JavaScript 基础

1. 第 1 章：JavaScript 简介

1.1. 什么是 JavaScript

1.1.1. 概念

1. 1995 年，布莱登 • 艾奇（美国人）在网景公司，用 10 天写的一门语言。
2. Javascript 是一门：动态的，弱类型的，解释型的，面向对象的脚本语言。
   (1) 动态：程序执行的时候才会确定数据类型。
   (2) 弱类型：数据类型不固定，可以随时改变。
   (3) 解释型：相对编译型的来说
      ① 编译型语言：程序在运行之前需要整体先编译（翻译）。
      ② 解释型语言：程序运行的时候，不会编译，拿一行解释执行一行。
   (4) 脚本：一般都是可以嵌在其它编程语言当中执行；

1.1.2. Java 和 JavaScript 的区别(了解)

Java 和 Javascript 没有任何关系；各自是各自。

起初叫做 livescript，后来改名叫做 JavaScript 借着 sun 公司的 Java 火了；

1.2. 为什么要有 JavaScript（作用）

刚开始只是为了解决表单验证而出现的。

而目前几乎什么都可以做。

表单验证

用户交互

游戏开发

后端开发

等等……

1.3. JavaScript 的组成部分

ECMAscript	JavaScript 的语法部分	主要包含 **JavaScript 语言语法**
DOM	文档对象模型	主要用来**操作页面**元素和样式
BOM	浏览器对象模型	主要用来**操作浏览器**相关功能

1.4. JavaScript 的第一个程序

alert()	以**弹出框**的形式输出内容
console.log()	以控制台**日志**形式打印内容
document.write()	将内容显示在页面上

```
alert('i love you~ zhao li ying!');//以弹出框的形式输出内容
console.log('i love you~ zhao li ying! two');//以后台日志的形式输出内容;
document.write('i love you~ zhao li ying! three');//把内容当做页面的内容去输出;
```

1.5. JavaScript 程序执行特点

1. JavaScript 的代码是**从上往下一行一行**执行的。
2. 每一行代表一句代码，分号是一句代码的结束标志，可以不写，但建议写上。
3. JavaScript 代码必须是**英文**书写。

4. JavaScript 的注释包含**单行注释//** 以及**多行注释/**/**

1.6.JavaScript 书写位置

1. **行内写法：**
   (1) **局限性很大**，只能针对事件进行添加，用的很少；
   (2) 它的代码**分离性最差**，可读性不强；
2. **内嵌写法：**
   (1) 在 body 的最下面 script 标签内去写我们的 js 代码，
   (2) 教学和项目用的最多
3. **外链写法：**
   (1) 在外部 js 文件当中去写 js 代码，
   (2) 最后通过 script 标签 src 引入到 html 当中，
   (3) 项目最终都会把代码文件进行分离；
   (4) 在外链的 script 标签之间不能去写代码

```
<!--第一种：行内的 js 写法几乎不用-->
<div id="box" onclick="alert('i love you~')">点击变色(行内的 js 写法)</div>
<!--第二种：这个 script 标签写在 body 内部的最下方，称为内嵌 js 写法-->
<script type="text/javascript">
    alert('i love you~ zhao li ying!');
    console.log('i love you~ zhao li ying! two');
    document.write('i love you~ zhao li ying! three');
</script>
<!--第三种 js 书写方式，外链式，项目的最后都要变为外链式去写-->
<script type="text/javascript" src="js/index.js"></script>
```

2. 第 2 章：变量

2.1. 什么是变量

变量：可以变化的量称作变量。

常量：不可以改变的量称作常量。

2.2. 为什么要有变量

1. 程序：代码的集合，一般指的是文件，静态的。
2. 进程：程序的一次执行过程，动态的。
   (1) 程序执行过程其实就是：输入数据、运算数据和输出数据的过程。
   (2) 数据需要存储，那么就需要容器，变量扮演的就是容器的角色。
3. 变量的本质归根结底就是一块内存（计算机的三大存储部件）

2.3. 变量的定义

1. 先定义再赋值
2. 定义的同时初始化
3. 不带 var 定义变量
4. 同时定义多个变量

```
//1、先定义后赋值
var a;//定义了一个变量，取名字叫 a
a = 100;//然后把 100 这个数据 存储到 a 这个变量当中
a = 200;//变量内部的东西是随时可以改变的；
console.log(a);//打印 a 内部的数据，


//2、初始化一个变量
```

```
var b = 100;//完全等价于  var b; b = 100;
console.log(b);
b = 300;
console.log(b);

//3、特殊情况特殊对待
c;//这样是错误的，它是在取 c 的值，而 c 没定义；
c = 500;//也是可以定义一个变量，但是它和带 var 定义的变量有区别，区别，后面说。现在你可以认为差不多；
console.log(c);

//4、一次定义多个变量；
var d = 20, e = 30;//var d = 20;var e = 30;是完全等价的；
var d = e = 30; //如果两个值相等，可以这么定义变量，但是后面的 e 是没有带 var 的
            // var d = 30; e = 30;
```

2.4.变量的命名规范

1. 标识符规则
   (1) 数字、字母、下划线及$组成
   (2) 不能以数字开头
   (3) 不能和关键字及保留字同名
2. 中英文大小写严格区分
3. 变量名字要见名思意
4. 变量名命名方法（大驼峰、小驼峰、下划线）

```
var a = 100;
var a1 = 200;
var 2a;//错误的，不能数字开头    报错 语法错误；

//语义化，存年龄
var age = 30;
//小驼峰是我们主打的写法
var PersonName;//大驼峰
var personName;//小驼峰
var person_name;//下划线
```

2.5.变量在内存当中的展现

两种方式交换两个变量的值

```
var a = 10;
var b = 20;
console.log(a,b);
a = 20;//修改 a 的值为 20
b = 10;//修改 b 的值为 10
console.log(a,b);
//上面的它不是交换；

//1、借用第三方变量实现两个变量值得交换
var c;
c = a;// = 出现 代表要把=右侧的值，赋值给 =左侧的变量， =左侧一定是变量，右侧只
```

```
要有值就行
console.log(a);//a 内部的值还是存在的
a = b;
console.log(a);
b = c;
console.log(a,b);


//2、不借助第三方变量实现交换（求和）；
a = a + b;//a = 30
b = a - b;// 30 - 20 = 10  b = 10
a = a - b;// 30 - 10 = 20  a = 20;
console.log(a,b);
```

3. 第 3 章：数据类型

3.1. 数据类型的分类

基本数据类型：数字 **Number**　字符串 **String**　布尔值 **Boolean**　**undefined**　**null**

对象数据类型：后面讲

3.2. 基本数据的使用

1. 数字：　整数　　小数　　二进制　八进制　　十六进制　科学计数法
2. 字符串：引号（单双引号都可以）包含的文本
3. 布尔值：true 和 false
4. undefined：undefined
5. null：null

```
//1、数字 Number
var num = 10;//整数
num = -10;
num = -1.23;//小数
//下面都是了解
num = 0b10;//存储二进制数，前面加 0b
num = 0o10;//存储八进制数，前面加 0o，可以简写成只加 0
num = 0x10;//存储十六进制数，前面加 0x
num = 1.23e3;//科学计数法
console.log(num);
//2、字符串 String  以单引号或者双引号包含的内容，统称字符串，其实就是我们说的文本内容
var str = '赵丽颖';
str = "asudgaklsdq8we2387,.f4jjashfdijiaas";
str = '10';//字符串 10
str = '      ';//空白串
str = '';//空串
str = "i'm fine";//单双引号根据实际情况，要交叉使用（内单外双，内双外单）；
console.log(str);
//3、布尔值 Boolean  用来表示真或者假   只有两个值
var bool = true;//真
bool = false;//假
console.log(bool);
//4、undefined 是一个基本数据类型    这个类型当中只有一个值，这个值就叫 undefined
//undefined 想要拿到这个值，定义变量不赋值，这个变量当中就是 undefined;
//undefined 的本意是未定义的意思，但是它在我们 js 当中并不是说变量没有定义，而是定义了变量没有赋值
```

```
var und;//里面会有一个值，undefined
console.log(und);
//5、null 是一个基本数据类型，这个类型当中也只有一个值，就叫 null;
//null 想要拿到这个值，必须定义了变量，自己赋值为 null,才能拿到 null
var nul = null;//null 经常用在初始化一个对象或者删除一个对象的时候使用;
console.log(nul);
```

3.3.基本数据类型的判断 typeof

1. typeof 是用来判断一个数据是什么类型的
2. typeof 之后的结果是什么
3. typeof 判断 null 的结果

```
var a = 10;//’ number ‘
a = '';//’ string ‘
a = true;//’ boolean ‘
a = undefined;//’ undefined ‘
a = null;//’ object ‘;//这个是一个设计缺陷，造成的;
console.log(typeof a);
//小面试题
console.log(typeof 100);//'number';
console.log(typeof typeof 100);//typeof 'number'//'string';
console.log(typeof typeof typeof 100)//'string';
```

4. 第 4 章：运算符和表达式

4.1.运算符和表达式的概念

运算符：就是参与**运算的符号**；

表达式：由变量或者常量与运算符组成的式子；**表达式是有值的**；

4.2.运算符的分类

4.2.1.算术运算符和表达式

1. 四则运算： + - * / %

   (1) %:求余或者取模 **/：求商 %：求余数**

   (2) %:作用很大，后期用的比较多：

   ① **判断一个数能否被另一个数整除**

   ② 可以**求出一个数字每个位上的数字**

   ③ 可以**求一个范围内的数**

   ④ 例如：任何的一个数字对 5 取余，肯定落在 0 - 4 之间

```
//算术运算符  + - * / %
//与其说讲运算符不如说讲表达式
var a = 10;
var b = 20;
console.log(a + b);//打印的是 a+b 这个整体的值，也就是说打印的是表达式的值；30
console.log(a - b);//-10
console.log(a * b);//200

console.log(a / b);//0.5    // /代表的是只求商，除不尽也要除，得到小数
```

```
console.log(a % b);//10      // %代表的是求余数，除不尽就拿余数

console.log(1725187231689 % 79 + 1); // 得到 1 - 79 之间的一个数字
```

2. 自增自减： ++ --

(1) ++ --只能和变量组成表达式

(2) 作用都是让变量的值自增或者自减 1

(3) a 和 a++ 是两个东西 a 代表变量 a++代表表达式

(4) 它们两个都是有值的；

(5) a++ 和 ++a 两个表达式的值

① ++在后，先把 a 的值赋值给表达式整体 a++ 然后让 a 自增 1；

② ++在前，先让 a 自增 1 再把 a 的值赋值给++a 这个表达式整体；

```
var a = 10;
//1
a++;//整体的值是 10 执行过后 a = 11 最终让 a 自增 1
++a;//整体的值是 12 执行过后 a = 12 最终也是让 a 自增 1
console.log(a);//12  打印 a 这个变量内部的值，
//2
console.log(a++);//10 打印 a++这个表达式整体的值 a = 11
console.log(a++);//11   a = 12
console.log(a);//12
//3
console.log(++a);//11 打印++a 这个表达式整体的值
console.log(a);  //11
```

作业：

```
var a = 7;
var b = 9;
var c = a + b++;
c = ++a + b++;
c = b + a++;
c = ++a + b++;
c = --a + b;
c = a + b--;
```

问：c 的值最终是多少；

4.2.2.赋值运算符和表达式

1. 赋值运算符： =
   (1) 是把= 右边的值 赋值给 = 左边的变量。
   (2) =左边一定是变量（或者对象属性）
   (3) =右边一定是有值的东西
2. 复合赋值运算符： += -= *= /= %=
   (1) a += b; <==等价于==> a = a + b
   (2) 其余的都是一样的

4.2.3.条件运算符和表达式

条件（比较）运算符 > < >= <= == != ===全等 !== 不全等

最终表达式的值都是布尔值；

4.2.4.逻辑运算符和表达式

&&与　　||或　　！非

1. 通用：　无论左右连接的是什么值都适用

(1) &&　　整个表达式的值：如果&&前面为真就取后面的值作为整个表达式的值如果&&前面为假就取前面的值作为整个表达式的值,后面的表达式根本不执行;

(2) ||　　整个表达式的值：如果||前面为真就取前面的值作为整个表达式的值，后面的表达式根本不执行;如果||前面为假就取后面的值作为整个表达式的值;

(3) ！　　要求后面必须值是布尔值，不是就转化为布尔值，必须是布尔值；

```
// && 与
//1）
var num = 0 && 10;//  非0即为真
console.log(num);
//2）
var a = 2;
console.log(0.2 && a++);//2  a3
console.log(a);


// ||或
//1
console.log(false || '赵丽颖');//
//2
var a = 30;
var b = 20;
console.log(a > b || a+b++);
```

```
console.log(a,b);


//!  非
console.log(!(a > 10));
console.log(!6);//非 0 即为真
console.log(!'zhaoliying');//非空即为真
```

2. 常用： 多个条件表达式的连接的时候

   (1) && 一假则假

   (2) || 一真则真

   (3) ! 非真即假 非假即真

```
var money = 100000;
var faceValue = 50;
console.log(money > 50000 && faceValue > 80);//false
console.log(money > 50000 || faceValue > 80);//true
console.log(!(money > 50000 || faceValue > 80))//false;
```

4.2.5.三目运算符和表达式

1. 一元 ++ -- ! +（正） -（负） 一侧有东西就能构成表达式
2. 二元 + - * / = 必须两侧都有东西才能构成表达式
3. 三元 ?: 三元（目）运算符只有一个
4. 三元表达式执行过程

   (1) 先看第一个表达式问号前面的值是否为真(不是布尔要转化为布尔)

   (2) 如果为真，则取冒号前面的值作为整个表达式的值，后面的不执行

(3) 如果为假，则取冒号后面的值作为整个表达式的值，前面的不执行

```
//1
var a = 10;
var b = 20;
var c = a < 0?a:b;
console.log(c);
//2
var a = 10;
var b = 10?++a:a--;
console.log(b, a);
//3
var a = 10;
var b = 20;
var c = a?a+b:b++;
console.log(a, b, c);
//4
var a = 10;
var b = 30;
var c = 0;
var d = (a > c || c)? a+b*c++ :a-c+b++;
console.log(a, b, c, d)
```

5. 第 5 章：数据类型转换

5.1. 强制类型转换

5.1.1. 其它类型转数字

Number() 强制将一个其它类型数据转化为数字类型，转不了就是 NaN

NaN（not a number）是数字类型，但 NaN 不是一个数字

1. 转化字符串
   (1) 如果字符串整体来看是一个数字，那么就转化为这个数字
   (2) 如果字符串整体来看不是一个数字，那么就转化为 NaN
   (3) 如果字符串是一个特殊的空字符串或者空白字符串，那么转化为 0
2. 转化 boolean
   (1) true 会转化为 1
   (2) false 会转化为 0
3. 转化 undefined　　undefined 会转化为 NaN
4. 转化 null　　null 会转化为 0

```
//1)字符串转数字
var str = 'ajsdg 赵丽颖';//NaNstr = '132quyeqoiu123';//NaN
str = '   -1.234';//数字-1.234
str = '    ';//0
str = '';//0
str = 'null';
console.log(Number(str));
//2)布尔值转化数字
var bool = true;//1
bool = false;//0
```

```
var result = Number(bool);
console.log(result);
//3）undefined 转数字
var und;//NaN
var result = Number(und);
console.log(result);
//4）null 转数字
var nul = null;//0
var result = Number(nul);
console.log(result);
```

5.1.2. 其它类型转字符串

String() 强制将一个其它类型数据转化为字符串类型

转化字符串没有什么特殊，给任何值，都会把这个值加引号变为字符串；

```
//1）数字转字符串
var num = 100;
var result = String(num);
console.log(result);
//2）布尔值转字符串
var bool = false;
var result = String(bool);
console.log(result);
//3）undefined 转字符串
var und;
```

```
var result = String(und);
console.log(result);
//4) null 转化字符串
var nul = null;
var result = String(nul);
console.log(result);
```

5.1.3. 其它类型转布尔值

Boolean() 强制将一个其它类型数据转化为布尔类型

1. 转化数字的时候，除了 **0** 和 **NaN** 是 **false**,其余都是 true;
2. 转化字符串的时候，除了**空字符串**是 **false**,其余都是 true
3. 转化 **undefined** 和 **null** 都是 **false**;

```
//1)数字转布尔
//除了 0 和 NaN 是 false,其余都是 true
var num = NaN;
var result = Boolean(num);
console.log(result);
//2)字符串转布尔值
//非空即为真
var str = 'asdfghald 打包会计师';
str = '1234124';
str = '    ';
str = '';
var result = Boolean(str);
```

```
console.log(result);
//3)undefined 转布尔  就是 false
var und;
var result = Boolean(und);
console.log(result);
//4)null 转布尔  就是 false
var nul = null;
var result = Boolean(nul);
console.log(result);
```

5.2. 手动类型转换

其实手动类型转换，就是从字符串当中提取数字，不是数字会先转化为数字

注意：字符串要求数字字符必须在字符串的前面（前面可以有空白字符），否则就是 NaN

parseInt()　　用来提取整数

parseFloat()　　用来提取浮点数（小数）

```
var str = '      112.2.3wUYGSKJFAKads';
var result = parseInt(str);
console.log(result);
result = parseFloat(str);
console.log(result);
```

5.3. 隐式类型转换

各种类型在适当的场合会发生隐式转换

主要是运算、比较及判等过程中

5.3.1. 同种数据类型之间的运算、比较和判等

1. 数字和数字

```
//数字和数字之间运算 比较  判等   都是该怎么算怎么算，没有任何坑；
var a = 10;
var b = 20;
//算术运算
console.log(a + b);
console.log(a - b);
console.log(a * b);
console.log(a / b);
console.log(a % b);
//比较
console.log(a > b);
console.log(a < b);
console.log(a >= b);
console.log(a <= b);
//判等
console.log(a == b);
console.log(a != b);


//下面都是了解
//数字的运算小数是不准确的，目前无法避免
console.log(0.1 + 0.2);
console.log(0/1);//0
```

```
console.log(1/0);//Infinity
console.log(-1/0);//-Infinity
console.log(0/0);//NaN
console.log(1%0);//NaN
console.log(0%0);//NaN
console.log(0%1);//0
//js 当中能表示的最大值和最小值
console.log(Number.MAX_VALUE);
console.log(Number.MIN_VALUE);
```

2. 字符串和字符串

```
var str1 = 'abc';
var str2 = 'acc';
//运算 除了+是拼接字符串，其余转数字
console.log(str1 + str2);//拼接字符串
console.log(str1 - str2);
console.log(str1 * str2);
console.log(str1 / str2);
console.log(str1 % str2);
//比较
//两个字符串比较 比较的是字符的 Unicode 码从左左到右依次比较每个字符，出现大小的结果就停止;
//0   48
//a   97
console.log(str1 > str2);
console.log(str1 < str2);
```

```
console.log(str1 >= str2);
console.log(str1 <=str2);
//判等
//同种数据在判等的时候，不会发生任何转化，就看两边一样不一样，一样就是相等，不一样就不相等；
console.log(str1 == str2);
console.log(str1 != str2);
```

3. 布尔和布尔

```
布尔和布尔
var bool1 = true;
var bool2 = false;
//运算，全部转数字
console.log(bool1 + bool2);
console.log(bool1 - bool2);
console.log(bool1 * bool2);
console.log(bool1 / bool2);
console.log(bool1 % bool2);

//比较，全部转数字
console.log(bool1 > bool2);
console.log(bool1 < bool2);
console.log(bool1 >= bool2);
console.log(bool1 <= bool2);

//同种数据在判等的时候，不会发生任何转化，就看两边一样不一样，一样就是相等，不一样就不相等
```

```
console.log(bool1 == bool2);
console.log(bool1 != bool2);
```

4. undefined 和 undefined

```
undefined 和 undefined
var und1;
var und2;
//运算，全部转数字
console.log(und1 + und2);
console.log(und1 - und2);
console.log(und1 * und2);
console.log(und1 / und2);
console.log(und1 % und2);
//比较，全部转数字
console.log(und1 > und2);
console.log(und1 < und2);
console.log(und1 >= und2);
console.log(und1 <= und2);
//判等 同种数据在判等的时候，不会发生任何转化，就看两边一样不一样，一样就是相等，不一样就不相等
console.log(und1 == und2);
console.log(und1 != und2);
console.log(NaN == NaN);
```

5. null 和 null

```
null 和 null
var null = null;
```

```
var nul2 = null;
//运算，全部转数字
console.log(null + nul2);
console.log(null - nul2);
console.log(null * nul2);
console.log(null / nul2);//NaN 1/0  Inifinity   0/0 是 NaN
console.log(null % nul2);
//比较，全部转数字
console.log(null > nul2);
console.log(null < nul2);
console.log(null >= nul2);
console.log(null <= nul2);
//判等  同种数据在判等的时候，不会发生任何转化，就看两边一样不一样，一样就是相等，不一样就不相等
console.log(null == nul2);
console.log(null != nul2);
```

5.3.2. 不同数据类型之间的运算、比较和判等

1. 数字和其它数据类型

```
//1、数字和字符串
var num = 100;
var str = 'zhaoliying';
//运算
//不同数据运算的时候，只要碰到字符串，+法都是拼接字符串，其余的算法都是转数字
console.log(num + str);//拼接字符串
```

```
console.log(num - str);
console.log(num * str);
console.log(num / str);
console.log(num % str);
//比较  全部转数字
console.log(num > str);
console.log(num < str);
console.log(num >= str);
console.log(num <= str);
//判等  不同数据类型之间的判等，也是要转数字
console.log(num == str);
console.log(num != str);

//2、数字和布尔值  全部转数字
var bool = true;
console.log(num + bool);
console.log(num - bool);
console.log(num * bool);
console.log(num / bool);
console.log(num % bool);
//比较  全部转数字
console.log(bool > num);
console.log(num < bool);
console.log(num >= bool);
console.log(num <= bool);
//判等  不同数据类型之间的判等，也是要转数字
console.log(num == bool);
```

```
console.log(num != bool);


//3、数字和 undefined   全部转数字
var und;
console.log(num + und);
console.log(num - und);
console.log(num * und);
console.log(num / und);
console.log(num % und);
//比较   全部转数字
console.log(num > und);
console.log(num < und);
console.log(num >= und);
console.log(num <= und);
//判等   不同数据类型之间的判等，也是要转数字
console.log(num == und);
console.log(num != und);


//4、数字和 null   全部转数字
var nul = null;
console.log(num + nul);
console.log(num - nul);
console.log(num * nul);
console.log(num / nul);
console.log(num % nul);
//比较   全部转数字
console.log(num > nul);
```

```
console.log(num < nul);
console.log(num >= nul);
console.log(num <= nul);
//判等  不同数据类型之间的判等，也是要转数字
console.log(num == nul);
console.log(num != nul);
```

2. 字符串和其它数据类型

```
//1）字符串和布尔
var str = 'yangmi';
var bool = false;
//运算
console.log(str + bool);//拼接字符串
console.log(str - bool);
console.log(str * bool);
console.log(str / bool);
console.log(str % bool);
//比较
console.log(str > bool);
console.log(str < bool);
console.log(str >= bool);
console.log(str <= bool);
//判等
console.log(str == bool);
console.log(str != bool);
// 2）字符串和 undefined
var und;
```

```
//运算
console.log(str + und);//拼接字符串
console.log(str - und);
console.log(str * und);
console.log(str / und);
console.log(str % und);
//比较
console.log(str > und);
console.log(str < und);
console.log(str >= und);
console.log(str <= und);
//判等
console.log(str == und);
console.log(str != und);
//3）字符串和 null
var nul = null;
//运算
console.log(str + nul);//拼接字符串 'yangminull'
console.log(str - nul);
console.log(str * nul);
console.log(str / nul);
console.log(str % nul);
//比较
console.log(str > nul);
console.log(str < nul);
console.log(str >= nul);
console.log(str <= nul);
```

```
//判等
console.log(str == nul);
console.log(str != nul);
```

3. 布尔值和其它数据类型

```
var bool = true;
// 1)布尔和 undefined
var und;
//运算
console.log(bool + und);
console.log(bool - und);
console.log(bool * und);
console.log(bool / und);
console.log(bool % und);
//比较
console.log(bool > und);
console.log(bool < und);
console.log(bool >= und);
console.log(bool <= und);
//判等
console.log(bool == und);
console.log(bool != und);

// 2)布尔和 null
var nul = null;
//运算
console.log(bool + nul);
```

```
console.log(bool - nul);
console.log(bool * nul);
console.log(bool / nul);
console.log(bool % nul);
//比较
console.log(bool > nul);
console.log(bool < nul);
console.log(bool >= nul);
console.log(bool <= nul);
//判等
console.log(bool == nul);
console.log(bool != nul);
```

4. undefined 和 null

```
var und;
var nul = null;
//运算
console.log(und + nul);
console.log(und - nul);
console.log(und * nul);
console.log(und / nul);
console.log(und % nul);
//比较
console.log(und > nul);
console.log(und < nul);
console.log(und >= nul);
```

```
console.log(und <= nul);
//判等
console.log(und == nul);//undefined 和 null 判等的时候，不会转化，特殊情况，他俩相等
console.log(und != nul);
```

5.4. 类型转换总结

1. 判等的时候：
   (1) 先看两边是不是同种数据类型，如果是直接看是不是一样；如果不是那么两边都转数字
   (2) 当遇到 null 的时候;会有特殊情况发生,
      ① 特殊情况：空串和 null 不相等
      ② 特殊情况：false 和 null 不相等
      ③ 特殊情况：0 和 null 不相等
      ④ 特殊情况：undefined 和 null 相等；
2. 运算和比较情况下：
   (1) 第一步：先看是不是 +
      ① 如果是 + 看有没有字符串，如果有就是拼接字符串
   (2) 第二步：再看是不是比较
      ① 如果是比较，看是不是两边都是字符串
      ② 如果两边全是字符串，比较的是字符串的 Unicode 码
   (3) 第三步：其余情况全部转数字
3. NaN:
   (1) 所有的东西和 NaN 进行算术运算都是 NaN
   (2) 所有的东西和 NaN 进行比较大小都是 false
   (3) 所有的东西和 NaN 都不相等（包括自己）

4. 全等和不全等：（不会出现类型转换）

(1) 他们在判等的时候，必须类型和值都相同。

(2) 如果有一个不一样，就不全等；

作业：晚自习强化练习

1、字符串拼接：个人简介

姓名 年龄 性别 期望薪资使用变量存储

console.log('我的名字叫**，我的年龄是**岁，我的性别是**，我期望毕业薪资是**K');

2、数据类型转换（课堂案例）

3、求出下面表达式的值

```
NaN + NaN
parseInt('    12.345.67sdjfg');
parseFloat('    12.345.67sdjfg');
parseFloat('    iloveyou12.345.67sdjfg');
null == ''
null == null
null == undefined
null == false
NaN == NaN
'100' + NaN
Number(NaN)
Boolean(NaN)
true == '       ';
true + '123';
+'45'-false;
```

6. 第 6 章：流程控制语句

6.1. 语句结构分类

顺序结构　分支结构　　循环结构

6.2. if 分支(判断)

6.2.1. if 单分支

1. 语法

if(一般都是表达式，但最终只要有值就行，这个值最终会转化为布尔值){

　　代码块；

}

2. 执行过程

　　(1) 第一步：先计算小括号当中的值，注意表达式、变量和值都是有值的

　　(2) 第二步：把最终小括号当中的值转化为布尔值

　　(3) 第三步：根据小括号当中值的真假决定是否执行花括号里的代码块

　　　① 如果最终的值是 **true** 就会执行花括号内部的代码块

　　　② 如果最终的值是 **false** 就跳过花括号执行下面的代码

```
//输入家产，如果家产大于 100 万，就去环游世界
var money = parseInt(prompt('请输入您的家产'));
if(money > 1000000){
    console.log('带你去东京和巴黎~');
}
```

6.2.2.双分支

1. 语法

if(一般都是表达式，但最终只要有值就行，这个值最终会转化为布尔值){

代码块；

}else{

代码块；

}

2. 执行过程

(1) 第一步：先计算小括号当中的值，注意表达式、变量和值都是有值的

(2) 第二步：把最终小括号当中的值转化为布尔值

(3) 第三步：根据小括号当中值的真假决定执行哪个花括号里的代码块

① 如果最终的值是 true 就会执行 if 花括号内部的代码块

② 如果最终的值是 false 就会执行 else 花括号内部的代码块

3. 双分支特殊情况

如果 if 双分支语句每个分支当中只有一条语句，可以改写为三元表达式。

```
//1、天气好就去看电影，不好就写代码
//如果我们碰到只有两种状态的数据，一般都是使用布尔值
var weather = true;
if(weather){
    console.log('我们一起去电影院');
}else{
    console.log('我们就在家写代码');
}
console.log('heihei');
//2、输入一个数，这个数如果大于 0 就-1，不大于 0 就+1
var num = parseFloat(prompt('请输入一个数字'));
```

```
if(num > 0){
    num--;
}else{
    num++;
}
console.log(num);
//双分支，并且双分支每个分支里面的代码只有一行，此时，我们可以把它改为三元表达式去写
console.log(num > 0?num--:num++);
console.log(num);
```

6.2.3.多分支

1. 语法

if(一般都是表达式，但最终只要有值就行，这个值最终会转化为布尔值){

代码块

}else if(一般都是表达式，但最终只要有值就行，这个值最终会转化为布尔值){

代码块

}else if(一般都是表达式，但最终只要有值就行，这个值最终会转化为布尔值){

代码块

}else{

代码块

}

2. 执行过程

(1) 第一步：先计算 if 小括号当中的值，注意表达式、变量和值都是有值的

(2) 第二步：把最终小括号当中的值转化为布尔值

(3) 第三步：根据小括号当中值的真假决定是执行 if 花括号代码块还是继续判断 else if 中条件

① 如果最终的值是 **true** 就会执行 if 花括号内部的代码块

② 如果最终的值是 **false** 就会判断接下来的 else if 条件（步骤同上）

(4) 第四步：如果 if 和所有的 else if 小括号内最终值都是 false，就执行 else 花括号当中的代码块，else 内部如果没有代码块，else 可以省略不写

3. 注意事项

(1) 多分支是从上到下依次去判断 if 后的小括号值是否为真

(2) 如果有为真的，就会执行相应花括号的代码块，其它的不再执行，也就是我们认为的跳楼现象

```
//1、输入体重判断属于什么样的体型
var weight = parseFloat(prompt('请输入您的体重'));//正常的成年人，小孩不算
if(weight >= 70 && weight < 90){
    console.log('骨感美~');
}else if(weight >= 90 && weight < 120){
    console.log('性感美~');
}else if(weight >= 120 && weight < 160){
    console.log('丰满美~');
}else if(weight >= 160 && weight < 200){
    console.log('肉感美~');
}else if(weight >= 200 && weight <= 300){
    console.log('胖胖美~');
}else{
    console.log('您输入的体重不合法,是 0 - 300 之间');
}
//2、输入一个数  如果这个数是在 0 - 100 之间  就让这个数+5，然后输出
//如果这个数是在 100 - 200 之间  就让这个数-5，然后输出
```

```
//如果这个数是在 200 - 300 之间 就让这个数*5，然后输出
//如果这个数是在 300 - 500 之间 就让这个数/5，然后输出
//如果这个数不是在 0 - 500 之间 原样输出
var num = parseFloat(prompt('请输入一个数字'));
if(num >= 0 && num < 100){
    num += 5;
    //console.log(num);
}else if(num >= 100 && num < 200){
    num -= 5;
    //console.log(num);
}else if(num >= 200 && num < 300){
    num *= 5;
    //console.log(num);
}else if(num >= 300 && num <= 500){
    num /= 5;
    //console.log(num);
}//else{
    //假设最终的这个 else 当中 没有代码 可以省略不写
//console.log(num);
//}
console.log(num);
```

6.3. switch....case 分支语句

6.3.1. 语法

switch(有值的东西，最后这里面是一个值，不会进行转化){

case 值：

代码块；

break;

case 值：

代码块；

break;

default:

代码块；

break;

}

6.3.2. switch 语句的作用

switch...case 语句的出现主要是为了解决 if 多分支，分支过多造成的代码可读性差。

6.3.3. switch 语句执行过程：

1. 先求出小括号当中的值，这个值不会做任何转化
2. 接着会拿这个值从上到下和所有的 case 标号后面的值进行绝对判等（全等）
3. 如果判等成功，就执行对比成功这个 case 标号下面的代码块；
4. 如果判等不成功，就接着往下对比
5. 如果所有的 case 标号都判等不成功，最终执行 default 的代码块

```
//1、输入一个数，根据输入的数字，打印是星期几
//if 写法
var num = parseInt(prompt('请输入一个数'));
if(num === 1){
    console.log('周一');
}else if(num === 2){
    console.log('周二');
}else if(num === 3){
    console.log('周三');
}else if(num === 4){
    console.log('周四');
}else if(num === 5){
    console.log('周五');
}else if(num === 6){
    console.log('周六');
}else if(num === 7){
    console.log('周日');
}else{
    console.log('请输入 1 - 7 的数字')
}
//switch 写法
switch(num){
    case 1:
        console.log('周一');
        break;//跳楼，跳出 switch 语句，没有 break，那么代码会继续往下执行后面的代
码块
    case 2:
```

```
        console.log('周二');
        break;
    case 3:
        console.log('周三');
        break;
    case 4:
        console.log('周四');
        break;
    case 5:
        console.log('周五');
        break;
    case 6:
        console.log('周六');
        break;
    case 7:
        console.log('周日');
        break;
    default:
        console.log('请输入 1 - 7 之间的数字');
        break;
}


//2、输入分数，判定优良差和不及格
//if 写法
var score = parseFloat(prompt('请输入你的分数'));
if(score >= 90 && score <= 100){
    console.log('A');
```

```
}else if(score >= 80 && score < 90){
    console.log('B');
}else if(score >= 70 && score < 80){
    console.log('C');
}else if(score >= 60 && score < 70){
    console.log('D')
}else if(score >= 0 && score < 60){
    console.log('不及格，重考');
}else{
    console.log('请输入合法的分值 0 - 100');
}


//switch 写法
switch(true){//假设我们 switch 后面括号当种的值，可能有很多种，那么我们就应该这样去考虑
    case score >= 90 && score <= 100:
        console.log('A');
        break;
    case score >= 80 && score < 90:
        console.log('B');
        break;
    case score >= 70 && score < 80:
        console.log('C');
        break;
    case score >= 60 && score < 70:
        console.log('D');
        break;
```

```
    case score >= 0 && score < 60:
        console.log('不及格，重考');
        break;
    default:
        console.log('请输入合法的分值 0 - 100');
        break;
}
```

6.4. for 循环

6.4.1. 循环的语法

for(一般初始化表达式；一般都是条件表达式；一般自增自减表达式){
    循环体（代码块）
}

6.4.2. 执行过程：

1. 第一次循环：
   (1) 首先执行初始化表达式
   (2) 接着执行条件表达式
   (3) 然后再去执行循环体
   (4) 最后执行自增自减表达式
2. 非第一次：
   (1) 执行条件表达式；
   (2) 执行循环体
   (3) 最后执行自增自减表达式

3. 每一次都要执行第二个条件表达式：
   (1) 当条件表达式的值为真的时候，会继续执行循环体
   (2) 当条件表达式的值为假的时候，代表循环结束，执行下面的代码
   (3) 如果首次执行的时候条件表达式的值就为假，那么循环一次都不执行

6.4.3.注意事项：

1. for 循环初始化表达式只是在第一次循环的时候执行，后面的循环全部都不执行了
2. for 循环当中的循环变量 i 有两个作用
   (1) 控制着循环的次数
   (2) 循环变量同时每次也是有不同值的

6.4.4.案例

```
// 1、打印 1-100 之间的整数
for(var i = 1; i <= 100; i++){
    console.log(i);
}
console.log(i);

// 2、打印 1-100 之间的偶数
for(var i = 1; i < 101; i++){
    if(i % 2 === 0){
        console.log(i);
    }
}
```

```
// 3、计算 1 到 100 的和
// 累加器思想
var sum = 0;
for(var i = 1; i <= 100; i++){
    sum += i;//sum = sum + i;
}
console.log(sum);

// 4、计算 1-100 之间所有偶数的和
var sum = 0;
for(var i = 1; i <= 100; i++){
    if(i % 2 === 0){
        sum += i;//sum = sum + i;
    }
}
console.log(sum);

// 5、求 100 的阶乘    1*2*3*4*。。。。*100
var product = 1;
for(var i = 1; i <= 100; i++){
    product *= i;//sum = sum + i;
}
console.log(product);

// 6、求 1! +2! +3! +……20!
var sum = 0;
var product = 1;
```

```
for(var i = 1; i <= 20; i++){
    product *= i;
    // 1!*2 ==== 2  ==== 2!
    // 2!*3 ==== 6  ==== 3!
    // 3!*4 ==== 24 ==== 4!
    // n! = (n-1)!*n
    sum += product;// sum = sum + product
}
console.log(sum);
```

6.5.for 嵌套循环

6.5.1.嵌套循环的定义：

在一个循环当中又会出现另外一个循环，外层循环执行一次，内层循环执行一轮

6.5.2.案例

```
//循环嵌套
//循环嵌套，循环的内部又是一个循环
//特点： 外层循环执行一次，内层循环执行一轮
//循环嵌套，嵌套的层次不宜过深；时间复杂度越大，代表效率越低
for(var i = 0; i < 100; i++){//100
    for(var j = 0; j < 100; j++){//100*100
        console.log('i love you~');
    }
```

```
}
//1、打印图形 打印矩形
//每次只能打印一个星  *
//外层循环控制行，内层循环控制列
//*****
//*****
//*****
//*****
//*****
for(var i = 0; i < 5; i++){
    //内部循环是为了打印每一行的星
    for(var j = 0; j < 5; j++){
        document.write('*');
    }
    //打印完一行换行要加
    document.write('<br>');
}


//2、打印图形 打印直角三角形
//*
//**
//***
//****
//*****
for(var i = 0; i < 5; i++){
    for(var j = 0; j < i + 1; j++){
        document.write('*');
```

```
    }
    document.write('<br>');
}
//3、打印 99 乘法口诀表
//1 * 1 = 1
//1 * 2 = 2 2 * 2 = 4
//1 * 3 = 3 2 * 3 = 6 3 * 3= 9
for(var i = 1; i <= 9; i++){
    for(var j = 1; j < i + 1; j++){
        //'1 * 1 = 1'     i*j = i*j
        document.write(j+'*' + i +'=' + i * j +'     ');
    }
    document.write('<br>');
}
```

作业：

1. 能手写乘法口诀表,第二天用纸默写
2. 针对 for 循环的练习一个

   循环 1-100 之间的所有数据，打印所有能被 3 或者 7 整除的数字
3. 针对 switch case 练习一个

   输入年龄 判定是童年少年青年中年老年
4. 熟练掌握课堂案例；

6.6. for 循环的特殊情况

6.6.1.语法

for 循环小括号内部，可以不写表达式，但是分号必须写。

```
//for 循环一般情况
for(var i = 0; i < 100; i++){
    console.log('i love you~' + i);
}
//特殊情况  for()内部的表达式可以不写，但是必须带上分号
var i = 0;
for(;i < 100;){
    console.log('i love you~' + i);
    i++;
}
```

6.6.2.死循环

1. for 循环的第二个表达式决定了循环是否继续，如果第二个表达式的值，永远为真，那么就构成死循环。
2. 在我们 js 当中死循环是不应该出现的，出现死循环用户体验相当差。

```
//for 循环的死循环
for(;true;){
    console.log('i love you~');
}
//最简单的死循环
for(;true;); //死循环   循环体内部如果没有代码块，可以省略大括号
for(;;);//死循环  for 中间第二个表达式的 true 可以省略
```

6.7. while 和 dowhile 循环

6.7.1. while 循环：

1. 语法

while(一般都是条件表达式){

循环体

}

2. 执行过程

(1) 先计算小括号当中的值

(2) 计算出来的值转化为布尔值

(3) 根据布尔值真假决定是否执行循环体

① 如果值为真就执行循环体

② 如果值为假就结束循环执行下面的代码

3. while 循环的死循环

while(true);

```
//while 循环
var i = 100;
while(i < 100){
    console.log('i love you~' + i);
    i++;
}
console.log(i);
```

6.7.2. do...while 循环：

1. 语法

do{

循环体

}while(一般都是条件表达式)

2. 执行过程

(1) 先执行一遍循环体

(2) 然后计算 while 后小括号当中的值

(3) 计算出来的值转化为布尔值

(4) 根据布尔值真假决定是否继续执行循环体

① 如果值为真就继续执行循环体

② 如果值为假就结束循环执行下面的代码

```
//do..while 是 while 的变种
var i = 100;
do{
    console.log('i love you~' + i);
    i++;
}while(i < 100);
console.log(i);
```

6.7.3.while 和 do...while 循环的区别：

1. **while 循环执行的时候，会先进行条件判断，如果条件为真，就执行循环体，如果为假，就跳出循环；**
2. **do...while 执行的时候，会先执行一遍循环体，然后再进行条件判断，如果条件为真，继续执行循环体，如果条件为假，就跳出循环；**
3. **在一开始条件为假的情况下，while 是循环体一次都不执行，但是 do..while 至少要执行一次循环体；**

6.8. break 和 continue 关键字作用 *****

6.8.1. break:作用

1. 在 switch 语句当中是跳出 switch
2. 在循环当中，跳出离它最近（包含它）的一层循环；

6.8.2. continue:作用：

结束本次循环，返回从下一次继续开始；

6.9. 强化练习

```
// 1、打印三位数位上有 3 或者 7 的数字       100 - 999
for(var i = 100; i <= 999; i++){
    //1、先求出这个数每个位上的数字
    var b = parseInt(i / 100);
    var s = parseInt(i / 10 % 10);
    var g = i % 10;
    //2、判断
    if(b === 3 || b === 7 || s === 3 || s === 7 || g === 3 || g === 7){
        console.log(i);
    }
}


// 2、求 100-999 之间的水仙花数。abc =a*a*a + b*b*b + c*c*c;
for(var i = 100; i <= 999; i++){
    var b = parseInt(i / 100);
```

```
    var s = parseInt(i / 10 % 10);
    var g = i % 10;

    if(b*b*b + s*s*s + g*g*g === i){
        console.log(i);
    }
}

//3、输出 1-100 之间所有的素数（质数）（只能被 1 和自身整除的数,但是 1 不是质数）
//3.1 标志位法
var flag = true;//true 最终标志着这个数是质数
for(var i = 1; i <= 100; i++){
    for(var j = 2; j < i; j++){
        //内层循环控制的是除数
        if(i % j == 0){
            flag = false;
            break;
        }
    }
    if(flag && i!==1){
        console.log(i);
    }
    flag = true; //重置标志位
}
//3.2 计数器法
var n = 0;//计数
for(var i = 1; i <= 100; i++){
```

```
    for(var j = 1; j <= i; j++){
        if(i % j == 0){
            n++;
        }
    }
    if(n == 2){
        console.log(i);
    }
    n = 0;//重置计数器
}



//4、完成一个等腰三角形的打印
//      *
//     ***
//    *****
//   *******
//  *********
// ***********
//*************
for(var i = 0; i < 7; i++){
    for(var j = 0; j < 6 - i; j++){
        document.write(' ');
    }
    for(var k = 0; k < 2*i+1; k++){
        document.write('*');
    }
```

```
    document.write('<br>');
}
```

7. 第 7 章：数组

7.1.数组概念，作用，定义，基本操作和遍历

7.1.1.什么是数组

1. 具有相同类型（或者不同类型）的数据有序集合；
2. 数组也是一种数据，它是属于对象数据类型；
3. 一次性存储多个数据

7.1.2.数组的定义

1. 字面量定义
2. 构造函数定义

```
//数组的定义：
//1、字面量（简便）定义：这个定义方式是我们以后常用
//array
var arr = [1,2,3,'赵丽颖',5];//定义了一个数组，数组里面有 5 个数据
var arr1 = [];//定义了一个空数组，里面没有任何数据；
var arr4 = [3];//定义了一个数组，这个数组只有一个数据 3


//2、构造函数定义；   这个定义是所有定义方式的本质，字面量定义是这个定义方式的简便方法
```

```
var arr2 = new Array(1,2,3);//[1,2,3]
var arr3 = new Array();//[]
//构造函数定义数组的坑
var arr6 = new Array('赵丽颖');//如果（）内部放的不是一个数字，那么就是一个数组内部只有一个元素
var arr5 = new Array(3);//[undefined,undefined,undefined];这个 3 代表的是数组的长度
```

7.1.3.数组长度和索引（下标）

1. 只要定义一个数组，数组会有一个默认的属性叫 **length**,它代表着**数组的长度**
2. **索引**也被称作**下标**，通常情况下只要我们知道了索引，就可以拿到这个数组对应的这个索引的值，当定义完一个数组的时候，**数组的索引最大值是数组的长度减 1**；

7.1.4.数组的基本操作和遍历

1. 数组的不同位置增加
   (1) 在数组的末尾加一个数
   (2) 数组头部加一个数
   (3) 在数组的中间加一个数

```
// 分为三个位置增：  末尾增一个   头部增一个   中间增一个
//末尾增一个：
arr[5] = 1000;
arr[arr.length] = 1000;//往数组末尾添加元素就是这一句
console.log(arr);
```

```
//头部增加一个
//循环遍历数组的时候，循环变量一般都是和数组的下标对应，因为下标是连续的
for(var i = arr.length - 1; i >= 0; i--){
    arr[i + 1] = arr[i];//arr[5] = arr[4];//往下标为 0 的位置添加一个 1000
}
//中间增加一个(主要是要知道往哪个下标去填)  往下标为 2 的位置添加一个 1000
for(var i = arr.length - 1; i >= 2; i--){
    arr[i + 1] = arr[i];//arr[5] = arr[4];
}
arr[2] = 1000;
console.log(arr);
```

2. 数组的不同位置删除
   (1) 在数组的末尾删一个数
   (2) 数组头部删一个数
   (3) 在数组的中间删一个数

```
//删
// 分为三个位置删:    末尾删一个     头部删一个     中间删一个
//末尾删一个(其实就是让数组的长度-1)
arr.length -= 1; //
arr.length = arr.length - 1;
arr.length--
console.log(arr);
console.log(arr[4]);//undefined，删除了就不再存在了

//头部删一个（头部没办法去直接操作）  //删除下标为 0 的那个元素
for(var i = 1; i <= arr.length - 1; i++){
```

```
    arr[i - 1] = arr[i];
}
arr.length--;
console.log(arr);

//中间删一个    删除下标为 1 的那个元素
for(var i = 2; i <= arr.length - 1; i++){
    arr[i - 1] = arr[i];
}
arr.length--;
console.log(arr);
```

3. 数组的改和查

(1) 知道了下标怎么都可以玩

(2) 不知道下标，我们需要去遍历一下数组，先拿到下标，再去改查；

```
//知道下标的情况下：
//把下标为 2 的值，修改为 666
//读取下标为 3 的值；
arr[2] = 666;
console.log(arr[3]);
console.log(arr);
//不知道下标的情况下，得先找到下标；
//把数组当中的 200，修改为 2000；
//得先遍历数组，找到 200 这个值对应的下标才能修改它
for(var i = 0; i < arr.length; i++){//数组遍历的固定模式，切记 i 对应的是数组的下标
```

```
    if(arr[i] === 200){
        arr[i] = 2000;
    }
}
console.log(arr);
```

4. 数组遍历

(1) 数组遍历使用 for 循环

(2) 循环变量和数组的下标对应

```
//数组的遍历（循环去找数组的所有元素）
//数组在遍历的时候，切记下标对应的就是循环变量
for(var i = 0; i < arr.length; i++){
    console.log(arr[i]);
}
```

7.2.数组案例 1

```
//1、数组求和
var arr = [1,2,3,4,5,6,7,8,9,10];
var sum = 0;
for(var i = 0; i < arr.length; i++){
    sum += arr[i];
}
console.log(sum);

// 2、求数组最大值，最小值，平均值
```

```
var arr = [88,66,99,520,30,8];
var max = arr[0];
var min = arr[0];
var sum = 0;
for(var i = 0; i < arr.length; i++){
    //在拿最大值
    if(max < arr[i]){
        max = arr[i];
    }
    //在拿最小值
    if(min > arr[i]){
        min = arr[i];
    }
    //求和
    sum += arr[i];
}
console.log(max,min,sum/arr.length);


//3、翻转数组*******
var arr = [1,2,3,4,5];
//1>、在原数组上进行翻转
console.log(arr);
for(var i = 0; i < arr.length/2; i++){
    var temp = arr[i];
    arr[i] = arr[arr.length - 1 - i];
    arr[arr.length - 1 - i] = temp;
}
```

```
console.log(arr);


//2>、在新数组当中翻转
var newArr = [];
for(var i = arr.length - 1; i >= 0; i--){
    newArr[newArr.length] = arr[i];//newArr[0] = arr[4]
}
console.log(newArr);
```

7.3.数组案例 2

```
// 1、合并数组，原生实现
var arr1 = [1,3,5,7,9];
var arr2 = [2,4,6,8,10];
//1>、合并在新数组当中
var newArr = [];
for(var i = 0; i < arr1.length; i++){
    //在把 arr1 内部的数据添加到 newArr 当中
    newArr[newArr.length] = arr1[i];
}
for(var i = 0; i < arr2.length; i++){
    //在把 arr1 内部的数据添加到 newArr 当中
    newArr[newArr.length] = arr2[i];
}
console.log(newArr);
//2>、在原数组上进行合并
```

```
for(var i = 0; i < arr1.length; i++){
    arr2[arr2.length] = arr1[i];
}
console.log(arr2);
// 2、数组去重    ***********************************
var arr = [1,2,3,4,1,1,2,1,3,2,4];  //去重完成 [1,2,3,4]
//数组去重都是在新数组当中去重的;
var newArr = [];
//代码是次要，首先是过程，过程明白了，代码多敲敲就熟练了
var flag = true;
for(var i = 0; i < arr.length; i++){
    //从老的数组当中拿数
    for(var j = 0; j < newArr.length; j++){
        //拿新的数组当中的值和老数组拿的值进行比较
        //而且从原数组当中拿的值，必须和新数组当中所有的值比较完成，才知道有没有
        if(arr[i] == newArr[j]){
            //代表着新数组内部有这个值;
            flag = false;
            break;
        }
    }
    //break 出来的，代表新数组当中有这个值
    //老老实实执行完内部循环出来的，代表新数组内部没有这个值;
    if(flag){
        newArr[newArr.length] = arr[i];
    }
    flag = true;//重置标志位
```

```
}
console.log(newArr);
// 3、冒泡排序**********************************
var arr = [66,36,88,520,10];//[10,36,66,88,520]
//外层循环控制轮数
for(var i = 0; i < arr.length - 1; i++){
    //外层循环控制轮数
    for(var j = 0; j < arr.length - 1 - i; j++){
        //内层循环控制的是比较的次数
        //同时内层循环的循环变量，也是比较时候拿的数组值得下标
        if(arr[j] > arr[j + 1]){//升序还是降序，自己决定，就看是大于还是小于
            var temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
console.log(arr);
```

作业：

找出数组中的所有偶数打印，奇数生成新数组

有一个从小到大排好序的数组。现输入一个数，要求按原来的规律将它插入数组中

8. 第 8 章：函数

8.1. 概念，定义（表达式，字面量），作用

8.1.1. 什么是函数：

1. 具有某种特定功能的代码块~
2. 函数其实本质也是一种数据，也是属于对象数据类型；

8.1.2. 为什么要有函数

1. 解决代码的冗余问题，形成代码复用；
2. 可以把整个代码项目，通过函数模块化；
3. 封装代码，让函数内部的代码对外部不可见；

8.1.3. 函数定义

1. 字面量定义：

```
function 函数名（）{
    代码块；（函数体）
}
```

2. 函数表达式定义：

```
var 变量名（函数名） = function(){
    代码块；（函数体）
}
```

3. 函数定义本质就是定义了变量赋值了一个函数数据
4. 函数定义的三要素

(1) 功能：函数名字要见名思意，看到名字就知道函数的功能

(2) 参数：形式参数，简称形参，本质是变量

(3) 返回值：

① 函数体 return 后面的值，最终是要返回给函数调用表达式

② 函数调用后，代码执行到 return，函数执行就立即结束，以下代码不再执行

8.1.4. 函数调用

1. 函数定义的时候函数内部代码是不执行的
2. 函数必须定义了才能调用，函数调用才会去执行函数定义的代码
3. 函数调用整体本质上是一个表达式
4. 函数调用的参数
   (1) 函数调用的参数被称作实际参数，简称实参
   (2) 函数调用的参数本质上是值（值，变量，表达式）
   (3) 函数调用的时候参数值是要对应赋值给函数定义的形参
5. 函数调用表达式的值
   (1) 函数调用表达式的值就是执行完函数定义内部代码 return 的值
   (2) 函数定义内部没有 return，默认返回的是 undefined

8.2. 封装函数强化练习

```
//编写求 1 到 n 的和函数
function add(n){
    var sum = 0;
    for(var i = 0; i <= n; i++){
        sum += i;
    }
    return sum;
```

```
}
var result = add(100);
console.log(result);


// 编写函数实现求一个数的阶乘
function getFactorial(n){
    var product = 1;
    for(var i = 1; i <= n; i++){
        product *= i;
    }
    return product;
}
var a = 3;
console.log(getFactorial(a+2));


// 编写函数求数组的最大值，最小值
function getMaxAndMinOfArray(arr){
    var max = arr[0];
    var min = arr[0];
    for(var i = 0; i < arr.length; i++){
        if(max < arr[i]){
            max = arr[i];
        }

        if(min > arr[i]){
            min = arr[i];
        }
```

```
    }
    return [max,min];      //只能返回一个值;
}
var arr = [11,22,3,4,5];//函数内外如果有同名的变量，这两个变量没有关系
var result = getMaxAndMinOfArray(arr);//var result = [max,min]
console.log(result[0],result[1]);

// 封装函数加工数组，每一项加 10 输出(打印)
function machiningArray(arr){
    for(var i = 0; i < arr.length; i++){
        console.log(arr[i] += 10);// 这里改不改数组的值都无所谓
    }
}
var arr = [1,2,3,4];
machiningArray(arr);

// 封装函数实现打印 1 到 N 的质数;
function printPrimeNumber(n){
    var flag = true;//true 最终标志着这个数是质数
    for(var i = 1; i <= n; i++){
        //外层循环控制的是拿的被除数
        //7   2 -- 6
        //8   2 -- 7
        //让拿进来的数从 2 初到自身减一;
        for(var j = 2; j < i; j++){
            //内层循环控制的是除数
            if(i % j == 0){
```

```
            flag = false;
            break;
        }
    }
    if(flag && i!==1){
        console.log(i);
    }
    flag = true; //重置标志位
  }
}
printPrimeNumber(100);

// 封装函数实现对数组进行排序;
function sortArray(arr){
  for(var i = 0; i < arr.length - 1; i++){
  //外层循环控制轮数
    for(var j = 0; j < arr.length - 1 - i; j++){
      if(arr[j] > arr[j + 1]){//升序还是降序，自己决定，就看是大于还是小于
        var temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
  return arr;
}
var arr = [1,3,5,7,9,2,4,6,8,10];
```

```
var result = sortArray(arr);
console.log(result);


// 封装函数实现对数组翻转
function reverseArray(arr){
    for(var i = 0; i < arr.length/2; i++){
        var temp = arr[i];
        arr[i] = arr[arr.length - 1 - i];
        arr[arr.length - 1 - i] = temp;
    }
    return arr;
}
var arr = [1,2,3,4];
console.log(reverseArray(arr));


// 封装函数实现对数组去重
function deduplicationArray(arr){
    //数组去重都是在新数组当中去重的;
    var newArr = [];
    //代码是次要，首先是过程，过程明白了，代码多敲敲就熟练了
    var flag = true;
    for(var i = 0; i < arr.length; i++){
        //从老的数组当中拿数
        for(var j = 0; j < newArr.length; j++){
            //拿新的数组当中的值和老数组拿的值进行比较
            //而且从原数组当中拿的值，必须和新数组当中所有的值比较完成，才知道有没
有
```

```
            if(arr[i] == newArr[j]){
                //代表着新数组内部有这个值；
                flag = false;
                break;
            }
        }
        if(flag){
            newArr[newArr.length] = arr[i];
        }
        flag = true;//重置标志位
    }
    return newArr;
}
var arr = [1,2,3,4,1,1,1,2,3,4,5,2,3,4];
console.log(deduplicationArray(arr))
```

8.3.作用域

8.3.1.作用域概念，作用，分类

1. 什么是作用域
   (1) 作用域是一个**抽象的概念**，看不见摸不着
   (2) 作用域就是**变量**起作用的**范围和区域**
   (3) **作用域在函数定义的时候就确定好了**
2. 作用域的**作用**就是用来**隔离变量**的
3. 作用域分类

(1) 在 ES5 当中只有全局作用域和局部作用域两种

(2) ES6 当中新增了块级作用域

8.3.2.全局变量和局部变量

1. 全局变量

   在全局作用域当中定义的变量称作全局变量，对于整个程序都可以使用

2. 局部变量

   在局部作用域当中定义的变量称作局部变量，只有局部（函数内部）可以使用

3. 全局变量和局部变量同名，互相不会有影响，各自使用各自的

4. 在函数内部遇到不带 var 的变量处理过程

   (1) 先看函数内部有没有带 var 的这个变量

   (2) 如果有当作局部变量去处理

   (3) 如果没有看形参有没有

   (4) 如果有还是当作局部变量去处理

   (5) 如果没有去函数外部看有没有定义这个变量

   (6) 如果有那就是函数内部在操作函数外部全局变量

   (7) 如果没有就是在函数外部定义了一个全局变量

```
var a = 10;
var b = 20;
function fn(){
    var a = 100;
    var b = 200;
    c = 300;//外部定义 c 和不定义 c 结局不同
    var d = 400;
    console.log(a,b,c,d);
```

```
}
fn();
console.log(a, b, c);
console.log(d);
```

8.4. 作用域链

1. 什么是作用域链
   (1) 作用域链描述的是程序在执行过程当中寻找变量的过程
   (2) 作用域链的作用就是在程序执行过程当中寻找变量；
2. 作用域链寻找变量的过程
   (1) 先从自身所在作用域去查找，如果没有再从上级作用域当中去查找，直到找到全局作用域当中。
   (2) 如果其中有找到，就不会再往上查找，直接使用。
   (3) 如果都没有找到，那么就会报引用错误提示变量没有定义。

```
var a = 0;
function fn1(){
    var a = 1;
    function fn2(){
        var a = 2;
        function fn3(){
            var a = 3;
            console.log(a);//3
        }
        fn3();
    }
```

```
    fn2();
}
fn1();
```

8.5. 预解析

8.5.1. 预解析

1. 程序在代码执行之前会先进行预解析（预解释、变量提升）
2. 预解析先去解析函数声明定义的函数，整体会被提升
3. 再去解析带 var 的变量
4. 函数重名会覆盖，变量重名会忽略
5. 变量如果不带 var，变量是不会进行预解析的
6. 只有带 var 的变量才会进行预解析
7. 表达式定义的函数也是当做变量去解析

8.5.2. 面试题:

```
//1
alert(a);
a = 0;


//2
alert(a);
var a = 0;
alert(a);
```

```
//3
alert(a);
var a = '我是变量';
function a(){ alert('我是函数') }
alert(a);

//4
alert(a);
a++;
alert(a);
var a = '我是变量';
function a(){ alert('我是函数') }
alert(a)

//5
alert(a);
var a = 0;
alert(a);
function fn(){
    alert(a);
    var a = 1;
    alert(a);
}
fn()
alert(a);

//6
```

```
alert(a);
var a = 0;
alert(a);
function fn(){
    alert(a);
    a = 1;
    alert(a);
}
fn()
alert(a);
```

8.6. IIFE，回调函数 函数递归 arguments

8.6.1. IIFE:

1. 什么是 IIFE
   (1) Immediately Invoked Function Expression 意为立即调用的函数表达式
   (2) 也叫匿名函数自调用
2. 语法：

(function(){
代码块；
})();

3. 特点：
   (1) 函数定义的时候同时执行
   (2) 只执行一次
   (3) 不会发生预解析（函数内部执行的时候会发生）
4. 作用：

(1) 防止外部命名空间污染

(2) 隐藏内部代码暴露接口

(3) 对项目的初始化

8.6.2. 函数实参伪数组 arguments

1. 什么是 arguments

(1) **ES5 中函数的内部**都会有一个特殊的**内置变量 arguments**

(2) arguments 是所有函数中都可用的局部变量

(3) 是一个对应于传递给函数的参数的类数组对象

2. 函数的形参可以不写

形参是用来接收实参的，而 **arguments** 就代表的是**实参伪数组**，函数调用传递的实参可以通过 arguments 获得

```
//1、其实函数定义的形参可写可不写（ES5 当中），但是虽然可以不写，建议以后写上形参
function add(){
    // console.log(arguments);
    // return arguments[0] + arguments[1];
    var sum = 0;
    for(var i = 0; i < arguments.length; i++){
        sum += arguments[i];
    }
    return sum;
}
console.log(add(10,20));
```

3. **arguments.length** 可以**让函数有多种功能**

arguments.length 是实参伪数组的**长度**，代表的是**传递实参的个数**

```
//2、arguments.length，拿得是实参的个数  可以让我们一个函数具有多种功能
function addOrSub(a,b,c){
    if(arguments.length == 2){
        return a - b;
    }else if(arguments.length == 3){
        return a + b + c;
    }
}
console.log(addOrSub(10,20));//传递两个实参就做减法
console.log(addOrSub(10,20,30));//传递的是三个实参就做加法
```

4. **arguments.callee** 代表了**函数本身**，常用在**函数递归调用**中

8.6.3.回调函数：

1. 回调函数三个条件
   (1) 函数是**我定义的**
   (2) **我没有调用**
   (3) 最终**执行了**
2. 常见回调函数
   (1) 事件
   (2) 定时器
   (3) ajax
   (4) 生命周期回调函数

8.6.4.函数递归调用

1. 函数递归调用的概念

(1) 一个函数如果调用的时候，内部又调用了自己，称作是函数的递归调用

(2) 函数递归调用很容易发生灾难（内存泄漏）而调用失败。

(3) 函数递归调用效率不高，能不用则不用

2. 函数递归调用成功的条件

(1) 必须有一个明显的结束条件

(2) 必须有一个趋近于结束条件的趋势

3. 案例

```
//1、灾难
function fn(){
    console.log('i love you zhao li ying~');
    fn();
}
fn();

//2、让灾难变成好事
//我想打印 10 遍后，就停止
var n = 0;
function fn(){
    if(n >= 10){
        return;
    }
    n++;
    console.log('i love you zhao li ying~');
    fn = 1;
    // fn();
    arguments.callee();//为了防止递归内部函数名字被修改，递归建议使用
arguments.callee,
```

```
                    //它仍然还是存储原来函数的地址；
}
fn();


//3、递归处理求一个数的阶乘
function getFactorial(n){
    if(n <= 1){
        return 1;
    }
    return  n * getFactorial(n - 1);
}
console.log(getFactorial(5));// 5! = 4! * 5
```

作业:

面试题: 函数，作用域，预解析（必须理解）

9. 第 9 章：对象

9.1. Object 的实例对象

9.1.1. 对象的概念

1. 面向对象和面向过程
   (1) 面向对象和面向过程是两种编程思想
   (2) C 语言是面向过程的语言
   (3) js 是面向对象的语言

2. 什么是对象及作用

(1) 无序的名值对的集合（键值对的集合）就叫做对象；

(2) 用来描述一个复杂的事物，属性方法的集合体（比如人、狗等等）

9.1.2.对象的定义方法

1. 字面量定义

(1) 对象是由属性和属性值组成的，就是我们说的键值对

(2) 属性的本质是一个字符串类型

(3) 属性值可以是任意数据

(4) 属性值是函数，那么这个属性也被称作方法

(5) 属性名不符合规范必须写上引号，符合规范的可以省略引号

2. 构造函数定义
3. 工厂函数模式
4. 三种方式创建对象，这些对象都被称作是 Object 的实例对象，Object 的实例对象类型不明确

```
//三种定义方式
//1、字面量定义
var obj = {
    name:'赵丽颖',
    'age':33,
    gender:'female',
    "height":165,
    eat:function(){
        console.log('吃货~');
    },
    'character-type':'淑女' //这个属性名必须使用引号;
```

```
};
var obj1 = {};//称作 obj 是 Object 的一个实例对象,只是这个对象当中没有任何属性
console.log(obj);
console.log(obj1);
//2、构造函数定义对象
var obj2 = new Object({
    name:'旺财',
    age:2,
    run:function(){
        console.log('跑的很快~');
    }
});
var obj3 = new Object();
console.log(obj2);
console.log(obj3);
//3、工厂函数定义对象
function createObject(){
    var obj = new Object();
    return obj;
}
var obj = createObject();
var obj2 = createObject();
```

9.1.3.对象的操作和遍历

1. 增删改查：点语法和[]语法操作

(1) 一般对象的操作都使用点语法

(2) 只有在以下情况下我们必须使用[]语法操作

① 如果对象的属性名不符合命名规范

② 如果需要使用变量的值作为属性名

2. 遍历

(1) for in 循环进行遍历对象

(2) for...in 遍历的是对象的所有属性

```
var obj = {
    name:'赵丽颖',
    'age':33,
    gender:'female',
    "height":165,
    eat:function(){
        console.log('吃货~');
    },
    'character-type':'淑女'//这个属性名必须使用引号；属性名不符合命名规范就得带引号
}
//增和改
//1、点语法，写起来简单，但是某些场合无法使用.去操作
obj.weight = 90; //有则更改，无则添加
//2、中括号语法，是通用的语法，但是写起来复杂
obj['weight'] = 100; //有则更改，无则添加
obj['character-type'] = '嘿嘿';//当对象的属性是不合法的名字必须使用[]
var a = true;
obj[a] = 'bbb';//以变量当中的值为属性去操作对象的话，必须使用[]
//删
delete obj.age;
```

```
delete obj['character-type'];//必须使用[]
delete obj[a];//必须使用[]
//查
console.log(obj.name);
console.log(obj['character-type']);//必须使用[]
console.log(obj[a]);//[]要先去找变量的值，然后把值转化为字符串，去对象找这个字符串属性操作。因为属性本质是字符串
console.log(obj.heihei);//对象没有这个属性，那么这个属性值是 undefined
//对象的遍历
for(var key in obj){//var key = 'name';
    //拿一次属性名赋值给变量 key    就会循环一次
    console.log(key,obj[key]);
}
```

3. 数组是数组数组也是对象

(1) 数组是特殊的对象，因为下标有序

(2) 数组使用 for 遍历的是数组对象的数字属性

(3) 数组使用 for...in 遍历的是数组对象的所有属性（length 除外）

```
//数组是数组，数组也是对象
var arr = [1,2,3,4];
arr.heihei = 'haha';//A.B 那么 A 一定是对象，B 一定是对象的属性，把数组当对象用加了一个属性和值
arr['100'] = 5;
console.log(arr);
//数组当数组用的使用，我们用 for 循环去遍历;
for(var i = 0; i < arr.length; i++){
    console.log(arr[i]);
```

```
}
//数组当对象用的时候，我们用 for  in 去遍历；
for(var key in arr){
    console.log(key,arr[key]);
}
```

4. 函数是函数函数也是对象

(1) 函数也是一种特殊的对象，因为函数能执行

(2) 函数有两种角色：函数和函数对象

```
//函数是函数，函数也是对象
//函数当函数使用
function fn(){
    console.log('i love you');
}
fn();
//函数当对象用
fn.hehe = 'heihei';    //A.B  那么代表这个函数是在当对象用；
console.log(fn);
console.dir(fn);
for(var key in fn){
    console.log(key,fn[key]);
}
```

9.2.构造函数创建特定实例对象

9.2.1.构造函数的概念

1. **构造函数就是一个函数**，只不过通常我们把构造函数的名字写成大驼峰
2. 在 js 当中，没有类的概念（ES5）,构造函数可以理解为类
3. 任何的函数都可以是普通函数，也可以是构造函数，就看怎么调用
   (1) 函数直接加小括号调用称作普通函数调用
   (2) 函数加小括号和 new 操作符调用称作构造函数调用

```
//Object 通过它创建的对象 类别不确切
//所以我们需要自己能创建出确切的类别对象
//ES5 当中没有类的概念，它是通过函数当类来用
//我们的构造函数来充当类的角色；
//创建一个类（函数）
function Car(name,color,price){
    this.name = name;
    this.color = color;
    this.price = price;
    this.run = function(){
        console.log('跑的很快~');
    }
}
//普通函数调用
var result = Car('奔驰','black',200000);
console.log(result);

console.log(color);
```

```
//构造函数调用
var c1 = new Car('劳斯莱斯','red',10000000);
console.log(c1);
c1.run();
```

9.2.2.this 的概念

1. 什么是 this
   (1) this 是 js 内置的一个变量，本质上是一个对象
   (2) 通常在函数当中使用，代表这个函数的调用者
   (3) 可以说谁执行了这个函数，this 就指向谁
2. 不同场合 this 的指向不同，js 当中，一般 this 有这些场合：
   (1) 如果 this 是在一个普通函数当中，this 一般指向 window
   (2) 如果 this 是在一个方法当中，this 一般指向这个方法的对象
   (3) 如果 this 是在一个构造函数当中，this 一般指向实例化对象
   (4) 如果 this 是在一个事件的处理（回调）函数当中，this 一般指向添加事件监听的事件源
   (5) 如果函数使用 call 和 apply，this 将由自己指定；

```
//this 的使用场合
//1、普通函数内部    一般执行的时候都是 window 去执行的
function fn1(){
    //叫函数，特殊对象 window 下的方法叫函数
    console.log(this);//window
}
fn1();//代表执行者是 window
```

```
function fn2(){
    console.log(this);//window
    function fn3(){
        console.log(this);//window
    }
    fn3();
}
fn2();
//2、方法内部    一般调用的时候执行者都是这个方法的对象
var obj = {
    eat:function(){
        //称作方法
        console.log(this);//obj 对象
        console.log('吃');
    }
}
obj.eat();//eat 方法执行者是 obj
//3、构造函数内部   this 一般代表的是实例化对象
//4、事件回调函数内部：一般代表的就是添加事件监听的事件源
//5、call 和 apply 调用  this 由自己指定
```

9.2.3.window 对象简介

1. window 是浏览器窗口对象，代表顶级对象
2. 全局变量和函数会作为 window 对象的属性对待
3. 打开浏览器，window 对象会自动生成

9.2.4.普通函数和构造函数的区别

1. 本质上都是函数，函数在调用的时候有多种方式
2. 构造函数去用
   (1) this 指向实例化对象
   (2) 返回值没有 return 或者 return 的是基本数据，返回的是实例化对象
   (3) 返回值 return 的是对象数据，那么返回的就是这个对象数据

3. 普通函数去用
   (1) this 指向 window
   (2) 返回值 return 的是什么就是什么
   (3) 返回值没有 return，返回 undefined

9.2.5.new 关键字做了什么

1. 开辟内存空间(堆)
2. this 指向该内存（让函数内部的 this）
3. 执行函数代码
4. 生成对象实例返回

9.3.原型对象和原型链

9.3.1.原型对象

1. 什么是原型对象:
   (1) 每个函数对象都会有一个 **prototype** 属性
   (2) 这个 **prototype** 属性指向了另外一个对象，我们称 **prototype** 这个对象是原型对象

(3) 原型对象默认是 Object 的实例对象

2. 显式原型对象和隐式原型对象

(1) 构造函数的实例化对象内部都会默认有一个属性叫做__proto_

(2) 这个__proto__和构造函数的 prototype 指向同一个对象（原型对象）

(3) 构造函数的 prototype 为显式原型对象

(4) 实例化对象的__proto__为隐式原型对象

3. 原型对象的作用：让所有的实例化对象资源共享节省内存

```
function Villa(size,styleType,price){
    this.size = size;
    this.styleType = styleType;
    this.price = price;
    // this.live = function(){
    //     console.log('住的很舒服');
    // }
}
Villa.prototype.live = function(){
    console.log('住的很舒服');
}//把方法添加在原型对象当中，让所有的实例化对象共享
console.dir(Villa);
var v1 = new Villa(1000,'欧美',10000000);
console.log(v1);
v1.live();
console.log(v1.styleType);

var v2 = new Villa(2000,'田园',20000000);
v2.live();
console.log(v2.styleType);
```

9.3.2. 原型链

1. 原型链描述的是对象查找属性或者方法的过程
2. 实例化对象在找属性的时候，先从自身去找看有没有这个属性，如果有，直接使用这个属性的值，如果没有，会继续顺着这个对象的隐式原型对象（__proto__）找到这个对象的原型对象（和它的构造函数的显式原型对象是同一个），看看原型对象是否存在这个属性，如果有就使用原型对象当中的这个属性值，如果还没有，再去找原型对象的隐式原型对象（默认就是 Object 显式原型对象），找到以后去看看有没有这个属性，如果有就使用这个属性值；如果没有就返回 undefined(代表已经找到顶了)

9.4. apply 和 call

1. 任何函数对象都有 apply 和 call 方法
   (1) apply 和 call 可以让函数或者方法的执行者（this）指向指定对象；
   (2) call 和 apply 可以让一个对象执行另外一个对象的方法；
   (3) 函数或者方法.apply(对象，【函数的参数】)
   (4) 函数或者方法.call(对象，函数的参数 1，函数的参数 2)；
2. call 和 apply 干了两件事：
   (1) 调用的时候先把 this 指向改为指定的对象
   (2) 自动执行使用的方法或者函数

```
var obj = {
    name:'赵丽颖',
    age:33
};
function Dog(name,age){
```

```
    this.name = name;
    this.age = age;
};
Dog.prototype.eat = function(a,b){
    console.log(this);
    console.log('吃肉');
    console.log(a,b);
};
var d1 = new Dog('旺财',3);
d1.eat(10,20);
obj.eat();//报错不是因为 obj.eat 报错而是因为 obj.eat 拿到了一个 undefined，然后加括号调用报错；
//狗可以吃肉
//赵丽颖没法吃肉
//如果我想让赵丽颖也吃肉
//1、给这个对象自己添加一个吃肉的方法
obj.eat = function(){
    console.log('吃肉');
}
obj.eat();
//2、不给这个对象添加，而是使用 apply 或者 call 借用 狗的吃肉方法去实现
d1.eat.apply(obj,[100,200]);//把狗的 eat 方法调用时的 this 指向修改为 obj，代表 obj 在执行狗的这个方法
d1.eat.call(obj,100,200);
```

9.5. instanceof 的用法

9.5.1. typeof 应用的场景

1. **typeof** 返回的是**数据类型**的**小写字符串**形式
   (1) 数字 'number'
   (2) 字符串 'string'
   (3) 布尔 'boolean'
   (4) undefined 'undefined'
   (5) null 'object'
   (6) 数组 'object'
   (7) 函数 'function'
   (8) 对象 'object'
2. typeof **可以判定 5 种**：
   (1) 数字 'number'
   (2) 字符串 'string'
   (3) 布尔 'boolean'
   (4) undefined 'undefined'
   (5) 函数 'function'
3. typeof **3 种数据判定不了**
   (1) null 'object'
   (2) 数组 'object'
   (3) 对象 'object'

9.5.2. instanceof 应用的场景

1. instanceof 用来判断一个对象是哪个构造函数的实例用的
2. A(对象数据) instanceof B（构造函数）

3. 专门用来解决判定数组和对象的时候使用

9.5.3.===

1. ===可以用来判定 null 和 undefined 的时候使用
2. 因为它们都是数据类型，而且数据类型当中只有一个值；
3. 通过 typeof　instanceof　以及 === 可以让我们判定 js 当中所有的数据类型

9.6.值类型和引用类型

9.6.1.值类型和引用类型的概念

1. 值类型：就是基本数据类型
2. 引用类型：就是对象数据类型也叫复杂数据类型

9.6.2.堆和栈的概念

1. 内存分为栈内存和堆内存
2. 在 C 语言和 java 当中程序运行数据存储严格区分栈内存和堆内存
3. js 运行的时候并没有严格的区分栈内存和堆内存
4. js 运行我们可以粗浅的认为数据都是在堆内存当中
5. 堆内存当中我们可以认为又区分栈结构和堆结构

9.6.3.堆空间释放

1. js 运行堆结构当中的数据是靠垃圾回收机制回收，释放内存的
2. 函数或者整个程序执行完成后，栈里面所有的东西都被释放销毁
3. 堆当中的数据可能还在，只是没有任何的变量指向（引用）

4. 回收机制会在适当的时候将垃圾对象清理回收；

9.6.4.值类型和引用数据类型面试题

```
//1、
var num1 = 10;
var num2 = num1;
num1 = 20;
console.log(num1);
console.log(num2);
//2、
var num = 50;
function f1(num) {
    num = 60;
    console.log(num);
}
f1(num);
console.log(num);
//3、
var num1 = 55;
var num2 = 66;
function f1(num, num1) {
    num = 100;
    num1 = 100;
    num2 = 100;
    console.log(num);
    console.log(num1);
```

```
    console.log(num2);
}
f1(num1, num2);
console.log(num1);
console.log(num2);
console.log(num);
// 4、
// 函数传参如果传的是基本数据类型和传引用（对象）有什么区别
//4-1、
var a = 10;
var b = 20;
function add(a,b){
    a = 30;
    return a + b;
}
add(a,b);
console.log(a);
//4-2、
function f1(arr){
    for(var i = 0; i < arr.length; i++){
        arr[i] += 2
    }
    console.log(arr);
}
var arr;
arr = [1,2];
f1(arr);
```

```
console.log(arr);
// 5、
// 两个对象是同一个对象，不同的操作有什么不同
var a = [1,2];
var b = a;
a[0] = 20;     如果 a = [20,2];
console.log(b);
//6、
function Person(name, age, salary) {
    this.name = name;
    this.age = age;
    this.salary = salary;
}
function f1(pp) {
    pp.name = "ls";
    pp = new Person("aa", 18, 10);
}
var p = new Person("zs", 18, 1000);
console.log(p.name);
f1(p);
console.log(p.name);
console.log(pp.name);
```

9.7. 内置对象 JSON

9.7.1. 什么是 json

1. json 是一种前后端数据交互的数据格式
2. json 本质上是一个字符串，简称 json 串

9.7.2. 怎么生成 json 串

1. 在前端 json 串的格式原形就是对象或者对象的数组
2. 先把数据存储为对象或者对象的数组，然后转化为 json 串进行交互
3. JSON 对象的方法 JSON.stringify 是专门把对象或者对象的数组格式化为 json
4. JSON 对象的方法 JSON.parse 是专门把 json 解析为对象或者对象的数组

```
var a = '赵丽颖';
var b = 100;
var c = '嘿嘿';
//第一步，把数据给封成一个对象
var obj = {
    a:a,
    b:b,
    c:c
}
//第二步，把对象给变为 json 串，依赖 JSON 对象的方法
console.log(obj);
var result = JSON.stringify(obj);//把对象转化为 json 串就可以传给后端
console.log(result);
//后端传递回来的数据，我们需要把这个 json 串转化为对象去操作
console.log(JSON.parse(result));
```

9.8. Math 工具对象

9.8.1. Math 对象常用属性和方法

round()	把小数四舍五入取整
floor()	把小数向下取整
ceil()	把小数向上取整
random()	取 0 到 1 之间的随机数，能取到 0 但是取不到 1
max()	取多个值之间的最大值
min()	取多个值之间的最小值
PI	是一个属性，取圆周率
pow()	幂运算
abs()	取绝对值
sin()	求三角函数正弦值

9.8.2. 案例：

1. 求任意两个整数之间的随机整数

```
function getRandomInt(a,b){
    return Math.floor(Math.random()*(b - a + 1) + a);
}
```

2. 随机验证码

```
function getRandomCode(n){
    var str = '1234567890';
```

```
    var code = '';
    for(var i = 0; i < n; i++){
        code += str[Math.floor(Math.random()*str.length)];
    }
    return code
}
```

9.9. Date 日期对象

9.9.1. Date 对象的方法

```
var date = new Date();
console.log(date);
console.log(date.getFullYear());
console.log(date.getMonth());
console.log(date.getDate());
console.log(date.getHours());
console.log(date.getMinutes());
console.log(date.getSeconds());
console.log(date.toLocaleTimeString());
console.log(date.toLocaleDateString());
console.log(date.getTime());//1970 年 1 月 1 日之间的毫秒数
```

9.9.2. 案例：

封装函数实现格式化日期 XXXX 年 XX 月 XX 日　　XX：XX：XX

```
function getDateAndTimeNow(){
```

```
    var date = new Date();
    var year = date.getFullYear();
    var month = date.getMonth() + 1;
    var day = date.getDate();
    var time = date.toLocaleTimeString();
    return '现在是：' + year + '年' + month + '月' + day + '日 '+ time;
}
```

9.10. 包装对象

1. 基本数据类型也可以调用方法是因为包装对象的存在
2. 包装对象有三种：数字包装对象、字符串包装对象和布尔值包装对象
3. 数字、字符串和布尔值在调用方法的时候其实是对应包装对象在调用

```
var a = 100;
console.log(a);
console.log(a instanceof Object);//false
console.log(a.toString());//A.B
console.log(a instanceof Object);//false
//a.toString 执行到这一行的时候，其实偷摸干了很多事
//1、a = new Number(a);  先把数字基本值转化为包装对象
//2、a.toString; 调用的其实是包装对象的原型当中的 toString 方法
//3、调用结束后自动再让 a 变回基本值  a = 100;
```

作业：

对象的增删改查，构造函数

创建两个同一品牌的手机对象（要求品牌和方法在原型当中定义共享使用）；接下来把实例化对象的过程以及原型链的原理图 每人画一遍；

10. 第 10 章：数组和字符串方法

10.1. 字符串方法 ES5/ES6

方法名	对应版本	功能	原字符串是否改变
charAt()	ES5-	返回指定索引的字符	n
charCodeAt(9)	ES5-	返回指定索引的字符编码	n
concat()	ES5-	将原字符串和指定字符串拼接，不指定相当于复制一个字符串	n
String.fromCharCode()	ES5-	返回指定编码的字符	n
indexOf()	ES5-	查询并返回指定子串的索引，不存在返回-1	n
lastIndexOf()	ES5-	反向查询并返回指定子串的索引，不存在返回-1	n
localeCompare()	ES5-	比较原串和指定字符串：原串大返回 1，原串小返回-1，相等返回 0	n
slice()	ES5-	截取指定位置的字符串，并返回。包含起始位置但是不包含结束位置，位置可以是负数	n

方法名	对应版本	功能	原字符串是否改变
substr()	ES5-	截取指定起始位置固定长度的字符串	n
substring()	ES5-	截取指定位置的字符串，类似 slice。起始位置和结束位置可以互换并且不能是负数	n
split()	ES5-	将字符串切割转化为数组返回	n
toLowerCase()	ES5-	将字符串转化为小写	n
toUpperCase()	ES5-	将字符串转化为大写	n
valueOf()	ES5-	返回字符串包装对象的原始值	n
toString()	ES5-	直接转为字符串并返回	n
includes()	ES6-	判断是否包含指定的字符串	n
startsWith()	ES6-	判断是否以指定字符串开头	n
endsWith()	ES6-	判断是否以指定字符串结尾	n
repeat()	ES6-	重复指定次数	n

10.2. 数组方法 ES5/ES6

方法名	对应版本	功能	原数组是否改变
concat()	ES5-	合并数组，并返回合并之后的数据	n
join()	ES5-	使用分隔符，将数组转为字符串并返回	n
pop()	ES5-	删除最后一位，并返回删除的数据	y
shift()	ES5-	删除第一位，并返回删除的数据	y
unshift()	ES5-	在第一位新增一或多个数据，返回长度	y
push()	ES5-	在最后一位新增一或多个数据，返回长度	y
reverse()	ES5-	反转数组，返回结果	y
slice()	ES5-	截取指定位置的数组，并返回	n
sort()	ES5-	排序（字符规则），返回结果	y
splice()	ES5-	删除指定位置，并替换，返回删除的数据	y
toString()	ES5-	直接转为字符串，并返回	n
valueOf()	ES5-	返回数组对象的原始值	n
indexOf()	ES5-	查询并返回数据的索引	n
lastIndexOf()	ES5-	反向查询并返回数据的索引	n
forEach()	ES5-	参数为回调函数，会遍历数组所有的项，回调函数	n

方法名	对应版本	功能	原数组是否改变
		接受三个参数，分别为 value，index，self；forEach 没有返回值	
map()	ES5-	同 forEach，同时回调函数返回数据，组成新数组由 map 返回	n
filter()	ES5-	同 forEach，同时回调函数返回布尔值，为 true 的数据组成新数组由 filter 返回	n
Array.from()	ES6-	将伪数组对象或可遍历对象转换为真数组	n
Array.of()	ES6-	将一系列值转换成数组	n
find	ES6-	找出第一个满足条件返回 true 的元素	n
findIndex	ES6-	找出第一个满足条件返回 true 的元素下标	n

注意：无论是字符串方法还是数组方法都要关注三要素：功能、参数、返回值

10.3. 强化练习（面试题）

1. 字符串反转
2. 打印字符串中出现字母最多的次数 *****

11. 第 11 章：DOM 基础

11.1. DOM 概念和操作方式

11.1.1. DOM 概念，作用，顶级对象

1. **DOM**：**文档对象模型**
   (1) **DOM 是**一个使程序和脚本有能力动态地访问和更新文档的内容、结构以及样式的平台和语言中立的**接口**。
   (2) DOM 描述了处理网页内容的方法和接口

2. 文档对象
   (1) document 是 window 对象的一个属性
   (2) **document** 代表整个 DOM **叫做文档对象**
3. 根元素

html 标签被称作是文档的**根元素**，也叫 root 元素

4. 文档树(DOM 树)
   (1) 以 **HTML 为根节点** 形成的一棵倒立的**树状结构**，我们称作 **DOM 树**；
   (2) DOM 树上所有的东西都叫节点，节点有 12 种，主要关注 4 种
      ① 元素节点　　标签
      ② 属性节点　　属性
      ③ 文本节点　　内容
      ④ 注释节点　　注释
5. DOM 对象

使用 **DOM 方法获取**到的**节点**被称作是 **DOM 对象**，任何节点都可以是 DOM 对象

11.2. window.onload

11.2.1. 等待页面加载完成事件

1. 一般我们都是等待页面加载完成之后才去操作 DOM 元素
2. 如果页面没有加载完成就去获取 DOM 元素，有可能获取不到；
3. window.onload = function(){
   }等待页面加载完成，系统会自动执行函数当中的代码；

11.2.2. 案例:

```
//等待页面加载完成事件,点击按钮弹出一个对话框
window.onload = function(){
    var btn = document.getElementById('btn');//通过 dom 方法获取到封装成 dom 对象,
给变量 btn
    btn.onclick = function(){
        //给 btn 添加点击事件监听
        //事件监听的函数都是回调函数，这个函数我们不会去执行，而是在特定的条件下系
统帮我们执行
        alert('我爱你，赵丽颖~');
    }
}
```

11.3. 鼠标事件

11.3.1. 事件三要素

1. 事件源 （承受事件的对象）

2. 事件类型
3. 事件处理回调函数

11.3.2. 事件处理三大步

1. 获取事件源 DOM 对象
2. 添加对应事件监听
3. 书写处理回调

注意：事件回调函数可以重复触发执行

11.3.3. 鼠标事件

1. **onclick** 鼠标点击
2. **onmousemove** 鼠标移动
3. **onmouseover** 鼠标移入
4. **onmouseout** 鼠标移出
5. **onmounseenter** 鼠标移入
6. **onmouseleave** 鼠标移出
7. **onmousedown** 鼠标按下
8. **onmouseup** 鼠标抬起

9. 简单案例： 点击按钮实现一个数字自增；

```
window.onload = function(){
    //事件是可以重复触发执行的
    var btn = document.getElementById('btn');
    var n = 0;
    btn.onclick = function(){
        n++;
        console.log(n);
```

```
    }
}
```

11.4. 获取 DOM 元素

11.4.1. document.getElementById

通过 id 获取到相关元素，这个方法只能获取一个 DOM 元素对象

11.4.2. document.getElementsByTagName

通过标签名去获取，获取到是多个返回一个伪数组，标签只有一个也是伪数组

11.4.3. document.getElementsByClassName

通过类名去获取，并且获取到是多个返回一个伪数组，哪怕只有一个标签是这个类也是伪数组

11.4.4. document.querySelector 和 querySelectorAll

1. 根据 CSS 选择器去获取，也就是说只要选择器能选择到，它们就可以获取到
2. querySelector 返回的是单个元素 dom 对象，选择器选中多个，只拿第一个
3. querySelectorAll 返回选择器选中的所有，返回也是伪数组

11.5. 操作 DOM 元素

11.5.1. 案例驱动：操作元素属性

1. 天生属性

(1) 点击按钮修改图片的路径（一般属性）

```
<body>
    <a href="https://www.baidu.com" id="aa">点击</a>
    <button id="btn">按钮</button>
    <img src="img/2.jpg" id="imgSrc" />
    <script type="text/javascript">
        window.onload = function(){
            var btn = document.getElementById('btn');
            var imgNode = document.getElementById('imgSrc');
            var aNode = document.getElementById('aa');
            btn.onclick = function(){
                //和之前操作对象一样，修改对象的属性值
                imgNode.src = 'img/3.jpg';
                imgNode['src'] = 'img/3.jpg';
                aNode.href = 'https://www.jd.com';
            }
        }
    </script>
</body>
```

(2) 点击按钮修改多选框选中（特殊属性）

```
<body>
```

```
    <button id="btn">按钮</button>
    <input type="checkbox" id="in"  />
    <script type="text/javascript">
        window.onload = function(){
            var btn = document.getElementById('btn');
            var inputNode = document.getElementById('in');
            //以后碰到元素当中属性和属性值相同的，js 去操作这个属性值的时候用布尔
值；
            btn.onclick = function(){
                inputNode.checked = true;
            }
        }
    </script>
</body>
```

(3) 点击按钮修改 p 标签的背景颜色（特殊属性）

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        .p1{
            background-color: hotpink;
        }
        .p2{
            background-color: skyblue;
        }
```

```
    </style>
</head>
<body>
    <p id="pp" class="p1">我爱你</p>
    <script type="text/javascript">
        window.onload = function(){
            //如果修改的是元素的类属性值，那么类属性 js 操作的时候，不是 class 而是
className
            var pp = document.getElementById('pp');
            pp.onclick = function(){
                pp.className = 'p2';
            }
        }
    </script>
</body>
```

2. 自定义属性

(1) setAttribute 和 getAttribute 的用法

```
<body>
    <p aa='bb' id="pp" class="p1">我爱你尚硅谷</p>
    <script type="text/javascript">
        window.onload = function(){
            var pp = document.getElementById('pp');
            pp.onclick = function(){
//                pp['aa'] = 'cc';自定义属性无法使用.语法和[]语法
//                console.log(pp.getAttribute('aa'));//这个方法是用来获取属性值的
                this.setAttribute('aa','cc');//这个方法是用来设置属性值的;
```

```
            this.setAttribute('class','p2');//这个方法设置 class 的时候，不用改为 className
        }
    }
</script>
</body>
```

(2) 点语法和他们有啥区别？

① 在操作元素属性的时候，.语法只能操作元素天生具有的属性

② 自定义的属性，通过.语法是无法操作的；只能通过 setAttribute 和 getAttribute 去操作

③ setAttribute 和 getAttribute 是通用的方法，无论元素天生的还是自定义的属性都可以操作

(3) 天生的属性就用.语法去操作，自定义的属性采用 setAttribute 和 getAttribute 去操作；

11.5.2. 案例驱动：操作元素内容

1. 修改单个 p 标签的内容

```
window.onload = function(){
    var pNode = document.querySelector('p');
    pNode.onclick = function(){
        //如果 p 内部没有其他标签，只有文本，两种都是读取文本
        console.log(this.innerHTML);//读取的时候如果内部有标签，把标签和文本一起读取
        console.log(this.innerText);//读取的时候如果内部有标签，只会读取文本
        //如果设置的内容当中不带标签，那么都是直接设置文本
```

```
        this.innerHTML = '<h2>我爱你戚薇</h2>';//设置的时候标签在页面上会生效
        this.innerText = '<h2>我爱你戚薇</h2>';//设置的时候标签在页面上以原样字符串显式
    }
}
```

2. 修改多个 p 标签文字内容

```
<body>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <script type="text/javascript">
    //点击任意的一个 p，所有的 p 内容都要改为呵呵
    window.onload = function(){
        var pNodes = document.querySelectorAll('p');
        //外部循环的时候，很快，它只是给 5 个 p 每个都添加了一个事件监听（把 p 元素内的事件属性指向一个函数，定义函数）
        //此时 5 个 p，每个都会有自己的事件监听，函数都是不同的，但是函数代码是一样的；而且此时函数的代码都不执行
        for(var i = 0; i < pNodes.length; i++){

            pNodes[i].onclick = function(){
                //内部循环是在点击了某个 p 之后才会执行的，并且执行的就是点击的这个 p 事件监听函数当中的代码；
                for(var j = 0; j < pNodes.length; j++){
```

```
                pNodes[j].innerText = '呵呵';
            }
        }
    }
</script>
</body>
```

3. 排他思想操作修改文本内容

```
<body>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <p>哈哈</p>
    <script type="text/javascript">
        //点击任意的一个p，只有点击的这个p内容为嘿嘿  其它的为呵呵
        window.onload = function(){
            var pNodes = document.querySelectorAll('p');
            for(var i = 0; i < pNodes.length; i++){
                pNodes[i].onclick = function(){
                    //排它处理
                    //1、先让所有的都处于同一种状态，都是呵呵
                    for(var j = 0; j < pNodes.length; j++){
                        pNodes[j].innerHTML = '呵呵';
                    }
                    //2、再让点击的哪一个单独变成嘿嘿
```

```
                //事件内部的循环绝对不能使用事件外部的循环变量，敢用就敢错；
                //外部循环执行的时候事件内部是不执行的，只有触发事件的时候，事件内部才执行。
                //而触发事件的时候，外部循环早都执行完了。当触发事件的时候，内部使用外部的 i，i 的值
                //永远是外部循环完成后的 i 的值 5；
//                  console.log(i);
//                  pNodes[i].innerHTML = '嘿嘿';
                this.innerHTML = '嘿嘿';
            }
        }
    }
</script>
</body>
```

4. 排他实现小圆点点击切换

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }
        ul,li{
            list-style: none;
```

```
}
img{
    display: block;
}
a{
    text-decoration: none;
}
input{
    outline: none;
}

.clearFix:after{
    content: '';
    display: block;
    clear: both;
}
.iconList{
    overflow: hidden;
}
.iconList li{
    float: left;
    width: 40px;
    height: 40px;
    background-color: grey;
    border-radius: 50%;
    margin-right: 10px;
```

```
        }
        .iconList li.current{
            background-color: red;
        }
    </style>
</head>
<body>
    <ul class="iconList">
        <li class="current"></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
    </ul>
    <script type="text/javascript">
        window.onload = function(){
            var liNodes = document.querySelectorAll('.iconList li');
            for(var i = 0; i < liNodes.length; i++){
                liNodes[i].onclick = function(){
                    //排它
                    for(var j = 0; j < liNodes.length; j++){
                        liNodes[j].className = '';
                    }
                    this.className = 'current';
                }
            }
        }
```

```
    </script>
</body>
```

5. textContent 及 innerText 的区别

   (1) textContent 可以获取隐藏元素的文本，包含换行和空白，只有高级浏览器认识

   (2) innerText 不可以获取，并且不包含换行和空格，所有浏览器都认识

11.5.3. 案例驱动：操作元素样式

1. 点击按钮修改元素的宽高以及背景色；

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }

        #box{
            width: 300px;
            height: 300px;
            background-color: yellow;
        }
    </style>
</head>
```

```
<body>
    <button>按钮</button>
    <div id="box">
    </div>
    <script type="text/javascript">
        window.onload = function(){
            var btn = document.querySelector('button');
            var box = document.getElementById('box');
            btn.onclick = function(){
//                box.style.width = '200px';
//                box.style.height = '200px';
//                box.style.backgroundColor = 'green';
                console.log(box.style.width);
            }
            //js 操作的 css 都是行内样式，因为行内权重比较高
            //js 一般操作样式的时候，box.style.width 只能设置，不能读取，读取的是行内的样式；开始时没有行内的；
        }
</body>
```

2. 导航鼠标移入高亮显示

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
```

```
        margin: 0;
        padding: 0;
    }
    ul,li{
        list-style: none;
    }
    a{
        text-decoration: none;
    }
    .list{
        width: 200px;
        height: 150px;
        border: 1px solid black;
    }

    .list li{
        width: 200px;
        height: 50px;
        background-color: greenyellow;
        text-align: center;
        line-height: 50px;
    }

    .list li:nth-child(2){
        border-bottom: 1px solid black;
        border-top: 1px solid black;
        box-sizing: border-box;
```

```
        }


    </style>
</head>
<body>
    <ul class="list">
        <li><a href="javascript:;">男装</a></li>
        <li><a href="javascript:;">女装</a></li>
        <li><a href="javascript:;">童装</a></li>
    </ul>

    <script type="text/javascript">
        window.onload = function(){
            var liNodes = document.querySelectorAll('.list li');

            for(var i = 0; i < liNodes.length; i++){
                //每一个 li 身上都添加了移入和移出两个事件监听;
                liNodes[i].onmouseover = function(){
                    this.style.backgroundColor = 'orange';
                };

                liNodes[i].onmouseout = function(){
                    this.style.backgroundColor = 'greenyellow';
                };
            }
        }
    </script>
```

```
</body>
```

3. 表格隔行变色

```
<body>
    <table>
        <!--tbody 没写不代表没有，页面渲染的时候 tbody 会自动加上-->
        <tr>
            <th>姓名</th>
            <th>性别</th>
            <th>年龄</th>
        </tr>

        <tr>
            <td>张三</td>
            <td>男</td>
            <td>18</td>
        </tr>

        <tr>
            <td>浩浩</td>
            <td>男</td>
            <td>18</td>
        </tr>

        <tr>
            <td>于敏红</td>
```

```
            <td>女</td>
            <td>18</td>
        </tr>
    </table>

    <script type="text/javascript">
        window.onload = function(){
            var trNodes = document.querySelectorAll('tr');
            //表格奇偶隔行变色
            for(var i = 0; i < trNodes.length; i++){
                if(i % 2 == 0){
                    trNodes[i].style.backgroundColor = 'skyblue'
                }else{
                    trNodes[i].style.backgroundColor = 'hotpink'
                }
            }
        }
    </script>
</body>
```

4. 开关状态改变 div 背景色；

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
```

```
        padding: 0;
    }

    #box{
        width: 200px;
        height: 200px;
        background-color: red;
    }
</style>
</head>
<body>
<div id="box">

</div>
<script type="text/javascript">
    window.onload = function(){
        var box = document.querySelector('#box');
        var flag = true;
        box.onclick = function(){
            if(flag){
                box.style.backgroundColor = 'green';
//                  flag = false;
            }else{
                box.style.backgroundColor = 'red';
//                  flag = true;
            }
            flag = !flag;
```

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

11.6. 键盘事件和焦点事件

11.6.1. onkeyup，onkeydown 键盘事件

1. 一般用的都是 keyup 事件
2. 它能够确保键盘事件只执行一次；
3. keyCode 存在于事件对象当中，可以通过它判断触发的是什么键
   (1) 事件对象就是我们回调函数的第一个形参值
   (2) 事件对象不是我们创建，当事件触发的时候，系统会创建好这个事件对象，并且自动传参调用
   (3) 事件对象包含了和事件触发相关的一切，每次事件触发都有对应的事件对象
4. 判断是否是回车事件

```
window.onload = function(){
    var inputNode = document.querySelector('input');
    //键盘事件通常是添加在表单类元素身上，还会添加在 doucument 身上
    document.onkeyup = function(event){
        if(event.keyCode == 13){
            console.log('回车');
        }
    }
}
```

11.6.2. onfocus 和 onblur 事件

获取焦点的时候 input 背景色和字体颜色固定

当失去焦点的时候背景颜色和字体颜色随机改变（变色龙）

```
window.onload = function(){
    var inputNode = document.querySelector('input');
    inputNode.onfocus = function(){
        //获取焦点事件
        this.style.backgroundColor = 'skyblue';
        this.style.color = 'hotpink';
    }
    inputNode.onblur = function(){
        //随机变色，变色龙
        var red = Math.floor(Math.random() * 256);
        var green = Math.floor(Math.random() * 256);
        var blue = Math.floor(Math.random() * 256);
        var red1 = Math.floor(Math.random() * 256);
        var green1 = Math.floor(Math.random() * 256);
        var blue1 = Math.floor(Math.random() * 256);
        this.style.backgroundColor = 'rgb('+ red +','+ green +','+ blue +')';
        this.style.color = 'rgb('+ red1 +','+ green1 +','+ blue1 +')';
    }
}
```

11.7. 综合案例

1. 案例：全选全不选反选

```
<body>
    <label for="ii">抽烟</label>
    <input type="checkbox" id="ii"/>

    <label>
        喝酒<input type="checkbox" />
    </label>

    <label>
        烫头<input type="checkbox" />
    </label>

    <label>
        打豆豆<input type="checkbox" />
    </label>

    <br />

    <button>全选</button>
    <button>全不选</button>
    <button>反选</button>

    <script type="text/javascript">
        window.onload = function(){
```

```
        var btns = document.querySelectorAll('button');
        var inputNodes = document.querySelectorAll('input[type="checkbox"]');
        //全选
        btns[0].onclick = function(){
            for(var i = 0; i < inputNodes.length; i++){
                inputNodes[i].checked = true;
            }
        }
        //全不选
        btns[1].onclick = function(){
            for(var i = 0; i < inputNodes.length; i++){
                inputNodes[i].checked = false;
            }
        }
        //反选
        btns[2].onclick = function(){
            for(var i = 0; i < inputNodes.length; i++){
                inputNodes[i].checked = !inputNodes[i].checked;
            }
        }
    }
</script>
</body>
```

2. 案例：轮播图按钮鼠标移入透明度渐变

```
<head>
<meta charset="UTF-8">
```

```
<title></title>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    ul,li{
        list-style: none;
    }

    img{
        display: block;
        /*vertical-align: middle;*/
    }

    a{
        text-decoration: none;
    }

    input{
        outline: none;
    }

    .clearFix:after{
        content: '';
        display: block;
```

```
    clear: both;
}

#box{
    position: relative;
    width: 600px;
    height: 300px;
    margin: 50px auto;
    overflow: hidden;
}

#box .list{
    width: 3000px;
    height: 300px;
}

#box .list li{
    float: left;
    width: 600px;
    height: 300px;
}

#box .list li img{
    width: 600px;
    height: 300px;
}
```

```
#box .iconList{
    position: absolute;
    left: 50%;
    transform: translateX(-50%);
    bottom: 10px;
    /*overflow: hidden;它和 position: absolute 绝对定位都能开启 BFC 达到清除浮动的效果*/
}

#box .iconList li{
    float: left;
    width: 30px;
    height: 30px;
    border-radius: 50%;
    background-color: grey;
    margin-right: 10px;
}

#box .iconList li.current{
    background-color: red;
}

#box span{
    position: absolute;
    top: 50%;
    transform: translateY(-50%);
```

```
        width: 50px;
        height: 100px;
        background-color: rgba(150,150,150,.7);
        font-size: 40px;
        text-align: center;
        line-height: 100px;
        color: white;
        opacity: 0;
        transition: opacity 10s;
    }

    #box .left{
        left: 0;
    }

    #box .right{
        right: 0;
    }

</style>
</head>
<body>
<div id="box">
    <ul class="list">
        <li><img src="img/3.jpg" alt="" /></li>
        <li><img src="img/22.jpg" alt="" /></li>
        <li><img src="img/24.jpg" alt="" /></li>
```

```
        <li><img src="img/17.jpg" alt="" /></li>
        <li><img src="img/5.jpg" alt="" /></li>
    </ul>

    <ul class="iconList">
        <li class="current"></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
    </ul>
    <span class="left">  <  </span>
    <span class="right">  >  </span>

</div>

<script type="text/javascript">
    window.onload = function(){
        var box = document.querySelector('#box');
        var spanNodes = document.querySelectorAll('#box span');

        box.onmouseover = function(){
            spanNodes[0].style.opacity = 1;
            spanNodes[1].style.opacity = 1;
        }

        box.onmouseout = function(){
```

```
            spanNodes[0].style.opacity = 0;
            spanNodes[1].style.opacity = 0;
        }
    }
</script>
</body>
```

3. 案例：二级菜单　选项卡　**********

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }

        ul,li{
            list-style: none;
        }

        a{
            text-decoration: none;
        }

        .list{
            position: relative;
```

```
        width: 200px;
        height: 150px;
        border: 1px solid black;
    }

    .list>li{
        width: 200px;
        height: 50px;
        background-color: hotpink;
        text-align: center;
        line-height: 50px;
    }

    .list>li:nth-child(2){
        border-bottom: 1px solid black;
        border-top: 1px solid black;
        box-sizing: border-box;
    }

    .list>li .listIn{
        display: none;
        position: absolute;
        left: 201px;
        top: -1px;
        width: 200px;
        height: 150px;
        border: 1px solid blueviolet;
```

```
        }

        .list>li .listIn li{
            width: 200px;
            height: 50px;
            background-color: greenyellow;
        }

        .list>li .listIn li:nth-child(2){
            border-bottom: 1px solid blueviolet;
            border-top: 1px solid blueviolet;
            box-sizing: border-box;
        }

    </style>
</head>
<body>
    <ul class="list">
        <li>
            <a href="javascript:;">男装</a>
            <ul class="listIn">
                <li>夹克</li>
                <li>格子衫</li>
                <li>皮鞋</li>
            </ul>
        </li>
        <li>
```

```
        <a href="javascript:;">女装</a>
        <ul class="listIn">
            <li>裙子</li>
            <li>高跟鞋</li>
            <li>打底裤</li>
        </ul>
    </li>
    <li>
        <a href="javascript:;">家电</a>
        <ul class="listIn">
            <li>电视</li>
            <li>洗衣机</li>
            <li>冰箱</li>
        </ul>
    </li>
</ul>

<script type="text/javascript">
    window.onload = function(){
        var liNodes = document.querySelectorAll('.list>li');
        var ulInNodes = document.querySelectorAll('.list>li .listIn');

        for(var i = 0; i < liNodes.length; i++){
            //打死也得记住
            liNodes[i].index = i;

            liNodes[i].onmouseover = function(){
```

```
            this.style.backgroundColor = 'orange';
            //在内部无法用 i 的值，也就是说在内部我们没办法使用 i 获取下标
            ulInNodes[this.index].style.display = 'block';

        };

        liNodes[i].onmouseout = function(){
            this.style.backgroundColor = 'hotpink';
            //在内部无法用 i 的值，也就是说在内部我们没办法使用 i 获取下标
            ulInNodes[this.index].style.display = 'none';
        };
    }

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

11.8. 兼容性封装设置读取内容函数

11.8.1. 浏览器兼容性讲解

1. 浏览器 IE8 级以下被称作低级浏览器
2. 其余浏览器我们称作高级浏览器
3. innerTxet 所有的浏览器都能使用这个属性操作内容
4. textContent 只有高级浏览器能使用
5. 兼容性是为了让高级浏览器和低级浏览器都能够自动根据浏览器版本选择

合适的属性操作内容。

11.8.2. 封装函数处理兼容性读写内容

```
<body>
    <p id="pp">尚硅谷</p>
    <script type="text/javascript">
        //兼容性封装函数实现 textContent 和 innerText
        var pNode = document.getElementById('pp');
//        console.log(pNode.textContent);
        //如果直接用 textContent，导致以后低版本浏览器用户无法看到这个值
        function setOrGetElementContent(node,content){
            if(arguments.length == 1){
                //读取内容
                //判断用户浏览器是高级还是低级
                if(node.textContent){//如果是高级浏览器拿到的就不是 undefiend
                    //高级浏览器
                    return node.textContent;
                }else{
                    //低级浏览器
                    return node.innerText;
                }
            }else if(arguments.length == 2){
                //设置内容
                if(node.textContent){
                    //高级
                    node.textContent = content;
```

```
            }else{
                //低级
                node.innerText = content;
            }
        }
    }
    console.log(setOrGetElementContent(pNode));
</script>
</body>
```

12. 第 12 章：DOM 节点操作

12.1. 节点的概念

12.1.1. 什么是节点

文档树所有包含的东西都可以称作节点

12.1.2. 节点的分类

1. 节点一共有 12 类，我们文档可以看到有下面四类
   (1) 元素　　标签
   (2) 文本　　文本内容
   (3) 属性　　元素属性
   (4) 注释　　注释
2. 我们最关注的就是元素节点，拿到元素节点可以操作文本、属性还有样式

12.2. 查找节点

12.2.1. 子节点和子元素节点

1. 子节点：childNodes (儿子节点):
   (1) 拿到的是某个元素的子节点
   (2) 包括元素子节点和文本子节点，如果有注释还有注释节点；
      ① 高级浏览器: 元素,文本(文本,空格,换行),注释
      ② 低版本浏览器: 元素,文本(不包括空格和换行),注释
2. 子元素节点：children(儿子元素):
   (1) 拿到的是某个元素的子元素节点
      ① 高级浏览器:元素
      ② 低版本浏览器:元素,注释
3. 节点的三大属性 nodeName、nodeType、nodeValue
4. 案例：获取元素的子节点和子元素

```
<body>
    <div id="box">
        <h2>我是一个标题</h2>
        <p>我是一个段落</p>
        <span>我是一个 span</span>
    </div>
    <script type="text/javascript">
        window.onload = function(){
            //1、子节点
            var box = document.getElementById('box');
            console.log(box.childNodes);//获取指定元素的子节点
            //                          nodeName    nodetype    nodeValue
```

```
        //文本节点              #text          3       文本内容
        //元素节点              大写元素名      1        null
        //注释节点              #comment       8       注释内容

        //在元素的子节点当中筛选过滤出子元素节点，放在真数组当中；
        var arr = [];
        for(var i = 0; i < box.childNodes.length; i++){
            if(box.childNodes[i].nodeType === 1){
                arr.push(box.childNodes[i]);
            }
        }
        console.log(arr);



        //2、子元素节点
        console.log(box.children);//指定元素的子元素节点，高版本拿到的只是元素低版本拿到的除了元素还有注释
        //从元素的子元素节点当中筛选子元素节点（低版本浏览器拿到的除了子元素还有注释）
        var arr = [];
        for(var i = 0; i < box.children.length; i++){
            if(box.children[i].nodeType === 1){
                arr.push(box.children[i]);
            }
        }
        console.log(arr);
    }
```

```
    </script>
</body>
```

12.2.2. 父节点和父元素节点

1. 父节点:parentNode 其实就是父元素(标签) 所有浏览器都能使用
2. 父元素:parentElement 也是父标签 所有浏览器都能用
3. 父节点:parentNode 和父元素:parentElement 获取的其实都一样是父元素

12.2.3. 查找节点的其它方式

1.	第一个子节点	都认识	firstChild
2.	第一个子元素节点	只有高级浏览器可以使用	firstElementChild
3.	最后一个节点	都认识	lastChild
4.	最后一个元素节点	只有高级浏览器可以使用	lastElementChild
5.	上一个兄弟节点	都认识	previousSibling
6.	上一个兄弟元素节点	只有高级浏览器可以使用	previousElementSibling
7.	下一个兄弟节点	都认识	nextSibling
8.	下一个兄弟元素节点	只有高级浏览器可以使用	nextElementSibling

```
<body>
    <ul>
        <li>我是列表项 1</li>
        <li>我是列表项 2</li>
        <li>我是列表项 3</li>
        <li>我是列表项 4</li>
        <li class="single">我是列表项 5</li>
        <li>我是列表项 6</li>
```

```
    <li>我是列表项 7</li>
    <li>我是列表项 8</li>
</ul>
<script type="text/javascript">
    window.onload = function(){
        var ulNode = document.querySelector('ul');
        console.log(ulNode.firstChild);
        console.log(ulNode.firstElementChild);
        console.log(ulNode.lastChild);
        console.log(ulNode.lastElementChild);
        var liNode = document.querySelector('.single');
        console.log(liNode.previousSibling);
        console.log(liNode.previousElementSibling);
        console.log(liNode.nextSibling);
        console.log(liNode.nextElementSibling);
        //兼容封装获取第一个子元素节点
        function getFirstElement(node){
            if(node.firstElementChild){
                //高级浏览器
                return node.firstElementChild;
            }else{
                //低级浏览器
                var ele = node.firstChild;
                while(ele && ele.nodeType != 1){
                    ele = ele.nextSibling;
                }
                return ele;
```

```
            }
        }
        var result = getFirstElement(ulNode);
        console.log(result);
    }
</script>
</body>
```

12.3. 创建节点

1. 第一种创建节点的方式：document.write()
2. 第二种创建方式：innerHTML
3. 第三种创建方式：DOM 方法 createElement()和 appendChild()

```
<body>
    <p>我是一个段落</p>
    <script type="text/javascript">
        window.onload = function(){
            //第一种创建方式
            document.write('<h2>嘿嘿</h2>');



            //几种特殊元素的获取方式
            console.log(document.body);  //直接可以获取 body 元素封装为 dom 对象
            console.log(document.head);  //直接可以获取 head 元素封装为 dom 对象
            console.log(document.documentElement);//直接可以获取 html 元素封装为 dom
对象
```

```
        //第二种创建方式
        document.body.innerHTML += '<h2>嘿嘿</h2>';

        //第三种方式创建  dom 方法去创建
        //第一步：先创建一个要添加的元素
        var h2Node = document.createElement('h2');
        //第二步：给新创建的 dom 对象添加内容
        h2Node.innerHTML = '嘿嘿';
        //第三步：把新创建好的 dom 对象添加到 dom 树上
        document.body.appendChild(h2Node);
    }
</script>
</body>
```

4. 案例：使用第二种和第三种创建一个列表

```
window.onload = function(){
    //使用第二种方式去创建列表
    var arr = ['赵丽颖','戚薇','李沁','迪丽热巴','贾玲','佟丽娅'];
    var str = '<ul>';
    for(var i = 0; i < arr.length; i++){
        str += '<li>'+arr[i]+'</li>';
    }
    str += '</ul>';
    document.body.innerHTML = str;

    //第三种方式创建
```

```
    var ulNode = document.createElement('ul');
    for(var i = 0; i < arr.length; i++){
        //创建 li
        var liNode = document.createElement('li');
        //给 li 里面添加内容
        liNode.innerHTML = arr[i];
        //把创建好的 li 添加给 ul
        ulNode.appendChild(liNode);
    }
    //把组装好的 ul 添加到文档树 body 里面
    document.body.appendChild(ulNode);
}
```

12.4. 节点的增删改

以下方法都是父元素调用,操作子元素

1. 插入节点: insertBefore(新节点,参照节点)
2. 替换节点: replaceChild(新节点,被替换的节点)
3. 删除节点: removeChild(被删除的节点)
4. 追加节点：appendChild(被追加的节点)

```
<body>
    <ul>
        <li>冰雨</li>
        <li>句号</li>
        <li>断点</li>
        <li>野狼 disco</li>
```

```
        <li>大碗宽面</li>
        <li>鸡你太美</li>
    </ul>
    <script type="text/javascript">

        window.onload = function(){
            //在指定的元素前面插入一个元素
            var ulNode = document.querySelector('ul');
            var liOld = document.querySelector('ul li:nth-child(4)');
            var liNew = document.createElement('li');
            liNew.innerHTML = '你的酒馆对我打了烊';
            ulNode.insertBefore(liNew,liOld);

            //替换指定的元素
            var ulNode = document.querySelector('ul');
            var liOld = document.querySelector('ul li:last-child');
            var liNew = document.createElement('li');
            liNew.innerHTML = '王妃';
            ulNode.replaceChild(liNew,liOld);

            //删除指定的元素
            var ulNode = document.querySelector('ul');
            var liOld = document.querySelector('ul li:nth-child(5)');
            ulNode.removeChild(liOld);
        }
    </script>
</body>
```

12.5. 综合案例

触发回车之后，把表单的内容动态创建 li 标签，并且所有的 li 标签移入变色

```
<body>
    <input type="text" placeholder="请输入电影名称" />
    <ul></ul>
    <script type="text/javascript">
        window.onload = function(){
            var inputNode = document.querySelector('input');
            var ulNode = document.querySelector('ul');
            inputNode.onkeyup = function(event){
                if(event.keyCode == 13){
                    //拿内容存储起来
                    var content = this.value;//表单类元素获取它的内容的时候拿的是
value 属性值，不是 innerText
                    //从元素当中拿到的内容都是字符串
                    //当内容有效再去创建
                    if(content.trim()){
                        //创建新的 li
                        var liNode = document.createElement('li');
                        //在创建的同时，就给每个 li 添加上事件监听，就不用后面重新再去
查找遍历；
                        liNode.onmouseover = function(){
                            this.style.backgroundColor = 'hotpink';
                        };
```

```
                liNode.onmouseout = function(){
                    this.style.backgroundColor = 'white';
                };

                liNode.innerText = content;
                ulNode.appendChild(liNode);
                //创建好 li 之后，把表单元素里面的内容清空
            }else{
                //没有内容或者是无效内容提示
                alert('请输入合法的电影名称');
            }
            //完成后把表单内容清空
            this.value = '';
        }
    }
</script>
</body>
```

13. 第 13 章：DOM 和 BOM

13.1. DOM

13.1.1. DOM 版本简介

1. DOM 版本有 DOM0、DOM1、DOM2 和 DOM3
2. **DOM0 和 DOM2** 有自己独立的**事件绑定和解绑方式**

3. DOM1 和 DOM3 没有事件绑定和解绑方式

13.1.2. Dom0 事件绑定和解绑

1. DOM0 事件绑定和解绑方式高低版本浏览器通用
2. DOM0 事件不能绑定同一类事件多次，如果绑定后面会覆盖前面的事件监听
3. DOM0 事件解绑本质上就是把事件回调函数和事件对象的事件属性断开指向

```
<body>
    <div id="box"></div>
    <button>解绑</button>
    <script type="text/javascript">
        window.onload = function(){
            var box = document.getElementById('box');
            var btn = document.querySelector('button');
            //dom0 绑定
            box.onclick = function(){
                console.log('i love you~ 赵丽颖!');
            }
            //dom0 事件如果添加同一类型事件多次，后面会把前面的覆盖掉
            box.onclick = function(){
                console.log('i love you~ 杨幂!');
            }
            //dom0 解绑  点击按钮之后解绑 dom0 事件
            btn.onclick = function(){
                //dom 事件解绑的本质就是把事件属性和事件函数断开连接
                box.onclick = null;
            }
```

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

13.1.3. Dom2 事件绑定和解绑

1. 可以添加同一类事件多次
2. 高级浏览器和低级浏览器绑定方式不同
3. 高级浏览器和低级浏览器解绑方式不同
4. dom2 事件解绑，函数必须放在外面去定义有名回调函数

```
<body>
  <div id="box"></div>
  <button id="btn">解绑</button>
  <script type="text/javascript">
    window.onload = function(){
      var box = document.getElementById('box');
      var btn = document.getElementById('btn');
      //1、高级浏览器
      //高级浏览器绑定
      //绑定的事件不需要解绑，可以这么定义
      box.addEventListener('click',function(){
        console.log('i love you~ zhao li ying~');
      })

      //dom2 事件添加同一类型事件多次，会依次执行事件，不会覆盖
      box.addEventListener('click',function(){
```

```
        console.log('i love you yangmi~');
    })

    //当绑定的事件需要解绑，那么事件回调定义在外面，写成有名函数
    function fn(){
        console.log('i love you~ zhao li ying~');
    }
    box.addEventListener('click',fn);

    //高级浏览器解绑
    //解绑的时候，有解绑的方法，而且方法当中传的参数 必须和绑定的时候参数完全一样才能解绑
    //如果你的事件需要解绑，那么函数必须在外面定义成有名函数，才能保证绑定和解绑的是同一个函数；
    btn.onclick = function(){
        box.removeEventListener('click',fn);
    }
    //2、低级浏览器绑定和解绑方法换成了 attachEvent 和 detachEvent，事件类型前面需要加 on
    //3、兼容性处理元素绑定 dom2 事件
    function addEvent(node,eventType,callback,isBubble){
        //
        if(node.addEventListener){
            //高级
            //高级浏览器添加事件的时候，第三个参数代表的是事件流，捕获(true)还是冒泡（false）
            node.addEventListener(eventType,callback,isBubble);
```

```
            }else{
                //低级
                //低级浏览器添加事件的时候，没有第三个参数，因为 ie 浏览器事件流主张的就是冒泡
                node.attachEvent('on'+eventType,callback);
            }
        }
    }
</script>
</body>
```

13.2. 事件流（事件传播）

13.2.1. 事件流的概念：

1. 捕获事件流（网景）
2. 冒泡事件流（ie）
3. DOM2 标准事件流
4. 标准事件流包含三个阶段： 捕获、目标、冒泡
5. 事件流对于每个事件（不是事件监听）都是客观存在的

13.2.2. 事件冒泡、事件捕获及阻止事件冒泡

```
<head>
    <meta charset="UTF-8">
    <title></title>
```

```
<style>
    *{
        margin: 0;
        padding: 0;
    }
    .laoda{
        position: relative;
        width: 500px;
        height: 500px;
        background-color: red;
    }
    .laoer{
        position: absolute;
        left: 50%;
        top: 50%;
        transform: translate(-50%,-50%);
        width: 300px;
        height: 300px;
        background-color: green;
    }
    .laomo{
        position: absolute;
        left: 0;
        right: 0;
        bottom: 0;
        top: 0;
        margin: auto;
```

```
            width: 100px;
            height: 100px;
            background-color: blue;
        }
    </style>
</head>
<body>
    <div class="laoda">
        <div class="laoer">
            <div class="laomo"></div>
        </div>
    </div>

    <script type="text/javascript">
        window.onload = function(){
            var laoda = document.querySelector('.laoda');
            var laoer = document.querySelector('.laoer');
            var laomo = document.querySelector('.laomo');
            //1、dom0 事件及 dom2 低版本浏览器事件都是冒泡
            laomo.onclick = function(event){
                console.log('我是老末');
                //阻止事件冒泡要使用事件对象
                event.stopPropagation();//阻止事件冒泡
            }

            laoer.onclick = function(e){
                console.log('我是老 er');
```

```
            //阻止事件冒泡要使用事件对象
            e.stopPropagation();//阻止事件冒泡     window.event 给低级浏览器用的，但是高级也可以使用
        }

        laoda.onclick = function(event){
            console.log('我是老 da');
            //阻止事件冒泡要使用事件对象
            event.stopPropagation();//阻止事件冒泡
        }
        document.body.onclick = function(){
            console.log('我是 body');
        };
        document.documentElement.onclick = function(){
            console.log('我是 html');
        };
        document.onclick = function(){
            console.log('我是祖宗');
        }

        //2、捕获我们可以在 dom2 事件高级浏览器上进行观察
        //了解就好
        laomo.addEventListener('click',function(){
            console.log('我是老末');
        },true);
        laoer.addEventListener('click',function(){
            console.log('我是老 er');
```

```
        },true);
        laoda.addEventListener('click',function(){
            console.log('我是老 da');
        },true);
    }
</script>
</body>
```

13.3. 事件委派

13.3.1. 什么是事件委派

1. 把子元素的事件监听添加给父（祖先）元素,把子元素发生的事件委托给父元素进行处理
2. 事件委派过程当中依赖了事件冒泡
3. 阻止事件冒泡是为了解决事件冒泡给我们带来的困扰
4. 事件冒泡的好处就是可以进行事件委派（事件委托，事件代理）

13.3.2. 事件委派用法，

1. 什么时候用：

   (1) 子元素很多，每个子元素都添加相同的事件

```
<body>
    <ul>
        <li>我是列表项 1</li>
        <li>我是列表项 2</li>
```

```
        <li>我是列表项 3</li>
        <li>我是列表项 4</li>
        <li>我是列表项 5</li>
        <li>我是列表项 6</li>
        <li>我是列表项 7</li>
        <li><span>我是列表项 8</span></li>
    </ul>

    <script type="text/javascript">
        window.onload = function(){
            //当子元素有很多的时候，并且子元素的事件监听里面的逻辑是相同的，最好使用事件委派
            //列表项移入高亮
            var ulNode = document.querySelector('ul');
            ulNode.onmouseover = function(event){
                //event.target 就代表的是目标元素节点
                if(event.target.nodeName == 'LI'){
                    //为什么要判断，确保拿到的就是我们所需要的目标元素，不能是 ul
                    event.target.style.backgroundColor = 'red';
                }else if(event.target.parentNode.nodeName == 'LI'){
                    event.target.parentNode.style.backgroundColor = 'red';
                }
            };
            ulNode.onmouseout= function(){
                if(event.target.nodeName == 'LI'){
                    event.target.style.backgroundColor = 'white';
                }else if(event.target.parentNode.nodeName == 'LI'){
```

```
                event.target.parentNode.style.backgroundColor = 'white';
            }
        };
    }
</script>
</body>
```

(2) 动态新添加元素，并且新添加的元素要和老的拥有同样的事件行为

```
<body>
    <ul>
        <li>我是列表项 1</li>
        <li>我是列表项 2</li>
        <li>我是列表项 3</li>
        <li>我是列表项 4</li>
        <li>我是列表项 5</li>
    </ul>
    <button>添加</button>
    <script type="text/javascript">
        //如果本身我们有一部分 li  后续我们还要动态的去添加一部分
        //如果这两部分都要具有相同的一些效果，那么此时事件最好是事件委派
        //事件委派可以让  新的和老的都有效果;
        window.onload = function(){
            var ulNode = document.querySelector('ul');
            var liNodes = document.querySelectorAll('li');
            var btn = document.querySelector('button');
            ulNode.onmouseover = function(e){
```

```
            //高级浏览器在调用回调的时候会把事件对象传递给第一个形参
            //低级浏览器在调用回调的时候不会事件对象给第一个形参，而是给了window 下的属性叫 event
            e = e || window.event;//高级浏览器和低级浏览器兼容写法
            var target = e.target || e.srcElement;//获取目标元素高级和低级浏览器兼容写法;
            if(target.nodeName == 'LI'){
                target.style.backgroundColor = 'red';
            }
        };

        ulNode.onmouseout= function(event){
            event = event || window.event;
            var target = event.target || event.srcElement;
            if(target.nodeName == 'LI'){
                target.style.backgroundColor = 'white';
            }
        };
        btn.onclick = function(){
            var liNode = document.createElement('li');
            liNode.innerHTML = '我是新的';
            ulNode.appendChild(liNode);
        }
    }
  </script>
</body>
```

2. 事件委派的做法
   (1) 给共有父元素添加事件监听
   (2) 不给子元素本身添加
   (3) 事件触发后在父元素的处理回调中通过事件对象去找真正发生事件的目标元素进行处理
3. 事件委派的好处：**节省内存、提高效率**。

13.4. 事件对象

13.4.1. 事件对象概念

1. **任何事件**都会有**事件对象**
2. 这个对象当中封装了和触发事件相关的一切信息

13.4.2. 事件对象兼容性处理

1. **高级浏览器**调用函数的回调函数，系统会把事件对象封装好传给回调函数的**第一个形参**
2. **低版本浏览器**去调用，系统会把事件对象封装好作为 window 的一个属性 **window.event**
3. 兼容性写法 **event = event || window.event**

13.4.3. 目标元素兼容处理

event.target || event.srcElement

13.4.4. 三种鼠标位置

1. clientX 和 clientY

(1) 拿的是鼠标相对视口的水平距离和垂直距离

(2) 相对的是视口的左上角（以视口左上角为原点）

2. pageX 和 pageY

(1) 拿的是鼠标相对页面的 水平距离和垂直距离

(2) 相对的是页面的左上角（以页面左上角为原点）

3. offsetX 和 offsetY

(1) 拿的是鼠标相对自身元素的 水平距离和垂直距离

(2) 相对的是自身元素左上角（以自身元素左上角为原点）

4. 鼠标跟随案例

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }
        img{
            display: block;
            width: 80px;
            height: 80px;
            position: absolute;
            left: 0;
            top: 0;
        }
    </style>
</head>
```

```
<body>
    <img src="img/10.jpg"/>
    <script type="text/javascript">
        window.onload = function(){
            var imgNode = document.querySelector('img');
            document.onmousemove = function(event){
                event = event || window.event
                //获取鼠标的位置  让图片的定位到这个位置
                console.log(event.clientX,event.clientY)
                imgNode.style.left = event.clientX + 'px';
                imgNode.style.top = event.clientY + 'px';
            }
        }
    </script>
</body>
```

13.5. BOM

13.5.1. window 对象

1. window 对象是 BOM 的顶级对象，称作浏览器窗口对象
2. 全局变量会成为 window 对象的属性
3. 全局函数会成为 window 对象的方法
4. window.onload
5. window.onresize
6. window.onscroll

13.5.2. location 对象

1. window.location 可以让用户获取当前页面地址以及重定向到一个新的页面
2. window.location.href 可以读也可以写，写的时候相当于转向另外一个页面

13.5.3. History 对象

1. window.history 对象包含浏览器的历史记录，window 可以省略。
2. 这些历史记录以栈的形式保存。页面前进则入栈，页面返回则出栈

13.5.4. navigator 对象

1. window.navigator 是一个只读对象
2. 它用来描述浏览器本身的信息，包括浏览器的名称、版本、语言、系统平台、用户特性字符串等信息

13.5.5. screen 对象

window.screen 提供了用户显示屏幕的相关属性，比如显示屏幕的宽度、高度，可用宽度、高度。

```
//window 对象
window.onload = function(){
    //等待页面加载完成事件
    window.onresize = function(){
        //浏览器窗口大小发生改变的事件
        console.log(document.documentElement.clientWidth);//获取浏览器视口宽度
        console.log(document.documentElement.clientHeight);//获取浏览器视口高度
    };
```

```
window.onscroll = function(){
    //系统滚动事件
    console.log('滚');
}

//history 对象  历史记录
history.back();//历史记录返回上一页
history.forward();//去到下一页
history.go(-2);//去到指定的历史记录页  0 代表当前页  -1 代表之前  1 代表之后

//location 对象  地址栏  可以修改和读取地址
console.log(window.location.href);
window.location.href = 'https://www.baidu.com';//重定向到百度

//navigator 对象，负责浏览器的版本、名称等信息
console.log(window.navigator.appName);//Netscape  网景  所有打印出来的都是这个网景
console.log(window.navigator.appVersion);//浏览器版本
console.log(window.navigator.appCodeName);//浏览器内核版本，但是打印出来一般都不准;

//screen  屏幕
console.log(window.screen.width);//拿到屏幕的宽  分辨率
console.log(window.screen.height);//拿到屏幕的高
}
```

13.6. 学生管理系统

```
<body>
    <table>
        <tr>
            <th>姓名</th>
            <th>年龄</th>
            <th>操作</th>
        </tr>

        <tr>
            <td>刘小东</td>
            <td>18</td>
            <td><a href="javascript:;">删除</a></td>
        </tr>

        <tr>
            <td>杨晨</td>
            <td>18</td>
            <td><a href="javascript:;">删除</a></td>
        </tr>

        <tr>
            <td>王凯凯</td>
            <td>18</td>
            <td><a href="javascript:;">删除</a></td>
        </tr>
```

```
</table>

<input type="text" placeholder="请输入您的姓名"/> <br />
<input type="text" placeholder="请输入你的年龄"/> <br />
<input type="button" value="增加"/>



<script type="text/javascript">
    window.onload = function(){
        //1、增加的逻辑
        var inputNodes = document.querySelectorAll('input');
        var tbodyNode = document.querySelector('tbody');
        inputNodes[2].onclick = function(){
            //1、获取填写的内容保存,拿到的内容都是字符串
            var username = inputNodes[0].value;
            var age = inputNodes[1].value;//拿到是字符串

            //判断内容的合法性，根据合法不合法决定是否继续创建
            if(username.trim()&&age.trim()){
                //2、创建组装一个 tr
                var trNode = document.createElement('tr');

                var tdName = document.createElement('td');
                tdName.innerHTML = username;
                trNode.appendChild(tdName);

                var tdAge = document.createElement('td');
```

```
            tdAge.innerHTML = age;
            trNode.appendChild(tdAge);

            var tdOperate = document.createElement('td');
            tdOperate.innerHTML = '<a href="javascript:;">删除</a>';
            trNode.appendChild(tdOperate);

            //3、把创建好的 tr 放到 tbody 当中，tbody 默认不写，但是存在
            tbodyNode.appendChild(trNode);


        }else{
            alert('用户名或者年龄不能为空');
        }
        //4、操作完成清空 input 当中的内容
        inputNodes[0].value = '';
        inputNodes[1].value = '';
    }

    //2、删除的逻辑    事件委派，因为老的和新的都有点击删除的效果

    tbodyNode.onclick = function(event){
        event = event || window.event;
        var target = event.target || event.srcElement;

        if(target.nodeName == 'A'){
            //删除点击的那个 a 相关的那一行
```

```
                this.removeChild(target.parentElement.parentNode);
            }


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

14. 第 14 章：定时器、元素位置大小及初始包含快

14.1. 定时器：

14.1.1. 单次定时器（延迟定时器）

1. 一般用来做延迟效果　定时炸弹
2. 案例 求出结果延迟 5 秒打印

```
<body>
    <button>清除</button>
    <script type="text/javascript">
        window.onload = function(){
            //延迟 5 秒再打印，延迟定时器 想成是定时炸弹
            var a = 10;
            var b = 20;
            var result = a + b;
            var btn = document.querySelector('button');
            console.log(result);
```

```
        //设置延迟定时器
        var timer = setTimeout(function(){
            //第一个参数代表到达时间后的回调函数
            //第二个参数，代表延迟的时间，单位是毫秒
            console.log(result);
        },5000);

        //清除延迟定时器
        btn.onclick = function(){
            clearTimeout(timer);
        }
    }
</script>
</body>
```

14.1.2. 多次定时器（循环定时器）

1. 和循环类似都是为了重复去做一件事　　闹钟
2. 案例　每隔 3 秒打印 i love you!

```
window.onload = function(){
    //每隔 3 秒打印一次 i love you ，打印 10 次停止
    //循环定时器，循环定时去做重复的事情
    //设置循环定时器
    var n = 0;
    var timer = setInterval(function(){
```

```
        console.log('i love you~');
        n++;
        if(n == 10){
            clearInterval(timer);
        }
    },3000);
}
```

14.1.3. 清除定时器：

1. 清除单次定时器
2. 清除多次定时器
3. 定时器的返回值要使用全局变量保存

14.1.4. 强化案例：

1. 年历

```
<body>
    <span></span>
    <script type="text/javascript">
        window.onload = function(){
            //每隔 1 秒，取一次时间，并且讲这个时间字符串放入 span 标签
            var spanNode = document.querySelector('span');
            function getDateTimeString(){
                var date = new Date();

                var year = date.getFullYear();
```

```
            var month = date.getMonth() + 1;
            var day = date.getDate();
            var time = date.toLocaleTimeString();

            return '现在是' + year + '年' + month + '月' + day + '日 ' + time;
        }
        setInterval(function(){
            var result = getDateTimeString();
            spanNode.innerHTML = result;
        },1000);
    }
</script>
</body>
```

2. 阅读协议

```
<body>
    <input type="button" value="阅读协议(5s)" disabled="disabled"/>
    <script type="text/javascript">
        window.onload = function(){
            var n = 5;
            var inputNode = document.querySelector('input');
            var timer = setInterval(function(){
                n--;
                if(n == 0){
                    clearInterval(timer);
                    inputNode.value = '确认';
```

```
                inputNode.disabled = false;
            }else{
                inputNode.value = '阅读协议('+ n +'s)';
            }
        },1000);
    }
</script>
</body>
```

14.1.5. 同步和异步：

1. 所有的定时器还有事件的回调函数都是异步操作
2. 对于我们的代码，以后我们可以认为代码分为同步代码和异步代码
3. 异步代码是要等同步代码执行完成之后才会执行的（js 是单线程的）

```
<script type="text/javascript">
    //js 是单线程的
    //回调函数的代码都是异步代码最后才按照先来后到执行
    var a = 0;

    setTimeout(function(){
        console.log('i love you~');
    },5000);

    setTimeout(function(){
        console.log('i love you1~');
    },3000);
```

```
    //以后我们见不到这样的，在页面上有一个很耗时的同步任务
    for(var i = 0; i < 50000; i++){
        for(var j = 0; j < 50000; j++){
            a++;
        }
    }
</script>
```

14.2. 元素的大小和位置

14.2.1. Client 系列　　只读

1. **clientWidth**

   拿的是盒子　内容 + padding 的宽；

2. **clientHeight**

   拿的是盒子　内容 + padding 的高；

3. clientLeft

   拿的是盒子左边框大小；

4. clientTop

   拿的是盒子上边框大小；

14.2.2. Offset 系列　　只读

1. **offsetWidth**

   拿的是盒子　内容 + padding + border 的宽

2. **offsetHeight**

   拿的是盒子　内容 + padding + border 的高

3. **offsetLeft**

   拿的是元素的偏移量：可以认为就是拿的绝对定位 left 值

4. **offsetTop**

   拿的是元素的偏移量：可以认为就是拿的绝对定位 top 值

14.2.3. Scroll 系列

1. scrollWidth　只读
   (1) 当内容比盒子小的时候，拿的是盒子的 clientWidth
   (2) 当内容比盒子大的时候，拿的是内容的 offsetWidth +一侧外边距+盒子的一侧内边距；
2. scrollHeight　只读
   (1) 当内容比盒子小的时候，拿的是盒子的 clientHeight
   (2) 当内容 p 比盒子大的时候，拿的是内容的 offsetHeight +一侧外边距+盒子的一侧内边距；
3. scrollTop　可读可写

   拿的是盒子内容向上滚动的距离
4. scrollLeft　可读可写

   拿的是盒子内容向左滚动的距离

14.2.4. 总结

1. 元素的大小：宽和高的获取：
   (1) 以后我们拿元素的宽和高先看元素有没有边框
   (2) 如果没有边框那么 clientWidth 和 offsetWidth 是一样的
   (3) 如果有边框，看你需要不，需要的话就用 offsetWidth 不需要就用

clientWidth

2. 元素的位置（偏移量）的获取

(1) 获取元素的位置直接通过 offsetLeft 和 offsetTop 去获取

(2) 注意相对的参照元素是谁（和绝对定位参照类似）

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;

        }

        #box1{
            position: absolute;
            width: 500px;
            height: 500px;
            margin: 50px;
            padding: 50px;
            border: 10px solid blue;
            background-color: red;
        }

        #box2{
            width: 300px;
            height: 300px;
```

```
        margin: 50px;
        padding: 50px;
        border: 10px solid black;
        background-color: green;
        overflow: scroll;
    }

    #box3{
        width: 400px;
        height: 400px;
        margin: 50px;
        padding: 50px;
        border: 10px solid deeppink;
        background-color: blueviolet;
    }
</style>
</head>
<body>
<div id="box1">
    <div id="box2">
        <div id="box3"></div>
    </div>
</div>
<button>获取</button>
<script type="text/javascript">
    window.onload = function(){
        var box2 = document.getElementById('box2');
```

```
        //获取大小及位置    三个系列
        console.log(box2.clientWidth);
        console.log(box2.offsetWidth);
        console.log(box2.scrollWidth);

        console.log(box2.clientLeft);
        console.log(box2.offsetLeft);
        document.querySelector('button').onclick = function(){
            console.log(box2.scrollLeft);//内容向左边滚动的距离
        }
    }
</script>
</body>
```

14.2.5. 视口宽高求法（固定的）

1. document.documentElement.clientWidth 获取视口宽
2. document.documentElement.clientHeight 获取视口高

14.2.6. 案例

1. 导航跟随

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
```

```
        *{
            margin: 0;
            padding: 0;
        }
        #box{
            width: 100%;
            height: 60px;
            background-color: red;
        }

        body{
            height: 3000px;
        }

    </style>
</head>
<body>
    <div id="box"></div>
    <script type="text/javascript">
        window.onload = function(){
            var box = document.getElementById('box');
            window.onscroll = function(){
                //1、得拿到视口的高度
                var H = document.documentElement.clientHeight
                //2、拿到内容滚动的距离
                //获取网页内容滚动的距离，有些浏览器认为是 body 的内容，有些浏览器认为是 html 的内容
```

```
            //如果是 body 的内容，那么 html 获取的内容滚动距离就是 0
            var scrollT = document.body.scrollTop ||
document.documentElement.scrollTop;
            if(scrollT >= H){
                box.style.position = 'fixed';
                box.style.left = 0;
                box.style.top = 0;
            }else{
                box.style.position = 'static';//定位默认值就是 static，不写定位其
实就是它
                box.style.left = 0;
                box.style.top = 0;
            }
        }
    }
</script>
</body>
```

2. 盒子来回移动

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
```

```
        }
        #box{
            position: absolute;
            left: 0;
            top: 0;
            width: 100px;
            height: 100px;
            background-color: red;
        }
    </style>
</head>
<body>
    <div id="box"></div>
    <script type="text/javascript">
        window.onload = function(){
            var box = document.getElementById('box');
            var step = 3;
            setInterval(function(){
                var startX = box.offsetLeft;
                var lastX = startX + step;

                if(lastX > document.documentElement.clientWidth - box.offsetWidth){
                    lastX = document.documentElement.clientWidth - box.offsetWidth;
                    step = -3;
                }else if(lastX < 0){
                    lastX = 0;
                    step = 3;
```

```
            }
            box.style.left = lastX + 'px';
        },16);
    }
</script>
</body>
```

14.3. 初始包含块及系统滚动条的控制

14.3.1. 初始包含块

1. 根元素的包含块（也称为初始包含块）由用户代理建立
2. 在 HTML 中，根元素就是 html 元素，有些浏览器会使用 body 作为根元素
3. 在大多数浏览器中，**初始包含块是一个视窗大小的矩形**

14.3.2. 系统滚动条的控制

1. html 和 body 这两个元素 overflow 的 scroll 属性，控制着系统的滚动条
2. 系统的滚动条有两个，一个是 body 身上的 一个是 document 身上的
   (1) body 或者 html 单独设置 overflow:scroll 滚动条打开的全部都是 document 的
   (2) 如果两个元素同时设置 overflow:hidden；那么两个滚动条全部被关闭；
   (3) 如果两个都设置 overflow:scroll,那么 html 会打开 document 身上的，而 body 会打开自己身上的滚动条；
   (4) html 和 body 同时设置 overflow 属性
      ① body 设置的是 scroll,html 设置是 hidden,那么 document 的滚动条被

关闭，body 身上的滚动条会打开。

② 相反，body 身上被关闭，document 身上的被打开。

14.3.3. 禁止系统滚动条

1. 设置方式

```
html,body{
    height:100%;
    overflow:hidden;
}
```

2. 为什么要加 height:100%

这个属性加上只是为了让设置更有说服力，只有内容超出才会被掩藏或者出现滚动条，如果不设置，那么 body 和 html 高度将由内容自动撑开，也就是说 body 当中的内容永远不会溢出。

15. 第 15 章：案例拖拽

15.1. 拖拽的原理

1. **元素最终的位置 = 元素的初始位置 + 鼠标的距离差**
2. 注意： 两个方向都要去照顾；

15.2. 在基础的拖拽事件上添加边界问题

1. 当元素在四周的时候，不能超出范围，做出范围界定；
2. 当元素距离四周边界 50px 时候立即吸附到边界（吸附效果）

15.3. 元素碰撞问题（九宫格）

计算元素到视口上方和左边的距离 getBoundingClientRect() 只能读不能写

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }
        #box{
            position: absolute;
            left: 0;
            top: 0;
            width: 150px;
            height: 100px;
            background-color: red;
        }
        img{
            position: absolute;
            left: 0;
            top: 0;
            bottom: 0;
            right: 0;
            margin: auto;
            width: 200px;
```

```
        height: 100px;
    }

</style>
</head>
<body>
<div id="box">
    尚硅谷
</div>

<img src="img/17.jpg" alt="" />

<script type="text/javascript">
    window.onload = function(){
        var box = document.getElementById('box')
        var imgNode = document.querySelector('img');
        box.onmousedown = function(event){
            event = event || window.event;
            //拿到元素的初始位置
            var eleX = box.offsetLeft;
            var eleY = box.offsetTop;
            //拿到鼠标按下的初始位置
            var startX = event.clientX;
            var startY = event.clientY;
            box.setCapture && box.setCapture();//对低版本浏览器设置全局捕获
            document.onmousemove = function(event){
                event = event || window.event;
```

```
                //获取鼠标移动后的位置，鼠标的结束位置
                var endX = event.clientX;
                var endY = event.clientY;
                //元素在鼠标动的时候，就跟着动，证明元素最终的位置是在移动当中求出来并且设置过去的；
                //元素的最终位置 = 元素的初始位置 + 鼠标的距离差
                //获取鼠标的距离差
                var disX = endX - startX;
                var disY = endY - startY;
                //求出元素的最终位置
                var lastX = eleX + disX;
                var lastY = eleY + disY;

                //判定临界值(吸附效果)
                if(lastX > document.documentElement.clientWidth - box.offsetWidth - 50){
                    lastX = document.documentElement.clientWidth - box.offsetWidth;
                }else if(lastX < 50){
                    lastX = 0;
                }

                if(lastY > document.documentElement.clientHeight - box.offsetHeight - 50){
                    lastY = document.documentElement.clientHeight - box.offsetHeight;
                }else if(lastY < 50){
```

```
                lastY = 0;
            }

            //设置给元素最终位置
            box.style.left = lastX + 'px';
            box.style.top = lastY + 'px';

            //元素设置之后，元素才有可能碰到中间的盒子
            var boxL = lastX + box.offsetWidth;
            var imgL = imgNode.getBoundingClientRect().left;//拿元素距离视口的位置，返回是一个对象（只能读不能写）
            var boxT = lastY + box.offsetHeight;
            var imgT = imgNode.getBoundingClientRect().top;
            var boxR = lastX;
            var imgR = imgNode.getBoundingClientRect().left + imgNode.offsetWidth;
            var boxB = lastY;
            var imgB = imgNode.getBoundingClientRect().top + imgNode.offsetHeight;

            if(boxL < imgL || boxT < imgT || boxR > imgR  || boxB > imgB){
                //代表碰不上
                imgNode.src = 'img/17.jpg';
            }else{
                //碰上
                imgNode.src = 'img/3.jpg';
            }
```

```
            };

            document.onmouseup = function(){
                //当鼠标抬起的时候，把 move 事件解绑，否则 pc 端 move 事件是不会自动消失的;
                document.onmousemove = document.onmouseup = null;
                box.releaseCapture && box.releaseCapture();
                //对低版本浏览器释放全局捕获，全局捕获有设置就有释放
            }

            return false;
        }
    }
    //浏览器默认行为按照 dom0 和 dom2 事件区分
    //如果是 dom0 事件，只要在事件的最后加上 return false 就可以
    //如果是 dom2 事件，需要使用 event.preventDefault()
</script>
</body>
```

15.4. 自定义滚动条

滑块的高度 / 滑槽的高度 = 滑槽的高度 / 内容的高度 = 滑块的滚动距离 / 内容的滚动距离

15.5. 鼠标滚轮事件

1. IE/chrome 的绑定方式

   mousewheel(dom2 的标准模式)

   event.wheelDelta

   上：120

   下：-120

2. fireFox 的绑定方式

   DOMMouseScroll(dom2 的标准模式)

   event.detail

   上：-3

   下：3

15.6. 为自定义滚动条添加滚轮事件

```
<head>
    <meta charset="UTF-8">
    <title></title>
    <style>
        *{
            margin: 0;
            padding: 0;
        }

        a{
            text-decoration: none;
        }
```

```
ul,li{
    list-style: none;
}

input{
    outline: none;
}

img{
    display: block;
}

.clearFix:after{
    content: '';
    display: block;
    clear: both;
}

html,body{
    height: 100%;
    overflow: hidden;
}

#wrap{
    position: relative;
    width: 100%;
    height: 100%;
```

```
        overflow: hidden;
        background-color: skyblue;
    }

    #wrap .content{
        position: absolute;
        left: 0;
        top: 0;
    }

    #wrap .scrollBar{
        position: absolute;
        right: 0;
        top: 0;
        width: 30px;
        height: 100%;
        border-left: 1px solid black;
        border-right: 1px solid black;
        background-color: hotpink;
    }

    #wrap .scrollBar .scrollIn{
        position: absolute;
        left: 50%;
        transform: translateX(-50%);
        top: 0;
        width: 26px;
```

```
        /*height: 100px;*/
        background-color: greenyellow;
        /*margin: 0 auto;*/
    }
</style>
</head>
<body>
<div id="wrap">

    <div class="content">

    </div>

    <div class="scrollBar">
        <div class="scrollIn"></div>
    </div>

</div>
<script type="text/javascript">
    window.onload = function(){
        var scrollIn = document.querySelector('#wrap .scrollBar .scrollIn');
        //模拟内容区域，内容要比 wrap 的高度要高
        var content = document.querySelector('#wrap .content');

        for(var i = 1; i < 200; i++){
            content.innerHTML += i + '<br>';
        }
```

```
        //动态设置滑块的高度
//          自定义滚动条的万能比例：
//          滑块的高度 / 滑槽的高度 = 滑槽的高度 / 内容的高度 = 滑块的滚动距离
/ 内容的滚动距离
        var scale = document.documentElement.clientHeight /
content.offsetHeight;
        scrollIn.style.height = document.documentElement.clientHeight * scale +
'px';
        //滑块滚动逻辑和内容的滚动逻辑
        scrollIn.onmousedown = function(event){
            event = event || window.event;
            var eleY = scrollIn.offsetTop;
            var startY = event.clientY;
            scrollIn.setCapture && scrollIn.setCapture();//对低版本浏览器设置全
局捕获
            document.onmousemove = function(event){
                event = event || window.event;
                var endY = event.clientY;
                var disY = endY - startY;
                var lastY = eleY + disY;

                //把滑块滑动的临界值加上
                if(lastY > document.documentElement.clientHeight -
scrollIn.offsetHeight){
                    lastY = document.documentElement.clientHeight -
scrollIn.offsetHeight;
```

```
            }else if(lastY < 0){
                lastY = 0;
            }

            scrollIn.style.top = lastY + 'px';//代表滑块滚动

            //让内容也跟着滚动
            var contentDis = -lastY / scale;
            content.style.top = contentDis + 'px';

        };
        document.onmouseup = function(){
            document.onmousemove = document.onmouseup = null;
            scrollIn.releaseCapture && scrollIn.releaseCapture();//对低版本浏览器释放全局捕获，全局捕获有设置就有释放，否则后果自负
        }
        return false;
    }

    //滚轮事件的逻辑
    //添加滚轮事件，两个都得添加，因为我们也不清楚用户使用的是什么浏览器
    //ie/chrome
    document.addEventListener('mousewheel',scrollMove);//假设我们填的事件和浏览器不对应，不会报错，只是没效果

    //firefox
    document.addEventListener('DOMMouseScroll',scrollMove);
```

```
//滚轮事件的回调函数，兼容 ie/chrome/firefox
var step = 0;
function scrollMove(event){
    event = event || window.event;
    //判断浏览器是什么浏览器
    if(event.wheelDelta){
        //ie/chrome
        if(event.wheelDelta > 0){
            //上
            step = -10;
        }else{
            //下
            step = 10;
        }
    }else if(event.detail){
        //firefox
        if(event.detail > 0){
            //下
            step = 10;
        }else{
            //上
            step = -10;
        }
    }
    //在此，我要拿到的最终结果是往上还是往下，我不关心是通过什么浏览器去滚动的
```

```
            var scrollDis = scrollIn.offsetTop + step;
            if(scrollDis > document.documentElement.clientHeight -
scrollIn.offsetHeight){
                scrollDis = document.documentElement.clientHeight -
scrollIn.offsetHeight;
            }else if(scrollDis < 0){
                scrollDis = 0;
            }
            scrollIn.style.top =  scrollDis + 'px'
            //内容的
            var contentDis = -scrollDis / scale;
            content.style.top = contentDis + 'px';
        }
    }
</script>
</body>
```

16. 第 16 章：轮播图逻辑及正则表达式

16.1. 轮播图逻辑

1. 点击按钮让图片先动起来
2. 当刚好元素走的位置和开始求出来的结束位置一样的时候，停止定时器；

```
<head>
    <meta charset="UTF-8">
    <title></title>
```

```
<style>
    *{
        margin: 0;
        padding: 0;
    }
    ul,li{
        list-style: none;
    }
    img{
        display: block;
    }

    #box{
        position: relative;
        width: 600px;
        height: 300px;
        margin: 50px auto;
        overflow: hidden;
    }

    #box .list{
        position: absolute;
        left: 0;
        top: 0;
        width: 3000px;
        height: 300px;
    }
```

```
#box .list li{
    float: left;
    width: 600px;
    height: 300px;
}

#box .list li img{
    width: 600px;
    height: 300px;
}

#box .iconList{
    position: absolute;/*它是可以让我们去达到清楚浮动的效果，因为开启了BFC*/
    /*overflow: hidden;/*它是可以让我们去达到清楚浮动的效果，因为开启了BFC*/
    left: 50%;
    bottom: 5px;
    transform: translateX(-50%);
}

#box .iconList li{
    float: left;
    width: 40px;
    height: 40px;
```

```
        margin-right: 10px;
        border-radius: 50%;
        background-color: grey;
    }

    #box .iconList li.current{
        background-color: red;
    }



    #box span{
        position: absolute;
        top: 50%;
        transform: translateY(-50%);
        width: 50px;
        height: 100px;
        background-color: rgba(100,100,100,.7);
        font-size: 40px;
        text-align: center;
        line-height: 100px;
        color: #ffffff;
        opacity: 0;
        transition: opacity 2s;
    }
```

```
        #box .left{
            left: 0;
        }

        #box .right{
            right: 0;
        }

    </style>
</head>
<body>
    <div id="box">
        <!--轮播图主体结构-->
        <ul class="list">
            <li><img src="img/17.jpg"/></li>
            <li><img src="img/22.jpg"/></li>
            <li><img src="img/25.jpg"/></li>
            <li><img src="img/3.jpg"/></li>
            <li><img src="img/9.jpg"/></li>
        </ul>

        <!--轮播图小圆点结构-->
        <ul class="iconList">
            <li class="current"></li>
            <li></li>
            <li></li>
            <li></li>
```

```
        <li></li>
    </ul>

    <span class="left">  <  </span>
    <span class="right">  >  </span>
</div>

<script type="text/javascript">
    window.onload = function(){
        var box = document.getElementById('box');
        var spanNodes = document.querySelectorAll('#box span');
        var ulNode = document.querySelector('#box .list');
        var timeAll = 600;//点击一次按钮，图片切换所需要的时间，我们设置为 600ms
        var perStepTime = 20;//设置每走一步所花费的时间
        var timer = null;

        box.onmouseenter = function(){
            spanNodes[0].style.opacity = 1;
            spanNodes[1].style.opacity = 1;
        };

        box.onmouseleave = function(){
            spanNodes[0].style.opacity = 0;
            spanNodes[1].style.opacity = 0;
        };

        //点击右侧按钮逻辑
```

```
        spanNodes[1].onclick = function(){
            //拿到 ul 当前的位置
            var startX = ulNode.offsetLeft;
            //拿到每次点击元素要移动的距离
            var perClickDis = -600;
            //求出每次点击元素最终的位置
            var lastX = startX + perClickDis;
            //把最终的位置设置给元素  如果直接把最终位置设置给元素,那么元素就会顺变到第二个 li，就成顺变轮播图
//              ulNode.style.left = lastX + 'px';

            //如果我们需要，慢慢走，那么-600px 不能一下子走过去，需要循环定时器
            //求出每一步走的距离
            var step = perClickDis / (timeAll / perStepTime);//每一步的距离
            timer = setInterval(function(){
                var left = ulNode.offsetLeft + step;//计算出每一步走完，元素的位置

                if(left == lastX){//如果元素一步一步 刚好和上面求出来的元素最终位置相同，定时器要清除
                    clearInterval(timer);
                }
                ulNode.style.left = left + 'px';

            },perStepTime);

        }
```

```
        //点击左侧按钮
        spanNodes[0].onclick = function(){
            //拿到 ul 当前的位置
            var startX = ulNode.offsetLeft;
            //拿到每次点击元素要移动的距离
            var perClickDis = 600;
            //求出每次点击元素最终的位置
            var lastX = startX + perClickDis;
            //把最终的位置设置给元素  如果直接把最终位置设置给元素，那么元素就会顺变到第二个 li，就成顺变轮播图
//              ulNode.style.left = lastX + 'px';

            //如果我们需要，慢慢走，那么-600px 不能一下子走过去，需要循环定时器
            //求出每一步走的距离
            var step = perClickDis / (timeAll / perStepTime);//每一步的距离
            timer = setInterval(function(){
                var left = ulNode.offsetLeft + step;//计算出每一步走完，元素的位置

                if(left == lastX){//如果元素一步一步 刚好和上面求出来的元素最终位置相同，定时器要清除
                    clearInterval(timer);
                }
                ulNode.style.left = left + 'px';
            },perStepTime);
        }
    }
</script>
```

```
</body>
```

3. 左边按钮和右边按钮一样，因此封装函数

4. 函数传参为 true 就是点右　false 就是点左

```
function move(flag){
    //flag 目前是为了确定我们点击的是左侧还是右侧按钮，传递 true 就代表是点的右侧，
false 代表点的是左侧；
    if(flag){
        //拿到每次点击元素要移动的距离
        var perClickDis = -600;
    }else{
        var perClickDis = 600;
    }
    //拿到 ul 当前的位置
    var startX = ulNode.offsetLeft;
    //求出每次点击元素最终的位置
    var lastX = startX + perClickDis;
    //把最终的位置设置给元素　如果直接把最终位置设置给元素，那么元素就会顺变到第
二个 li，就成顺变轮播图
    //如果我们需要，慢慢走，那么-600px 不能一下子走过去，需要循环定时器
    //求出每一步走的距离
    var step = perClickDis / (timeAll / perStepTime);//每一步的距离
    timer = setInterval(function(){
        var left = ulNode.offsetLeft + step;//计算出每一步走完，元素的位置

        if(left == lastX){//如果元素一步一步 刚好和上面求出来的元素最终位置相同，
```

```
定时器要清除
            clearInterval(timer);
        }
        ulNode.style.left = left + 'px';
    },perStepTime);
}
```

5. 函数内部添加无缝的操作

```
function move(flag){
    //flag 目前是为了确定我们点击的是左侧还是右侧按钮，传递 true 就代表是点的右侧，
false 代表点的是左侧；
    if(flag){
        //拿到每次点击元素要移动的距离
        var perClickDis = -600;
    }else{
        var perClickDis = 600;
    }

    //拿到 ul 当前的位置
    var startX = ulNode.offsetLeft;
    //求出每次点击元素最终的位置
    var lastX = startX + perClickDis;
    //把最终的位置设置给元素  如果直接把最终位置设置给元素，那么元素就会顺变到第
二个 li，就成顺变轮播图
//              ulNode.style.left = lastX + 'px';

    //如果我们需要，慢慢走，那么-600px 不能一下子走过去，需要循环定时器
```

```
    //求出每一步走的距离
    var step = perClickDis / (timeAll / perStepTime);//每一步的距离


    timer = setInterval(function(){
        var left = ulNode.offsetLeft + step;//计算出每一步走完，元素的位置
        if(left == lastX){//如果元素一步一步 刚好和上面求出来的元素最终位置相同，
定时器要清除
            clearInterval(timer);
            //点完一张，刚好停下来的时候，添加无缝逻辑
            if(left == -3600){
                left = -600;
                //右侧无缝逻辑
            }else if(left == 0){
                left = -3000;
                //左侧无缝逻辑
            }
        }
        ulNode.style.left = left + 'px';
    },perStepTime);
}
```

6. 小圆点变色在函数内部最后添加

```
//小圆点变色  （排他）
//1、让所有的小圆点全部变灰
for(var i = 0; i < iconNodes.length; i++){
    iconNodes[i].className = '';
```

```
}


//2、让相关的那个小圆点变红  （让准备显式的那张图片对应的小圆点变红）
//关键是求出对应的这个小圆点的下标，就可以变红
//要求小圆点的下标，小圆点没法直接拿下标是多少，我们得根据显式的图片的下标去求小圆点下标，他们有关系
//最终要求，显示的这个图片它的下标，和显示这个图片的时候，元素的位置相关
var index = (-lastX / 600) - 1; //图片的下标 - 1  刚好就是我们要的小圆点下标（前后两张需要判断）
if(index > 4){
    index = 0;
}else if(index < 0){
    index = 4;
}
iconNodes[index].className = 'current';
```

7. 点击小圆点在函数外部添加

```
for(var i = 0; i < iconNodes.length; i++){
    iconNodes[i].index = i;
    iconNodes[i].onclick = function(){
        move((this.index + 1)*-600);
        //需要把点击的这个小圆点元素所处的最终位置，传递到函数当中，
        // 用来求出点击小圆点元素需要走的距离
        //这个距离就不是点击按钮走的那个 600 了，得去计算
    }
}
```

8. 连续点击按钮或者小圆点定时器叠加问题解决（函数最开始添加）

```
if(isMove){
    //在第二次进来的时候，先判断 flag，看看图片是不是在切换中
    return;
}
isMove = true;//代表目前图片已经开始切换
setTimeout(function(){
    isMove = false;
},600)
```

9. 在函数外部添加自动轮播

```
autoRun();
function autoRun(){//以后要重启定时器，就把定时器封装为函数，哪里需要在哪里调用就重新启动了
    autoTimer = setInterval(function(){
        move(true);//自动轮播其实就是模拟点击右侧按钮的逻辑
    },2000);
}
```

10. 移入之后，停止自动轮播，移出再次自动轮播

```
box.onmouseenter = function(){
    spanNodes[0].style.opacity = 1;
    spanNodes[1].style.opacity = 1;
    clearInterval(autoTimer);
};
```

```
box.onmouseleave = function(){
    spanNodes[0].style.opacity = 0;
    spanNodes[1].style.opacity = 0;
    autoRun();//本质上相当于重新设置了一遍定时器;
};
```

16.2. 正则表达式

16.2.1. 正则的概念

1. 是什么？
   (1) 正则表达式是描述字符模式的对象。
   (2) 正则表达式用于对字符串模式匹配及检索替换，是对字符串执行模式匹配的强大工具。
   (3) 正则表达式是一种字符串匹配规则；正则就是让我们用来在一个字符串当中去查找符合正则规则的字符串或者判断字符串是否符合正则规则；
2. 为什么？
   (1) 假设我想要知道一个字符串当中是否有 6，该如何去做
   (2) 假设我想知道字符串当中是否有数字又该如何
   (3) 假设我想从字符串当中找到 abcd 怎么去做
   (4) 假设我想知道这个电话号码是否合法
3. 创建方式
   (1) 字面量创建

   var patt=/pattern/modifiers;

(2) 构造函数创建

var patt=new RegExp(pattern,modifiers);

(3) pattern（模式） 描述了表达式的模式

(4) modifiers(修饰符) 用于指定全局匹配、区分大小写的匹配和多行匹配

(5) 注意：当使用构造函数创造正则对象时，需要常规的字符转义规则（在前面加反斜杠 \）

16.2.2. 正则表达式规则写法

1. 修饰符

(1) 修饰符用于执行区分大小写和全局匹配:

(2) i:忽略大小写

(3) g: 执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。

(4) m:执行多行匹配

2. 方括号

(1) 方括号用于查找某个范围内的字符：

(2) [abc] 查找 abc 任意一个

(3) [^abc] 查找不是 abc 的任意一个

(4) [0-9] 查找任意一个数字 \d

(5) [a-z] 查找任意一个小写字母

(6) [A-Z] 查找任意一个大写字母

3. 元字符

(1) . 匹配任意字符不包含\n（换行和结束符）

(2) \d 任意数字 等价于[0-9]

(3) \D 任意非数字 等价于[^0-9]

(4) \w 任意单词字符 数字 字母 下划线 [a-z A-Z 0-9 _]

(5) \W 任意非单词字符 [^a-z A-Z 0-9 _]

(6) \s 任意空白字符

(7) \S 任意非空白字符

(8) $ 结尾

(9) ^ 开头

(10) \b 单词边界

(11) \B 非单词边界

(12) \n 换行符

(13) \f 换页符

(14) \r 回车符

(15) \t 制表符

(16) \v 垂直制表符

4. 量词

(1) + 1 个或者多个前一个字符 \d+

(2) * 0 个或者多个前一个字符 \d*

(3) ？ 0 个或者 1 个前一个字符 \d?

(4) {n} n 个前一个字符 \d{2} (\d{11})\1;

(5) {m,n} m 到 n 个前一个字符 \d{2,4}

(6) {m,} 至少 m 个前一 个字符 \d{2,}

5. 分组 （）分组后的反向引用

6. 量词后面的？代表非贪婪

16.2.3. 正则相关方法

1. 正则对象的方法：

(1) Reg.test()

① test() 方法用于检测一个字符串是否匹配某个模式

② 如果字符串中含有匹配的文本，则返回 true，否则返回 false

(2) Reg.exec()

① exec() 方法用于检索字符串中的正则表达式的匹配。

② 该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。

③ 注意：此方法每次只会返回一个结果，如果要找到所有的，需要循环去调用而且必须全局匹配修饰；

2. 字符串方法使用正则

(1) str.match()

① match()在字符串中搜索符合规则的内容

② 搜索成功就返回内容，格式为数组，失败就返回 null。

③ 如果正则不加 g，那么返回第一次符合的结果，加 g 返回所有结果

④ 找一个是详细进行展示

⑤ 找多个是在数组中展示找到的所有内容子串，不详细

(2) str.search()

① search()在字符串搜索符合正则的内容

② 搜索到就返回出现的位置， 如果搜索失败就返回 -1

③ 只能返回第一次

(3) str.replace()

① replace()查找符合正则的字符串，就替换成对应的字符串。

② 返回替换后的内容。

16.2.4. 案例:

1. 给一个字符串，如果里面含有 2 个到 10 个连续数字就打印 i love you!
2. 判断手机号码是否合法
3. 判断邮箱是否合法
4. 将手机号的中间 4 位替换位****
5. 将身份中号的最后 4 位替换位****
6. 从一篇文章当中匹配所有的电话号码

JavaScript 高级

17. 第 17 章：重点总结及执行上下文

17.1. 内存

1. 计算机：cpu（寄存器） 内存 硬盘
   (1) 硬件内存条：
   (2) 存储 1 或者 0（通电和不通电）
   (3) 内存地址（内存编号）
   (4) 内存的东西断电消失
   (5) 硬盘的东西永久性存储
2. 内存结构
   (1) 栈内存和堆内存 （c 语言、java）
   (2) 但是在 js 当中所有的内存都属于堆内存
   (3) 堆内存内部又分为栈结构和堆结构

17.2. 变量

1. 程序和进程概念
2. 变量：可以变化的量：就是存储数据的容器，本质上是内存空间；
3. 属性的赋值优先级比变量高（连等的情况下）

17.3. 数据

17.3.1. 分类(2 大类)

1. 基本(值)类型
   (1) Number: 任意数值
   (2) String: 任意文本
   (3) Boolean: true/false
   (4) undefined: undefined
   (5) null: null
2. 对象(引用)类型
   (1) Object: 任意对象
   (2) Array: 一种特别的对象类型(下标/内部数据有序)
   (3) Function: 一种特别的对象类型(可执行)
   (4) 备注：准确来说 Array 和 Function 属于一种特别的 Object 类型。

17.3.2. 数据存储

1. 基本数据存储　　数据本身
2. 对象数据存储　　地址值
3. 备注：内存栈结构当中存储的都是值，一个存的是数据本身，另一个存的是数据所在的内存（堆结构）地址；

17.3.3. 鉴别数据类型

1. typeof
   (1) typeof 的返回值是类型的字符串名称（首字母是小写的）
   (2) 可以区别: 数值, 字符串, 布尔值, undefined, function

(3) 不能区别: null 与数组、对象

(4) 一个不正确：typeof null 的结果是 object

(5) 一个不精确：typeof []的结果是 object

2. instanceof:

(1) A instanceof B ==> 判断 A 是否是 B 这个构造函数的实例对象

(2) 专门用来判断对象数据的具体类型: Object, Array

3. ===

(1) 可以判断: undefined 和 null

(2) 备注：因为 undefined 类型只有一个值、null 类型也只有一个值，所以可以用===判断

17.3.4. 所有数据类型转化布尔值：

1. 基本数据类型：

(1) ‘’ 0 NaN undefined null false

(2) 其余全是 true

2. 对象数据类型：全是 true

3. 备注：js 当中只有 6 个 false 值，其余全是 true；

17.3.5. 所有的数据类型进行判等 ==

1. 基本数据类型

(1) 如果是同种的基本数据类型，那么就直接比较两个值是否一样

(2) 如果是不同的基本数据类型，那么两个值都要去转数字；

(3) 不同的基本类型当出现 null 会出现特殊情况

① 0 和 null 不等

② ''和 null 不等

③ false 和 null 不等

④ undefined 和 null 相等

2. 对象数据类型

(1) 两边都是对象类型判等，判的是地址，如果引用（地址）相同就相等

(2) 其余情况下对象数据类型转基本数据然后再进行

① 对象和对象的比较及运算，对象要转化基本数据

② 对象和基本数据的比较、判等及运算，对象要转化基本数据

(3) 对象数据转基本数据规则

① 首先看对象能不能转化为一个基本值类型　看看对象能不能使用 valueOf 方法转化一个基本值类型,调用的全是 Object 的原型对象当中的 valueOf 方法；

② 所有的对象(非包装对象)在调用 valueOf 方法的时候都返回自身

③ 如果对象调用 valueof 方法返回的不是基本值类型，那么会接着调用 toString（各自是各自的）方法转化基本值；

17.3.6. undefined 与 null 的区别

1. undefined 代表定义变量没有赋值　　var a;
2. null 代表赋值了，只是值为 null　　var a = null;
3. 什么时候给变量赋值为 null 呢?

(1) 对象变量初始化

(2) 删除对象

17.3.7. 关于引用变量赋值问题

1. 2 个引用变量指向同一个对象，通过一个引用变量修改对象内部数据，另一个引用变量也看得见
2. 2 个引用变量指向同一个对象,让一个引用变量指向另一个对象，另一个引用变量还是指向原来的对象

17.3.8. 关于函数传参的问题

1. 传基本数据值
2. 传引用数据值

17.3.9. 关于内存释放（堆结构）的问题

垃圾回收机制：堆结构当中的对象数据，要想被回收释放，必须成为垃圾对象，也就是没有人再指向这个对象数据；

17.3.10. 面试题

```
//1
var a = {n:1};
var b = a;
//变量和属性同时连等赋值的时候，属性的优先级比变量要高
a.x = a = {n:2};
console.log(a.x);
console.log(b.x);
//2、
function add(a,b){
   return a + b;
}
console.log(add instanceof Object);
console.log(Object instanceof Object);
console.log(1 instanceof Object);
console.log(Array instanceof Object);
//3、
```

```
var b1 = {
    b2:[2,'atguigu',console.log],
    b3:function () {
        alert('hello')
    }
}
console.log(b1,b1.b2,b1.b3)
console.log(b1 instanceof Object,typeof b1)
console.log(b1.b2 instanceof Array,typeof b1.b2)
console.log(b1.b3 instanceof Function,typeof b1.b3)
console.log(typeof b1.b2[2]);
console.log(typeof b1.b2[2]('atguigu'))
```

17.4. 对象基础

17.4.1. 什么是对象?

1. 代表现实中的某个事物, 是该事物在编程中的抽象(无序的键值对的集合)
2. 多个数据的集合体(封装体)
3. 用于保存多个数据的容器

17.4.2. 为什么要用对象?

1. 便于对多个数据进行统一管理
2. 为了去描述某个复杂的事物

17.4.3. 对象的组成

1. 属性
   (1) 代表现实事物的状态数据
   (2) 由属性名和属性值组成
   (3) 属性名都是字符串类型，属性值是任意类型
2. 方法
   (1) 代表现实事物的行为数据
   (2) 备注：方法是一种特别的属性==>属性值是函数

17.4.4. 如何访问对象内部数据?

1. 对象.属性名: 编码简单，但有时不能用
2. 对象['属性名']: 编码麻烦，但通用
3. 问题: 什么时候必须使用[]的方式?
   (1) 属性名不是合法的标识符　要带引号
   (2) 属性名是变量的值　不能给变量带引号
      ① 对象[变量名]:变量里面的值会被转化成字符串，作为属性名

17.4.5. 面试题

```
var a = {};
var obj1 = {m:2}
var obj2 = {n:2}
var obj2 = [1,2,3];
var obj3 = function(){};
a[obj1] = 4
a[obj2] = 5
```

```
a.name ='kobe'
a[obj3] = 6;
console.log(a[obj1])
console.log(a)
```

17.5. 函数基础

17.5.1. 什么是函数?

1. 具有特定功能的 n 条语句的封装体
2. 只有函数是可执行的, 其它类型的数据是不可执行的
3. 函数也是对象

17.5.2. 为什么要用函数?

1. 提高代码复用
2. 便于阅读和交流
3. 把一个项目模块化

17.5.3. 如何定义函数?

1. 函数声明（字面量）
2. 函数表达式
3. 构造函数

17.5.4. 函数的 2 种角色

1. 函数: 通过()使用

2. 对象: 通过.使用 ==> 称之为: 函数对象

17.5.5. 如何调用(执行)函数?

1. test()
2. new test()
3. obj.test()
4. new obj.test()
5. test.call/apply(obj)

17.5.6. 普通函数和构造函数的区别

1. 本质都是函数
2. 调用方式不同
3. this 指向不同
4. 返回值不同

17.5.7. 回调函数

1. 什么函数才是回调函数?
   (1) 你定义的
   (2) 你没有直接调用
   (3) 但最终它执行了(在特定条件或时刻)
2. 常见的回调函数?
   (1) DOM 事件函数
   (2) 定时器函数
   (3) ajax 回调函数(后面学)
   (4) 生命周期回调函数(后面学)

17.5.8. IIFE

1. 理解
   (1) 全称: Immediately-Invoked Function Expression 立即调用函数表达式
   (2) 别名: 匿名函数自调用
2. 作用
   (1) 隐藏内部实现
   (2) 不污染外部命名空间
3. 特点;
   (1) 只能调用一次
   (2) 函数不会预解析（内部执行时候会预解析）
   (3) 定义的时候同时调用;

17.5.9. this 的总结

1. this 是什么?
   (1) 一个关键字, 一个内置的引用变量
   (2) 在函数中都可以直接使用 this
   (3) this 代表调用函数的当前对象（函数的调用者）
   (4) 在定义函数时, this 还没有确定, 只有在执行时才动态确定(绑定)的
2. 如何确定 this 的值?
   (1) test()
   (2) obj.test()
   (3) new test()
   (4) 事件的回调函数中指向事件源
   (5) test.call/apply(obj)
3. 总结: 函数的调用方式决定了 this 是谁

17.5.10. 函数的递归调用

1. 函数调用的时候内部又调用自身
2. 斐波那契数列（兔子数列）

17.6. 执行上下文

17.6.1. 什么是执行上下文

1. 程序在解析和运行的时候所依赖和使用的环境称作执行上下文；
2. 全局执行上下文 和 函数执行上下文

17.6.2. 什么是执行上下文栈：

程序为了管理执行上下文（确保程序的执行顺序）所创建的一个栈数据结构，被称作执行上下文栈；

17.6.3. 预解析（变量提升）

1. 先解析函数：函数重名覆盖
2. 再解析变量：变量重名忽略

17.6.4. 作用域

1. 变量起作用的范围；
2. 作用域用来隔离变量，防止变量命名污染；
3. 作用域定义时候确定

17.6.5. 作用域链：

1. 真实存在的，作用域链是使用执行上下文当中变量对象所组成的数组结构
2. 查找过程是先去自身的变量对象当中查找，如果没有，去上级执行上下文的变量对象当中去查找，直到找到全局执行上下文的变量对象
3. 注意：函数调用的时候上一级的变量对象其实是在函数定义的时候都已经确定好的

17.6.6. 执行上下文的阶段

1. 分为**创建阶段和执行阶段**，**代码**开始**执行之前和之后**
2. 创建上下文阶段：
   (1) **收集变量形成变量对象** （函数 var 的变量会收集）
      ① 预解析（其实在创建变量对象的时候已经做了预解析）
   (2) **确定 this 指向**（可以认为确定执行者）
   (3) **创建自身执行上下文的作用域链**
      ① 注意：同时确定函数在调用时候的上级作用域链。（根据 ECMA 词法去确定，看内部是否引用外部变量确定）
3. 执行上下文阶段
   (1) 为变量真正赋值
   (2) 顺着作用域链查找要使用的变量或者函数执行

17.6.7. 面试题

```
//1
var x = 10;
function fn() {
    console.log(x);
```

```
}
function show(f) {
    var x = 20;
    f();
}
show(fn);


//2
var a;
function a() {}
console.log(typeof a)


//3
if (!(b in window)) {
    var b = 1;
}
console.log(b)


//4
var c = 1;
function c(c) {
    console.log(c)
    var c = 3
}
c(2)


//5
```

```
var fn = function () {
    console.log(fn)
}
fn()
function fn(){console.log(fn)};
fn();

//6
var obj = {
    fn2: function () {

        console.log(fn2)
    }
}
obj.fn2()
```

18. 第 18 章：重点总结及闭包

18.1. 闭包

18.1.1. 如何产生闭包(条件)?

1. 函数嵌套
2. 内部函数引用外部函数的局部变量
3. 使用（调用）外部函数

18.1.2. 闭包到底是什么?

1. 理解一: 闭包是嵌套的内部函数(绝大部分人)
2. 理解二: 包含被引用变量(外部函数)的对象(极少数人)
3. 理解三： 所谓的闭包是一个引用关系，该引用关系存在于内部函数中，引用的是外部函数的变量的对象(深入理解)

18.1.3. 常见的闭包

1. 将函数作为另一个函数的返回值
2. 将函数作为实参传递给另一个函数调用
3. 使用闭包实现私有方法操作独立的私有属性

18.1.4. 闭包的作用

1. 延长外部函数变量对象的生命周期
2. 让函数外部可以操作(读写)到函数内部的数据(变量/函数)
3. 注意： 浏览器为了性能后期将外部函数中不被内部函数使用的变量清除了

18.1.5. 闭包的生命周期

1. 产生: 在嵌套内部函数定义完时就产生了(不是在调用)
2. 死亡: 在嵌套的内部函数成为垃圾对象时

18.1.6. 自定义模块

1. 具有特定功能的 js 文件
2. 将所有的数据和功能都封装在一个函数内部(私有的)

3. 只向外暴露一个包信 n 个方法的对象或函数
4. 模块的使用者，只需要通过模块暴露的对象调用方法来实现对应的功能

18.1.7. 闭包的缺点和解决

1. 内存泄漏 ： 内存无法释放
2. 内存溢出 ： 内存被撑爆
3. 解决方式 ： f = null

18.1.8. 面试题

```
//代码片段一
var name = "The Window";
var object = {
    name: "My Object",
    getNameFunc: function () {

        return function () {
            return this.name;
        };
    }
};
console.log(object.getNameFunc()());

//代码片段二
var name2 = "The Window";
var object2 = {
```

```
    name2: "My Object",
    getNameFunc: function () {
        var that = this;
        return function () {
            return that.name2;
        };
    }
};
console.log(object2.getNameFunc()());


//代码片段三
function fun(n, o) {
    console.log(o)
    return {
        fun: function (m) {
            return fun(m, n)
        }
    }
}
var a = fun(0)
a.fun(1)
a.fun(2)
a.fun(3)
var b = fun(0).fun(1).fun(2).fun(3)
var c = fun(0).fun(1)
c.fun(2)
c.fun(3)
```

```
//代码片段四
function Foo() {
    getName = function () { alert (1); };
    return this;
}
Foo.getName = function () { alert (2);};
Foo.prototype.getName = function () { alert (3);};
var getName = function () { alert (4);};
function getName() { alert (5);}

//请写出以下输出结果：
Foo.getName();
getName();
Foo().getName();
getName();
new Foo.getName();
new Foo().getName();
new new Foo().getName();
```

18.2. 原型和原型链、终极原型链

18.3. 面向对象

1. 面向对象三大特性：封装继承多态
2. 继承
   (1) 原型继承

(2) 借用构造函数继承

(3) 组合继承

3. 多态

(1) 方法重写

(2) 方法重载

18.4. 事件循环机制

18.4.1. 多进程和多线程

1. 进程：程序的一次执行，它占有一片独有的内存空间
2. 线程： CPU 的基本调度单位，是程序执行的一个完整流程
3. 进程与线程

(1) 一个进程中一般至少有一个运行的线程: 主线程

(2) 一个进程中也可以同时运行多个线程，我们会说程序是多线程运行的

(3) 一个进程内的数据可以供其中的多个线程直接共享

(4) 多个进程之间的数据是不能直接共享的

4. 浏览器运行是单进程还是多进程?

(1) 有的是单进程

① Firefox

② 老版 IE

(2) 有的是多进程

① Chrome

② 新版 IE

5. 浏览器运行是多线程
6. js 是单线程的

(1) 如何证明 js 执行是单线程的?

① setTimeout()的回调函数是在主线程执行的

② 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行

(2) 为什么 js 要用单线程模式，而不用多线程模式?

① JavaScript 的单线程，与它的用途有关。

② 作为浏览器脚本语言，JavaScript 的主要用途是与用户互动

③ 这决定了它只能是单线程，否则会带来很复杂的同步问题

18.4.2. js 引擎执行代码的基本流程

1. 先执行初始化代码:
   (1) 包含一些特别的代码
   (2) 设置定时器
   (3) 绑定监听
   (4) 发送 ajax 请求
2. 后面在某个时刻才会执行回调代码

18.4.3. 事件模型

1. 模型的两个重要组成部分
   (1) 事件管理模块
   (2) 回调队列
2. 模型的运转流程
   (1) 执行初始化代码，将事件回调函数交给对应模块管理
   (2) 当事件发生时，管理模块会将回调函数及其数据添加到回调列队中
   (3) 只有当初始化代码执行完后(可能要一定时间)，才会遍历读取回调队列中的回调函数执行

18.5. Web Workers 模拟多线程

1. H5 规范提供了 js 分线程的实现，取名为: Web Workers
2. 相关 API
   (1) Worker: 构造函数，加载分线程执行的 js 文件
   (2) Worker.prototype.onmessage: 用于接收另一个线程的回调函数
   (3) Worker.prototype.postMessage: 向另一个线程发送消息
   (4) 每个线程可以向不同线程发送消息也可以接收不同线程传来的消息
3. 不足
   (1) worker 内代码不能操作 DOM(更新 UI)
   (2) 不能跨域加载 JS
   (3) 不是每个浏览器都支持这个新特性
4. 计算得到 fibonacci 数列中第 n 个数的值
   (1) 在主线程计算: 当位数较大时，会阻塞主线程，导致界面卡死
   (2) 在分线程计算: 不会阻塞主线程