JavaScript基础

1 - 循环

1.1 for循环

- 语法结构

```
for(初始化变量; 条件表达式; 操作表达式 ){
    //循环体
}
```

```
// 循环的目的：可以重复执行某些代码
        console.log('我想你了');
        console.log('我想你了');
        console.log('我想你了');
        console.log('----------------------');
        for (var i = 1; i <= 1000; i++) {
            console.log('我想你了');
        }
```

名称	作用
初始化变量	通常被用于初始化一个计数器，该表达式可以使用 var 关键字声明新的变量，这个变量帮我们来记录次数。
条件表达式	用于确定每一次循环是否能被执行。如果结果是 true 就继续循环，否则退出循环。
操作表达式	用于确定每一次循环是否能被执行。如果结果是 true 就继续循环，否则退出循环。

执行过程：

1. 初始化变量，初始化操作在整个 for 循环只会执行一次。

- 执行条件表达式，如果为true，则执行循环体语句，否则退出循环，循环结束。

1. 执行操作表达式，此时第一轮结束。
2. 第二轮开始，直接去执行条件表达式（不再初始化变量），如果为 true ，则去执行循环体语句，否则退出循环。
3. 继续执行操作表达式，第二轮结束。
4. 后续跟第二轮一致，直至条件表达式为假，结束整个 for 循环。

```
// 3. 初始化变量 就是用var 声明的一个普通变量， 通常用于作为计数器使用
        // 4. 条件表达式 就是用来决定每一次循环是否继续执行 就是终止的条件
        // 5. 操作表达式 是每次循环最后执行的代码 经常用于我们计数器变量进行更新（递增或者递减）
        // 6. 代码体验 我们重复打印100局 你好
        for (var i = 1; i <= 100; i++) {
            console.log('你好吗');
        }
```

断点调试：

断点调试是指自己在程序的某一行设置一个断点，调试时，程序运行到这一行就会停住，然后你可以一步一步往下调试，调试过程中可以看各个变量当前的值，出错的话，调试到出错的代码行即显示错误，停下。断点调试可以帮助观察程序的运行过程

断点调试的流程：
1、浏览器中按 F12--> sources -->找到需要调试的文件-->在程序的某一行设置断点
2、Watch：监视，通过watch可以监视变量的值的变化，非常的常用。
3、摁下F11，程序单步执行，让程序一行一行的执行，这个时候，观察watch中变量的值的变化。

- for 循环重复相同的代码

```
// for 循环可以执行相同的代码
        for (var i = 1; i <= 10; i++) {
            console.log('我想你了');
        }
        // 我们可以让用户控制输出的次数
        var num = prompt('请您输入次数');
        for (var i = 1; i <= num; i++) {
            console.log('我想你了');
        }
```

- for 循环重复不相同的代码

  例如，求输出1到100岁：

```
//  基本写法
for (var i = 1; i <= 100; i++) {
      console.log('这个人今年' + i + '岁了');
}
```

  例如，求输出1到100岁，并提示出生、死亡

```
// for 里面是可以添加其他语句的
for (var i = 1; i <= 100; i++) {
 if (i == 1) {
    console.log('这个人今年1岁了， 它出生了');
 } else if (i == 100) {
    console.log('这个人今年100岁了，它死了');
  } else {
      console.log('这个人今年' + i + '岁了');
  }
}
```

for循环因为有了计数器的存在，还可以重复的执行某些操作，比如做一些算术运算。

```
        var sum = 0; // 求和 的变量
        for (var i = 1; i <= 100; i++) {
            // sum = sum + i;
            sum += i;
        }
        console.log(sum);
```

```
  // 1. 求1-100之间所有数的平均值   需要一个 sum 和的变量 还需要一个平均值 average 变量
        var sum = 0;
        var average = 0;
        for (var i = 1; i <= 100; i++) {
            sum = sum + i;
        }
        average = sum / 100;
        console.log(average);
        // 2. 求1-100之间所有偶数和奇数的和   我们需要一个偶数的和变量 even  还需要一个奇数 odd
        var even = 0;
        var odd = 0;
        for (var i = 1; i <= 100; i++) {
            if (i % 2 == 0) {
                even = even + i;
            } else {
                odd = odd + i;
            }
        }
        console.log('1~100 之间所有的偶数和是' + even);
        console.log('1~100 之间所有的奇数和是' + odd);
        // 3. 求1-100之间所有能被3整除的数字的和
        var result = 0;
        for (var i = 1; i <= 100; i++) {
            if (i % 3 == 0) {
                // result = result + i;
                result += i;
            }
        }
        console.log('1~100之间能够被3整数的数字的和是：' + result);
```

```
      //求学生成绩案例
```

```
// 弹出输入框输入总的班级人数(num)
// 依次输入学生的成绩（ 保存起来 score ）， 此时我们需要用到
// for 循环， 弹出的次数跟班级总人数有关系 条件表达式 i <= num
// 进行业务处理：计算成绩。 先求总成绩（ sum ）， 之后求平均成绩（ average ）
// 弹出结果
var num = prompt('请输入班级的总人数:'); // num 总的班级人数
var sum = 0; // 求和的变量
var average = 0; // 求平均值的变量
for (var i = 1; i <= num; i++) {
    var score = prompt('请您输入第' + i + '个学生成绩');
    // 因为从prompt取过来的数据是 字符串型的需要转换为数字型
    sum = sum + parseFloat(score);
}
average = sum / num;
alert('班级总的成绩是' + sum);
alert('班级平均分是：' + average);
```

```
// 一行打印五个星星
// console.log('★★★★★');
// for (var i = 1; i <= 5; i++) {
//     console.log('★');
// }
// var str = '';
// for (var i = 1; i <= 5; i++) {
//     str = str + '★';
// }
// console.log(str);
var num = prompt('请输入星星的个数');
var str = '';
for (var i = 1; i <= num; i++) {
    str = str + '★'
}
console.log(str);
```

1.2 双重for循环

- 双重 for 循环概述

> 循环嵌套是指在一个循环语句中再定义一个循环语句的语法结构，例如在for循环语句中，可以再嵌套一个for循环，这样的 for 循环语句我们称之为双重for循环。

- 双重 for 循环语法

```
for（外循环的初始；外循环的条件；外循环的操作表达式）{
    for（内循环的初始；内循环的条件；内循环的操作表达式）{
        需执行的代码；
    }
}
```

  - 内层循环可以看做外层循环的循环体语句
  - 内层循环执行的顺序也要遵循 for 循环的执行顺序

- 外层循环执行一次，内层循环要执行全部次数

- 打印五行五列星星

```
var star = '';
for (var j = 1; j <= 3; j++) {
    for (var i = 1; i <= 3; i++) {
      star += '☆'
    }
    // 每次满 5个星星 就 加一次换行
    star += '\n'
}
console.log(star);
```

核心逻辑：

1.内层循环负责一行打印五个星星

2.外层循环负责打印五行

- for 循环小结
  - for 循环可以重复执行某些相同代码
  - for 循环可以重复执行些许不同的代码，因为我们有计数器
  - for 循环可以重复执行某些操作，比如算术运算符加法操作
  - 随着需求增加，双重for循环可以做更多、更好看的效果
  - 双重 for 循环，外层循环一次，内层 for 循环全部执行
  - for 循环是循环条件和数字直接相关的循环

```
// 2. 我们可以把里面的循环看做是外层循环的语句
    // 3. 外层循环循环一次， 里面的循环执行全部
    // 4. 代码验证
    for (var i = 1; i <= 3; i++) {
        console.log('这是外层循环第' + i + '次');
        for (var j = 1; j <= 3; j++) {
            console.log('这是里层的循环第' + j + '次');
        }
    }
```

```
// 打印五行五列星星
    var str = '';
    for (var i = 1; i <= 5; i++) { // 外层循环负责打印五行
        for (var j = 1; j <= 5; j++) { // 里层循环负责一行打印五个星星
            str = str + '★';
        }
        // 如果一行打印完毕5个星星就要另起一行 加 \n
        str = str + '\n';
    }
    console.log(str);
```

```
// 打印n行n列的星星
        var rows = prompt('请您输入行数:');
        var cols = prompt('请您输入列数:');
        var str = '';
        for (var i = 1; i <= rows; i++) {
            for (var j = 1; j <= cols; j++) {
                str = str + '★';
            }
            str += '\n';
        }
        console.log(str);
```

```
// 打印倒三角形案例
        var str = '';
        for (var i = 1; i <= 10; i++) { // 外层循环控制行数
            for (var j = i; j <= 10; j++) { // 里层循环打印的个数不一样  j = i
                str = str + '★';
            }
            str += '\n';
        }
        console.log(str);
```

```
    // 九九乘法表
        // 一共有9行，但是每行的个数不一样，因此需要用到双重 for 循环
        // 外层的 for 循环控制行数 i ，循环9次 ，可以打印 9 行
        // 内层的 for 循环控制每行公式  j
        // 核心算法：每一行 公式的个数正好和行数一致， j <= i;
        // 每行打印完毕，都需要重新换一行
        var str = '';
        for (var i = 1; i <= 9; i++) { // 外层循环控制行数
            for (var j = 1; j <= i; j++) { // 里层循环控制每一行的个数  j <= i
                // 1 × 2 = 2
                // str = str + '★';
                str += j + '×' + i + '=' + i * j + '\t';
            }
            str += '\n';
        }
        console.log(str);
```

1.3 while循环

while语句的语法结构如下：

```
while（条件表达式）{
    // 循环体代码
}
```

执行思路：

- 1 先执行条件表达式，如果结果为 true，则执行循环体代码；如果为 false，则退出循环，执行后面代码
- 2 执行循环体代码

- 3 循环体代码执行完毕后，程序会继续判断执行条件表达式，如条件仍为true，则会继续执行循环体，直到循环条件为 false 时，整个循环过程才会结束

注意：

- 使用 while 循环时一定要注意，它必须要有退出条件，否则会成为死循环

```
        var num = 1;
         while (num <= 100) {
             console.log('好啊有');
             num++;
         }
         // 4. 里面应该也有计数器 初始化变量
         // 5. 里面应该也有操作表达式  完成计数器的更新 防止死循环
```

```
// while循环案例
        // 1. 打印人的一生，从1岁到100岁
        var i = 1;
        while (i <= 100) {
            console.log('这个人今年' + i + '岁了');
            i++;
        }
        // 2. 计算 1 ~ 100 之间所有整数的和
        var sum = 0;
        var j = 1;
        while (j <= 100) {
            sum += j;
            j++
        }
        console.log(sum);
        // 3. 弹出一个提示框， 你爱我吗？  如果输入我爱你，就提示结束，否则，一直询问。
        var message = prompt('你爱我吗?');
        while (message !== '我爱你') {
            message = prompt('你爱我吗?');
        }
        alert('我也爱你啊！');
```

1.4 do-while循环

do... while 语句的语法结构如下：

```
do {
    // 循环体代码 - 条件表达式为 true 时重复执行循环体代码
} while(条件表达式);
```

执行思路

1 先执行一次循环体代码

2 再执行条件表达式，如果结果为 true，则继续执行循环体代码，如果为 false，则退出循环，继续执行后面代码

注意：先再执行循环体，再判断，do...while循环语句至少会执行一次循环体代码

```
 var i = 1;
       do {
           console.log('how are you?');
           i++;
       } while (i <= 100)
```

```
// while循环案例
       // 1. 打印人的一生，从1岁到100岁
       var i = 1;
       do {
           console.log('这个人今年' + i + '岁了');
           i++;
       } while (i <= 100)
       // 2. 计算 1 ~ 100 之间所有整数的和
       var sum = 0;
       var j = 1;
       do {
           sum += j;
           j++;
       } while (j <= 100)
       console.log(sum);
       // 3. 弹出一个提示框， 你爱我吗？  如果输入我爱你，就提示结束，否则，一直询问。
       do {
           var message = prompt('你爱我吗?');
       } while (message !== '我爱你')
       alert('我也爱你啊');
```

1.5 continue、break

continue 关键字用于立即跳出本次循环，继续下一次循环（本次循环体中 continue 之后的代码就会少执行一次）。

例如，吃5个包子，第3个有虫子，就扔掉第3个，继续吃第4个第5个包子，其代码实现如下：

```
// continue 关键字   退出本次（当前次的循环）  继续执行剩余次数循环
       for (var i = 1; i <= 5; i++) {
           if (i == 3) {
               continue; // 只要遇见 continue就退出本次循环 直接跳到 i++
           }
           console.log('我正在吃第' + i + '个包子');

       }
       // 1. 求1~100 之间， 除了能被7整除之外的整数和
       var sum = 0;
       for (var i = 1; i <= 100; i++) {
           if (i % 7 == 0) {
               continue;
           }
           sum += i;
       }
       console.log(sum);
```

break 关键字用于立即跳出整个循环（循环结束）。

```
// break 退出整个循环
        for (var i = 1; i <= 5; i++) {
            if (i == 3) {
                break;
            }
            console.log('我正在吃第' + i + '个包子');
        }
```

2 - 代码规范

2.1 标识符命名规范

- 变量、函数的命名必须要有意义
- 变量的名称一般用名词
- 函数的名称一般用动词

2.2 操作符规范

```
// 操作符的左右两侧各保留一个空格
for (var i = 1; i <= 5; i++) {
   if (i == 3) {
       break; // 直接退出整个 for 循环，跳到整个for循环下面的语句
   }
   console.log('我正在吃第' + i + '个包子呢');
}
```

2.3 单行注释规范

```
for (var i = 1; i <= 5; i++) {
   if (i == 3) {
       break; // 单行注释前面注意有个空格
   }
   console.log('我正在吃第' + i + '个包子呢');
}
```

2.4 其他规范

关键词、操作符之间后加空格

```
if (true) {

}
for (var i = 0; i <= 100; i++) {

}
```

JavaScript基础

1 - 数组

1.1 数组的概念

- 数组可以把一组相关的数据一起存放，并提供方便的访问(获取）方式。
- 数组是指**一组数据的集合**，其中的每个数据被称作**元素**，在数组中可以**存放任意类型的元素**。数组是一种将一组数据存储在单个变量名下的优雅方式。

1.2 创建数组

JS 中创建数组有两种方式：

- 利用 new 创建数组

```
var 数组名 = new Array() ;
var arr = new Array();   // 创建一个新的空数组
```

注意 Array () ，A 要大写

- 利用数组字面量创建数组

```
//1. 使用数组字面量方式创建空的数组
var  数组名 = [];
//2. 使用数组字面量方式创建带初始值的数组
var  数组名 = ['小白','小黑','大黄','瑞奇'];
```

  - 数组的字面量是方括号 [ ]
  - 声明数组并赋值称为数组的初始化
  - 这种字面量方式也是我们以后最多使用的方式

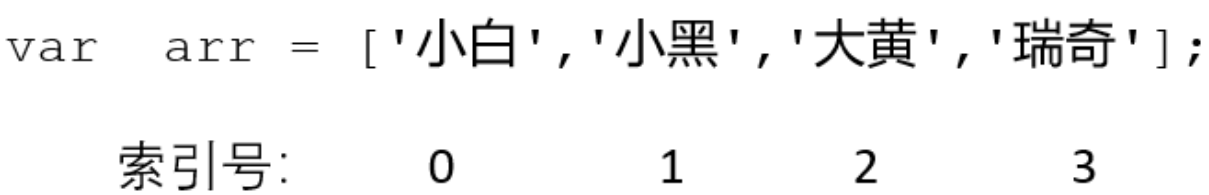
- 数组元素的类型

  数组中可以存放任意类型的数据，例如字符串，数字，布尔值等。

```
var arrStus = ['小白',12,true,28.9];
```

1.3 获取数组中的元素

索引(下标)：用来访问数组元素的序号（数组下标从 0 开始）。



数组可以通过索引来访问、设置、修改对应的数组元素，可以通过“数组名[索引]”的形式来获取数组中的元素。

```
// 定义数组
var arrStus = [1,2,3];
// 获取数组中的第2个元素
alert(arrStus[1]);
```

注意：如果访问时数组没有和索引值对应的元素，则得到的值是undefined

1.4 遍历数组

- 数组遍历

  把数组中的每个元素从头到尾都访问一次（类似学生的点名），可以通过 for 循环索引遍历数组中的每一项

```
var arr = ['red','green', 'blue'];
for(var i = 0; i < arr.length; i++){
    console.log(arrStus[i]);
}
```

- 数组的长度

  数组的长度：默认情况下表示数组中元素的个数

  使用“数组名.length”可以访问数组元素的数量（数组长度）。

  ```
  var arrStus = [1,2,3];
  alert(arrStus.length);  // 3
  ```

  注意：

  - 此处数组的长度是数组元素的个数 ，不要和数组的索引号混淆。
- 当我们数组里面的元素个数发生了变化，这个 length 属性跟着一起变化
  - 数组的length属性可以被修改：
- 如果设置的length属性值大于数组的元素个数，则会在数组末尾出现空白元素；
  - 如果设置的length属性值小于数组的元素个数，则会把超过该值的数组元素删除

1.5 数组中新增元素

数组中可以通过以下方式在数组的末尾插入新元素：

```
数组[ 数组.length ] = 新数据;
```

2 - 函数

2.1 函数的概念

在 JS 里面，可能会定义非常多的相同代码或者功能相似的代码，这些代码可能需要大量重复使用。虽然 for循环语句也能实现一些简单的重复操作，但是比较具有局限性，此时我们就可以使用 JS 中的函数。

函数：就是**封装了一段可被重复调用执行的代码块。**通过此代码块可以**实现大量代码的重复使用。**

2.2 函数的使用

声明函数

```
// 声明函数
function 函数名() {
    //函数体代码
}
```

- function 是声明函数的关键字,必须小写
- 由于函数一般是为了实现某个功能才定义的， 所以通常我们将函数名命名为动词，比如 getSum

调用函数

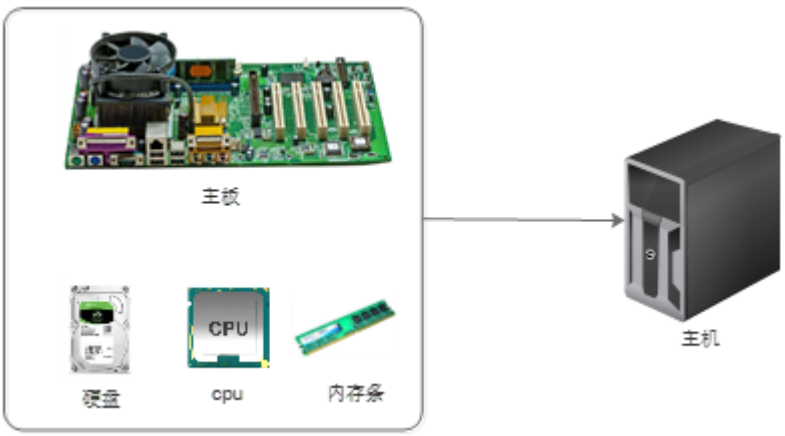
```
// 调用函数
函数名();  // 通过调用函数名来执行函数体代码
```

- 调用的时候千万不要忘记添加小括号
- 口诀：函数不调用，自己不执行

  注意：声明函数本身并不会执行代码，只有调用函数时才会执行函数体代码。

函数的封装

- 函数的封装是把一个或者多个功能通过函数的方式封装起来，对外只提供一个简单的函数接口
- 简单理解：封装类似于将电脑配件整合组装到机箱中（类似快递打包）



例子：封装计算1-100累加和

```
/*
   计算1-100之间值的函数
*/
// 声明函数
function getSum(){
  var sumNum = 0;// 准备一个变量，保存数字和
  for (var i = 1; i <= 100; i++) {
    sumNum += i;// 把每个数值 都累加 到变量中
  }
  alert(sumNum);
}
// 调用函数
getSum();
```

2.3 函数的参数

函数参数语法

- 形参：函数定义时设置接收调用时传入
- 实参：函数调用时传入小括号内的真实数据

参数	说明
形参	形式上的参数 函数定义的时候 传递的参数 当前并不知道是什么
实参	实际上的参数 函数调用的时候传递的参数 实参是传递给形参的

参数的作用：在函数内部某些值不能固定，我们可以通过参数在调用函数时传递不同的值进去。

函数参数的运用：

```
// 带参数的函数声明
function 函数名(形参1, 形参2 , 形参3...) { // 可以定义任意多的参数，用逗号分隔
  // 函数体
}
// 带参数的函数调用
函数名(实参1, 实参2, 实参3...);
```

1. 调用的时候实参值是传递给形参的
2. 形参简单理解为：不用声明的变量
3. 实参和形参的多个参数之间用逗号（,）分隔

函数形参和实参数量不匹配时

参数个数	说明
实参个等于形参个数	输出正确结果
实参个数多于形参个数	只取到形参的个数
实参个数小于形参个数	多的形参定义为undefined，结果为NaN

注意：在JavaScript中，形参的默认值是undefined。

小结：

- 函数可以带参数也可以不带参数
- 声明函数的时候，函数名括号里面的是形参，形参的默认值为 undefined
- 调用函数的时候，函数名括号里面的是实参
- 多个参数中间用逗号分隔
- 形参的个数可以和实参个数不匹配，但是结果不可预计，我们尽量要匹配

2.4 函数的返回值

return 语句

返回值：函数调用整体代表的数据；函数执行完成后可以通过return语句将指定数据返回 。

```
// 声明函数
function 函数名 ( ) {
    ...
    return  需要返回的值；
}
// 调用函数
函数名();    // 此时调用函数就可以得到函数体内return 后面的值
```

- 在使用 return 语句时，函数会停止执行，并返回指定的值
- 如果函数没有 return ，返回的值是 undefined

break ,continue ,return 的区别

- break ：结束当前的循环体（如 for、while ）
- continue ：跳出本次循环，继续执行下次循环（如 for、while ）
- return ：不仅可以退出循环，还能够返回 return 语句中的值，同时还可以结束当前的函数体内的代码

2.5 arguments的使用

当不确定有多少个参数传递的时候，可以用 arguments 来获取。JavaScript 中，arguments实际上它是当前函数的一个内置对象。所有函数都内置了一个 arguments 对象，arguments 对象中存储了传递的所有实参。arguments展示形式是一个伪数组，因此可以进行遍历。伪数组具有以下特点：

- 具有 length 属性
- 按索引方式储存数据
- 不具有数组的 push , pop 等方法

  注意：在函数内部使用该对象，用此对象获取函数调用时传的实参。

2.6 函数案例

函数内部可以调用另一个函数，在同一作用域代码中，函数名即代表封装的操作，使用函数名加括号即可以将封装的操作执行。

2.7 函数的两种声明方式

- 自定义函数方式(命名函数)

  利用函数关键字 function 自定义函数方式

  ```
  // 声明定义方式
  function fn() {...}
  // 调用
  fn();
  ```

  - 因为有名字，所以也被称为命名函数
  - 调用函数的代码既可以放到声明函数的前面，也可以放在声明函数的后面

- 函数表达式方式(匿名函数)

  利用函数表达式方式的写法如下：

  ```
  // 这是函数表达式写法，匿名函数后面跟分号结束
  var fn = function(){...};
  // 调用的方式，函数调用必须写到函数体下面
  fn();
  ```

  - 因为函数没有名字，所以也被称为匿名函数
  - 这个fn 里面存储的是一个函数
  - 函数表达式方式原理跟声明变量方式是一致的
  - 函数调用的代码必须写到函数体后面