每个元素的值进行统计，得出一个汇总的结果。

var result=arr.reduce(function(prev,elem,i,arr){

//prev: 获得截止目前的临时汇总值

return prev+elem;

},

base //prev起始时的基数

)

三. **ES6新规**

不改变原理的基础上，让API变的更简化

let: 代替var用于声明变量

var的缺点:

1. 声明提前:

2. 没有块级作用域:

let的优点:

1. 阻止了声明提前

2. 让let所在的块({...})也变为一级作用域

let原理：相当于创建匿名函数自调用，并变量更名

(function(){

var \_a=100；

…

})()

**参数增强:**

1. 默认值: ES6中可以给形参提前设置一个默认值，调用时，如果没有给实参，就自动启用默认值代替。

function fun(形参1, 形参2, ... ,形参n=默认值){

…

}

强调: 带默认值的形参必须出现在参数列表的末尾。

2. 剩余参数(rest) 代替 arguments

arguments的问题:

1. 不是纯正的数组类型，而是类数组对象，导致数组的所有API，arguments不能使用

2. 只能接住所有参数，不能有选择的接住部分参数

如何使用剩余参数:

function fun(形参1, 形参2, ...数组名){

//数组中就接住了除形参1,形参2以外的剩余的实参

//且"数组"是纯正的数组类型，可以使用所有数组家的API

}

剩余参数的优点:

1. 是纯正的数组，可使用所有数组家API；  
 2. 可有选择的获得剩余参数。

|  |
| --- |
| 示例：  function calc(sname,...salary){  //...salary可以接住除是sname以外的剩余实参，并保存进salary数组中  var total=salary.reduce(  function(prev,elem){  return prev+elem;  },  0  )  console.log(`${sname}的总工资为：${total}`);  }  calc("lilei",10000,2000,3000); |

3. 打散/散播(spread):

代替apply，用于专门打散数组类型参数为单个值

apply的问题: 其实apply的主要功能是替换this，顺便打散数组类型参数

何时: 只要单纯希望打散数组类型参数为单个值时

调用时: fun(...数组名)

先把数组打散后，再分别传给fun

相当于: fun(数组[0],数组[1],数组[2], ... )

|  |
| --- |
| 示例：  var arr=[1,3,7,5,2]  console.log(Math.max(…arr)) // 7 |

**day07**

**ES6**

**箭头函数**: 简写一切匿名函数

1. 去function变 =>

2. 如果只有一个参数，可省略 ()

3. 如果函数体只有一句话，可省略 {}

如果仅有的一句话是return ...，则必须省略return

改箭头函数后的特性——内外this通用！

如果和this无关时，或希望内外this通用时，可改箭头函数

如果不希望内外this通用时，就不能用箭头函数

**解构**: 提取出一个大的对象中的部分成员，单独直接使用变量。

1. 数组解构: 下标对下标

var arr=[elem1, elem2,...];

|| ||

var [ 变量1,变量2,... ]=arr;

2. 对象解构: 属性对属性

var obj={属性1:值1, 属性2:值2, ... }

|| ||

var { 属性1:变量1, 属性2:变量2, ... }=obj

如果不打算改变成员在原变量中的名称(属性1和变量1的名字一样时)，可以简写为：var{ 属性1 }=obj;

|  |
| --- |
| 示例：  var user={  age:24,  address:{city:"北京",street:"万寿路"},  login:function(){  console.log("登录...");  }  }  //var {  // login : login,  // address : address  //}=user; 规范写法  var {  login,  address  }=user; //简写  login();  console.log(address); |

3. 参数解构: 其实是对象解构在函数传参时的应用

何时: 多个形参都不确定是否有值时

如何：2步

|  |  |
| --- | --- |
| 定义函数时: 将所有形参都定义在对象结构中  function fun( {  url:url,  type:type,  data:data,  dataType:dataType  } ){ … }  可简写为: function fun({url, type, data, dataType}){ … } | 调用时: 所有实参都要放在一个对象结构中整体传入  fun({  url:"http://localhost:3000/index",  type:"get",  //data: ,  dataType:"json"  }) |

结果:

形参 <=赋值= 实参

url = "http://localhost:3000/index

type = "get"

data = undefined

dataType = "json"

优点:

1. 与参数个数无关

2. 与参数顺序无关

**遍历索引数组**

1. for ——何时：需要控制顺序和步调

for(var i=0;i<arr.length;i++){

arr[i] //当前元素

}

2. forEach ——何时：及关心内容又关心位置

arr.forEach( (elem,i,arr)=>{ elem //当前元素值 })

不灵活:

不能控制顺序

不能控制步调

3. for of：自动依次取出arr中的每个属性值 ——何时：只关心内容，不关心位置、顺序和步调

for(var elem of arr){

elem //当前元素值

}

不灵活: 只能获得元素值，不能获得元素位置

vs for in:

for of: 可遍历数字下标的索引数组或类数组对象

for in: 只能遍历非数字下标的关联数组或对象

|  |
| --- |
| 遍历索引数组： for for of forEach map：保护原数组，并生成新数组  遍历类数组对象：for for of  遍历对象和关联数组: for in |

**class 面向对象的简化：**

1. 封装：

(1) 对单个对象直接量的简化:

var obj={

属性1: 值1,

属性2: 变量, //如果属性名刚好和变量名相同，则只写一个 —直接写—>属性2,

方法名:function(){ //所有对象中的方法，不再需要 :function —直接写—>方法名(){ … }

... ...

}

}

(2) 构造函数

对一种类型定义的简化: class

用class{ }包裹构造函数和原型对象方法

将构造函数名提升为类型名，构造函数统一更名为constructor

所有原型对象方法不用再添加 "类型名.Prototype" 前缀，而是直接写在class内，与构造函数平级（中间不要逗号）

class Plane {

constructor (fname,speed,score){

this.fname=fname;

this.speed=speed;

this.score=score;

}

fly(){ ... }

getScore(){ ... }

}

2. 继承:

何时：如果多个class中包含部分相同的属性和方法，就要定义抽象父类型(只包含重复部分)，让所有子类型继承抽象父类型。

如何：2步

(1) 定义需要继承的父类型:

class Father{

constructor(参数,…){

this.xxx=xxx;

…

}

fly(){…}

}

(2) 子类型extends父类型；子类型构造函数中必须先借用父类型构造函数。

class Child extends Father{ //继承Father的方法(extends相当于原来的Object.setPrototypeOf(Child,Father)

constructor( 所有参数 ){

super(参数,...); //借父类型的构造函数，可以使用Father内的属性

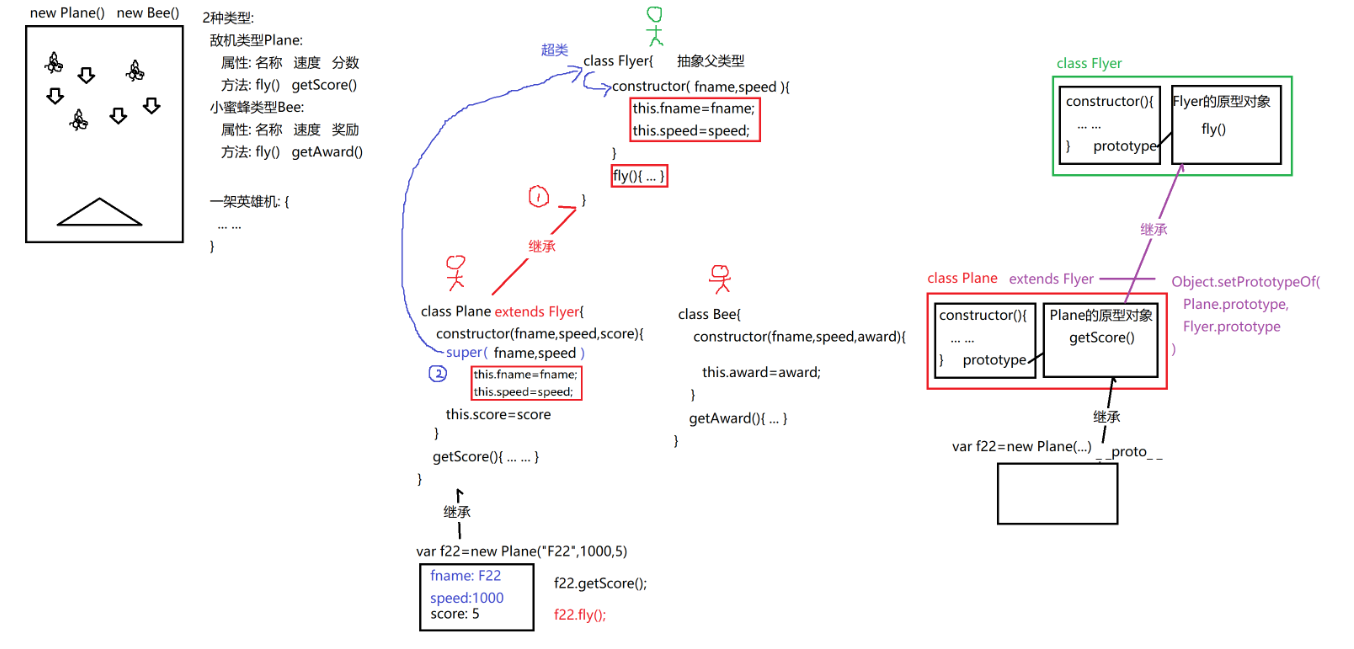
this.xxx=xxx; //创建自身属性

}

}

强调：super中不要加this！

super()必须是第一句话，以防止爹的属性把孩子的属性给覆盖了！！



|  |
| --- |
| 示例：  class Father{  constructor(fname,speed){  this.fname=fname;  this.speed=speed;  }  fly(){  console.log(`${this.fname}以时速${this.speed}飞行`);  }  }  class Child extends Father{  constructor(fname,speed,score){  super(fname,speed);  this.score=score;  }  getScore(){  console.log(`击落${this.fname}获得${this.score}分`);  }  }  var f22=new Child("F22",100,5);  f22.fly();  f22.getScore();  console.log(f22); |

3. Promise

(1)Promise是一个构造函数 new Promise() 得到一个Promise一个实例

(2)在Promise上有两个函数分别是resolve（成功之后的回调函数）和reject（失败之后的回调函数）

(3)在Promise构造函数的prototype属性上，有一个.then()方法，也就是只要是Promise构造函数创建的实例，都可以访问到.then()方法

(4)如果Promise表示一个异步操作，每当我们new一个Promise的实例，这个实例就表示一个具体的异步操作

(5)既然Promise创建的实例是一个异步操作那么这个异步操作的结果只能有两种状态

①状态一：异步执行成功 需要在内部调用，成功的回调函数resolve 把结果返回给调用者

②状态二：异步执行失败了 需要在内部调用，成功的回调函数reject把结果返回给调用者

③由于Promise的实例是一个异步操作，所以拿到操作结果后，无法使用return把操作结果返回给调用者，这个时候只能使用回调函数的形式，来把成功或者失败的结果返回给调用者

(6)我们可以在new出来的Promise实例上，调用.then()方法【预先】为这个Promise一部操作，指定成功（resolve）和失败（reject）回调函数

需求: 多个异步函数，必须顺序执行！

错误: 单靠先后调用多个异步函数，无法保证顺序执行。而是一开始就都开始执行，异步同时执行。

（1）使用回调函数callback

正确: 2步:

定义前一项任务时，就要定义回调函数参数，并在自己的任务执行完之后自动调用回调函数

function liang(callback){

console.log("亮起跑...");

setTimeout(function(){

console.log("亮到达终点！");

callback(); //调用参数带进来的函数

},6000)

}

调用时，将要做的下一件事，包裹在一个匿名函数中作为参数传入前一个函数中:

liang( //参数callback传入函数

function(){ ran(); }

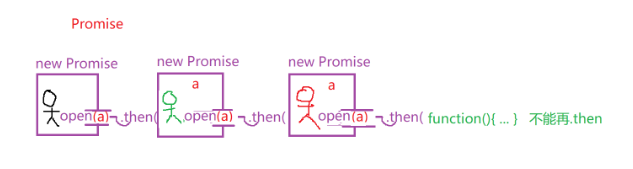
);

问题: 回调地狱 callback hell——调用层次太多，看着像地狱

|  |
| --- |
| 示例：  function liang(callback){  console.log("亮起跑...");  setTimeout(function(){  console.log("亮到达终点！");  callback();  //写在这里的代码一定会在亮到达终点后才执行，callback:function(){ran();}  //callback()->ran()  },6000)  }  function ran(callback){  console.log("然起跑...");  setTimeout(function(){  console.log("然到达终点！");  callback();  },4000)  }  function dong(callback){  console.log("东起跑...");  setTimeout(function(){  console.log("东到达终点！");  callback();  },2000)  }  liang(//参数callback传入函数  //在亮开始调用时，就把下一项任务提前托付给亮，任务中写了一句话: "ran();"  function(){  ran(  function(){  dong(  function(){  console.log("比赛结束");//最后输出:"比赛结束!"  }  );  }  );  }  ); |

（2）解决: 使用Promise

类似将每个人装在小框(new Promise)里，并且设置有开关(open)，可以通过open传递参数a，每个小框后面都会用then()连接后面的小框



|  |
| --- |
| 示例：  function liang(){  return new Promise(function(open){  var bang="亮的棒";  console.log("亮起跑...");  setTimeout(function(){  console.log("亮到达终点！");  open(bang);  },6000)  })  }  function ran(bang){  return new Promise(function(open){  console.log(`然接到${bang}`);  console.log("然起跑...");  setTimeout(function(){  open(bang);  console.log("然到达终点！");  },4000)  })  }  function dong(bang){  return new Promise(function(open){  console.log(`东接到${bang}`);  console.log("东起跑...");  setTimeout(function(){  open(bang);  console.log("东到达终点！");  },2000)  })  }  liang()  .then(ran) //ran不加()，是因为要liang执行完后才会执行ran，而不是立即执行的！  .then(dong)  .then(  function(bang){  console.log(`${bang}也到达终点`);  console.log("比赛结束");  }  ); |

**day01**

正课:

1. 什么是DOM

2. DOM Tree

3. 查找元素:

1. 什么是DOM: Document Object Model

DOM: 专门操作网页内容的API的标准——W3C

JS=ES(核心语法)+

DOM(专门操作网页内容的API)+

BOM(专门操作浏览器软件的API)

为什么: 为了统一所有浏览器的操作网页的API标准

何时: 只要操作网页的内容，都要用DOM

包括: 增，删，改，查，事件绑定

|  |
| --- |
| //DOM 4步  //1. 查找触发事件的元素  //2. 绑定事件处理函数  //3. 查找要修改的元素  //4. 修改元素 |

2. DOM Tree

什么是DOM Tree: 内存中保存一个网页中所有内容的树形结构

为什么: 因为树形结构是最容易保存复杂的上下级包含关系的结构

如何:

形成: 浏览器在加载一个HTML网页时，都会创建一棵DOM数。包括:

唯一的根节点对象: document；

每个网页内容(元素、属性、文字)都会成为树上的节点对象。

每个节点对象都有三个属性:

节点类型 .nodeType

何时: 需要判断节点类型时

值是一个数字:

document 9

element 1 (元素)

attribute 2 (属性)

text 3 (文本)

问题: 只能区分节点类型，不能更细致的区分元素的名称

节点名 .nodeName

何时: 判断当前节点的元素名时

包括: document #document

element 全大写的标签名

attribute 属性名

text #text

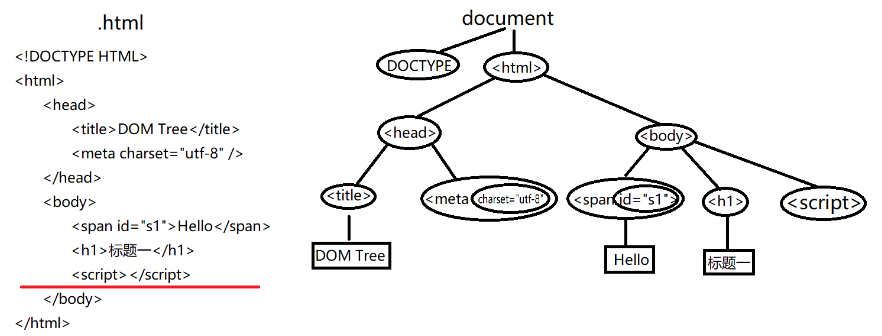
节点值 .nodeValue——几乎不用

包括: document null

element null

attribute 属性值

text 文本内容



**一. 查找元素**: 4种:

(1) 不用查找就可直接获得的元素: 4个

<html> document.documentElement

<head> document.head

<body> document.body

<form> document.forms[i/id]

(2) 按节点间关系查找:

何时: 在首先获得了一个元素后，要找周围附近的元素时

**节点树:** 包含所有节点对象的完整树结构

2大类关系:

父子关系:

.parentNode 当前元素的父节点对象

.childNodes 当前元素的所有直接子节点对象

获得父节点下第i个位置的直接子节点: 父节点.childNodes[i] i从0开始

.firstChild 当前元素的第一个直接子节点对象

.lastChild 当前元素的最后一个直接子节点对象

兄弟关系:

.previousSibling 当前元素的前一个兄弟节点对象

.nextSibling 当前元素的下一个兄弟节点对象

问题: 不但包含元素节点，还有看不见的空字符节点干扰。（节点 > 元素）

**元素树:** 仅包含元素节点的树结构

元素树不是一棵新树，而是节点树中的一个子集而已。

优点: 只包含元素节点，不包含文本节点，不会受干扰。

包括:

父子关系:

.parentElement 当前元素的父元素对象（几乎等同于.parentNode）

.children 当前元素的所有直接子元素对象

获得父元素下第i个位置的直接子元素: 父元素.children[i] i从0开始

.firstElementChild 当前元素的第一个直接子元素对象

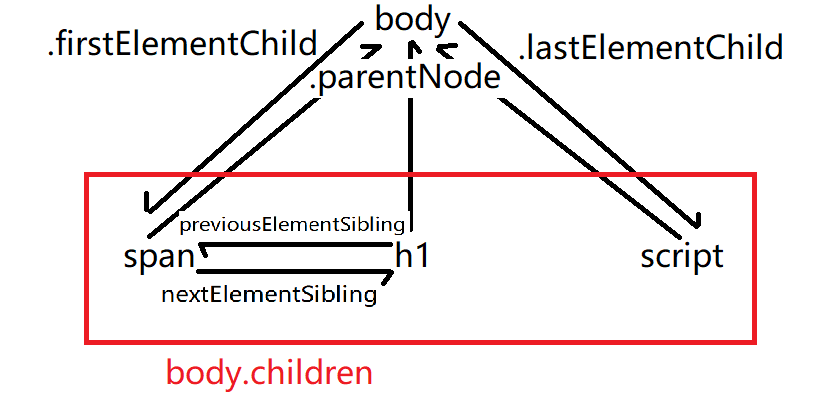
.lastElementChild 当前元素的最后一个直接子元素对象

兄弟关系:

.previousElementSibling当前元素的前一个兄弟元素对象

.nextElementSibling 当前元素的下一个兄弟元素对象

总结: 今后只要按节点间关系查找元素时，首选元素树的属性



|  |
| --- |
| 练习: 查找一个节点下的所有后代节点:  2步:  先定义函数，仅遍历直接子元素:  在函数内，对每个直接子元素，调用和父元素完全相同的当前函数本身。  <body>  <span>Hello World !</span>  <ul id="nav">  <li>电影</li>  <li>综艺  <ul>  <li>{{sname}}</li>  <li>爸爸</li>  <li>{{score}}</li>  </ul>  </li>  <li>剧集</li>  </ul>  <script>  var sname="dingding";  var score=1000;  //定义一个函数，仅遍历直接子元素  function getChildren(parent){  //更换找到元素中带{{ }}，并进行更换值  if(parent.innerHTML=="{{sname}}"){  parent.innerHTML=sname;  }else if(parent.innerHTML=="{{score}}"){  parent.innerHTML=score;  }  console.log(parent.nodeName); //打印节点名  var children=parent.children; //获得当前父元素的直接子元素列表  for(var child of children){ //遍历直接子元素列表中的每个孩子  getChildren(child); //再对每个孩子执行和自己完全相同的操作（递归）  }  }  getChildren(document.body); //遍历body的所有直接子元素  </script>  </body> |

(3) 按HTML特征查找: 4种:

何时：只要一个条件，就可以获得想要的元素时。

按id查找一个元素:

何时: 当要查找的元素身上有id属性时，都首选用id查找；

如何: var elem=document.getElementById("id名")

强调: getElementById只能用document调用

问题： 1. 不是所有元素都有id；

2. 一次只能找一个元素。

|  |
| --- |
| 示例：  html中: <div id="header"></div>  <div id="main"></div>  <div id="footer"></div>  var elemMain=document.getElementById("main")  结果: elemMain: <div id="main"></div> |

按标签名查找多个元素:

何时: 当元素上没有id，name，class时，就可用标签名查找；

如何:

var elems=任意父元素.getElementsByTagName("标签名")

强调: 1. 可用任意父元素找；

2. 不仅查找直接子元素，而是在所有后代中查找符合条件的元素；

返回值：多个元素组成的类数组对象。

按name属性查找多个元素：

何时：专门用于查找表单中的有name属性的表单元素。

如何：var elem=document.getElementsByName("name")

强调: getElementsByName只能用document调用。

返回值：包含多个相同name属性值的元素的类数组对象。

|  |
| --- |
| 示例：  <form>  用户名: <input name="uname">  性别: <input type="radio" name="sex" value="1">男  <input type="radio" name="sex" value="2">女  //查找所有性别单选按钮:  var radios=document.getElementsByName("sex");  返回值: {  0: <input type="radio" name="sex" value="1">,  1: <input type="radio" name="sex" value="2">  }  //只查找一个姓名文本框:  var txtName=document.getElementsByName("uname")**[0]**;  返回值: { 0: <input name="uname"> }  txtName=<input name="uname"> |

按class属性查找多个元素：

如何：var elem=父元素.getElementsByClassName("name")

在指定"父元素"内，查找class为指定的"类名"的多个元素。

强调： 1. 可用任意父元素调用，仅当前父元素内查找，缩小查找范围；

2. 不但查找直接子元素，且在所有后代中查找；

3. 只需要一个class，就可匹配元素，不需要整个class完整匹配。

返回值：包含多个相同classname属性值的元素的类数组对象。

|  |
| --- |
| 示例：  <ul>.getElementsByClassName("menu")  <li>  <ul>  <li class="menu active">  <li class="menu">  <li class="menu">  <li>  <ul>  <li class="menu active">  <li class="menu">  <li class="menu">  返回值: {  0: <li class="menu active">,  1: <li class="menu">,  2: <li class="menu">,  3: <li class="menu active">,  4: <li class="menu">,  5: <li class="menu">  } |

问题：一次只能按一个条件查找，当查找条件复杂时，代码会很繁琐。

4. 按选择器查找

何时: 当查找条件复杂，如果单靠一个条件/特性无法精确的找到元素时。

如何: 2个API:

只查找一个元素:

var elem=任意父元素.querySelector("选择器")

|  |
| --- |
| 比如:  查找tfoot中最后一个td:  var td=table.querySelector("tfoot>tr>td:last-child")  如果没找到: 返回null |

查找多个元素：

var elems=任意父元素.querySelectorAll("选择器")

|  |
| --- |
| 比如:   1. 查找tbody中每行最后一个td:   var tds=table.querySelectorAll("tbody>tr>td:last-child")  返回值: tds:{ 0:<td>, 1:<td>, 2:<td> }  如果没找到，返回空类数组对象: { }.length=0   1. var as=document.querySelectorAll("[data-toggle=tab]");   属性选择器：查找属性值为data-toggle=tab的元素 |

总结: 因为按选择器查找的效率不如按HTML特征查找。

如果靠一个条件就可找到想要的元素时，首选按HTML特征查找；

如果查找条件复杂时，首选按选择器查找。

**day02**

**二. 修改:**

1. **内容**: 3种:

获取或修改原始双标签的HTML片段内容:

elem.innerHTML

示例：<p>来自<a href="http://xinhuashe.com">&lt;&lt;新华社&gt;&gt;</a>的消息</p>

p.innerHTML => 来自<a href="http://xinhuashe.com">&lt;&lt;新华社&gt;&gt;</a>的消息

获取或修改纯文本内容:

elem.textContent

比innerHTML多做两件事：

1. 翻译特殊符号为正文；

2. 去掉内嵌的子标签，只保留文字。

示例：p.textContent => 来自<<新华社>>的消息

获取表单元素的值:

elem.value

2. **属性**: 3种:

(1) HTML标准属性: 2种:

核心DOM API: 4个

获取属性值： var value=elem.getAttribute("属性名")

修改属性值： elem.setAttribute("属性名","新值")——有则修改，无则添加

移除属性： elem.removeAttribute("属性名")

判断是否包含属性: var bool=elem.hasAttribute("属性名") true/false

HTML DOM API: 对常用核心DOM API的简化

HTML DOM提前将所有HTML标准属性，定义在内存中的元素对象上：

elem.属性名

|  |
| --- |
| 比如: <a href="http://tmooc.cn">go to tmooc</a>  a:{  id:"",  title:"",  href: "http://tmooc.cn",  target:"",  name:"",  ... ...  }  获取a的href属性值: a.href  修改a的title为"由此前往tmooc":  a.title="由此前往tmooc"  移除a的title属性: a.title="";  判断是否包含a: a.title!=="" 就是包含 |

特例: class是ES标准中的关键字

DOM中不允许再使用class作为属性名

class一律都要改为className

DOM中的className属性等于HTML中的class

总结：只要访问元素的HTML标准属性，都用 点 ( . )elem.className

|  |
| --- |
| 示例：(下拉菜单)  //获得所有span  var spans=document.getElementsByTagName("span");  //var spans=document.querySelectorAll("ul.tree>li>span");  console.log(spans);  for(var span of spans){  span.onclick=function(){  //如果span的class为open，就删除open  if(this.className=="open"){  //this.removeAttribute("class");  this.className="";  }else{  //否则就给span添加open  this.className="open";  }  }  } |

(2) 状态属性: enabled disabled selected checked

值都是bool类型，不是字符串类型，不能用核心DOM API修改。

只能用HTML DOM的 点 ( . ) 来访问。

|  |
| --- |
| 示例：  <button disabled></button>  <select>  <option selected></option>  <input type="checkbox" checked>  问题: 值都是bool类型，不能用核心DOM API访问  解决: 只能用HTML DOM的.访问  btn.disabled=true;  chb.checked=true; |

补充: CSS3中有一种特殊的选择器: 状态伪类:

:enabled :disabled :checked :selected

比如：<input type="radio" name="sex" value="1" checked>男

input:checked

<input type="radio" name="sex" value="0">女

input:not(:checked) //选择没有checked属性的input元素

(3) 自定义扩展属性:

什么是: HTML标准中没有规定的，程序员自行添加的属性

何时: 2种:

用自定义扩展属性作为条件，选择要绑定事件的元素，可让事件不受元素、样式更改的影响。

为什么不用id,class,元素 选择器：

1. id只能选择一个元素；

2. class是定义样式用的，经常变化；

元素选择器限制太死。实现一种效果，可能用不同的元素都行。

在客户端元素上临时缓存业务数据。

好处: 避免重复请求服务端，造成延迟

获取或修改自定义扩展属性值：

<ANY 自定义属性名="值">

核心DOM API:

强调：HTML DOM API不能操作自定义扩展属性

因为：自定义扩展属性是后天自定义的，HTML标准中没有。所以不包含在元素对象中，不能用点(.)直接访问。

比如：<img src="1.png" md="1-md.png" lg="1-lg.png">

获取img的md属性值： ~~img.md 不可用~~

img.getAttribute("md") => "1-md.png"

修改：img.setAttribute("md","2-md.png")

支持HTML5标准中:

<ANY data-自定义属性名="值">

elem.dataset.自定义属性名 => 可自动获得所有 "data-" 开头的属性

比如：<img src="1.png" data-md="1-md.png" data-lg="lg.png">

获取img的md属性值：img.dataset.md => "1-md.png"（不要写data-）

修改：img.dataset.md="2-md.png"

|  |
| --- |
| 示例：  <body>  <h2>实现多标签页效果</h2>  <div class="tabs">  <ul id="tab">  <li><a href="javascript:;" data-toggle="tab" data-target="#content1">10元套餐</a></li>  <li><a href="javascript:;" data-toggle="tab" data-target="#content2">30元套餐</a></li>  <li><a href="javascript:;" data-toggle="tab" data-target="#content3">50元包月</a></li>  </ul>  <div id="container">  <div id="content1">  10元套餐详情：<br />&nbsp;每月套餐内拨打100分钟，超出部分2毛/分钟  </div>  <div id="content2">  30元套餐详情：<br />&nbsp;每月套餐内拨打300分钟，超出部分1.5毛/分钟  </div>  <div id="content3">  50元包月详情：<br />&nbsp;每月无限量随心打  </div>  </div>  </div>  <script>  //1.查找触发事件的元素  var as=document.querySelectorAll("[data-toggle=tab]");  //2.绑定事件  for(var a of as){  a.onclick=function(){  //3.查找需要修改的元素  var divs=document.querySelectorAll("#container>div");  //4.修改元素  //遍历每个子div，并将每个子div隐藏  for(var div of divs){  div.style.display="none";  }  //获得当前a对应的div的id  var id=this.getAttribute("data-target");  //id:"#content1"只能用选择器查找  //用id找到对应的div，并设置其显示  document.querySelector(id).style.display="block";  }  }  </script>  </body> |

**3. 样式**:

(1) 修改内联样式:

elem.style.css属性名="值"

等效于: <ELEM style=" css属性名:值"

强调: css属性名中如果带 -，需要去横线变驼峰:

z-index => zIndex

list-style => listStyle

background-color => backgroundColor

如果是带单位的数字属性:

修改时: 必须手动拼接单位到结尾:

.style.width=12+"px";

获取时: 必须去掉结尾的px，才能做计算

width="12.5px"

parseFloat(width) => 12.5

问题: elem.style 仅代表内联样式:

修改时，如果elem.style，优先级最高！

获取时，只能获取内联样式，无法获取外部样式表中的样式。

(2) 获取计算后的样式: 最终应用到当前元素上的所有样式的集合。

如何: 2步:

先获得计算后的样式的集合对象:

var style=getComputedStyle(elem)

从css的集合style中，获取一个css属性的值:

style.css属性名

强调: 因为计算后的样式属性来源不确定，怕牵一发而动全身，所以都是只读的。

问题: elem.style修改样式，一句话只能修改一个css属性

如果同时修改多个css属性，代码会很繁琐

解决: 今后，只要批量修改css属性，都要首选class方式修改。

**三. 添加**

添加: 3步:

1. 创建一个空元素对象

var a=document.createElement(**"**a**"**)

a: <a></a>

2. 设置关键属性

a.href="http://tmooc.cn";

a.innerHTML="go to tmooc";

-->a: <a href="http://tmooc.cn"> go to tmooc </a>

3. 将新对象挂载到DOM树上指定位置：3种

在当前父元素下的结尾，追加一个新元素:

父元素.appendChild(a)

在父元素下的某个子元素child之前插入a:

父元素.insertBefore(a, child)

用a替换父元素的某个子元素child:

父元素.replaceChild(a, child)

优化: 尽量减少修改DOM树的次数

页面加载过程:

html -> DOM树

↓

加载树 -> 排版(layout) -> 绘制

↑ 非常耗时

css -> COM模型

每次修改DOM树都会导致，浏览器重排重绘页面。

如何: 2种:

1. 如果同时添加父元素和子元素时，应该先在内存中将子元素添加到父元素中，最后再一次性将父元素添加到DOM树。

2. 如果父元素已经在页面上了，要同时添加多个平级子元素时。先将多个子元素临时加入文档片段对象中。再一次性将文档片段对象添加到DOM树上。

文档片段对象将子元素送到DOM树后，自动释放。

文档片段: 内存中，临时保存多个平级子元素的，虚拟的父元素。

如何:

先创建一个文档片段对象:

var frag=document.createDocumentFragment();

将子元素添加到frag中

frag.appendChild(子元素)

将frag整体添加到DOM树

父元素.appendChild(frag)

**四. 删除**

1. 删除父元素下的子元素:

父元素.removeChild(child);

elem.remove();

**day03**

1. HTML DOM**常用对象**:

Image Select/option Table/... Form/...

(1) Image对象：代表页面上的一个<img>元素

例如：创建image对象var img=new Image();

(2) Select对象: 代表页面上一个<select>元素

属性: .selectedIndex 快速获得当前选中项option的位置

.options 快速获得select下所有option的集合

.options.length 获得select下option的个数

.length => 等同于 .options.length

.value 获得select中当前选中项的值（如果选中项没有value，则用内容作为value）

方法: .add(option) 向select中添加一个option

问题: 不支持文档片段（最好使用append）

.remove(i) 移除select中i位置的option

事件: .onchange 当(选中项)改变时

Option对象: 代表select中每个option元素

创建并添加(内容,value)：var opt=new Option(text,value)

(3) Table对象: 代表页面上一个<table>元素

table管着**行分组**:

添加行分组: var thead=table.create**TH**ead()

【相当于原来2步: ❶var thead=document.createElement("thead"); ❷table.appendChild(thead);】

var tbody=table.createTBody()

var tfoot=table.createTFoot()

删除行分组: table.deleteTHead()

table.deleteTFoot()

获取行分组: table.tHead

table.tFoot

table.t**B**odi**es**[i] 一个table中包含多个tbody

行分组管着**行**:

添加行: var tr=行分组.insertRow(i)

【相当于原来2步：❶var tr=document.createElement("tr"); ❷分组行. insertBefore(tr, child);】

在当前行分组中i位置，插入一个新行；如果i位置有行，则向后挤，而不是替换！

固定用法:

1. 行分组.insertRow(0) 开头插入

2. 行分组.insertRow() 末尾追加

删除行: 行分组.deleteRow(i)

删除行分组内的i位置的行

强调: i是相对于行分组内的下标

问题: 行分组内的下标位置无法自动获得

解决: 今后只要删除行，都用:

**table**.deleteRow(tr.rowIndex) 【tr.rowIndex可获得当前tr在整个table表中的下标位置】

获取行: 行分组.row**s**[i] 获得当前行分组中的第i行。

行管着**格**:

添加格: var td=tr.insertCell(i)

【相当于原来2步：❶var td=document.createElement("td"); ❷tr. insertBefore(td, child);】

固定用法: tr.insertCell() 在行末尾追加新格

问题: 只能创建td，不能创建th

解决：使用css轻松为表头中的td添加样式。

删除格: tr.deleteCell(i)

获取格: tr.cell**s**[i]

(4) Form对象: 代表页面上一个表单元素

获取: var form=document.forms[i];

属性: .elements 获得表单中所有表单元素的集合

.elements.length 获得表单中所有表单元素的个数

.length => 等同于 .elements.length

方法: form.submit() 手动调用程序控制提交表单

获取表单元素input:

form.elements[i/id/name]

如果表单元素上有name属性:

form.name属性值

方法: elem.focus() 让当前表单元素获得焦点

elem.blur() 让当前表单元素失去焦点

|  |
| --- |
| 示例：  //获取form中倒数第二个input元素  var btn=form.elements[form.length-2];  btn.onclick=function(){  var rname=vali(form.username,/^\w{1,10}$/);  var rpwd=vali(form.pwd,/^\d{6}$/);  //判断，如果都满足，才提交表单  if(rname==true && rpwd==true){  form.submit();  }  } |

**二. 事件**(event)

1. 事件: 浏览器启动触发的或用户手动触发的页面内容状态的改变

什么是事件处理函数: 事件发生时，自动执行的函数

(1) 如何绑定事件处理函数: 3种:

①在HTML中绑定:

<ANY onclick="play()">

function play(){ ... }

问题: 不符合内容与行为分离的原则，不便于维护

②在js中绑定，用赋值方式绑定:

元素.onclick=function(){

//this->当前单击的.前的元素

}

③当事件发生在元素上时，浏览器自动执行函数:

元素.onclick()

问题: 一个事件属性，只能绑定一个处理函数

在js中绑定，用添加事件监听对象的方式:

元素.addEventListener("click",事件处理函数)

添加事件监听对象

事件监听对象: 保存一个事件相关的元素+事件名+处理函数的对象

|  |
| --- |
| 比如:  btnShoot.addEventListener("click",shoot1)  事件监听对象={  元素: btnShoot,  事件名: click  处理函数: shoot1  } |

其实，浏览器有一个事件队列，所有事件监听对象都会自动添加到事件队列中。

当浏览器中触发事件时，浏览器会遍历事件队列，并查找出符合条件的事件监听对象，执行其处理函数。

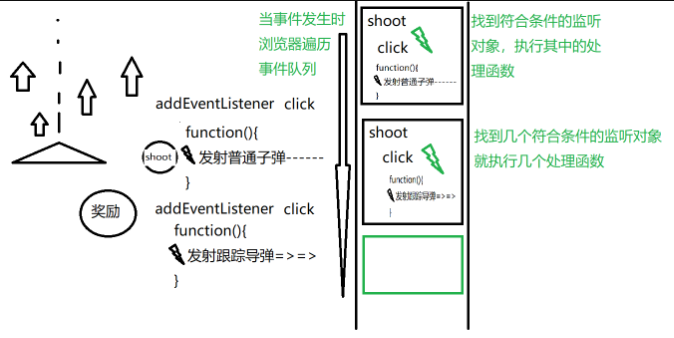
(2) 移除事件监听对象:

元素.removeEventListener(

"click", 原函数名

)

强调: 如果一个处理函数可能被移除，则绑定时，就不能用匿名函数，必须用有名称的函数（因为使用匿名函数function就是创建一个新的函数，虽然长的一样，但是储存地址不一样）。



|  |
| --- |
| 示例：  <body>  <button id="btnShoot">shoot</button><br>  <button id="btnAward">获得跟踪导弹</button><br>  <button id="btnBreak">失去跟踪导弹</button><br>  <script>  //获取shoot的btn元素  var btnShoot=document.getElementById("btnShoot");  //往事件队列中添加事件  btnShoot.addEventListener(  "click",  function(){  console.log("发射子弹------");  }  );  //获取“获得追踪弹”的btn元素  var btnAward=document.getElementById("btnAward");  function shoot2(){  alert("发射跟踪导弹=>=>=>");  }  btnAward.onclick=function(){  btnShoot.addEventListener("click",shoot2)  }  //获得“失去跟踪导弹”的btn元素  var btnBreak=document.getElementById("btnBreak");  btnBreak.onclick=function(){  btnShoot.removeEventListener("click",shoot2);  }  </script>  </body> |

2. 事件模型(event module)

事件模型: 事件发生时，到所有处理函数执行完，所经历的过程

三个阶段:

捕获: 由外向内，记录各级父元素上绑定的处理函数

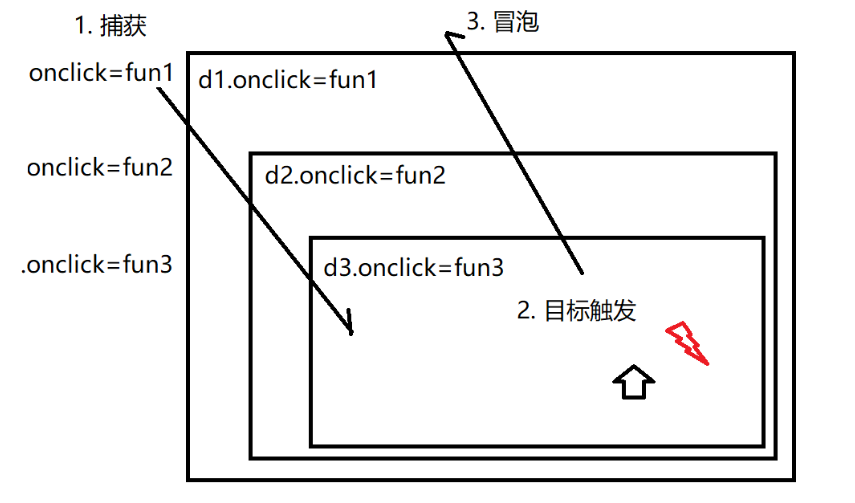
只记录，不触发

目标触发: 优先触发目标元素上的处理函数

目标元素(target): 最初想单击的那个元素

事件模型中，都是先触发目标元素上的处理函数

冒泡: 由内向外，依次触发各级父元素上绑定的处理函数



为什么: 所有浏览器的开发者一致认为，点在子元素，等效于也点在父元素中了。

事件对象:

什么是: 事件发生时，自动创建的保存事件信息的对象

何时: 2种:

获取事件相关信息

改变事件的默认行为

如何:

获取: 事件对象总是作为处理函数的第一个参数自动传入

元素.onclick=function(e){

//e->当前事件对象

}

事件发生时自动执行，默认传入参数event:

元素.onclick(event);

API:

取消冒泡: e.stopPropagation();

停止 蔓延

利用冒泡:

优化: 尽量减少事件监听的个数

为什么: 浏览器触发事件，是靠遍历方式找到监听对象执行。所以，监听对象的个数，决定了遍历的速度。监听对象多，遍历就慢，事件响应就会延迟。监听对象少，遍历就快，事件响应就快！

如何: 今后，只要多个平级元素，要绑定相同的事件时，其实只要在父元素绑定一次，所有子元素就可共用！

2个难题:

this不再指向实际单击的元素，而是指向父元素

解决: e.target代替this

e.target会始终保存着最初触发事件的目标元素，且不随冒泡而改变

实际点击的子元素不一定是想要的

解决: 凡是利用冒泡，都要先判断是不是想要的。

只有目标元素是想要的元素，才执行操作，否则就什么也不干。

|  |
| --- |
| 示例：  <body>  <div id="keys">  <button>1</button>  <button>2</button>  <button>3</button>  <button>4</button><br>  <button>C</button>  <button>+</button>  <button>-</button>  <button>=</button>  </div>  <textarea id="sc" style="resize:none;width:200px; height:50px;" readonly></textarea>  <script>  var keys=document.getElementById("keys");  //仅为父元素绑定一次事件监听  keys.onclick=function(e){  //this->当前单击按钮  //e.target->永远保存着目标元素  //如果单击的是按钮时  if(e.target.nodeName=="BUTTON"){  //查找id为sc的显示屏文本框  var sc=document.getElementById("sc");  //判断目标元素的内容  switch(e.target.innerHTML){  //如果是"C"  case "C":  //就清除显示屏文本框的内容  sc.innerHTML="";  break;  //如果是"="  case "=":  //就取出显示屏文本框的内容，放入eval()函数中计算表达式的值，结果，再放回显示屏文本框中显示出来  try{  sc.innerHTML=eval(sc.innerHTML)  //如果瞎点按钮，生成错乱的表达式，程序的控制台也不能报红色错误！而是友好的显示屏文本框中提示错误信息  }catch(err){  sc.innerHTML="请输入正确的计算方式";  }  break;  default:  //其余按钮  //将目标元素按钮的内容追加到显示屏文本框的内容中  sc.innerHTML+=e.target.innerHTML;  }  }  }  </script>  </body> |

3. 阻止默认行为:

比如: <a href="#top"> 默认会在地址栏中结尾添加#top

如果不希望a，篡改地址栏中的url:

e.preventDefault();

阻止 默认的行为

|  |
| --- |
| 示例：  <script>  var a=document.body.children[0];  a.onclick=function(e){  e.preventDefault(); //在地址栏最后就不会出现默认的#top  alert("疼！");  }  </script> |

4. 鼠标坐标:

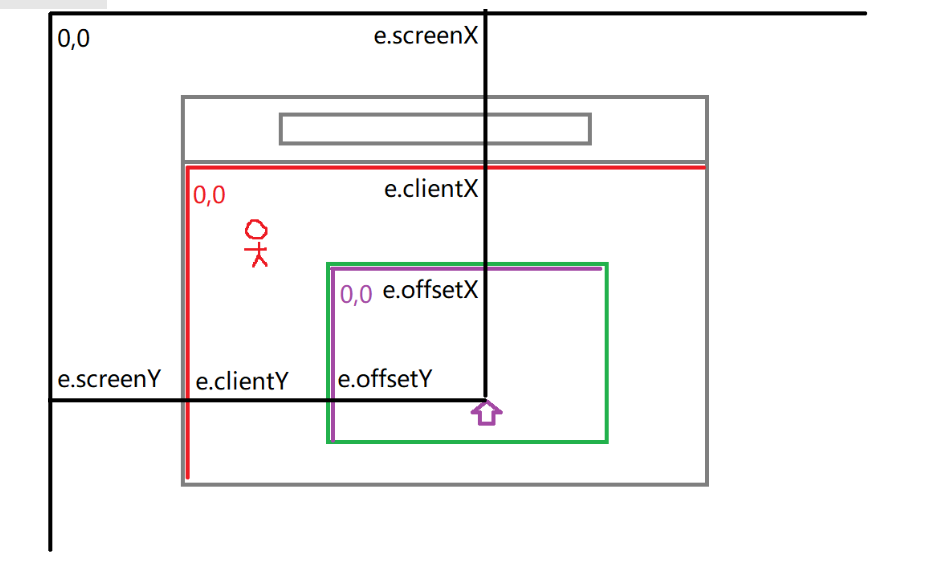
3组坐标:

相对于屏幕左上角: e.screenX e.screenY

相对于文档显示区左上角: e.clientX e.clientY

相对于当前元素左上角: e.offsetX e.offsetY

坑: 事件绑定在哪个元素，offset就只能获得距离那个绑定事件的元素左上角的偏移量。



|  |
| --- |
| 示例：鼠标点击拖拽图片到任意位置  <div id="pop"></div>  <script>  //查找id为pop的div  var div=document.getElementById("pop");  //定义开关变量控制div能否跟随鼠标一定  var canMove=false;  //定义两个变量，保存图片移动过程中不变的offsetX和offsetY  var offsetX,offsetY;  //当鼠标在div上按下时，打开开关  div.onmousedown=function(e){  canMove=true;  //获得绿线  offsetX=e.offsetX;  offsetY=e.offsetY;  }  //当鼠标在window范围内移动时  window.onmousemove=function(e){  //只有打开开关时  if(canMove==true){  //才随时获得鼠标的位置，并计算div的位置  //获得红线  //红线-绿线=蓝线  var left=e.clientX-offsetX;  var top=e.clientY-offsetY;  //设置div的top和left蓝线  div.style.left=left+"px";  div.style.top=top+"px";  }  }  //当鼠标在div上抬起时，关闭开关  div.onmouseup=function(){  canMove=false;  }  </script> |

**day04**

**一. BOM**: Browser Object Model

BOM: 专门操作浏览器窗口/软件的一套API

问题: 没有标准

何时: 希望操作除网页内容以外的浏览器窗口或读取浏览器软件的信息时

包括:

1. **window**: 3个角色

(1) 代替ES中的global充当全局作用域对象

(2) 包含所有ES，DOM和BOM的API

又包括: history, location, document, navigator, screen, event

(3) 代表当前打开的窗口

获取窗口大小: 2组:

完整的窗口大小: window.outerWidth

window.outerHeight

文档显示区大小: window.innerWidth

window.innerHeight

打开和关闭窗口:

打开窗口: window.open()

4种:

在当前窗口打开，可后退:

html:<a href="url" target="\_self">

js: open("url", "\_self");

在当前窗口打开，禁止后退:

js: location.replace("新url")

打开新的url网址，替换当前位置旧的url网址。

在新窗口打开，可打开多个:

html:<a href="url" target="\_blank">

js: open("url", "\_blank");

在新窗口打开，只能打开一个:

html: <a href="url" target="自定义窗口名">

js: open("url", "自定义窗口名");

原理: 浏览器内存中，每个窗口都有一个唯一的窗口名；

打开窗口时，也可通过target属性，为新窗口指定自定义的窗口名。

浏览器规定同名(name)的窗口只能打开一个；后打开的同名窗口会自动将先打开的同名窗口覆盖掉。

内置窗口名:

\_self: 自动获取当前窗口自己的名字，作为新窗口的名字。

\_blank: 不指定窗口名，在运行时，浏览器会自动随机分配窗口名。

关闭窗口: window.close()

2. **history对象**: 内存中保存当前窗口打开后，成功访问过的url的历史记录栈；

每成功访问一个url，就会向history中push一个url记录。

何时: 用程序模拟前进，后退或刷新时

如何: history.go(1) 前进一步

history.go(-1) 后退一步

history.go(-2) 后退两步

history.go(0) 刷新

3. **location对象**: 保存当前窗口正在打开的url的对象

属性: location.href 完整url

location.protocol 协议

location.host 主机名+端口号；比如localhost:3000

location.hostname 主机名: 比如localhost

location.port 端口号: 3000

location.pathname 相对路径

location.hash #锚点地址；比如：#top

location.search ?查询字符串

方法:

(1) 也可实现在当前窗口打开，可后退

location.assign("新url")

location.href="新url"

location="新url"

(2) 在当前窗口打开，实现禁止后退:

location.replace("新url")

原理: 用新url替换history中旧的url

(3) 刷新: 2种:

普通刷新: 优先从浏览器本地缓存中获取资源。缓存中没有或过期，才去服务器下载新的。

F5

history.go(0)

location.reload()

强制刷新: 跳过浏览器缓存，总是从服务器下载最新的资源

location.reload(true) true表示强迫

4. **navigator对象**: 保存浏览器配置信息的对象

何时: 只要想获得浏览器的配置信息时

包括:

(1) plugins：保存所有浏览器安装的插件信息的集合。

navigator.plugins["插件名"] 返回值为undefined，则未安装

|  |
| --- |
| 示例：  //判断浏览器是否安装浏览器pdf插件  if(navigator.plugins["Chrome PDF Viewer"]==undefined){  document.write(`未安装PDF插件<a href="javascript:;">点此下载</a><br>`);  }else{  document.write(`已安装PDF插件<br>`);  } |

(2) userAgent：保存浏览器名称和版本号的字符串

使用Microsoft Edge浏览器打开，按F12，在"仿真"中更改"用户代理字符串"，可以模仿其它浏览器。

MSIE ——> IE

Trident ——> IE version：11

Firefox ——> Firefox

OPR Chrome Safari ——> OPR

Edge Chrome Safari ——> Edge

Chrome Safari ——> Chrome

Safari ——> Safari

|  |
| --- |
| 示例：  document.write(navigator.userAgent+"<br>")  //用Edge打开->F12->仿真->用户代理字符串->选择不同的浏览器，查看不同的userAgent字符串  var ua=navigator.userAgent;  var browser,version;  //先判断当前浏览器的名称  //只能靠猜  //如果ua中有MSIE,当前浏览器就是IE  if(ua.indexOf("MSIE")!=-1){  browser="IE"  }else if(ua.indexOf("Trident")!=-1){  //否则,如果ua中没有IE,却有Trident内核，当前浏览器就是IE 11  browser="IE";  version=11;  }else if(ua.indexOf("Firefox")!=-1){  //否则,如果ua中有Firefox,当前浏览器就是Firefox  browser="Firefox"  }else if(ua.indexOf("OPR")!=-1){  //否则,如果ua中有OPR,当前浏览器就是OPR  browser="OPR"  }else if(ua.indexOf("Edge")!=-1){  //否则,如果ua中有Edge,当前浏览器就是Edge  browser="Edge"  }else if(ua.indexOf("Chrome")!=-1){  //否则,如果ua中有Chrome,当前浏览器就是Chrome  browser="Chrome"  }else if(ua.indexOf("Safari")!=-1){  //否则,如果ua中有Safari,当前浏览器就是Safari  browser="Safari"  }  document.write(browser+"<br>");  //求版本号:  //如果还不知道版本号时  if(version===undefined){  //在ua中找到浏览器名称的位置i  var i=ua.indexOf(browser);  //再加名称的长度+1，调到版本号的第一位，  i+=browser.length+1;  //再从i位置开始截取ua中的3位，转为整数  version=parseInt(ua.slice(i,i+3))  }  document.write(version) |

(3) cookieEnabled: 判断当前浏览器是否启用了cookie true/false

什么是cookie: 在客户端本地持久保存用户私密信息的小文件。

为什么: 程序中的数据，都是临时的。程序退出后，都释放。

何时: 只要希望程序关闭，数据也不丢失时

正课:

1. 什么是jQuery:

2. 如何使用jQuery:

3. 查找:

[ ]——里面内容，可写，可不写

1. 什么是jQuery:

jQuery是 第三方开发的 —— 先下载才能使用

执行DOM操作的 —— 学jQuery依然在学DOM

极简化的 —— 对DOM API的终极简化

函数库 —— jQuery中一切用函数来实现

为什么: 2个原因:

jQuery是DOM操作的终极简化:

简化了: 增删改查

事件绑定

动画

ajax

解决了大部分浏览器兼容性问题:

凡是jQuery让用的都没有兼容性问题:

何时: 在PC端，几乎所有旧的大型项目或框架的底层DOM操作，都是由jQuery实现的

2. 如何使用jQuery:

下载: jquery.com

版本: 1.x 兼容IE8

1.x.js 未压缩版本

拥有完备的注释，代码格式，见名知意的变量名

优点: 可读性好——适合学习或开发时使用

缺点: 字符太多，体积大，不便于下载和传输

1.x.min.js 压缩版本

去掉所有注释和代码格式，极简化所有变量名

缺点: 可读性差

优点: 字符少，体积小，便于下载和传输——生产环境

2.x 不再兼容IE8

3.x 不再兼容IE8，增加了新特性

运行在严格模式下

用for of 代替了 .each()方法

支持Promise，可以用.then()

如何使用:

(1) 先引入jQuery到页面中，再编写自定义js:

<script src="js/jquery-1.11.3.js"></script>

原理:

引入jquery-1.11.3.js，其实是在内存中添加了一种新的类型: jQuery

也包含两部分:

构造函数: function jQuery(){…}

专门用于创建jQuery类型的子对象

原型对象: jQuery.prototype

专门保存所有jQuery类型的子对象共有的简化版API

(2) 今后，只要想使用jQuery家简化版API，都要先创建jQuery家的子对象:

如何创建: 2种:

$("选择器"):

创建一个jQuery子对象 =相当于=> new jQuery()

用选择器找到符合条件的DOM元素，并将DOM元素保存到jQuery子对象中。

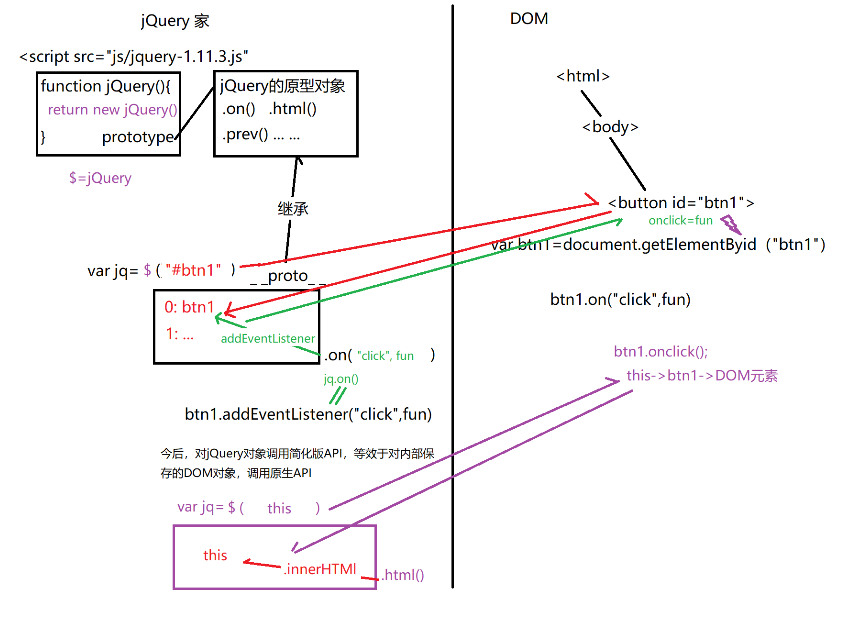
$(DOM元素) //this, e.target

创建一个jQuery子对象并将现有的一个DOM元素对象直接封装进jQuery子对象中

(3) 调用jQuery家的API:

jQuery子对象.简化版API()

对jQuery子对象调用简化版API，相当于对子对象中保存的DOM元素，执行原生的DOM API。由jQuery对象自动一对一翻译。



3. 查找元素:

按选择器查找:

何时: 只要什么元素都没有获得时，第一次查找，都用按选择器查找。

如何:

var $elems=$("选择器")

(1) jQuery中支持所有CSS选择器

基本选择器:

#id 元素 .class \* 选择器1,选择器2

层级选择器:

父 后代

父>直接子代

兄+弟

兄~弟弟们

属性过滤 等效于 CSS属性选择器

[属性名] [属性名=值] [属性名^=开头]

[属性名$=结尾] [属性名\*=内容] [属性名!=值]

状态过滤: :enabled :disabled :selected :checked

否定伪类: :not(其它选择器)

而且: 多个选择器无缝连用

(2) jQuery中又扩展出了个别新**兄弟选择器**

css中的子元素过滤 vs jQ中的基本过滤选择器(i为下标)

:first-child :first 父元素的第一个孩子 = .first()

:last-child :last 父元素的最后一个孩子 = .last()

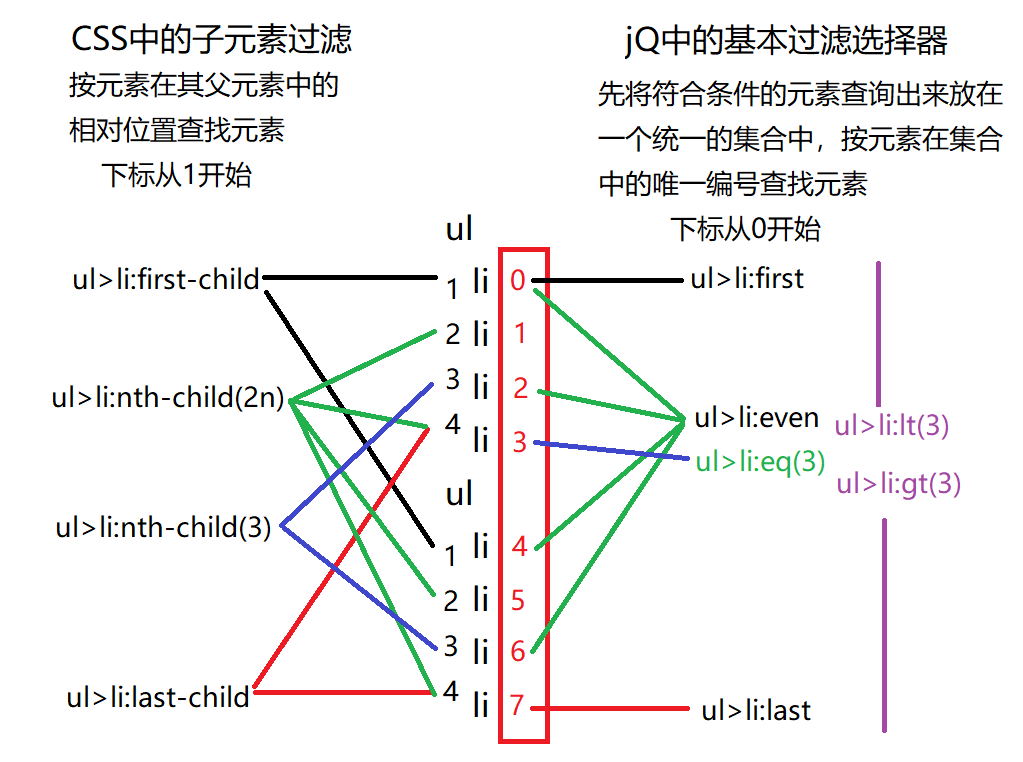
:nth-child(n) n从1开始 :eq(i) 下标等于i的元素

:nth-child(2n) 选择偶数 :even 下标为偶数

:nth-child(2n+1) 奇数 :odd 下标为奇数

:gt(i) 下标大于i的所有元素

:lt(i) 下标小于i的所有元素



|  |
| --- |
| 示例：  <style>  /\* 1. 使用css实现  .dropdown:hover>.dropdown-menu{  display:block;  } \*/  </style> |
| <body>  <h1>下拉菜单</h1>  <div class="dropdown">  <a data-trigger="dropdown" href="#">Dropdown</a>  <ul class="dropdown-menu">  <li><a href="#">Action</a></li>  <li><a href="#">Another action</a></li>  <li><a href="#">Something else here</a></li>  <li><hr></li>  <li><a href="#">Separated link</a></li>  </ul>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //2. 使用jQuery实现鼠标移入下拉  var $div=$("[data-trigger=dropdown]").parent();  //当鼠标移入时，添加事件($div，mouseover，function)到事件队列中  $div.on("mouseover",function(){  //让div下的最后一个孩子ul显示  $(this).children(":last").show();  //.css("display","block"); =简化于=> .show()  });  //当鼠标移出时  $div.on("mouseout",function(){  //让div下的最后一个孩子ul隐藏  $(this).children(":last").hide();  //.css("display","none"); =简化于=> .hide()  });  </script>  </body> |

正课:

1. 查找

2. 修改

3. 按节点间关系查找

4. 添加/删除/替换/克隆

1. 查找:

可见性过滤: jQuery新增

:visible 匹配显示的元素

:hidden 匹配隐藏的元素

display:none

<input type="hidden">

~~visibility: hidden~~

~~opacity:0~~

内容过滤: jQuery新增

:contains(文本) 匹配元素的内容中包含指定“文本”的元素

:has(选择器) 匹配包含满足选择器要求的子元素的父元素

:parent 选择内容非空的元素 =等效于=> :not(:empty)

:empty 选择内容为空的元素

|  |
| --- |
| 示例：  <div class="alert" id="alert1">  第一个警告框  </div>  <div class="alert" id="alert2">  <span class="close">×</span>  第一个警告框  </div>  <script>  //选择包含"提交"二字的按钮，变为绿色按钮  $("button:contains(提交)").css("background","#0f0");  //选中包含.close按钮的.alert元素，让它们变为红色的警告框  $(".alert:has(.close)").css("background","#f0f");  //选中不包含.close按钮的.alert元素，让它们变为绿色的警告框  $(".alert:not(:has(.close))").css("background","#0a0");  </script> |

表单元素过滤: jQuery新增:

:input 匹配所有表单元素: input select textarea button

:[type] 匹配type为指定类型的input元素:

比如： :text <input type="text">

:password <input type="password">

:...

方法：$("form").submit(function(){

return true/false; //手动控制form表单提交，返回true提交，返回false不提交

})

|  |
| --- |
| 示例：  <body>  <form>  用户名:<input disabled></br>  密码:<input type="password" disabled></br>  <input type="checkbox">我同意本站的使用条款<br>  <input type="submit" value="提交注册信息" disabled/>  </form>  <script src="js/jquery-1.11.3.js"></script>  <script>  //1.选中type为checkbox的input元素  var $chb=$(":checkbox");  //2.绑定单击事件处理函数  $chb.on("click",function(){  //3.修改checkbox外的其余表单元素  var $others=$(":input:not(:checkbox)")  //如果当前chb是选中的  if($(this).is(":checked")){  //4.修改元素为可以使用  $others.prop("disabled",false);  }else{  $others.prop("disabled",true);  }  })  </script>  </body> |

2. 修改:

(1) 内容: 3种:

原始HTML片段: .innerHTML（DOM）

$elem.html("html片段") 一个函数两用(获取、修改)

获取HTML片段：var html=$elem.html()

修改HTML片段：$elem.html("新html片段")

纯文本内容: .textContent（DOM）

$elem.text("文本内容") 一个函数两用

表单元素的值: .value（DOM）

$elem.val("值") 一个函数两用

(2) 属性:

DOM中：

HTML标准属性:

2种:

1. 核心DOM: getAttribute() setAttribute()

2. HTML DOM: .属性名 .属性名=值

状态属性

HTML DOM: .状态属性

自定义扩展属性:

核心DOM: getAttribute() setAttribute()

jQuery中：

核心DOM API: 一个函数两用

获取属性值：var value=$elem.attr("属性名")

修改属性值：$elem.attr("属性名","新值")

HTML DOM API: 一个函数两用

获取属性值：var value=$elem.prop("属性名")

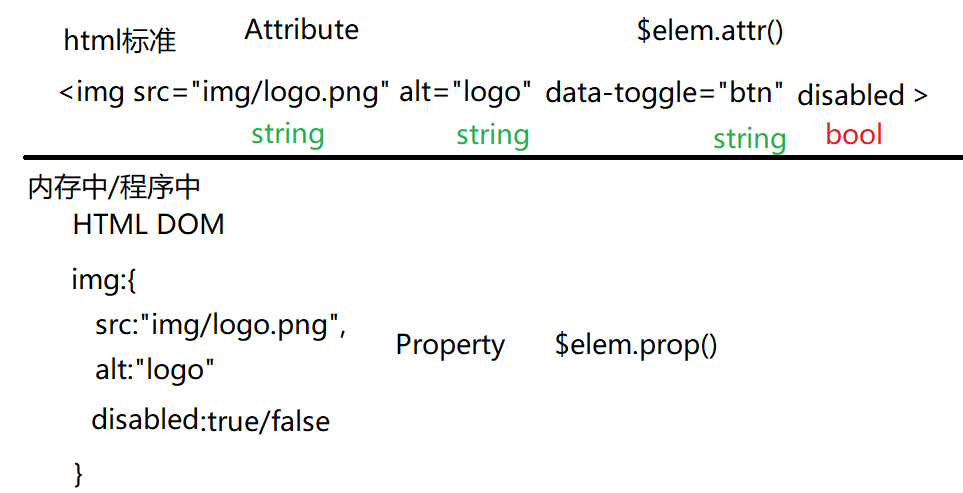
修改属性值：$elem.prop("属性名","新值")

四大状态属性: enabled disabled checked selected

只能用$elem.prop()——存放在内存中，布尔类型

自定义扩展属性: data-\*="..."

只能用$elem.attr()



问题: 一次只能修改一个属性，如果同时修改多个属性时，代码会很繁琐:

解决: 同时修改多个属性:

$elem.attr/prop({

属性:值,

... : ...

})

|  |
| --- |
| 示例：（点击图片，就更换为下一张图片）  <body>  <h1>操作元素的属性</h1>  <img src="img/1.jpg" alt="1">  <script src="js/jquery-1.11.3.js"></script>  <script>  //单击图片，切换下一张  $("img").on("click",function(){  //使用alt来临时储存数据  var alt=parseInt($(this).attr("alt"));  //每点击一次就将alt加1，超过图片数量就返回1从头开始  if(alt<4){  alt+=1;  }else{  alt=1;  }  //将src中的数字变为alt  var src=`img/${alt}.jpg`;  //把img中的src和alt同时更换  $(this).attr({src,alt})  /\*等效于  $(this).attr({  arc:src,  alt:alt  })  也等效于  $(this).attr("src",src);  $(this).attr("alt",alt);  \*/  });  </script>  </body> |

(3) 样式:

DOM中：

修改样式：elem.style.css属性="值"

获取属性：get.ComputedStyle()

jQuery中：

修改样式：$elem.css("css属性","值")

获取属性：var value=$elem.css("css属性")

简写：如果同时修改多个css属性时

$elem.css({

css属性:值,

… : …

})

问题: 如果同时修改多个css属性，用.css()代码繁琐

解决: 用操作class来批量修改样式

<button class="btn btn-danger active"

$elem.addClass("class1 class2 ...")

$elem.removeClass("class1")

$elem.hasClass("class1")

=等同于=> .is(".class1")

比如: if($btn.hasClass("down")) //如果有down

$btn.removeClass("down"); //就移除down

else//否则

$btn.addClass("down"); //就添加down

只要在有和没有一个class之间来回切换时，采用 .toggleClass("class")—如果没有，自动添加，如果有，自动清除！

福利：不用写单位

ODM中： div.style.width="120px"

jQuery中：#div.css({width:120})

|  |
| --- |
| 示例：  <style>  .alert{  border:1px solid #a33;  color:#922;  background:#fee;  padding:10px;  position:fixed;  overflow: hidden;  }  .alert>.close{  float:right;  font-weight:bold;  opacity:.5;  cursor:pointer;  }  .alert>.close:hover{opacity:1}  /\*定义淡入淡出动画的初始状态,常驻元素\*/  .fade{  top:50%;left:50%;  width:0px;height:50px;  margin-left:0px;margin-top:-35px;  opacity:0;  transition:all 1s linear;  }  /\*定义淡入淡出动画的结束状态\*/  .in{  top:50%;left:50%;  width:350px;height:50px;  margin-left:-185px;margin-top:-35px;  opacity:1;  }  </style> |
| <body>  <button id="btn">click me!</button>  <div class="alert fade">  <span class="close">×</span>  <p>您的浏览器版本太低!请下载最新版本的浏览器</p>  </div>  <script src="js/jquery-1.11.3.js"></script>  <script>  //点击按钮为对话框添加in class  $("#btn").on("click",function(){  $(".alert").addClass("in");  });  //点击x按钮移除对话框in class  $(".close").on("click",function(){  $(".alert").removeClass("in");  });  </script>  </body> |

jQuery API 三大特性:

自带遍历效果

|  |
| --- |
| 示例：（点击小图片，下方my-big中显示大图片）  <body>  <img src="img/1.jpg" data-target="img/1-l.jpg" class="my-small">  <img src="img/2.jpg" data-target="img/2-l.jpg" class="my-small">  <img src="img/3.jpg" data-target="img/3-l.jpg" class="my-small">  <img src="img/4.jpg" data-target="img/4-l.jpg" class="my-small">  <hr/>  <img src="img/1-l.jpg" class="my-big">  <script src="js/jquery-1.11.3.js"></script>  <script>  //找到class为my-small的img并绑定触发事件函数,jQuery自带遍历  $(".my-small").on("click",function(){  //找到小图片data-target的值(大图片的地址)  var src=$(this).attr("data-target")  //找到需要修改的元素，class为my-big的img  //将data-target的值，给小图片的src  $(".my-big").attr("src",src);  });  </script>  </body> |

一个函数两用

如果没有参数传入，就获取其内容；

如果通过参数传入了新值，就执行修改。

多数函数都自动返回正在操作的jQuery对象本身——>链式操作

3. 按节点间关系查找:

何时: 只要已经获得一个元素，找周围附近的元素时

DOM中: 2大类:

(1) 父子关系

elem.parentNode 获得elem的父节点/元素

jq => $elem.parent();

elem.children 获得elem的直接子元素

jq => $elem.children(["选择器"]);——[ ]的意思：选择器，可写，可不写

如果不写"选择器"参数，则默认拿全部直接子元素

如果写了"选择器"参数，则只获取符合选择器条件的部分直接子元素

比如: $ul.children() //获得所有li

$ul.children(".active") //仅获得所有li中class为active的部分li

elem.firstElementChild 获得elem下第一个子元素

jq => $elem.children(":first")

elem.lastElementChild 获得elem下最后一个子元素

jq => $elem.children(":last")

新增的API:

问题: .children()只能获得直接子元素

解决: 如何在所有后代中查找:

$elem.find("选择器")

比如: $ul.children() 只获取直接子元素li

$ul.find("li") 可获得ul下各级菜单中所有的li

(2) 兄弟关系

elem.previousElementSibling 获得elem的前一个兄弟元素

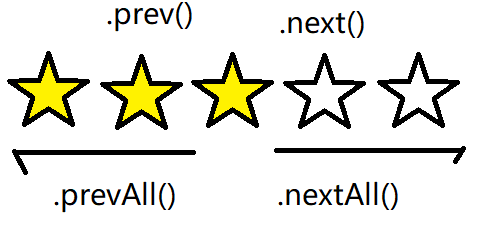
jq => $elem.prev();

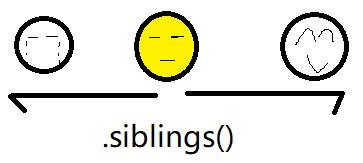
新增的API: $elem.prevAll() 选择当前元素之前所有兄弟元素

elem.nextElementSibling 获得elem的后一个兄弟元素

jq => $elem.next();

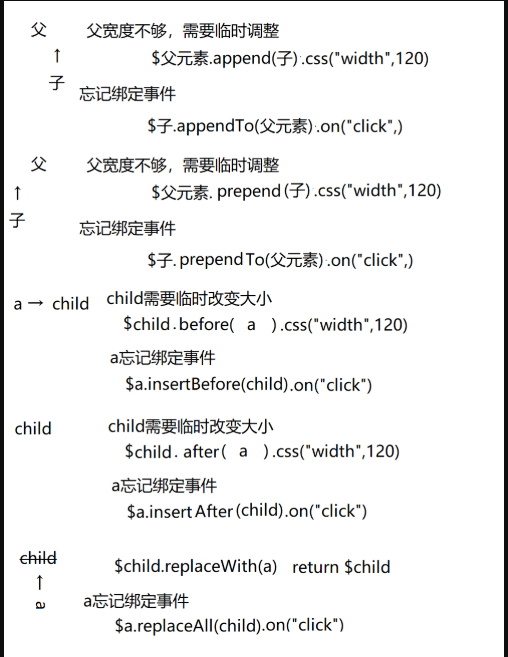
新增的API: $elem.nextAll() 选择当前元素之后所有兄弟元素

 新增的API: $elem.siblings(["选择器"]) 选择当除前元素外[满足选择器的]所有兄弟元素



|  |
| --- |
| 示例：  <body>  <h1>使用属性选择器实现标签页头的切换</h1>  <ul class="tabs">  <li class="active"><a data-toggle="tab" href="#">十元套餐</a></li>  <li><a data-toggle="tab" href="#">二十元套餐</a></li>  <li><a data-toggle="tab" href="#">三十元套餐</a></li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  $("ul.tabs>li").on("click",function(e){  e.preventDefault();  //先给自己加active class  //再给自己以外的所有兄弟去除active class  $(this).addClass("active")  //第3个特性，相当于执行完后return $(this)，后面可以连这写  .siblings().removeClass("active") //链式操作  })  </script>  </body> |

**1. 添加/删除/替换/克隆**

 添加: 2步:

(1) 用html片段创建新元素:

var $a=$(`<a href="http://tmooc.cn">go to tmooc</a>`);

(2) 将新元素添加到DOM树上（后面都可以对主语进行链式操作）

追加:

$父元素.append($a) .对父元素做其它操作 末尾追加

$a.appendTo(父元素).对a做其它操作 末尾追加

$父元素.prepend($a) 在开头插入

$a.prependTo(父元素) 在开头插入

中间插入：

$child.before($a) .对child做其它操作 将a插入到child之前

$a.insertBefore(child) .对a做其它操作 将a插入到child之前

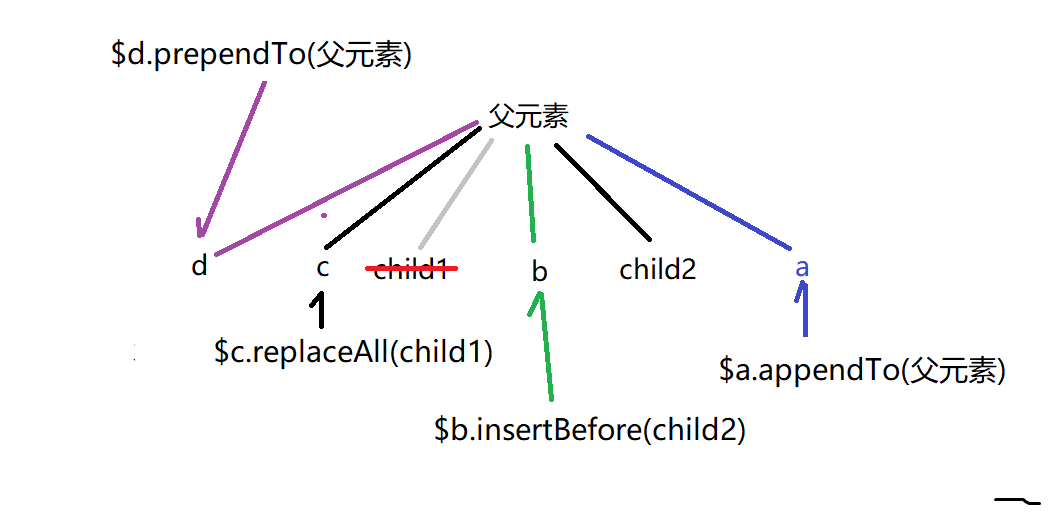
$child.after($a) 将a插入到child之后

$a.insertAfter(child) 将a插入到child之后

替换:

$child.replaceWith($a) 用a代替child

$a.replaceAll(child).对a做其它操作 用a代替child



删除: $elem.remove();

克隆: var $elem2=$elem.clone();

**2. 事件**

(1) 绑定事件处理函数:

.on("事件名",处理函数); //代替addEventListener()

对于常用的21种事件，可进一步简写为（on里面只要事件名和处理函数这2个参数）:

.事件名(处理函数)

(2) 解绑: .off("事件名",原函数名);——要解绑的函数必须是有名的函数

(3) 利用冒泡:

$父元素.on("事件名","选择器",function(){ … })

2个难题:

this 又可以用了——可以不用加参数e了 =相当于=> $(e.target)

~~if($xx.is("条件选择器"))~~ 不用再写

仅把选择器提前到第二个参数上，on会自动盘判断目标元素是否符合选择器条件的要求，只有符合条件的选择器，才进入函数中执行。

强调: "选择器"必须相对于$父元素内编

**3. 页面加载**

问题: 直接放在js文件中，不属于任何事件的代码，只要页面引入就立刻执行。如果在head中引入，极可能找不到页面元素，导致错误。

解决: 所有代码都要放在“页面加载完成”事件中，等待事件发生时，才执行。

2个"加载完成事件"

仅DOM内容加载完成：DOMContentLoaded 提前

仅等待DOM(HTML和JS)加载完就提前触发——>创建DOM树

何时: 与css，图片无关的代码，都要放在DOMContentLoaded中（比如：事件绑定，不需要网页图片加载完成，就可以更早操作）

如何: $(document).ready(function(){

… //会在document(DOM)准备好之后自动触发

})

简化：$().ready(function(){ … })

再简化：$(function(){ … })——几乎所有jQuery都要放在这个函数中。

整个网页加载完成: Load 延迟

需要等整个网页(HTML,JS,CSS,图片)都完成才触发——> DOM树上元素和CSS,图片结合排版，绘制成页面

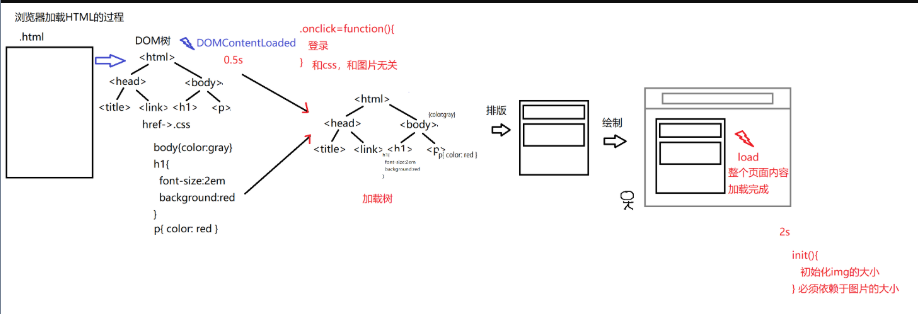
何时: 依赖css和图片的代码，只能放在load中

如何: DOM中：window.onload=function(){ … }

jQ中：$(window).load(function(){ //.load =相当于=> .addEventListener("load",...)

… //所有页面内容都加载完才能触发

})



**4. 鼠标事件**

.mouseover()

.mouseout()

问题: 即使进出子元素，也会频繁冒泡触发父元素上的处理函数——效率低

解决:

.mouseenter()

.mouseleave()

如果同时不绑定mouseenter和mouseleave

其实，只需要绑定一个hover:

$elem.hover( //等效于同时绑定了mouseenter和leave

function(){ ... }, //给mouseenter

function(){ ... } //给mouseleave

)

更简化: 如果是两个处理函数并且只改变一种属性Xxx，可用toggleXxx简写为一个

其实只要给一个处理函数即可：

$elem.hover(function(){

$(this).toggleXxx("…") //比如：$(this).toggleClass("hover")—进出时，添加或移除$elem的 .hover

})

**5. 模拟触发**(trigger)

没有点击按钮，也可以执行按钮的处理函数。

如何: $("按钮").trigger("事件名")

比如: $("#btnSearch").trigger("click")

自动触发"查询按钮"上的click处理函数

可以更简化:

$("按钮").事件名()

|  |
| --- |
| 示例：模拟百度搜索  <body>  <input type="text"><button>百度一下</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //查找触发事件的元素，并绑定事件处理函数  $("button").on("click",function(){  //获得input的值  var value=$("input").val()  //如果value不为空字符，才执行搜索  if(value.trim().length!==0){  console.log(`搜索内容为${value}的信息`);  }  })  //当在input上按下回车键再抬起时（不要使用keydown，不然按着不放，就会一直触发）  $("input").keyup(function(e){  //当按下的是回车键（回车键对应的keyCode代码为13）  if(e.keyCode===13){  //就执行和按钮完全相同的单击事件处理函数  //$("button").trigger("click")  $("button").click();  }  })  //让input文本框自动获得焦点  .focus();  </script>  </body> |

**1. 动画API**（异步）：

(1) 简单动画: 写死的三种动画效果

显示隐藏: $elem.show() $elem.hide() =合并=> $elem.toggle()

默认没有动画效果，瞬间显示隐藏

原理: .show()和.hide()没有参数时，默认使用display:block/none实现显示隐藏，所以不支持过渡

如何添加动画效果: 添加"动画持续时间"参数，毫秒数

原理: 用js定时器+DOM，反复修改宽、高、透明度属性模拟过渡效果

上滑下滑: slideUp() slideDown() =合并=> slideToggle()

淡入淡出: fadeIn() fadeOut() =合并=> fadeToggle()

缺点: 1. 用js程序写死的动画效果，不可维护。

2. 用js程序模拟的动画效果，效率不如css的transition

总结: 简单动画API很少使用

|  |
| --- |
| 示例：点击时，展开/隐藏部分商品列表  <body>  <ul id="list">  <li>尼康(234)</li>  <li>佳能(22)</li>  <li>索尼(932)</li>  <li>宾得(11)</li>  <li>爱国者(3234)</li>  <li>欧巴(32)</li>  <li>海鸥(2334)</li>  <li>卡西欧(334)</li>  <li>三星(834)</li>  <li>松下(234)</li>  <li>其它品牌(2343)</li>  </ul>  <button data-toggle="brandlist">精简显示品牌</button>  <script src="js/jquery-1.11.3.js"></script>  <script>  //为data-toggle="brandlist"绑定单击事件:  $("[data-toggle=brandlist]").on("click",function(){  var $lis=$("#list>li:gt(4):not(:last)");  //如果ul#list下有隐藏的元素，就显示全部  if($(this).html()=="显示所有品牌"){  $lis.show();  //顺便改当前按钮内容为"精简显示品牌"  $(this).html("精简显示品牌")  }else{//否则,隐藏5以上，但除最后一个元素之外的剩余元素  $lis.hide();  //顺便改当前按钮内容为"显示所有品牌"  $(this).html("显示所有品牌")  }  })  </script>  </body> |

(2) 万能动画: 对任意css属性的变化应用动画效果

$elem.animate({

css属性: 目标值,

… : …

},持续时间)

强调: 只能设置单个数值的属性

不支持颜色过渡和CSS3变换的过渡

也是用js定时器+DOM模拟的过渡效果

(3) 排队和并发:

排队: 多个css属性先后依次变化

如何: 对同一个元素，先后调用多个动画API，多个动画API是排队执行的

原理: 每个元素其实都有一个动画队列

调用动画API不是立刻执行动画的意思，而仅是将动画加入队列中等待执行。

并发: 多个css属性同时变化

如何: 在一个animate中同时定义多个css属性

(4) 停止动画: $elem.stop()

问题: 默认仅停止当前正在播放的第一个动画，队列中后续动画依然执行

解决: .stop(true) 停止动画，并清空队列！

(5) :animated 专门匹配正在播放动画的元素

(6) 动画结束后自动执行:

如何: 为动画API添加第三个参数: 回调函数

$elem.animate({

css属性: 目标值,

… : …

},

持续时间,

function(){ //回调函数就会在动画结束后自动执行！

$(this). … //jq：this-> animate点前的主语

})

|  |
| --- |
| 示例：点击小星星，变大、变淡.... 直至消失  <body>  <img id="s4" src="img/star.png"><br/>  <script src="js/jquery-1.11.3.js"></script>  <script>  $("#s4").on("click",function(){  alert("疼！")  $(this).animate({  top:-200,  left:-28, //原图片大小为128px，因为随着图片增大，中心位置会偏移，所以在图片增大的同时，设置偏移的量 (184-128)/2  width:184,  height:184,  opacity:0 //这个在隐藏后仍处于当前文档流，点击任然可以弹框“疼！”  },2000,  function(){  $(this).hide() //display:none; 脱离文档流消失  })  })  </script>  </body> |

**2. 类数组对象**操作:

jQuery查找的数据是类数组对象。

比如：var $lis=$("ul>li")

$lis => {0:li, 1:li, 2:li, 3:li}

(1) 遍历类数组对象:

$elems.each(function(i, elem){

$(elem) //elem->当前正在遍历的DOM元素对象，所以要使用$封装，才能使用jq的API

})

(2) 查找一个元素在jQuery对象中的位置:

var i=$elems.index(要查找DOM/jq对象)

找到返回元素下标，找不到返回-1。

|  |
| --- |
| 示例：评分，打几星  <body>  <h3>请打分</h3>  <ul class="score">  <li></li>  <li></li>  <li></li>  <li></li>  <li></li>  </ul>  <script src="js/jquery-1.11.3.js"></script>  <script>  //利用冒泡，为ul绑定点击事件  $("ul.score").on("click","li",function(){  //找到当前点击li的位置  var i=$("ul.score>li").index($(this));  //让自己以及以前的背景都变为红色  $(`ul.score>li:lt(${i+1})`).css("background","#f00");  // i之后的都变为白色  $(`ul.score>li:gt(${i})`).css("background","#fff");  })  </script>  </body> |

**3. 自定义**组件:

(1) 添加自定义函数:

定义自定义函数:

jQuery.fn.自定义函数=function(){ //fn=prototype

this ->将来调用当前函数的.前，已经是jQuery类型的子对象$elems，所以不用$( )来封装this

}

调用自定义函数:

$elems.自定义函数()

(2) 封装自定义插件:

插件/组件: 网页中拥有独立的HTML，CSS和js的页面区域。

为什么: 重用！

何时: 今后在页面中只要反复使用的独立页面区域都要封装为插件。

❶ 使用官方插件: jQuery UI

按照插件要求编写html内容，不用添加class

引入插件的jquery-ui.css文件

先引入jQuery.js，再引入jQuery ui.js

添加自定义脚本，找到插件的父元素，调用插件API

问题: vs bootstrap

兼容多种设备:

jQuery UI仅用于PC端网站。  
 bootstrap同时支持PC端和移动端网页。

可维护性:

jQuery UI采用侵入方式自动添加class，导致可维护性差。

bootstrap强迫手动添加class，但可维护性好。

❷ 用jQuery Ui方式封装自定义插件:

前提: 已经用传统的HTML,css和js实现了插件的效果；

封装插件的过程，其实是一个提取的过程：

提取插件的css到独立的css文件中；

要求：凡是一个插件的css样式类，必须都以一个统一的父级类名作为开头。

定义独立的js文件

a. 定义插件函数，向jQuery的原型对象fn中添加插件函数——侵入

\*侵入: 对插件中的元素自动添加所需的class

b. 绑定事件处理函数: 提取原页面中的事件绑定代码到插件函数中

使用jQueryUI风格的插件:

引入插件.css

按插件要求编写HTML(不加class)

引入jquery.js和插件.js  
 查找插件父元素调用插件API

|  |
| --- |
| 示例：实现手风琴效果  css中：  /\* accordion \*/  .accordion{width:80%; margin:0 auto;}  .accordion>.title{  background:#eee; border:1px soild #aaa;  padding:6px; font-size:1.5em;  font-weight:bold; cursor:pointer;  }  .accordion>.content{  border-left:1px solid #eee;  border-right:1px solid #eee;  }  .accordion>:last-child{  border-bottom:1px solid #eee;  }  .accordion .fade{  height:0;  opacity:0;  overflow:hidden;  transition:all .5s linear;  }  .accordion .in{  height:84px;  opacity:1;  } |
| js中自定义插件：  // 1.自定义函数  jQuery.fn.accordion=function(){  //this->$父元素  $parent=this;  //在jQuery的原型对象中添加自定义的API  $parent.addClass("accordion")  //给$parent下的奇数位置的三个div的class上添加title  .children(":even").addClass("title")  //给$parent下的奇数位置的三个div的兄弟class上添加content fade  .next().addClass("content fade")  //给$parent下的奇数位置的三个div的兄弟的第一个clas上添加in  .first().addClass("in");  //2.绑定事件  $parent.on("click",".title",e=>  $(e.target).next(".content").toggleClass("in")  .siblings(".content").removeClass("in")  );  } |
| html中：  <head>  <title> new document </title>  <meta charset="utf-8">  <style>  </style>  <link rel="stylesheet" href="my-ui/my-ui.css">  <script src="js/jquery-1.11.3.js"></script>  <script src="my-ui/my-ui.js"></script>  </head>  <body>  <h1>使用“高度动画”实现“手风琴”组件</h1>  <div id="my-accordion">  <div>《西游记》简介</div>  <div>一个和尚和四个动物的故事: Lorem ipsum dolor sit amet</div>  <div>《水浒传》简介</div>  <div>105个男人和三个女人的故事: Lorem ipsum dolor sit amet</div>  <div>《红楼梦》简介</div>  <div>一个男人和一群女人的故事: Lorem ipsum dolor sit amet</div>  </div>  <script>  $("#my-accordion").accordion();  // $(".accordion").on("click",".title",e=>  // $(e.target).next(".content").toggleClass("in")  // .siblings(".content").removeClass("in")  // );  </script>  </body> |

**4. AJAX**

$.ajax({ //{…}——参数解构，放置多个参数，不确定有没有

url:"服务端接口地址url",

type:"get/post",

data:"uname=dingding&upwd=123456", //jq支持对象：{uname:"dingding", upwd:"123456"}

dataType:"json" //自动将将json字符串执行，JSON.parse(),转为js中的数组和对象，可直接使用

//jQuery 版本为1.x和2.x

success:function(服务器返回结果—result){ //当请求成功时自动执行

...

}

//jQuery 版本为3.x才支持promise

.then(function(服务器返回结果—result){

…

})

})

|  |
| --- |
| 引入页头：  index.html  <head>  <script src="js/header.js"></script>  </head>  <body>  <header id="header" class="container"> </header>  </body> |
| header.js  $(function(){  //1.发送ajax请求，请求header.  $.ajax({  //请求头部html  url:"header.html",  type:"get",  success:function(header){  //用请求的header去替换当前页面的header  $(header).replaceAll("header")  //添加当前页面header的样式  $(`<link rel="stylesheet" href="css/header.css">`).appendTo("head")  }  })  }) |

**5. 跨域**

什么是: 一个域名向的网页，向另一个域名下的服务器发送请求

包括:

域名不同:

http://www.a.com/index.html -> http://www.b.com/service

二级域名不同:

http://mail.tedu.cn/index.html -> http://hr.tedu.cn/server

端口不同:

http://localhost:5500/index.html ->http://localhost:3000/index

协议不同:

http://localhost/index.html -> https://localhost/index

端口： :80 :443

相同机器上: 域名->IP

http://localhost/index.html -> http://127.0.0.1/index

问题: ajax不允许发送跨域请求:

错误: No "Access-Control-Allow-Origin" headers

解决:

方式一：服务区端设置允许跨域:

//也称CORS跨域： Cross Origin Resource Share

res.writeHead(200,{

"Access-Control-Allow-Origin":"\*" //\*—请求来源的所有地址

})

~~res.send(服务端返回结果)~~ 不允许

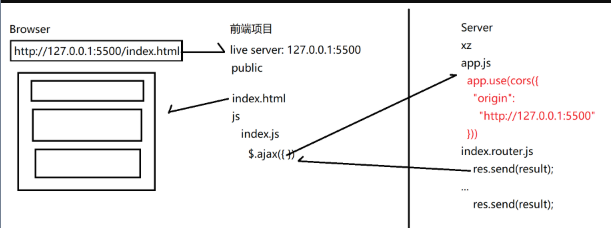
会和为了跨域而单独手写的reswriteHead()冲突，因为只能写一次，所以要拆开写。

//res.send=res.writeHead()+res.write(JSON.stringify(result))+res.end();

res.write(JSON.stringify(服务端返回结果/result))——只能转换为字符串发送

res.end();

|  |
| --- |
| 可以使用第三方插件cors解决，后面任然可以使用res.send(result)  //引入跨域请求  const cors=require("cors");  //启用cors跨域，只允许http://127.0.0.1:5500的请求访问该服务端  app.use(cors({  "origin":["http://127.0.0.1:8080","http://localhost:8080"],  credentials:true  })); |



方式二：jsonp跨域：json with padding(填充)

本质：既然ajax不能跨域，就请别的可以跨域的东西帮忙发送请求

~~<img> <link> <iframe>~~

选<script src="任意位置的js文件">代替ajax发送请求

本质上<script>拿到的js代码，并立刻执行。

方案1: 客户端: 请<script>向服务端发请求:

服务端: 将要返回的数据，填充在一条合法的js语句中返回。

问题: 服务端将客户端执行的程序写死了，众口难调

方案2: 客户端: 提前定义一个处理函数，可接收服务端的数据并执行在客户端本地自定义的操作

服务端: 将普通js语句修改为函数调用语句，且函数名必须和客户端提前定义的函数名一致

问题: 服务端将函数名写死，众口难调

方案3: 客户端: <script发送请求时，通过参数将客户端自定义的函数名发送到服务端

服务端: 接收函数名参数，动态拼接到要返回的js语句中

问题: 客户端<script>是写死的，只能在页面加载时，请求一次。无法按需反复请求

方案4: 在事件处理函数中，动态创建<script>

问题: <script>会堆积

方案5: 在处理函数的结尾，自动删除最后一个<script>

|  |
| --- |
| // 1. 定义一个 回调函数 handleResponse 用来接收返回的数据  function handleResponse(data) {  console.log(data);  };  // 2. 动态创建一个 script 标签，并且告诉后端回调函数名叫handleResponse  var body=document.getElementsByTagName('body')[0];  var script = document.createElement('script');  script.type = 'text/javascript';  script.src = 'http://www.laixiangran.cn/login?callback=handleResponse';  body.appendChild(script); |

其实$.ajax是支持跨域请求：

$.ajax({

url:"服务端接口地址url",

type:"get/post",

data:"uname=dingding&upwd=123456",

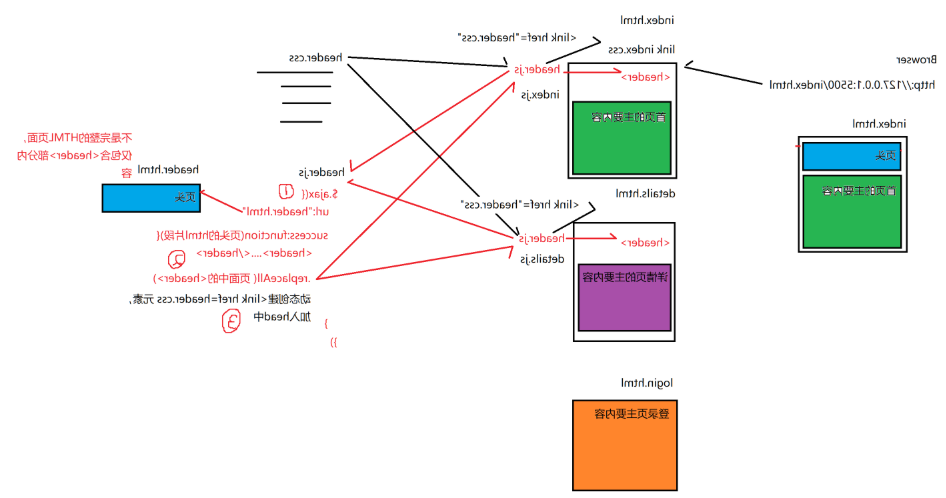
dataType:"jsonp",

…

})

|  |
| --- |
| 追加一种解决跨域方式：  jQuery插件：jquery-jsonp。  有第一种方式的基础，使用jsonp插件也就比较简单了，server端代码无需任何改动。  var url="http://localhost:8080/WorkGroupManagment/open/getGroupById" +"?id=1&callback=?";  $.jsonp({  "url": url,  "success": function(data) {  $("#current-group").text("当前工作组:"+data.result.name);  },  "error": function(d,msg) {  alert("Could not find user "+msg);  }  }); |

**项目中通过ajax请求页头：**



|  |
| --- |
| $(function(){  //1.发送ajax请求，请求header.  $.ajax({  //请求头部html  url:"header.html",  type:"get",  success:function(header){  //用请求的header去替换当前页面的header  $(header).replaceAll("header")  //添加当前页面header的样式  $(`<link rel="stylesheet" href="css/header.css">`).appendTo("head")  }  })  }) |

