可以将js代码编写到外部js文件中，然后通过script标签引入

写到外部文件中可以在不同的页面中同时引用，也可以利用到浏览器的缓存机制

推荐使用的方式

script标签一旦用于引入外部文件了，就不能在编写代码了，即使编写了浏览器也会忽略

如果需要则可以在创建一个新的script标签用于编写内部代码

可以将js代码编写到script标签

<script type="text/javascript">

alert("我是script标签中的代码！！");

</script>

可以将js代码编写到标签的onclick属性中

当我们点击按钮时，js代码才会执行

虽然可以写在标签的属性中，但是他们属于结构与行为耦合，不方便维护，不推荐使用

<button onclick="alert('讨厌，你点我干嘛~~');">点我一下</button>

可以将js代码写在超链接的href属性中，这样当点击超链接时，会执行js代码

<a href="javascript:;">你也点我一下</a>

多行注释

JS注释

多行注释，注释中的内容不会被执行，但是可以在源代码中查看

要养成良好的编写注释的习惯，也可以通过注释来对代码进行一些简单的调试

\* 1.JS中严格区分大小写

\* 2.JS中每一条语句以分号(;)结尾

\* - 如果不写分号，浏览器会自动添加，但是会消耗一些系统资源，

\* 而且有些时候，浏览器会加错分号，所以在开发中分号必须写

\* 3.JS中会忽略多个空格和换行，所以我们可以利用空格和换行对代码进行格式化

\* 字面量，都是一些不可改变的值

\* 比如 ：1 2 3 4 5

\* 字面量都是可以直接使用，但是我们一般都不会直接使用字面量

\*

\* 变量 变量可以用来保存字面量，而且变量的值是可以任意改变的

\* 变量更加方便我们使用，所以在开发中都是通过变量去保存一个字面量，

\* 而很少直接使用字面量

\* 可以通过变量对字面量进行描述

标识符

\* - 在JS中所有的可以由我们自主命名的都可以称为是标识符

\* - 例如：变量名、函数名、属性名都属于标识符

\* - 命名一个标识符时需要遵守如下的规则：

\* 1.标识符中可以含有字母、数字、\_、$

\* 2.标识符不能以数字开头

\* 3.标识符不能是ES中的关键字或保留字

\* 4.标识符一般都采用驼峰命名法

\* - 首字母小写，每个单词的开头字母大写，其余字母小写

\* helloWorld xxxYyyZzz

\*

\* - JS底层保存标识符时实际上是采用的Unicode编码，

\* 所以理论上讲，所有的utf-8中含有的内容都可以作为标识符

\* 数据类型指的就是字面量的类型

\* 在JS中一共有六种数据类型

\* String 字符串

\* Number 数值

\* Boolean 布尔值

\* Null 空值

\* Undefined 未定义

\* Object 对象

\*

\* 其中String Number Boolean Null Undefined属于基本数据类型

\* 而Object属于引用数据类型

\* String字符串

\* - 在JS中字符串需要使用引号引起来

\* - 使用双引号或单引号都可以，但是不要混着用

\* - 引号不能嵌套，双引号不能放双引号，单引号不能放单引号

在字符串中我们可以使用\作为转义字符，

当表示一些特殊符号时可以使用\进行转义

\" 表示 "

\' 表示 '

\n 表示换行

\t 制表符

\\ 表示\

\* 在JS中所有的数值都是Number类型，

\* 包括整数和浮点数（小数）

\*

\* JS中可以表示的数字的最大值

\* Number.MAX\_VALUE

\* 1.7976931348623157e+308

\*

\* Number.MIN\_VALUE 大于0的最小值

\* 5e-324

\*

\* 如果使用Number表示的数字超过了最大值，则会返回一个

\* Infinity 表示正无穷

\* -Infinity 表示负无穷

\* 使用typeof检查Infinity也会返回number

\* NaN 是一个特殊的数字，表示Not A Number

\* 使用typeof检查一个NaN也会返回number

可以使用一个运算符 typeof

来检查一个变量的类型

语法：typeof 变量

检查字符串时，会返回string

检查数值时，会返回number

如果使用JS进行浮点运算，可能得到一个不精确的结果

\* 所以千万不要使用JS进行对精确度要求比较高的运算

\* Boolean 布尔值

\* 布尔值只有两个，主要用来做逻辑判断

\* true

\* - 表示真

\* false

\* - 表示假

\*

\* 使用typeof检查一个布尔值时，会返回boolean

\* Null（空值）类型的值只有一个，就是null

\* null这个值专门用来表示一个为空的对象

\* 使用typeof检查一个null值时，会返回object

\*

\* Undefined（未定义）类型的值只有一个，就undefind

\* 当声明一个变量，但是并不给变量赋值时，它的值就是undefined

\* 使用typeof检查一个undefined时也会返回undefined

\* 强制类型转换

\* - 指将一个数据类型强制转换为其他的数据类型

\* - 类型转换主要指，将其他的数据类型，转换为

\* String Number Boolean

\* 将其他的数据类型转换为String

\* 方式一：

\* - 调用被转换数据类型的toString()方法

\* - 该方法不会影响到原变量，它会将转换的结果返回

\* - 但是注意：null和undefined这两个值没有toString()方法，

\* 如果调用他们的方法，会报错

\* 方式二：

\* - 调用String()函数，并将被转换的数据作为参数传递给函数

\* - 使用String()函数做强制类型转换时，

\* 对于Number和Boolean实际上就是调用的toString()方法

\* 但是对于null和undefined，就不会调用toString()方法

\* 它会将 null 直接转换为 "null"

\* 将 undefined 直接转换为 "undefined"

\* 将其他的数据类型转换为Number

\* 转换方式一：

\* 使用Number()函数

\* - 字符串 --> 数字

\* 1.如果是纯数字的字符串，则直接将其转换为数字

\* 2.如果字符串中有非数字的内容，则转换为NaN

\* 3.如果字符串是一个空串或者是一个全是空格的字符串，则转换为0

\* - 布尔 --> 数字

\* true 转成 1

\* false 转成 0

\*

\* - null --> 数字 0

\*

\* - undefined --> 数字 NaN

\* 转换方式二：

\* - 这种方式专门用来对付字符串

\* - parseInt() 把一个字符串转换为一个整数

\* - parseFloat() 把一个字符串转换为一个浮点数

调用Number()函数来将a转换为Number类型

\* parseInt()可以将一个字符串中的有效的整数内容去出来，

\* 然后转换为Number

parseFloat()作用和parseInt()类似，不同的是它可以获得有效的小数

\* 如果对非String使用parseInt()或parseFloat()

\* 它会先将其转换为String然后在操作

\* 在js中，如果需要表示16进制的数字，则需要以0x开头

\* 如果需要表示8进制的数字，则需要以0开头

\* 如果要要表示2进制的数字，则需要以0b开头

\* 但是不是所有的浏览器都支持

\* 将其他的数据类型转换为Boolean

\* - 使用Boolean()函数

\* - 数字 ---> 布尔

\* - 除了0和NaN，其余的都是true

\*

\* - 字符串 ---> 布尔

\* - 除了空串，其余的都是true

\*

\* - null和undefined都会转换为false

\*

\* - 对象也会转换为true

调用Boolean()函数来将a转换为布尔值

\* 运算符也叫操作符

\* 通过运算符可以对一个或多个值进行运算,并获取运算结果

\* 比如：typeof就是运算符，可以来获得一个值的类型

\* 它会将该值的类型以字符串的形式返回

\* number string boolean undefined object

\*

\* 算数运算符

\* 当对非Number类型的值进行运算时，会将这些值转换为Number然后在运算

\* 任何值和NaN做运算都得NaN

\*

\* +

\* +可以对两个值进行加法运算，并将结果返回

\* 如果对两个字符串进行加法运算，则会做拼串

\* 会将两个字符串拼接为一个字符串，并返回

\* 任何的值和字符串做加法运算，都会先转换为字符串，然后再和字符串做拼串的操作

\* -

\* - 可以对两个值进行减法运算，并将结果返回

\*

\* \*

\* \* 可以对两个值进行乘法运算

\* /

\* / 可以对两个值进行除法运算

\* %

\* % 取模运算（取余数）

任何值和字符串相加都会转换为字符串，并做拼串操作

\* 我们可以利用这一特点，来将一个任意的数据类型转换为String

\* 我们只需要为任意的数据类型 + 一个 "" 即可将其转换为String

\* 这是一种隐式的类型转换，由浏览器自动完成，实际上它也是调用String()函数

一元运算符，只需要一个操作数

\* + 正号

\* - 正号不会对数字产生任何影响

\* - 负号

\* - 负号可以对数字进行负号的取反

\*

\* - 对于非Number类型的值，

\* 它会将先转换为Number，然后在运算

\* 可以对一个其他的数据类型使用+,来将其转换为number

\* 它的原理和Number()函数一样

自增 ++

\* - 通过自增可以使变量在自身的基础上增加1

\* - 对于一个变量自增以后，原变量的值会立即自增1

\* - 自增分成两种：后++(a++) 和 前++(++a)

\* 无论是a++ 还是 ++a，都会立即使原变量的值自增1

\* 不同的是a++ 和 ++a的值不同

\* a++的值等于原变量的值（自增前的值）

\* ++a的值等于新值 （自增后的值）

\*

自减 --

\* - 通过自减可以使变量在自身的基础上减1

\* - 自减分成两种：后--(a--) 和 前--(--a)

\* 无论是a-- 还是 --a 都会立即使原变量的值自减1

\* 不同的是a-- 和 --a的值不同

\* a-- 是变量的原值 （自减前的值）

\* --a 是变量的新值 （自减以后的值）

var n1=10;

var n2=20;

var n = n1++; //n1 = 11 n1++ = 10

console.log('n='+n); // 10

console.log('n1='+n1); //11

n = ++n1 //n1 = 12 ++n1 =12

console.log('n='+n); //12

console.log('n1='+n1); //12

n = n2--;// n2=19 n2--=20

console.log('n='+n); //20

console.log('n2='+n2); //19

n = --n2; //n2=18 --n2 = 18

console.log('n='+n); //18

console.log('n2='+n2); //18

\* JS中为我们提供了三种逻辑运算符

\* ! 非

\* - !可以用来对一个值进行非运算

\* - 所谓非运算就是值对一个布尔值进行取反操作，

\* true变false，false变true

\* - 如果对一个值进行两次取反，它不会变化

\* - 如果对非布尔值进行元素，则会将其转换为布尔值，然后再取反

\* 所以我们可以利用该特点，来将一个其他的数据类型转换为布尔值

\* 可以为一个任意数据类型取两次反，来将其转换为布尔值，

\* 原理和Boolean()函数一样

\*

\* && 与

\* - &&可以对符号两侧的值进行与运算并返回结果

\* - 运算规则

\* - 两个值中只要有一个值为false就返回false，

\* 只有两个值都为true时，才会返回true

\* - JS中的“与”属于短路的与，

\* 如果第一个值为false，则不会看第二个值

\*

\* || 或

\* - ||可以对符号两侧的值进行或运算并返回结果

\* - 运算规则：

\* - 两个值中只要有一个true，就返回true

\* 如果两个值都为false，才返回false

\* - JS中的“或”属于短路的或

\* 如果第一个值为true，则不会检查第二个值

\* && || 非布尔值的情况

\* - 对于非布尔值进行与或运算时，

\* 会先将其转换为布尔值，然后再运算，并且返回原值

\* - 与运算：

\* - 如果第一个值为true，则必然返回第二个值

\* - 如果第一个值为false，则直接返回第一个值

\*

\* - 或运算

\* - 如果第一个值为true，则直接返回第一个值

\* - 如果第一个值为false，则返回第二个值

与运算：如果两个值都为true，则返回后边的

与运算：如果两个值中有false，则返回靠前的false

或运算：如果第一个值为true，则直接返回第一个值

或运算：如果第一个值为false，则直接返回第二个值

\* =

\* 可以将符号右侧的值赋值给符号左侧的变量

\* +=

\* a += 5 等价于 a = a + 5

\* 通过关系运算符可以比较两个值之间的大小关系，

\* 如果关系成立它会返回true，如果关系不成立则返回false

\*

\* > 大于号

\* - 判断符号左侧的值是否大于右侧的值

\* - 如果关系成立，返回true，如果关系不成立则返回false

\*

\* >= 大于等于

\* - 判断符号左侧的值是否大于或等于右侧的值

\*

\* < 小于号

\* <= 小于等于

\*

\* 非数值的情况

\* - 对于非数值进行比较时，会将其转换为数字然后在比较

\* - 如果符号两侧的值都是字符串时，不会将其转换为数字进行比较

\* 而会分别比较字符串中字符的Unicode编码

\* 在字符串中使用转义字符输入Unicode编码

\* \u四位编码

在网页中使用Unicode编码

&#编码; 这里的编码需要的是10进制

\* 相等运算符用来比较两个值是否相等，

\* 如果相等会返回true，否则返回false

\*

\* 使用 == 来做相等运算

\* - 当使用==来比较两个值时，如果值的类型不同，

\* 则会自动进行类型转换，将其转换为相同的类型

\* 然后在比较

\* 不相等

\* 不相等用来判断两个值是否不相等，如果不相等返回true，否则返回false

\* - 使用 != 来做不相等运算

\* - 不相等也会对变量进行自动的类型转换，如果转换后相等它也会返回false

\*

\*

\* ===

\* 全等

\* - 用来判断两个值是否全等，它和相等类似，不同的是它不会做自动的类型转换

\* 如果两个值的类型不同，直接返回false

\* !==

\* 不全等

\* - 用来判断两个值是否不全等，和不等类似，不同的是它不会做自动的类型转换

\* 如果两个值的类型不同，直接返回true

\* undefined 衍生自 null

\* 所以这两个值做相等判断时，会返回true

NaN不和任何值相等，包括他本身

\* 可以通过isNaN()函数来判断一个值是否是NaN

\* 如果该值是NaN则返回true，否则返回false

\* 条件运算符也叫三元运算符

\* 语法：

\* 条件表达式?语句1:语句2;

\* - 执行的流程：

\* 条件运算符在执行时，首先对条件表达式进行求值，

\* 如果该值为true，则执行语句1，并返回执行结果

\* 如果该值为false，则执行语句2，并返回执行结果

\* 如果条件的表达式的求值结果是一个非布尔值，

\* 会将其转换为布尔值然后在运算

\* , 运算符

\* 使用,可以分割多个语句，一般可以在声明多个变量时使用,

\* 就和数学中一样，在JS中运算符也有优先级，

\* 比如：先乘除 后加减

\* 在JS中有一个运算符优先级的表，

\* 在表中越靠上优先级越高，优先级越高越优先计算，

\* 如果优先级一样，则从左往右计算。

\* 但是这个表我们并不需要记忆，如果遇到优先级不清楚

\* 可以使用()来改变优先级

\* 我们的程序是由一条一条语句构成的

\* 语句是按照自上向下的顺序一条一条执行的

\* 在JS中可以使用{}来为语句进行分组,

\* 同一个{}中的语句我们称为是一组语句，

\* 它们要么都执行，要么都不执行，

\* 一个{}中的语句我们也称为叫一个代码块

\* 在代码块的后边就不用再编写;了

\*

\* JS中的代码块，只具有分组的的作用，没有其他的用途

\* 代码块内容的内容，在外部是完全可见的

\* 流程控制语句

\* - JS中的程序是从上到下一行一行执行的

\* - 通过流程控制语句可以控制程序执行流程，

\* 使程序可以根据一定的条件来选择执行

\* - 语句的分类：

\* 1.条件判断语句

\* 2.条件分支语句

\* 3.循环语句

\*

\*

\* 条件判断语句：

\* - 使用条件判断语句可以在执行某个语句之前进行判断，

\* 如果条件成立才会执行语句，条件不成立则语句不执行。

\* - if语句

\* - 语法一：

\* if(条件表达式){

\* 语句...

\* }

\*

\* if语句在执行时，会先对条件表达式进行求值判断，

\* 如果条件表达式的值为true，则执行if后的语句，

\* 如果条件表达式的值为false，则不会执行if后的语句。

\* if语句只能控制紧随其后的那个语句,

\* 如果希望if语句可以控制多条语句，

\* 可以将这些语句统一放到代码块中

\* if语句后的代码块不是必须的，但是在开发中尽量写上代码块，即使if后只有一条语句

\* if语句

\* 语法二:

\* if(条件表达式){

\* 语句...

\* }else{

\* 语句...

\* }

\*

\* if...else...语句

\* 当该语句执行时，会先对if后的条件表达式进行求值判断，

\* 如果该值为true，则执行if后的语句

\* 如果该值为false，则执行else后的语句

\*

\* 语法三：

\* if(条件表达式){

\* 语句...

\* }else if(条件表达式){

\* 语句...

\* }else if(条件表达式){

\* 语句...

\* }else{

\* 语句...

\* }

\*

\* if...else if...else

\* 当该语句执行时，会从上到下依次对条件表达式进行求值判断

\* 如果值为true，则执行当前语句。

\* 如果值为false，则继续向下判断。

\* 如果所有的条件都不满足，则执行最后一个else后的语句

\* 该语句中，只会有一个代码块被执行，一旦代码块执行了，则直接结束语句

\* 条件分支语句也叫switch语句

\* 语法：

\* switch(条件表达式){

\* case 表达式:

\* 语句...

\* break;

\* case 表达式:

\* 语句...

\* break;

\* default:

\* 语句...

\* break;

\* }

\*

\* 执行流程：

\* switch...case..语句

\* 在执行时会依次将case后的表达式的值和switch后的条件表达式的值进行全等比较，

\* 如果比较结果为true，则从当前case处开始执行代码。

\* 当前case后的所有的代码都会执行，我们可以在case的后边跟着一个break关键字，

\* 这样可以确保只会执行当前case后的语句，而不会执行其他的case

\* 如果比较结果为false，则继续向下比较

\* 如果所有的比较结果都为false，则只执行default后的语句

\*

\* switch语句和if语句的功能实际上有重复的，使用switch可以实现if的功能，

\* 同样使用if也可以实现switch的功能，所以我们使用时，可以根据自己的习惯选择。

\* 循环语句：

\* 通过循环语句可以反复的执行一段代码多次

\*

\* while循环

\* - 语法：

\* while(条件表达式){

\* 语句...

\* }

\*

\* - while语句在执行时，

\* 先对条件表达式进行求值判断，

\* 如果值为true，则执行循环体，

\* 循环体执行完毕以后，继续对表达式进行判断

\* 如果为true，则继续执行循环体，以此类推

\* 如果值为false，则终止循环

\*

\* do...while循环

\* - 语法：

\* do{

\* 语句...

\* }while(条件表达式)

\*

\* - 执行流程：

\* do...while语句在执行时，会先执行循环体，

\* 循环体执行完毕以后，在对while后的条件表达式进行判断，

\* 如果结果为true，则继续执行循环体，执行完毕继续判断以此类推

\* 如果结果为false，则终止循环

\*

\* 实际上这两个语句功能类似，不同的是while是先判断后执行，

\* 而do...while会先执行后判断，

\* do...while可以保证循环体至少执行一次，

\* 而while不能

向这种将条件表达式写死为true的循环，叫做死循环

该循环不会停止，除非浏览器关闭，死循环在开发中慎用

可以使用break，来终止循环

创建一个循环，往往需要三个步骤

1.创初始化一个变量

var i = 11;

2.在循环中设置一个条件表达式

while(i <= 10){

3.定义一个更新表达式，每次更新初始化变量

document.write(i++ +"<br />")

}

\* prompt()可以弹出一个提示框，该提示框中会带有一个文本框，

\* 用户可以在文本框中输入一段内容，该函数需要一个字符串作为参数，

\* 该字符串将会作为提示框的提示文字

\*

\* 用户输入的内容将会作为函数的返回值返回，可以定义一个变量来接收该内容

\* for语句，也是一个循环语句，也称为for循环

\* 在for循环中，为我们提供了专门的位置用来放三个表达式：

\* 1.初始化表达式

\* 2.条件表达式

\* 3.更新表达式

\*

\* for循环的语法：

\* for(①初始化表达式;②条件表达式;④更新表达式){

\* ③语句...

\* }

\*

\* for循环的执行流程：

\* ①执行初始化表达式，初始化变量（初始化表达式只会执行一次）

\* ②执行条件表达式，判断是否执行循环。

\* 如果为true，则执行循环③

\* 如果为false，终止循环

\* ④执行更新表达式，更新表达式执行完毕继续重复②

通过一个for循环来输出图形

这个for循环执行几次，图形的高度就是多少

它可以用来控制图形的高度

\* 在循环的内部再创建一个循环，用来控制图形的宽度

\* 目前我们的外部的for循环执行1次，内部的就会执行5次

\* 内层循环可以来决定图形的宽度，执行几次图形的宽度就是多少

//创建外层循环，用来控制乘法表的高度

for(var i=1 ; i<=9 ; i++ ){

//创建一个内层循环来控制图形的宽度

for(var j=1 ; j<=i ; j++){

document.write("<span>"+j+"\*"+i+"="+i\*j+"</span>");

}

//输出一个换行

document.write("<br />");

}

/\*

\* 打印出1-100之间所有的质数

\*/

//打印2-100之间所有的数

for(var i=2 ; i<=100 ; i++){

//创建一个布尔值，用来保存结果，默认i是质数

var flag = true;

//判断i是否是质数

//获取到2-i之间的所有的数

for(var j=2 ; j<i ; j++){

//判断i是否能被j整除

if(i%j == 0){

//如果进入判断则证明i不是质数,修改flag值为false

flag = false;

}

}

//如果是质数，则打印i的值

if(flag){

console.log(i);

}

}

\* break关键字可以用来退出switch或循环语句

\* 不能在if语句中使用break和continue

\* break关键字，会立即终止离他最近的那个循环语句

\* 可以为循环语句创建一个label，来标识当前的循环

\* label:循环语句

\* 使用break语句时，可以在break后跟着一个label，

\* 这样break将会结束指定的循环，而不是最近的

\* continue关键字可以用来跳过当次循环

\* 同样continue也是默认只会对离他最近的循环循环起作用

\* JS中数据类型

\* - String 字符串

\* - Number 数值

\* - Boolean 布尔值

\* - Null 空值

\* - Undefined 未定义

\* - 以上这五种类型属于基本数据类型，以后我们看到的值

\* 只要不是上边的5种，全都是对象

\* - Object 对象

\*

\*

\* 基本数据类型都是单一的值"hello" 123 true,

\* 值和值之间没有任何的联系。

\*

\* 在JS中来表示一个人的信息（name gender age）：

\* var name = "孙悟空";

\* var gender = "男";

\* var age = 18;

\* 如果使用基本数据类型的数据，我们所创建的变量都是独立，不能成为一个整体。

\*

\* 对象属于一种复合的数据类型，在对象中可以保存多个不同数据类型的属性。

\*

\* 对象的分类：

\* 1.内建对象

\* - 由ES标准中定义的对象，在任何的ES的实现中都可以使用

\* - 比如：Math String Number Boolean Function Object....

\*

\* 2.宿主对象

\* - 由JS的运行环境提供的对象，目前来讲主要指由浏览器提供的对象

\* - 比如 BOM DOM

\*

\* 3.自定义对象

\* - 由开发人员自己创建的对象

创建对象

\* 使用new关键字调用的函数，是构造函数constructor

\* 构造函数是专门用来创建对象的函数

\* 使用typeof检查一个对象时，会返回object

\* 在对象中保存的值称为属性

\* 向对象添加属性

\* 语法：对象.属性名 = 属性值;

\* 读取对象中的属性

\* 语法：对象.属性名

\*

\* 如果读取对象中没有的属性，不会报错而是会返回undefined

\* 修改对象的属性值

\* 语法：对象.属性名 = 新值

\* 删除对象的属性

\* 语法：delete 对象.属性名

\* 向对象中添加属性

\* 属性名：

\* - 对象的属性名不强制要求遵守标识符的规范

\* 什么乱七八糟的名字都可以使用

\* - 但是我们使用是还是尽量按照标识符的规范去做

\* 如果要使用特殊的属性名，不能采用.的方式来操作

\* 需要使用另一种方式：

\* 语法：对象["属性名"] = 属性值

\* 读取时也需要采用这种方式

\*

\* 使用[]这种形式去操作属性，更加的灵活，

\* 在[]中可以直接传递一个变量，这样变量值是多少就会读取那个属性

\* 属性值

\* JS对象的属性值，可以是任意的数据类型

\* 甚至也可以是一个对象

\* in 运算符

\* - 通过该运算符可以检查一个对象中是否含有指定的属性

\* 如果有则返回true，没有则返回false

\* - 语法：

\* "属性名" in 对象

\* 基本数据类型

\* String Number Boolean Null Undefined

\*

\* 引用数据类型

\* Object

\*

\* JS中的变量都是保存到栈内存中的，

\* 基本数据类型的值直接在栈内存中存储，

\* 值与值之间是独立存在，修改一个变量不会影响其他的变量

\*

\* 对象是保存到堆内存中的，每创建一个新的对象，就会在堆内存中开辟出一个新的空间，

\* 而变量保存的是对象的内存地址（对象的引用），如果两个变量保存的是同一个对象引用，

\* 当一个通过一个变量修改属性时，另一个也会受到影响

\* 当比较两个基本数据类型的值时，就是比较值。

\* 而比较两个引用数据类型时，它是比较的对象的内存地址，

\* 如果两个对象是一摸一样的，但是地址不同，它也会返回false

\* 使用对象字面量，可以在创建对象时，直接指定对象中的属性

\* 语法：{属性名:属性值,属性名:属性值....}

\* 对象字面量的属性名可以加引号也可以不加，建议不加,

\* 如果要使用一些特殊的名字，则必须加引号

\*

\* 属性名和属性值是一组一组的名值对结构，

\* 名和值之间使用:连接，多个名值对之间使用,隔开

\* 如果一个属性之后没有其他的属性了，就不要写,

\* 函数 function

\* - 函数也是一个对象

\* - 函数中可以封装一些功能（代码），在需要时可以执行这些功能（代码）

\* - 函数中可以保存一些代码在需要的时候调用

\* - 使用typeof检查一个函数对象时，会返回function

我们在实际开发中很少使用构造函数来创建一个函数对象

创建一个函数对象

可以将要封装的代码以字符串的形式传递给构造函数

封装到函数中的代码不会立即执行

函数中的代码会在函数调用的时候执行

调用函数 语法：函数对象()

当调用函数时，函数中封装的代码会按照顺序执行

\* 使用 函数声明 来创建一个函数

\* 语法：

\* function 函数名([形参1,形参2...形参N]){

\* 语句...

\* }

\* 使用 函数表达式 来创建一个函数

\* var 函数名 = function([形参1,形参2...形参N]){

\* 语句....

\* }

\* 定义一个用来求两个数和的函数

\* 可以在函数的()中来指定一个或多个形参（形式参数）

\* 多个形参之间使用,隔开，声明形参就相当于在函数内部声明了对应的变量

\* 但是并不赋值

\* 在调用函数时，可以在()中指定实参（实际参数）

\* 实参将会赋值给函数中对应的形参

\* 调用函数时解析器不会检查实参的类型,

\* 所以要注意，是否有可能会接收到非法的参数，如果有可能则需要对参数进行类型的检查

\* 函数的实参可以是任意的数据类型

\* 调用函数时，解析器也不会检查实参的数量

\* 多余实参不会被赋值

\* 如果实参的数量少于形参的数量，则没有对应实参的形参将是undefined

\* 创建一个函数，用来计算三个数的和

\*

\* 可以使用 return 来设置函数的返回值

\* 语法：

\* return 值

\*

\* return后的值将会会作为函数的执行结果返回，

\* 可以定义一个变量，来接收该结果

\*

\* 在函数中return后的语句都不会执行

\*

\* 如果return语句后不跟任何值就相当于返回一个undefined，

\* 如果函数中不写return，则也会返回undefined

\*

\* return后可以跟任意类型的值

调用函数

变量result的值就是函数的执行结果

函数返回什么result的值就是什么

\* 返回值可以是任意的数据类型

\* 也可以是一个对象，也可以是一个函数

函数对象()

\* 立即执行函数

\* 函数定义完，立即被调用，这种函数叫做立即执行函数

\* 立即执行函数往往只会执行一次

(function(a,b){

console.log("a = "+a);

console.log("b = "+b);

})(123,456);

\* 函数也可以称为对象的属性，

\* 如果一个函数作为一个对象的属性保存，

\* 那么我们称这个函数时这个对象的方法

\* 调用这个函数就说调用对象的方法（method）

\*

\* 但是它只是名称上的区别没有其他的区别

\* 语法：

\* for(var 变量 in 对象){

\*

\* }

\*

\* for...in语句 对象中有几个属性，循环体就会执行几次

\* 每次执行时，会将对象中的一个属性的名字赋值给变量

\* 作用域

\* - 作用域指一个变量的作用的范围

\* - 在JS中一共有两种作用域：

\* 1.全局作用域

\* - 直接编写在script标签中的JS代码，都在全局作用域

\* - 全局作用域在页面打开时创建，在页面关闭时销毁

\* - 在全局作用域中有一个全局对象window，

\* 它代表的是一个浏览器的窗口，它由浏览器创建我们可以直接使用

\* - 在全局作用域中：

\* 创建的变量都会作为window对象的属性保存

\* 创建的函数都会作为window对象的方法保存

\* - 全局作用域中的变量都是全局变量，

\* 在页面的任意的部分都可以访问的到

\*

\* 2.函数作用域

\* 变量的声明提前

\* - 使用var关键字声明的变量，会在所有的代码执行之前被声明（但是不会赋值），

\* 但是如果声明变量时不适用var关键字，则变量不会被声明提前

\*

\* 函数的声明提前

\* - 使用函数声明形式创建的函数 function 函数(){}

\* 它会在所有的代码执行之前就被创建，所以我们可以在函数声明前来调用函数

\* 使用函数表达式创建的函数，不会被声明提前，所以不能在声明前调用

\* 函数作用域

\* - 调用函数时创建函数作用域，函数执行完毕以后，函数作用域销毁

\* - 每调用一次函数就会创建一个新的函数作用域，他们之间是互相独立的

\* - 在函数作用域中可以访问到全局作用域的变量

\* 在全局作用域中无法访问到函数作用域的变量

\* - 当在函数作用域操作一个变量时，它会先在自身作用域中寻找，如果有就直接使用

\* 如果没有则向上一级作用域中寻找，直到找到全局作用域，

\* 如果全局作用域中依然没有找到，则会报错ReferenceError

\* - 在函数中要访问全局变量可以使用window对象

\* 在函数作用域也有声明提前的特性，

\* 使用var关键字声明的变量，会在函数中所有的代码执行之前被声明

\* 函数声明也会在函数中所有的代码执行之前执行

\* 在函数中，不适用var声明的变量都会成为全局变量

\* 定义形参就相当于在函数作用域中声明了变量

\* 解析器在调用函数每次都会向函数内部传递进一个隐含的参数,

\* 这个隐含的参数就是this，this指向的是一个对象，

\* 这个对象我们称为函数执行的 上下文对象，

\* 根据函数的调用方式的不同，this会指向不同的对象

\* 1.以函数的形式调用时，this永远都是window

\* 2.以方法的形式调用时，this就是调用方法的那个对象

\* 使用工厂方法创建的对象，使用的构造函数都是Object

\* 所以创建的对象都是Object这个类型，

\* 就导致我们无法区分出多种不同类型的对象

\* 创建一个构造函数，专门用来创建Person对象的

\* 构造函数就是一个普通的函数，创建方式和普通函数没有区别,

\* 不同的是构造函数习惯上首字母大写

\*

\* 构造函数和普通函数的区别就是调用方式的不同

\* 普通函数是直接调用，而构造函数需要使用new关键字来调用

\*

\* 构造函数的执行流程：

\* 1.立刻创建一个新的对象

\* 2.将新建的对象设置为函数中this,在构造函数中可以使用this来引用新建的对象

\* 3.逐行执行函数中的代码

\* 4.将新建的对象作为返回值返回

\*

\* 使用同一个构造函数创建的对象，我们称为一类对象，也将一个构造函数称为一个类。

\* 我们将通过一个构造函数创建的对象，称为是该类的实例

\*

\* this的情况：

\* 1.当以函数的形式调用时，this是window

\* 2.当以方法的形式调用时，谁调用方法this就是谁

\* 3.当以构造函数的形式调用时，this就是新创建的那个对象

\* 使用instanceof可以检查一个对象是否是一个类的实例

\* 语法：

\* 对象 instanceof 构造函数

\* 如果是，则返回true，否则返回false

\* 所有的对象都是Object的后代，

\* 所以任何对象和Object左instanceof检查时都会返回true

\* 创建一个Person构造函数

\* - 在Person构造函数中，为每一个对象都添加了一个sayName方法，

\* 目前我们的方法是在构造函数内部创建的，

\* 也就是构造函数每执行一次就会创建一个新的sayName方法

\* 也是所有实例的sayName都是唯一的。

\* 这样就导致了构造函数执行一次就会创建一个新的方法，

\* 执行10000次就会创建10000个新的方法，而10000个方法都是一摸一样的

\* 这是完全没有必要，完全可以使所有的对象共享同一个方法

\* 将函数定义在全局作用域，污染了全局作用域的命名空间

\* 而且定义在全局作用域中也很不安全

\* 原型 prototype

\*

\* 我们所创建的每一个函数，解析器都会向函数中添加一个属性prototype

\* 这个属性对应着一个对象，这个对象就是我们所谓的原型对象

\* 如果函数作为普通函数调用prototype没有任何作用

\* 当函数以构造函数的形式调用时，它所创建的对象中都会有一个隐含的属性，

\* 指向该构造函数的原型对象，我们可以通过\_\_proto\_\_来访问该属性

\*

\* 原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，

\* 我们可以将对象中共有的内容，统一设置到原型对象中。

\*

\* 当我们访问对象的一个属性或方法时，它会先在对象自身中寻找，如果有则直接使用，

\* 如果没有则会去原型对象中寻找，如果找到则直接使用

\*

\* 以后我们创建构造函数时，可以将这些对象共有的属性和方法，统一添加到构造函数的原型对象中，

\* 这样不用分别为每一个对象添加，也不会影响到全局作用域，就可以使每个对象都具有这些属性和方法了

\* 原型对象也是对象，所以它也有原型，

\* 当我们使用一个对象的属性或方法时，会现在自身中寻找，

\* 自身中如果有，则直接使用，

\* 如果没有则去原型对象中寻找，如果原型对象中有，则使用，

\* 如果没有则去原型的原型中寻找,直到找到Object对象的原型，

\* Object对象的原型没有原型，如果在Object原型中依然没有找到，则返回undefined

可以使用对象的hasOwnProperty()来检查对象自身中是否含有该属性

当我们直接在页面中打印一个对象时，事件上是输出的对象的toString()方法的返回值

如果我们希望在输出对象时不输出[object Object]，可以为对象添加一个toString()方法

\* 垃圾回收（GC）

\* - 就像人生活的时间长了会产生垃圾一样，程序运行过程中也会产生垃圾

\* 这些垃圾积攒过多以后，会导致程序运行的速度过慢，

\* 所以我们需要一个垃圾回收的机制，来处理程序运行过程中产生垃圾

\* - 当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象，

\* 此时这种对象就是一个垃圾，这种对象过多会占用大量的内存空间，导致程序运行变慢，

\* 所以这种垃圾必须进行清理。

\* - 在JS中拥有自动的垃圾回收机制，会自动将这些垃圾对象从内存中销毁，

\* 我们不需要也不能进行垃圾回收的操作

\* - 我们需要做的只是要将不再使用的对象设置null即可