

JavaScript基础

JavaScript基础

JavaScript概述

JavaScript到底是什么？

JavaScript 运行在什么地方

JavaScript的注释

JavaScript的调试方法

ECMAScript基础

ECMAScript的语法基础规则

变量标识符

关键字

保留字

JavaScript数据类型

字符串类型String

数字类型Number

关于数字类型NaN

布尔类型Boolean

未定义类型undefined

空类型Null

数据类型检测

数据类型转换

字符串转十进制数值

布尔类型转数字

null与undefined转换为数值

其它类型转字符串

其它类型转布尔值

数字进制之间的转换

十进制转其它进制

其它进制转十进制

语法格式

JavaScript的运算符

一元操作符

递增递减操作符

递增操作符

递减操作符

特殊类型值的递增递减操作

一元加减操作符

一元加法操作符

一元减法操作符

加法操作符

减法操作符

乘法操作符

除法操作符

取余操作符

布尔操作符[逻辑操作符]

逻辑非操作符

逻辑与操作符

逻辑或操作符

相等操作符

全等操作符

不相等操作符

关系操作符

条件操作符

赋值运算符

逗号操作

操作符的优先级

小练习

二进制运算符

按位非 (NOT)

按位与 (AND)

按位或 (OR)

按位异或 (XOR)

左移

有符号的右移

无符号右移

注意事项

JavaScript语句

if...else

for循环语句

循环的本质

while循环语句

do...while循环语句

break与continue

嵌套的循环

label语句

switch语句

练习

函数

函数的定义

函数的调用执行

函数的检测

函数作用域 【重点】

函数的参数

形参与实参

函数的重载

函数的返回值

函数的本质 【重点】

递归函数

带参数与返回值的递归函数 【难点】

注意事项

练习

JavaScript数组

数组的概念

数组的定义

通过 `new Array()` 的方式来创建

通过 `[]` 字面量来定义

数组的检测

通过 `instanceof` 关键字来完成检测

通过 `Array.isArray()` 来检测

数组的取值与赋值

JS数组与其他语言数组的区别

数组的遍历

数组的常用属性及方法

课堂练习

JavaScript函数（二）

函数的定义

函数的参数

`arguments`实参数组

`arguments`解析

函数的调用

立即执行函数

函数表达式的执行

闭包调用

匿名函数

回调函数 【重点】

匿名函数的回调

数组的高级方法

迭代方法

forEach方法

map方法

filter方法

some方法

every方法

注意事项

归并方法

reduce方法

reduceRight方法

排序方法

数组方法的总结

课堂练习

课后练习

二维数组

二维的数组的定义

二维数组的特殊情况

二维数组的取值与赋值

二维数组的应用点

案例与练习

面向对象（一）

对象的概念

对象创建

通过 Object 来创建

通过字面量 {} 来创建

对象属性的调用方式

通过 . 的方式来调用

通过 [] 的方式来调用

变量做为属性名

使用工厂模式创建对象

使用构造函数创建对象 【重点】

构造函数与普通函数的区别

保证函数以构造的方式执行

对象中的方法

基础篇总结

面向对象（二）

`delete`关键字

`Object.defineProperty`来定义属性

 数据属性

 访问器属性

`Object.defineProperties`来定义属性

遍历对象的属性

`for...in`遍历对象

 通过`Object.keys()`遍历对象

`Object.getOwnPropertyNames()`方法遍历

构造函数与特殊属性结合

获取对象属性的描述信息

判断对象是否具备某一个属性

 通过`in`关键字来检测

 通过`hasOwnProperty()`来完成检测

练习

JavaScript面向对象（三）

`this`关键字

对象中的`this`

构造函数里面的`this`

函数调用方式的不同决定`this`指向

补充

 通过`call()`的方式来调用

 通过`apply()`的方式来调用

练习

面向对象（四）

通过`call/apply`来继承

通过原型继承【重点，难点】

理解原型

通过原型继承(简单版)

构造函数的原型继承【过渡】

构造函数的原型继承2 【过渡】

构造函数的原型继承3 【标哥推荐版】

面向对象的补充点

关于 `toString()` 方法

关于 `valueOf()` 方法

练习

对象在内存当中的存储

数据的内存存储

对象的拷贝

对象的浅拷贝

对象深拷贝

简单对象的深拷贝 【1】

简单数组深拷贝 【2】

通过 `Object.assign()` 拷贝简单对象 【3】

复杂对象的深拷贝 【终极】

跨对象调用方法 【重点】

执行上下文栈

什么是执行上下文

执行上下文堆栈

执行上下文的建立过程

执行上下文建立阶段以及代码执行阶段

作用域

自由变量与作用域链

作用域与执行上下文

案例

变量会随着执行环境而建立吗

常用内置对象

Math对象

Date对象

延时调用与循环调用

延时调用

取消延时调用

循环调用

取消循环调用

正则表达式对象

包装对象

String对象

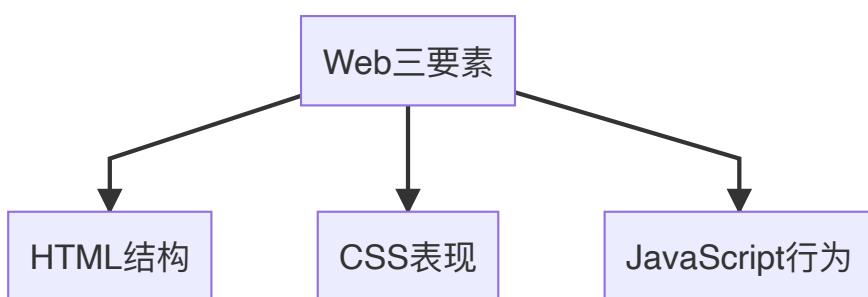
Number对象

Boolean对象

练习

JavaScript概述

首先回顾一个点，Web三要素



1. 一个页面显示什么是由HTML决定的
2. 一个页面上面的元素怎么显示，显示成什么样式是表现形式决定（由CSS样式决定）
3. 一个页在上面具备什么功能则是由JS决定

JavaScript到底是什么？

当看到这个单词的时候，很多同学可能就会联想到另一个单词 **Java**，它与我们的 **JavaScript** 是一个完全不同的语言，本身不存在任何包含或隶属关系，它是一门独立的弱类型的脚本语言

JavaScript语言是经过了多年了的展，期间经过了几次脚本语言的更新换代，才体现出了JavaScript的特点

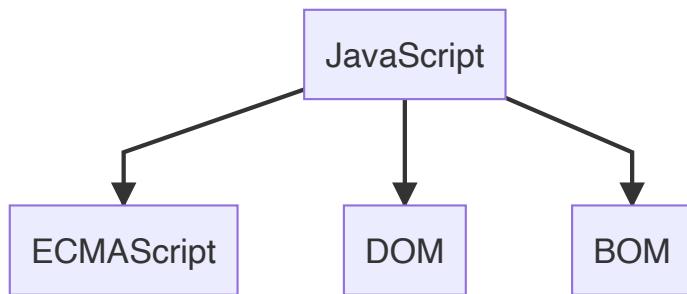
在网页当中，脚本的语言的发展经过了几个阶段，我们现在了解一下

1. **asp+VBScript** 这是微软推出第一套网页开发者的技术，本来的目的是为了更好的开发网页，但是这套脚本语言又可以同时运行在window的电脑上面，这样就非常不安全，如果某一个网站存在安全漏洞，则有可能会被注入VBS病毒，这样就会极大的提高病毒的传播面

同时这套技术对IE浏览器的支持度非常高，而IE被慢慢淘汰以后，这个套技术也被淘汰了

2. **J#+JScript** 这是微软推出的第二套网页开发技术，用于弥补之前 **VBS** 下面的bug及不足， **J#** 因为侵权 **Java** 所以停止了， **JScript** 因为侵权 **ECMAScript** 所以被停止了
3. **NetScape** 基于 **Java** 的特性及语言特点，并且在遵守 **ECMAScript** 的规范下面推出了 **JavaScript** 脚本语言，这就是目前我们所使用的 **JS** 语言

JavaScript是一门特殊的语言，它里面一共有3大部分



1. **ECMAScript** 简称ES，目前我们主要学习的版本是 **ES5/ES6**，**ECMAScript** 规定当前语言里面的语法，关键字，流程控制，运行符，面向对象，数据类型等
2. **DOM** 全称是 **document object model** 文档对象模型，JS毕竟是一门脚本语言，它需要运行在网页里面，所以它主要的任何就是操作网页，**DOM** 这个技术点就是将JS与网页结合，能够让JS更快更高效的操作网页，实现网页的交互效果
3. **BOM** 全称 **browser object model** 浏览器对象模型，这一种技术就是让JS去操作浏览器，通过这种技术我们可以让JS去调用设备的硬件信息，如实现缓存，调用蓝牙，GPS，网格状态，摄像头及麦克风等信息

JavaScript 运行在什么地方

JavaScript作为一种脚本语言，它不可能独立运行(**nodejs** 除外)，它依附于网页的存在，它的代码可以写在三个地方

行内的写法

```
<button type="button" onclick="alert('标哥哥真帅气')">按钮</button>
```

这一种写法相当于 **CSS** 里面的行内样式的写法，把 **JS** 代码写在某一个属性的后面

内部代码块

```
<script>
    alert("吃了没，世界");
</script>
```

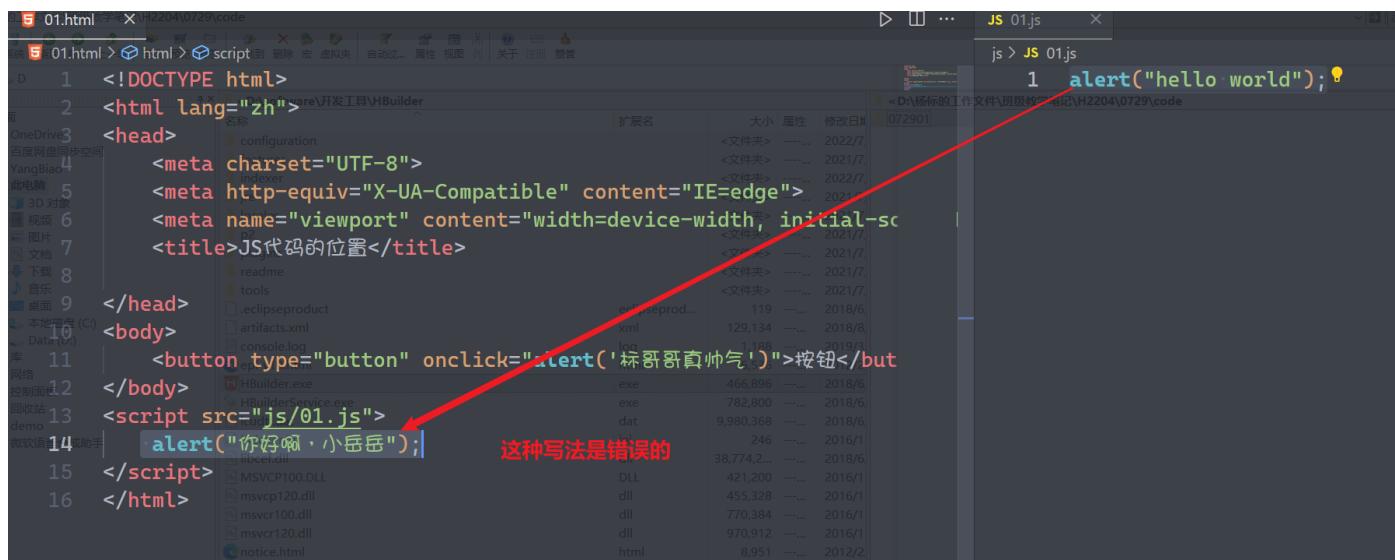
这一种写法与 **CSS** 里面的内部样式块很相似，它必须写在一对 **script** 的标签里面

这里要注意，正常情况下， **script** 标签应该写在 **body** 标签结束的位置，而不是写在 **head** 标签里面

写在单独的JS文件里面

```
<script src="js/01.js"></script>
```

注意事项



如果一个 **script** 标签引入外部的JS文件，则不能在这个标签的内部去写代码

JavaScript的注释

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JS的注释</title>
    <style>
        /* 这是CSS的注释 */
    </style>
</head>
```

```

<body>
    <!-- 这是HTML的注释 -->

</body>
<script>
    // 单行注释

    /*
        这是代码块的注释
        可以注释多行代码
    */

</script>
</html>

```

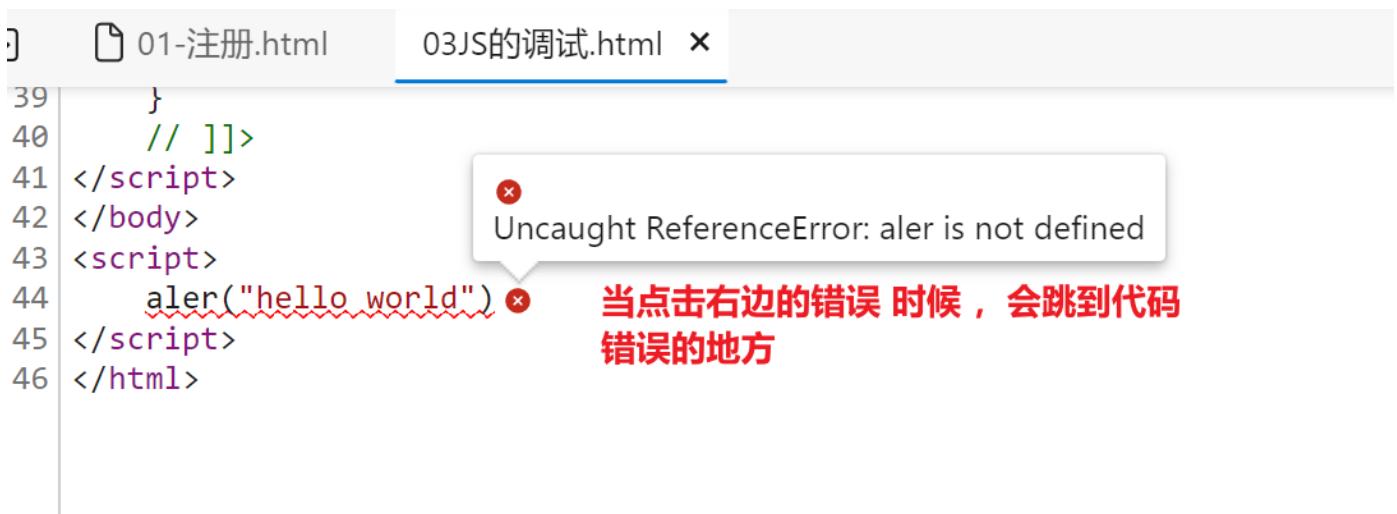
在一个页面里面，不同的地方要使用不同的注释方式，千万不要搞错了

JavaScript的调试方法

之前在讲HTML与CSS的时候，如果代码写错了，页面上面最多是没有效果的，但是如果是JS代码，则要注意，代码出现错误以后是会在浏览器的控制台表现出来的



它会在控制台告诉我们错误在什么地方



ECMAScript基础

JS分为三部分，其中 **ECMAScript** 规定了这门语言的语法基础，数据类型，流程控制，运行符，类型转换，函数，面向对象等一系列的基础知识，它是入门点

ECMAScript的语法基础规则

```
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>基础规则</title>
    <STYLE>
        INPUT{
            BACKGROUND-COLOR:RED;
        }
    </STYLE>
</head>
<body>
    <input type="text">
    <INPUT TYPE="TEXT" />
</body>
<script>
    alert("陈怡静, 刘诗霞, 芦伊菲");
    // ALERT("陈怡静, 刘诗霞, 芦伊菲");    这里不行
</script>
</html>
```

1. **JS** 代码的基础单位是行，一行代码结束以后要以分号结尾 【如果省略了这个分号也是可以的，但是不推荐】
2. **JS** 是一门严格区分大小写的语言，它与之前的 **HTML**、**CSS** 不一样
3. **JS** 编程里面，所有的符号都是英文状态下面的符号

变量标识符

变量其实就是一个可以变化内容或值的一个符号，在之前的CSS里面，我们已经接触过了变量的概念

```
:root{  
    --primaryColor:red;  
}  
.box{  
    color:var(--primaryColor);  
}
```

通过上面的过程，我们可以大概知道变量是，它定义的过程和使用的过程是什么样式

在JS里面也是一样的，在JS当中定义变量有一个特殊的**关键字** var 来实现的

var 的全称叫 variable，英文单词就是变量的意思

```
var a;
```

上面的代码就是定义了变量 a。这行代码的意思就是告诉浏览器，我 var 后面跟着的这个 a 它是一个变量,但是这个变量没有赋值，没有赋值的变量叫未初始化的变量

```
var a = 18;  
//我们可以在定义变量的时候，直接给这一个变量赋一个初始的值，这个赋值始值的过程叫 变量初始化
```

上面的代码其实也可以分2部去写

```
var a; //定义变量  
a = 18; //赋值
```

通过上面的代码我们已经知道关键字 var 就是用来定义变量的，但是这里有2个细节需要同学样注意

1. 在 JS 里面，定义所有的变量都使用同一个关键字 var，这一点与其它的语言不一样

```
var a = 18;  
var b = "标哥哥";  
var c = 19.9;  
var d = true;
```

JS它是一门弱型的语言【弱类型就是在定义变量的时候，不去强调变量的类型，同时变量的类型也是可以后期进行相互转换的】

```
//下面是java的代码  
int a = 18;  
String name = "标哥哥";  
char sex ='男';  
float money = 11.2f;
```

2. 变量的命名一定要规范

- 所有的变量名应该遵守见名知意
- 所有的变量命名应该是驼峰命名法，驼峰命名法就是第一个单词首字母小写，后面单词的首字母大写
- 所有的变量命名应该避开关键字与保留字
- 变量只能是字母，下划线或 \$ 开头

```
//定义年龄使用age 见名知意  
var age = 18;  
//定义姓名使用username 见名知意  
var userName = "标哥";  
// 学生信息列表的变量  
var studentInfoList;  
  
// var var;    错误的变量命名 使用了关键字做变量名  
// var for;    错误的变量命名 使用了关键字做变量名  
  
//var 1age;    错误的变量命名 使用了数字开头  
// var user-name;  错误的变量命名 使用了特殊字符做变量名  
// var #age;    错误的变量命名 使用了特殊字符做变量名  
  
var $age = 20;  
var user_age = 20;  
var _age = 20;
```

关键字

关键字是编译系统里面为了实现一个或多个功能所特定命名的一个关键词

ECMA-262 描述了一组具有特定用途的关键字，这些关键字可用于表示控制语句的开始或结束，或者用于执行特定操作等。按照规则，关键字也是语言保留的，不能用作标识符。以下就是 ECMAScript 的全部关键字（带*号上标的是第 5 版新增的关键字）：

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger*	function	this	with
default	if	throw	
delete	in	try	

保留字

保留字也是关键字，它在当前的版本中没有使用，但是可能会有后面的版本里面去使用

ECMA-262 还描述了另外一组不能用作标识符的保留字。尽管保留字在这门语言中还没有任何特定的用途，但它们有可能在将来被用作关键字。以下是 ECMA-262 第 3 版定义的全部保留字：

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

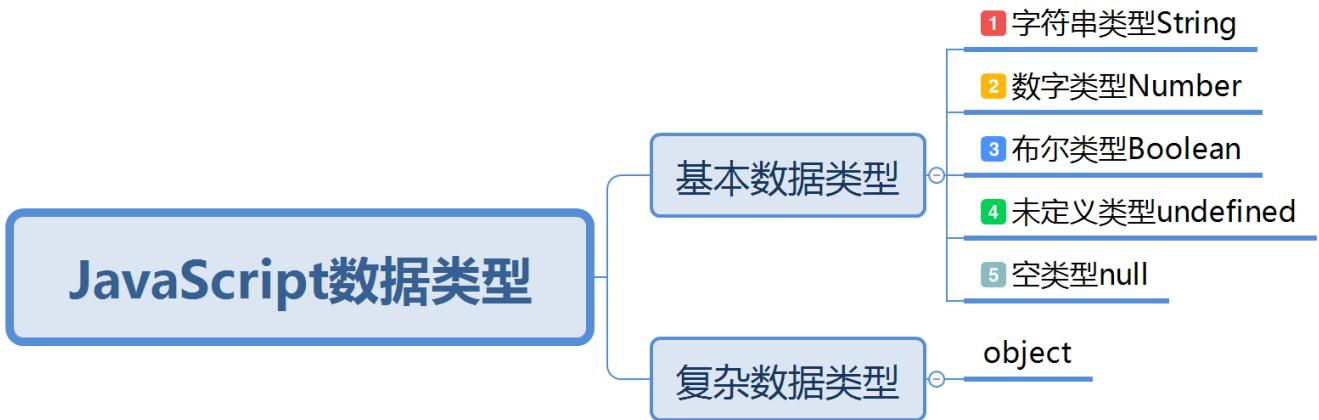
现在我们已经知道变量的定义是使用关键字 `var` 了，那么我们就来通过这个关键来带出新的知识点

```
var teacherName = "标哥";
var age = 18;
var height = 180.5;
var money = null;
```

我们可以看到变量后面跟着的这些是不一样的，要弄清楚上面的问题，我们就需要学习 `ECMAScript` 当中的数据类型

JavaScript 数据类型

在 JavaScript 里面根据不同的场景需求，我们要使用一些特殊的数据来去描述，如数字，如逻辑判断的真或假，如小数等，这些都称之为数据类型



在JS的数据类型里面，我们把数据类型分为2个大类

1. 基本数据类型

基本数据类型描述数据的根本类型，在JS里面一定分为很多小的类

2. 复杂数据类型【引用类型】

字符串类型String

字符串是所有编程语言当中最常见的数据类型，同时在JS里面也是最安全的数据类型，所有的数据类型都可以转换成这种类型，它也非常好识别（所有使用引号包裹的都是字符串）

```
var a = "标哥哥";
var b = "hello";
var c = "100";
```

上面变量赋的值全都被引号包裹了，但是要注意，符号全都是英文的

```
var d = "标哥哥";
```

上面的代码就是错误的，它使用了中语文的符号，编程语言里面所有的符号应该都是英文的

在字符串类型里面所有被引号包裹的就是字符串，没有单引号与双引号的区别，**只要是成双成对的出现就可以了**

```
var sex = '男';           //正确
var stuName = '黄相立';   //正确
var girlFriend = '赵金麦'; //这是不对的，因为引号不是成对的出现的
```

现在请同学样来判断一下，下面是不是字符串

```
var a = 18;           //数字
var b = "18";         //字符串
var c = '18';         //字符中
var d = c;            //相当于 var d = '18'，它也是字符串
var e = 'c';
//第一个a没有打引号，所以不是字符串
//最后的d是接收了c的值，而c的值是'18'，所以变量d最后的值也是'18'
```

数字类型Number

数字类型是JavaScript当中用于表述数字的，最简单的数字类型就是平时使用的10进制数据类型（后面我们还会讲到8进制，16进制及2进制）

后面会讲到2进制及进制转换，所以这里只写10进制

同时JS里面的数字类型与其它的语言不一样，它不区分小数，整数，长整型，短整型，单精度，双精度等，所有的数字全称之为 **Number** 类型

```
var money = 0;          //整数
var money1 = 25;         //数字
var money2 = 35.5;       //小数
var money3 = -100;        //负数
var a = "999";           //只要加了引号，就是字符串类型
```

现在再列举一下16进制的情况 [【了解】](#)，所以的16进制的数字都是以 **0x** 开头的16进制方式来赋值

```
var a = 255;             //十进制
var b = 0xff;             //0x一定是16进制 后面的ff代表提16进制的值
```

16进制的值一旦赋值结束以后，系统会自动的转换成10进制的值

所有的十进制数字都是由0~9构成的，但是仍然有一些特殊的数字要记住

关于数字类型NaN

NaN 是一个非常特殊的数字类型，全称叫 **not a number**，这个值代表本来经过运算以后应该得到一个数字的，结果又得不到数字出现了意外，就会显示这个值

```
var a = 100;
var b = "标哥";
a - b; //这个时候的结果就是NaN
//减法操作以后按结果应该是一个数字，结果我又得不到数字，出现了意外，这个时候就会得到NaN
```

布尔类型Boolean

布尔类型是绝大多数编程语言里面都具备的数据类型，这个类型下面只有2个值，分别是 `true` 和 `false`，用于在编程过程当中判断条件真或假

```
var isTeacher = true;
var isGirl = false;
var x = 1;
var y = 2;
var z = (x < y); //这条件是成立的，所以它的结果是一个true
```

请同学们分析一下下面的数据类型是什么

```
var a = "true";
var b = false;
var c = 'false';
```

变量a被引号包含，所以它是字符串，变量b是 `false` 所以是Boolean类型，变量c被单引号包裹，也是字符串

未定义类型undefined

未定义类型是 `JavaScript` 当中独有的一种特殊类型，它指的是一个变量定义了以后没有初始化或在初始化的时候给了一个 `undefined` 的值

```
var a = true; //boolean
var b = 123; //number
var c = "hello"; //string
```

上面的3个变量都有数据类型，原因是因为它们都有值，但是如果一个变量定义了以后没有值，会是什么类型

```
var d; //这个时候就只定义了变量，而没有初始化，它就是一个未定义的类型 undefined
```

还有一种情况也可以是未定义类型，就是直接赋一个 `undefined` 的值

```
var e = "赵今麦";           //string类型  
e = undefined;           //undefined类型
```

所以 **undefined** 代表没有的意思

空类型Null

null与我们上面所学习的 **undefined** 非常相似，它也代表没有，但是2个类型有本质性的区别

```
var a ;           //不赋值它就是undefined类型  
var b = null;           //空类型
```

b变量null相当于一个房子，这个房子它是一个空房子,a是undefined，它代表空，它连房子都没有

场景一对话：undefined的情况对象

杨标：“颜一鸣，来，我今天晚上给一个大大的礼物给你”

颜一鸣听了以后非常高兴，跑跑跳跳就回到座位

李心悦上线了

问：“嘿，小伙子，刚刚标哥叫你过去干什么”

颜一鸣神神秘秘的说道：“标哥说晚上给一个礼物我”

李心悦就问：“啥礼物，多少钱？”

颜一鸣听后说道：“我也不知道，标哥说晚上给我”

场景二对话： null 的情况的对话

杨标：“颜一鸣，来，小伙子，这里有个盒子，盒子里面装的是你今天晚上的惊喜礼物”

抱着这个盒子，颜一鸣欢乐的走向自己的位置

李心悦又上线了

“哟，不错啊，又有东西过来了，你和标哥啥关系，怎么老有礼物给你，快看看，到底是个啥？”

颜一鸣很好奇的就打开了这个盒子，然后发现，这个盒子是空的

颜一鸣大失所望，然后就开始慢慢问候标哥了

上面的2个场景对话就充分的说明了 **null** 与 **undefined** 的场景，而给颜一鸣的盒子就相当于是内存当中的空间，**null** 是会在内存里面占据一个空间，只是这个空间里面没有放任何东西，而 **undefined** 在内存里面是没有空间的

数据类型检测

JavaScript有5种基本数据类型，如果一个变量赋了值以后就有了初始的数据类型，那么这个数据类型是否会发生变化

```
var a = "hello";
a = "你好";
```

上面的代码里面，我们定义了一个变量，同时对这个变量进行了初始化，它的初始数据类型是 **String** 字符串类型，后面把 **a** 重新赋值为“你好”的时候它还是字符串类型

```
var a = "hello";      //字符串
a = 100;             //这里不会报错，现在赋值数字了，它就是number字符串类型
```

从上面的代码上面我们可以看到， **JavaScript** 的数据类型是可以发生改变的，这一种现象我们叫 **弱类型机制**

在 **JS** 当中变量是没有固定的数据类型的，它的数据类型是由后面的值来决定的，我们赋什么类型值，它就是什么数据类型

这一种现象与其它的编程语言是不一样的， **java** 与 **c** 等都是强类型机制，一旦确定了变量的数据类型，就不可能更改

```
int a;
double b;
float c;
String d;
boolean f;
```

正是因为JS的数据类型在更改，所以如果我们想确定一个变量的数据类型，可以通过**数据类型检测**来完成

在JS里面，基本数据类型的检测使用的是关键字 `typeof`

```
var a = 1;
var b = "你好";
var c = false;
var d = undefined;
var e = null;

// 现在下一个检测
typeof a; // 'number'类型
typeof b; // 'string'类型
typeof c; // 'boolean'类型
typeof d; // 'undefined'类型
typeof e; // 'object'类型
```

在前面的4个数据检测的时候，得到的结果与我们预期的结果是一致的，只有 `null` 在检测的时候得到的结果是 `object` 对象类型

基本数据类型检测以后得到的结果有以下几种情况

1. `number` 数字类型
2. `string` 字符串类型
3. `boolean` 布尔类型
4. `undefined` 未定义类型
5. `object` 对象类型，这个结果比较特殊，我们的 `null` 检测出来会是这个类型【单独对待】

数据类型检测的语法

```
typeof 变量名;
```

除了上面的方式以外，还可以使用下面的方式也行

```
typeof(变量名);
```

上面2种写法它的结果是一样的，只是写法不一样而已

顺带着说一句

```
typeof NaN; // 'number' 类型

var a = 123;
typeof typeof a;

// 分析过程
// typeof(typeof(a)); // 'string'
// typeof a // 'number'

// typeof 'number' // 结果 'string'
```

数据类型转换

JavaScript里面有5种基本数据类型，这5种基本数据类型之间是可以发生相互的转换的

字符串转十进制数值

```
var a = "999";
var b = "9a9";
var c = "99.99";
var d = "a99";
var e = "99.99.88";
var f = "标哥";
var g = "-99";
var h = "";
```

这里我们定义了7个字符串的变量，那么如何将上面的这些字符串转换成相应的数字 **Number** 类型呢？

String 字符串转 **Number** 有三个方法

1. 通过 **Number(value?: any): number** 方法来完成转换

通过这种方式的转换，只能实现合法的数字字符串转换

```
var a1 = Number(a);          //得到结果999, 它是一个number
var b1 = Number(b);          //得到结果NaN, 转换失败, 它也是一个number类型
var c1 = Number(c);          //得到结果99.99 它是一个number类型
var d1 = Number(d);          //得到结果NaN
var e1 = Number(e);          //得到结果NaN
var f1 = Number(f);          //NaN
var g1 = Number(g);          //得到结果-99 它是一个number类型
var h1 = Number(h);          //得么结果0
```

同时我们再看下面的几种情况

```
var h = "0001";
var h1 = Number(h);      //1

var aa = "0x0001";
var aa1 = Number(aa);    //1
```

`Number` 方法的转换本质上就是直接去掉了字符串的引号，然后再看它是否是一个合法的数字

后期所有的隐式类型转换全都默认使用 这个方法进行转换

2. 使用 `parseInt(value: string, radix?: number): number` 进行转换

这个方法只能将字符串转换成整数

```
var a1 = parseInt(a);    //999
var b1 = parseInt(b);    //9 尝试进行转换, 成功就保留, 失败了就不要了
var c1 = parseInt(c);    //99
var d1 = parseInt(d);    //NaN
var e1 = parseInt(e);    //99
var f1 = parseInt(f);    //NaN
var g1 = parseInt(g);    //-99
```

`parseInt` 的转换是逐个转换，成功就保留，失败就停止，如果一开始就失败了则返回 `NaN`。同时要注意，它的结果成功以后也只会是一个整数

3. 使用 `parseFloat(value:string):number` 进行转换

这个结果与上面的结果非常相似，只是会保留小数点以后的内容

```
var a1 = parseFloat(a); //999
var b1 = parseFloat(b); //9
var c1 = parseFloat(c); //99.99
var d1 = parseFloat(d); //NaN
var e1 = parseFloat(e); //99.99
var f1 = parseFloat(f); //NaN
var g1 = parseFloat(g); //-99

//特殊情况
var x = parseFloat("99.0"); //99
var y = parseFloat(".9"); //0.9
```

布尔类型转数字

这里指的是布尔类型转10进制数字

布尔类型如果想转 `number` 类型，只能通过 `Number(value?: any): number` 来进行

```
var a = true;
var b = false;

//如何将上面的2个值转换成Number类型

var a1 = Number(a); //结果1
var b1 = Number(b); //结果0
```

? 思考：为什么这里不使用 `parseInt/parseFloat` 去转换

null与undefined转换为数值

```
var a = null;
var b = undefined;

// 想转数字，到底什么方法？
// Number(value?: any): number
// parseInt(value: string, radix?: number): number
// parseFloat(value:string):number
```

通过分析以后，我们得到应该是要通过 `Number` 去转换

```
var a1 = Number(null); //0
var b1 = Number(undefined); //NaN
```

其它类型转字符串

字符串是JS里面最根本的数据类型，所以的数据类型都可以转换为字符串

```
var a = 1;
var b = false;
var c = null;
var d = undefined;
var e = 3.14;
var f = -10;
var g = NaN;
```

上面的所有变量都是非字符串类型的，那么我们现在要将这些数据转换成字符串，怎么办呢？

字符串的转换非常简单，并且方法还非常多

1. 使用 `String(value?:any):string` 方法来进行转换

```
var a1 = String(a);           // "1";
var b1 = String(b);           // "false";
var c1 = String(c);           // "null";
var d1 = String(d);           // "undefined";
var e1 = String(e);           // "3.14";
var f1 = String(f);           // "-10";
var g1 = String(g);           // "NaN";
```

2. 直接将这个变量加上一个空字符串

```
// 直接加空字符串
var a1 = a + "";           // "1";
var b1 = b + "";           // "false";
var c1 = c + "";           // "null";
var d1 = d + "";           // "undefined";
var e1 = e + "";           // "3.14";
var f1 = f + "";           // "-10";
var g1 = g + "";           // "NaN";
```

这个特性非常重要，在JS里面，执行的是字符串优先

3. 通过 `toString()` 的方法来完成，这一种方法只适用于非 `null` 与非 `undefined` 的数据类型

```
// 通过toString来转换
var a1 = a.toString();           // "1";
var b1 = b.toString();           // "false";
// var c1 = c.toString();         // 直接报错, 因为null不具备 toString()的
方法
// var d1 = d.toString();         // 直接报错, 因为undefined不具备
toString()的方法
var e1 = e.toString();           // "3.14";
var f1 = f.toString();           // "-10";
var g1 = g.toString();           // "NaN";
```

字符串的转换有3种方法，经过对比我们发现前2种方法的转换是没有任何限制条件的，后面的 `toString` 方法则不能在 `null` 和 `undefined` 的下面使用

其它类型转布尔值

重点中的重点

在JS所有的数据类型里面，只有6个明确的 `false` 条件，它分别是 空字符串“”，
`0, false, null, undefined, NaN`

这一个转换过程非常麻烦，因为如果布尔类型的结果不明确，那么我们后面在学习流程控制的时候就会有问题，如 `if...else, for, while, do...while`

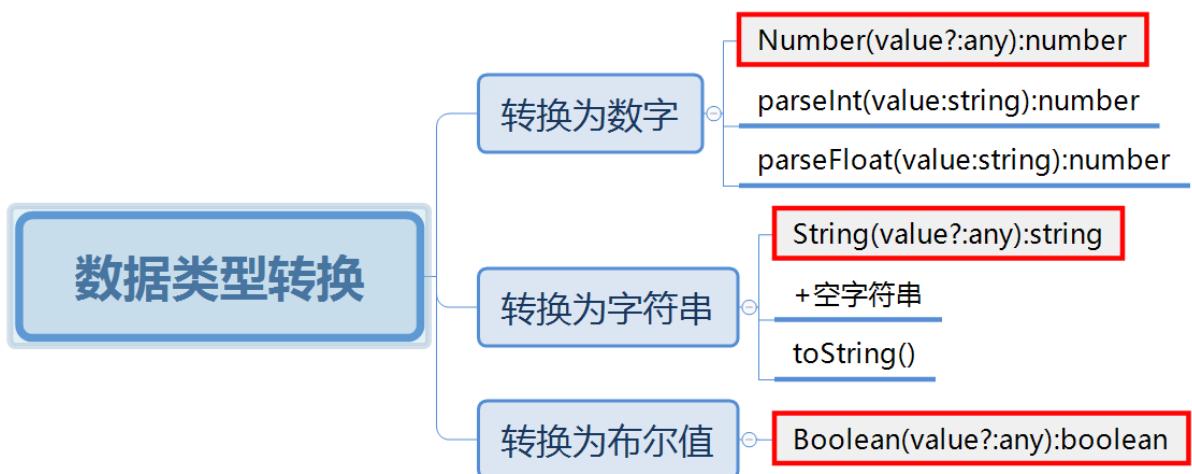
在JS里面，如果要将其它类型的值转换成布尔类型，则优先使用 `Boolean(value?:any):boolean` 这一个方法来进行

```
var a = 1;
var b = 0;
var c = "true";
var d = "false";
var e = "";
var f = null;
var g = undefined;
var h = NaN;
var i = 123;
var j = "标哥哥";

// 上面的所有值都要通过Boolean去转换
```

在上面的变里里面，我们把所有的数据类型基本上都涵盖了，现在将它们转换成布尔类型，看结果是什么

```
var a1 = Boolean(a);      //true
var b1 = Boolean(b);      //false
var c1 = Boolean(c);      //true
var d1 = Boolean(d);      //true
var e1 = Boolean(e);      //false
var f1 = Boolean(f);      //false
var g1 = Boolean(g);      //false
var h1 = Boolean(h);      //false
var i1 = Boolean(i);      //true
var j1 = Boolean(j);      //true
```



所有打了红色标记的，都是后面隐式数据类型转换所使用的默认方式

数字进制之间的转换

之前我们在学习 `Number` 数据类型的时候，我们只是对10进制的数值做了学习，其它数字还可以转换成其它的进制，也可以将其它的进制转换成10进制，现在我们来看看它如何操作

十进制转其它进制

十进制转其它进制，只能转到2~36

```
var a = 37;
var b = 18;
```

上面的2个数就是10进制的数，如何将这2个数转换成2进制呢

在大学里面学习的时候，老师都教过我们可以使用除2取余法来进行

$$\begin{array}{r} 2 \overline{)37} & | \\ 2 \overline{)18} & \square \\ 2 \overline{)9} & | \\ 2 \overline{)4} & \square \\ 2 \overline{)2} & | \\ & | \\ & 1, 0, 5, 10, 1 \end{array}$$

上面的结果是手动算出来的，但是我们程序员也可以通过程序来计算

在数字类型 `Number` 里面，有一个方法叫 `toString(radix?:number)`，在数字类型转进制的时候，我们需要调用这个方法，其中 `radix` 代表进制基数，也叫目标进制

```
var a1 = a.toString(2);           //二进制的'100101'  
var b1 = b.toString(2);           //二进制的'10010'  
var c1 = c.toString(2);           //二进制的'1000001'
```

所以10进制转换成其它进制是非常简单的只需要将调用 `toString(目标进制)` 就可以了，如果不写目标进制，则默认转换成10进制

```
toString(目标进制);    //不填目标进制就是10进制
```

这个方法不仅可以转换成二进制，还可以转换成任何其它的进制都可以

```
var d = 255;  
d.toString(16);           //十六进制的"ff"
```

其它进制转十进制

如果原来的进制数是0，则会直接忽略

```
var a = "10010";           //这是2进制的10010  
var b = "1001001";         //这是2进制的1001001
```

如何将上面的2个数转换成10进制呢

在大学里面，我们也学过了一个手动转换的规则叫**指数相加法**

1	0	0	1	0
---	---	---	---	---

$$(2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0)$$

$$16 + 0 + 0 + 2 + 0$$

上面的图就是一个手动计算的过程，但是我们也可以通过程序来计算

其实转10进制的方法我们也学习过了，它主要是使

用 `parseInt(value:string, radix:number):number` 来实现的，虽然说我们之前已经学习过了这个方法，但是我们并没有使用第二个参数，这个方法后面的第二个值代表这个要转换的数原来的进制

```
parseInt(原来的值, 原位的进制); //如果不填，则默认就是10
```

现在有了上面的代码基础以后，我们可以试着来进行相关的操作

```
var a1 = parseInt(a,2); //18  
var b1 = parseInt(b,2); //73
```

现在请同学们看一下下面这个代码会是什么结果

```
var c1 = parseInt("ff"); //转换失败，得到结果NaN  
var d1 = parseInt("ff",16); //转换成功，得到结果255  
var e1 = parseInt("3",2); //转换失败，得到结果NaN  
var f1 = parseInt("3",3); //转换失败，得到结果NaN  
var g1 = parseInt("3",0); //转换成功，忽略0进制，直接使用10进制，结果是3
```

语法格式

```
0  var i = '你可' ,  
1  var g = "-99";
```

这里我们定义了7个字符串的变量，那么如何将上面的这些字符串转换成相应的数字 Number 类型呢？

String 字符串转 Number 有三个方法

1. 通过 `Number(value?: any): number` 方法来完成转换

可以有，也可以没有

放进去的值

结果是什么类型

放进去值的类型

JavaScript的运算符

JavaScript里面的运算符是一个承上启下的知识点，它对接后面的语句流程控制，承接上面数据类型转换

JavaScript中有很多运算符，请不要将这些运算符与其它编程语言相比较，如 C/C++/Java 等，它们这些语言都是强类型语言，而我们是弱类型语言（弱类型语言会存在隐式类型转换）

一元操作符

只能操作一个值的操作符我们叫一元操作符。一元操作符是ECMAScript当中最简单的一种操作符。一元操作符的结果一定是一个Number类型

递增递减操作符

递增操作符

```
var a = 3;  
var b = 3;  
  
a = a + 1; //4  
b++; //4
```

经过上面的对比，我们发现 `a=a+1` 与 `b++` 所得到的结果是一模一样的。这就是递增操作符。

递增操作符的本质是指在自己的原来的数值上面去加1，递增操作符的符号是 `++`，这个符号可以放在变量的前面，也可以放在变量的后面

思考：放在前面与放在后面有什么区别？看下面代码

```
var a = 3;  
var b = 3;  
  
a++;  
++b;  
console.log(a); //4  
console.log(b); //4
```

上面的代码经过运行以后，我们发现2个变量 `a` 与 `b` 最终的结果都是4！难道符号在前与符号在后是一样的吗？

符号在前面与符号在后面是有本质性的区别的，我们现在把上面的代码改变一下

```
var a = 3;  
var b = 3;  
  
console.log(a++); //3  
console.log(++b); //4
```

1. 符号在后，先使用这个变量，使用完以后，再去变化这个变量
2. 括号在前，先将这个变量变化以后，再去使用这个变量

练习：根据上面的特点我们来完成下面的一个小练习

```
var a = 11;  
var b = 12;  
var c = 13;  
  
a++; //11  
//12  
++a; //13  
//13 13 13 15  
var d = a++ + ++b + c++ + ++a;  
  
//请说出a,b,c,d的值各是多少?  
//a:15  
//b:13  
//c:14  
//d:54
```

递减操作符

递减操作符与上面的递增操作符原理是一样的

```
var a = 8;  
var b = 8;  
  
a--;  
b = b - 1;  
console.log(a);  
console.log(b);
```

递减操作符就是在原来的值上面去减1，它的操作符号是 `--`，同样，这个符号也可以放在变量的前面，还可以放在变量的后面

```
var a = 8;  
var b = 8;  
  
console.log(a--); //8  
console.log(--b); //7
```

特殊类型值的递增递减操作

递增操作与递减操作不仅仅可以适用于数值，还可以在其它的数据类型下面使用，这一点是JS这种弱类型编程语言独有的特点

```
var a = "1";  
var b = "hello";  
var c = "";  
var d = true;  
var e = false;  
var f = null;  
var g = undefined;  
var h = NaN;
```

上面的值都不属于我们的数字类型，这个时候当他们进行递增递减操作的时候会有什么情况呢？

操作原则

1. 一元操作符的结果一定是一个 `Number` 类型
2. 如果是非数字执行的递增与递减操作，则先将这个值进行 `Number()` 的转换
3. `NaN` 不参与运算，即使参与运算，结果也一定是 `NaN`

4. 符号在后，先使用自己，再变化自己；符号在前，先变化自己，再使用自己

一元加减操作符

一元加减操作符是JS里面一种特殊的操作符，它相当于执行了 **Number** 的类型转换，它的结果也一定是Number类型

```
var a = "01";
var b = "1.1";
var c = "z";
var d = false;
var e = 1.1;
var f = NaN;
var g = null;
var h = undefined;
```

我们现在就将这8个变量执行一元加减操作符

一元加法操作符

```
a = +a;           //1
b = +b;           //1.1
c = +c;           //NaN
d = +d;           //0
e = +e;           //1.1
f = +f;           //NaN
g = +g;           //0
h = +h;           //NaN
```

一元减法操作符

```
a = -a;          //-1
b = -b;          //-1.1
c = -c;          //NaN
d = -d;          //0
e = -e;          //-1.1
f = -f;          //NaN
g = -g;          //0
h = -h;          //NaN
```

操作原则

1. 结果一定是 **Number** 类型

2. 非数值的直接通过 `Number()` 去转换
3. `Nan` 不能于计算
4. 如果是一元减法操作，则通过 `Number()` 转换以后，再乘以 `-1`

加法操作符

加法操作符是我们使用 `+` 来表示，加法操作符的结果不一定是 `Number` 类型

因为JS是弱类型的缘故，所以在执行加法运算的时候，不一定是数字在相加，还有可能是其它类型的值在加相

```
console.log("hello" + "world");      // "helloworld"
console.log(1 + 2);                  // 3
console.log("1" + 2);                // "12"
console.log(true + 1);               // 2
console.log(true + "1");              // "true1"
console.log(NaN + 1);                // NaN
console.log(NaN + "1");              // "NaN1"
console.log(1 + undefined);          // NaN
console.log("1" + undefined);        // "1undefined"
console.log(1 + null);               // 1
console.log(true + true);             // 2
console.log(true + null);             // 1
console.log(true + undefined);        // NaN
```

操作原则

1. 如果执行加法的时候有字符串，则结果一定是一个拼接型的字符串
2. `Nan` 不参与运算，只要参与计算，则结果一定是 `Nan`
3. 对于非数字类型进行加法运算，则先使用 `Number()` 进行转换一下

隐藏规则：在加法运算里面，字符串是老大， `NaN` 是老二；俗称加法字符串优先

根据上面的规则，我们可以来看一下下面的题目

```
console.log(1 + "2" + "2");          // "122"
console.log('1' + 2 + 3);              // "123"
console.log(1 + +"2" + "2");           // "32"
```

减法操作符

减法操作符也是算术运算符，它使用 `-` 来表示，减法操作符的结果一定是 `Number` 类型

```
console.log(2 - 1);           //1
console.log("2" - 1);          //1
console.log("2" - "1");        //1
console.log("2" - true);       //1
console.log(2 - false);        //2
console.log(1 - null);         //1
console.log(2 - undefined);    //NaN
console.log(2 - NaN);          //NaN
console.log(null - true);      //-1
console.log("a" - 1);          //NaN
console.log("a" - "b");        //NaN
console.log(" " - 1);          //-1
console.log(" " - NaN);        //NaN
```

操作规则

1. 减法操作符得到的结果一定是一个 `Number` 类型
2. `Nan`与任何值相减都是`Nan`
3. 如果执行减法的不是数字，则通过 `Number` 去转换一次

乘法操作符

乘法操作符使用的是 `*` 来表示，它的结果与操作原则与减法保持了一致

```
console.log(2 * 1);           //2
console.log("2" * 1);          //2
console.log("2" * "1");        //2
console.log("2" * true);       //2
console.log(2 * false);        //0
console.log(1 * null);         //0
console.log(2 * undefined);    //NaN
console.log(2 * NaN);          //NaN
console.log(null * true);      //0
console.log("a" * 1);          //NaN
console.log("a" * "b");        //NaN
console.log(" " * 1);          //0
console.log(" " * NaN);        //NaN
```

操作原则

1. 乘法操作符得到的结果一定是一个 Number 类型
2. NaN与任何值相乘都是NaN
3. 如果执行乘法的不是数字，则通过 Number 去转换一次

除法操作符

除法操作符使用 / 来表示，除法操作符的结果一定是 Number 类型，它的操作规则与减法操作符保持一致

```
console.log(2 / 1);          //2
console.log("2" / 1);        //2
console.log("2" / "1");      //2
console.log("2" / true);     //2
console.log(2 / false);      //Infinity
console.log(1 / null);       //Infinity
console.log(2 / undefined);  //NaN
console.log(2 / NaN);        //NaN
console.log(null / true);    //0
console.log("a" / 1);         //NaN
console.log("a" / "b");      //NaN
console.log("") / 1);        //0
console.log("") / NaN);      //NaN
```

操作原则

1. 除法操作符得到的结果一定是 Number 类型
2. NAN与任何数相除都是NaN
3. 如果执行除法的不是数字，则通过 Number 去转换一次
4. 如果除数为0，则最终的结果为 Infinity 无穷大

取余操作符

取余操作符是针对两个数相除以后取余数，使用 % 来表示，它的结果一定是Number类型，遵守减法操作规则

```
console.log(2 % 1);           //0
console.log("2" % 1);          //0
console.log("2" % "1");         //0
console.log("2" % true);        //0
console.log(2 % false);         //NaN
console.log(1 % null);          //NaN
console.log(2 % undefined);      //NaN
console.log(2 % NaN);           //NaN
console.log(null % true);        //0
console.log("a" % 1);            //NaN
console.log("a" % "b");          //NaN
console.log("") % 1);             //0
console.log("") % NaN);          //NaN
```

操作原则

1. 取余操作符得到的结果一定是 **Number** 类型
2. **NaN**不参与运算，只要参于运算，结果就是**NaN**
3. 如果执行取余运算的不是数字，则通过 **Number** 去转换一次
4. 如果取余的时候除数为0，则结果就是 **NaN**

布尔操作符[逻辑操作符]

在一门编程语言中，布尔操作符的重要性堪比相等操作符。如果没有测试两个值关系的能力，那么诸如 `if...else` 和循环之类的语句就不会有用武之地了。布尔操作符一共有 3 个：非 (NOT) 、与 (AND) 和或 (OR) 。

布尔操作符无非就是真或假的运算过程，所以在这里要首先就弄清楚，在JS的数据里面，有哪6个明确的 **false** 值？

这6个值 `0, "", false, null, undefined, NaN`

逻辑非操作符

逻辑非操作符使用的是 **!** 来表示，它执行的是非真即假，非假即真的操作，它的结果一定是一个布尔类型

```
var a = true;
var b = !a;                  //false
var c = !(100<101);        //false;
```

如果仅仅只看上面的代码，我们会发现，非运算符是非常简单的，但是要注意JavaScript是一门弱类型语言，所有的类型都可以进行逻辑非的运算（这一点和其它的强类型语言是不一样的），所以它们还可以这样写

```
console.log(!true);           //false
console.log(!123);            //false
console.log(!undefined);      //true
console.log(!0);               //true
console.log(!null);            //true
console.log(!NaN);             //true
console.log(!"");
console.log(!"biaogege");     //false
console.log(!!"bgg");         //true
```

操作原则

1. 它的结果一定是布尔类型
2. 非真即假，非假即真
3. 对于不是布尔类型的值的，先通过 Boolean 做隐式类型再换，再取非
4. 一定要注意6个明确的 false 条件

逻辑与操作符

逻辑与操作符执行的是 `&&` 这个符号进行操作，它执行一假即假的原则，它的结果不一定是布尔类型

现在我们先通过简单的案例来学习一下

```
var a = true && true;           //true
var b = true && false;            //false
var c = false && false;           //false
var d = false && true;            //false
var e = true && false && true;      //false
```

在上面的代码里面，一假即假的操作是非常好判断的。但是也要注意，还是因为JS是一门弱类型语言，在执行逻辑与的操作的时候，不一定全都是布尔值在进行操作，还可以是其它的任何类型，请看下面代码

```
console.log(true && false);           //false
console.log(true && 123);              //123
console.log(false && 123);             //false
console.log(true && "");               //""
console.log(true && "" && 123);        //""
console.log(NaN && false);            //NaN
console.log(NaN && null);             //NaN
console.log(null && "abc");           //null
console.log(123 && "abc");            //"abc"
console.log(0 && 123);                //0
```

操作原则

1. 它的结果不一定是 Boolean 类型
2. "一假即假"的操作
3. 对于不是布尔类型的要拿 Boolean 去测试一下
4. 对于6个明确的 false 的条件要条件
5. 短路原则：当一个表达式已经能够得到结果，就不会再向后面运算

逻辑或操作符

逻辑或操作符使用 || 来表示，它执行一真即真的操作，它的结果不一定是布尔类型

还是一样的，先通过几个简单的案例来看一下

```
var a = true || false;                  //true
var b = false || false;                //false
var c = false || true;                 //true
var d = true || true;                  //true
var e = true || false || true;         //true
var f = false || true || false;        //true
```

同理，这个操作与上面的操作也是相类似的，它也是可以与其它数据类型进行操作的。请看下面代码

```
console.log(true || 123 || false);      //true
console.log(false || 123 || false);       //123
console.log(666 || true || "hello");     //666
console.log(NaN || false || null);        //null
console.log(null || "标哥哥");           //"标哥哥"
console.log(false || NaN);               //NaN
```

操作原则

1. 它的结果不一定是 Boolean 类型
2. "一真即真"的操作
3. 对于不是布尔类型的要拿 Boolean 去测试一下
4. 对于6个明确的 false 的条件要条件
5. 短路原则：当一个表达式已经能够得到结果，就不会再向后面运算

逻辑非，与，非这三个符都逻辑为布尔操作符，我们在工作当中也经常会将这三个操作符结合在一起使用，这个时候怎么办呢：

? 小练习：根据之前所学习的知识以及操作原则，请计算下面的操作结果

```
console.log(true || "" && NaN); //true
console.log(666 || !"" || false); //6666
console.log(666 && NaN || null); //null
console.log("标哥哥" || 123 && NaN); //"标哥哥"
console.log(null || 666 && undefined); //undefined
console.log(!null || 123 && undefined); //true
```

总的操作原则

1. 或：一真即真；与：一假即假，非：非真即假，非假即真
2. 先非，再与，最后或
3. 非布尔类的型的要用 Boolean 去测试或转换
4. 弄清楚6个明确的 false 条件
5. 短路原则

相等操作符

在 JavaScript 里面，我们如果想判断两个内容是否相等，我们需要使用 == 来表示

JavaScript里面的相等操作符与其它强类型语言里面的相等操作符不一样，因为JS是一门弱类型的语言，有时候在判断的时候，它在符号左右会发生隐式的类型转换

```

console.log(1 == 2);                                //false
console.log(1 == 1);                                //true
console.log(1 == true);                             //true 第4条原则

console.log(0 == false);                            //true 第4条原则
console.log("0" == false);                          //true 先适用于第4条
原则，再适用于第3条
console.log("10" == 10);                           //true 第3条原则
console.log("") == 0);                            //true 第3条原则
console.log(NaN == 0);                           //false 第2条原则
console.log(NaN == NaN);                          //false 第2条原则
console.log(null == undefined);                   //true 第1条原则
console.log("") == null);                         //false 第1条原则或第
5条原则
console.log(0 == undefined);                      //false 第1条原则或第
5条原则

```

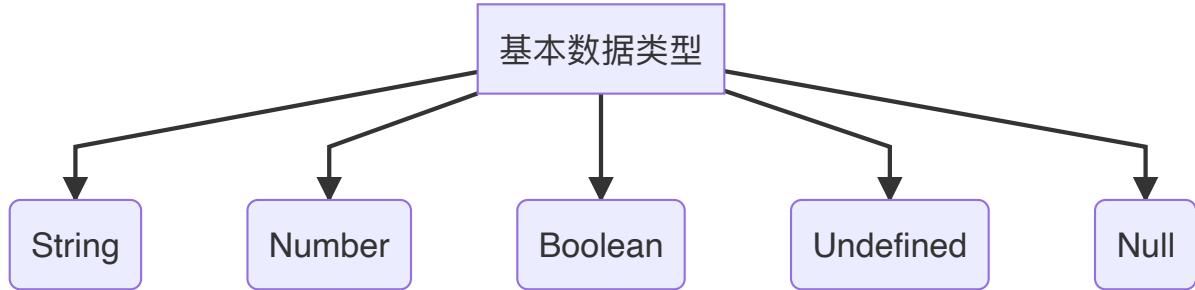
操作原则

1. `null` 与 `undefined` 相等，也可以与自身相等，除此之外，不与任何值相等
2. `NaN` 不参与比较，只要比较就是 `false`
3. 如果一个是字符串，另一个操作数是数值，则将字符串转成数值
4. 如果是布尔类型，则通过 `Number` 去转换成数字
5. 在比较之前，`null` 与 `undefined` 不参与类型转换
6. 相等操作符的结果一定是布尔类型

全等操作符

全等操作符也叫严格相等操作符，它使用 `==` 来表示，它是弱类型语言里面独有的，因为在上面的相等操作符里面，我们如果直接判断相等，它会做隐匿类型转换，这样做并不严格，有些时候我们希望要判断它的数据类型和值都要相等，这个时候我们就使用严格相等

全等操作符一定是符号两边的数据类型和数据的值都相同，结果才为 `true`，否则一律为 `false`



```

console.log(1 === 2);           //false
console.log(1 === 1);           //true
console.log(1 === true);        //false
console.log(0 === false);       //false
console.log("0" === false);     //false
console.log("10" === 10);        //false
console.log("0" === 0);          //false
console.log(NaN === 0);          //false
console.log(NaN === NaN);        //false
console.log(null === undefined); //false
console.log(" " === null);       //false
console.log(0 === undefined);    //false

```

操作原则

1. 符号两边数据类型要相同
2. 符号两边数据的值相同
3. `Nan` 不参与比较，只要比较就是 `false`
4. 结果一定是布尔类型

不相等操作符

不相等操作主要就是 `!=` 与 `!==`，它其实就是把相等操作符或全等操作的结果取反就可以了

```

console.log(1 != 1);           //false
console.log(1 !== "1");         //true
console.log(1 != "1");          //false
console.log(null != undefined); //false;
console.log(NaN != NaN);        //true

```

关系操作符

小于 (`<`) 、大于 (`>`) 、小于等于 (`<=`) 和大于等于 (`>=`) 这几个关系操作符用于对两个值进行比较，比较的规则与我们在数学课上所学的一样。这几个操作符都返回一个布尔值，

```
var result1 = 5 > 3;          //true
var result2 = 5 < 3;          //false
```

上面的关系操作符的结果都是准备的，但是要注意一点，JavaScript是一个弱类型的语言，它有可能会发生隐式数据类型转换，所以我们符号左右两边的数可能并不是同一个数据类型，这怎么办呢？

console.log("1" < 2);	//true	第2条原则
console.log("1" < "2");	//true	第4条原则
console.log("11" < 2);	//false	第2条原则
console.log("11" < "2");	//true	第4条原则
console.log(true < 2);	//true	第3条原则
console.log(true < "2");	//true	先适用第3条，再适用第2条
console.log("aa" < "b");	//true	第4条原则
console.log("aa" < "ab");	//true	第4条原则
console.log("A" < "a");	//true	第4条原则
console.log(null < 1);	//true	第6条原则
console.log(NaN <= 0);	//false	第7条原则
console.log(undefined < 1);	//false	先适用第6条，再适用于7条
console.log("我" > "你");	//true	第5条原则
console.log("0x12" <= 12);	//false	第2条原则，注意前面是16进制
console.log("0x12" <= "12");	//true	第4条原则

操作原则

1. 如果2个数是数值，则直接比较大小
2. 如果有1个是数值，另一个不是数值，则将另一个数通过 `Number` 转换一下
3. 如果有一个操作数是布尔值，则先通过 `Number` 去转换
4. 如果2个操作数都是非中文的字符串，则比较 `ascii` 码
5. 如果有中文字符串，则比较 `unicode` 编码【目前这一块，你们没有学，要到后面讲面向对象才会讲到】
6. 对于非数字类型的，则通过 `Number` 隐式类型转换一下
7. `Nan` 不参与比较，只要比较就是 `false`

0011		01/0		0101		0110		0111	
3	4	5	6	7	8	9	10	11	
十进制	字符								
48	0	64	@	80	P	96	`	112	p
49	1	65	A	81	Q	97	a	113	q
50	2	66	B	82	R	98	b	114	r
51	3	67	C	83	S	99	c	115	s
52	4	68	D	84	T	100	d	116	t
53	5	69	E	85	U	101	e	117	u
54	6	70	F	86	V	102	f	118	v
55	7	71	G	87	W	103	g	119	w
56	8	72	H	88	X	104	h	120	x
57	9	73	I	89	Y	105	i	121	y
58	:	74	J	90	Z	106	j	122	z
59	;	75	K	91	[107	k	123	{
60	<	76	L	92	\	108	l	124	
61	=	77	M	93]	109	m	125	}
62	>	78	N	94	^	110	n	126	~
63	?	79	O	95	_	111	o	127	△

条件操作符

条件操作符也叫三目运算符，它是根据一个条件来进行赋，在有些编程语言里面也叫三元表达式。条件操作符的基本语法格式如下

```
var a = 条件是否成立?条件成立的值:条件不成立的值;
```

现在先看几个简单的例子

```
var a = true ? "标哥哥" : "帅哥哥";           //标哥哥
var b = false ? "帅气" : "多金";                //多金
var c = 1 > 2 ? "颜一鸣" : "小鸣";              //小鸣
```

按照条件操作符的要求，前面的条件一定要是 Boolean 值才行，但是因为JS是一门弱类型的语言，它的前面不一定会得到布尔类型的值，这个时候我们就要将这个条件使用 Boolean 去测试一下，再去进行后面的操作

```
var a = 666 ? "你好" : "我好";                  //你好
var b = null ? "hello" : "world";               //world
var c = 1 > NaN ? "你好" : null;                 //null
var d = c ? "你好" : "大家好";                   //大家好
```

条件表达式在它是根据一个条件来进行赋值，在平常开发当中它的应用非常广泛，如下

```
var a = 3;  
var b = 4;  
var max;  
//请使用条件表达式，来将a与b当中较大的一个值赋值给max  
max = a > b ? a : b;
```

上面的条件表达式，可以将较大的一个值赋值给 `max`，我们现在再将上面的案例升级一下

```
var a = 3;  
var b = 4;  
var c = 5;  
var max;  
//请使用条件表达式，将a,b,c三个当中最大的一个值给max;
```

第一种写法

```
//第一步：先计算a,b当中比较大的一个值  
var temp = a > b ? a : b;  
//这个时候的temp就是a和b当中大的那个值  
//第二步：再将temp与c相比较  
var max = temp > c ? temp : c;
```

现在再将上面的2行代码转换成1行代码

```
var max = (a > b ? a : b) > c ? (a > b ? a : b) : c;
```

第二种写法

```
// var max = a > b ? (成立以后a,c比较) : (不成立b,c比较);  
var max = a > b ? (a>c?a:c) : (b>c?b:c);
```

条件操作符的本质就是根据一个条件去执行某些事件 【后期会带出流程控制】

赋值运算符

赋值运算符使用的是 `=` 来表示，这个在之前讲变量的时候已经讲过了。赋值运算符是将符号右边的值赋值给符号左边

```
var a = 123; //将右边的123赋值给左边的变量q
```

赋值运算符是一个非常简单的操作符，只有一个注意事项就是符号的左边必须是一个定值（在赋值的一瞬间是一个定值）

赋值运算符是一个特殊的运算符，它是从右向左再进行操作，所以在操作的一瞬间，它左边的值是不允许发生变化的

```
var a = 10;
var b = 20;
a = b;          //赋值的时候a是一个固定的值，没有发生变化，所以它不会报错，赋值完成以后它就是20
a++ = b;        //错误的，b赋值给了a，a说不要你的值，我自己在变化，这个时候左边的值就没有固定
+a = b;        //错误的，原因仍然是左边的值不是定值
a = b++;
```

赋值运算符主要是以下的几种

```
var a = 10;
a = a + 20;
//上面的写法还可以简写成下面的形式
a += 20;
var b = 5;
b -= 3;        //b = b-3;

var c = 10;
c += c;
```

上面的这种操作叫复合赋值运算符，根据上面的情况，我们可以依次类推

- 乘/赋值(*=)
- 除/赋值(/=)
- 加/赋值(+=)
- 减/赋值(-=)
- 模/赋值(%=)

逗号操作

逗号运算符是一个最最最基础的运算符，它代表一条语句没有完，或一个关键字的作用没有完，逗号代表一个操作没有结束

请看下面代码

```
// var a ;  
// var b;  
// var c;  
var a, b, c;
```

同理，我们还可以继续向下面看

```
// var a = 10;  
// var b = 20;  
// var c = 30;  
var a = 10, b = 20, c = 30;
```

操作符的优先级

现在我们已经学习了很多运算符，如果运算符结合在一起了，就会有优先级的问题，具体的优先级请看下面

1. `. () []`
2. 一元操作符 `++`, `--`
3. 算术运算符[先乘除，后加减]
4. 关系操作符 `>`, `<`, `>=`, `<=`, `==`, `!=`
5. 逻辑操作符[先非，再与，最后或]
6. 三目运算符
7. 赋值运算符

```
var num = 10;  
var x = 5 == num/2 && (2+2*num).toString() == "22";  
console.log(x);
```

上面的代码，我们来通过运算符的优先级来计算一下

```
//var x = 5 == num/2&&(2+2*10).toString() == "22";  
//var x = 5 == num/2&&(2+20).toString() == "22";  
//var x = 5 == num/2&&(22).toString() == "22";  
//var x = 5 == num/2&&"22" == "22";  
//var x = 5 == 10/2&&true;  
//var x = 5 == 5&&true;  
//var x = true&&true;  
var x = true;
```

优先级	运算符	说明	结合性
1	[]、.、()	字段访问、数组索引、函数调用和表达式分组	从左向右
2	++ -- -~!delete new typeof void	一元运算符、返回数据类型、对象创建、未定义的值	从右向左
3	*、/、%	相乘、相除、求余数	从左向右
4	+、-	相加、相减、字符串串联	从左向右
5	<<、>>、>>>	左位移、右位移、无符号右移	从左向右
6	<、<=、>、>=、instanceof	小于、小于或等于、大于、大于或等于、是否为特定类的实例	从左向右
7	==、!=、====、!==	相等、不相等、全等，不全等	从左向右
8	&	按位“与”	从左向右
9	^	按位“异或”	从左向右
10		按位“或”	从左向右
11	&&	短路与（逻辑“与”）	从左向右
12		短路或（逻辑“或”）	从左向右
13	?:	条件运算符	从右向左
14	=、+=、-=、*=、/=、%=、&=、 =、^=、<、<=、>、>=、>>=	混合赋值运算符	从右向左
15	,	多个计算	按优先级计算，然后从右向左

GSDN @@剁椒鱼头

小练习

1. 计算下面的结果

```
var a = 10;
++a; //11
var b = ++a+2;
```

```
console.log(b); //14

var c = 10;
c++; //10
//11
var d = c+++2; //11+2 = 13
console.log(d); //d=13 c = 12

var e = 10;
var f = e++ + ++e; //10 + 12 = 22
console.log(f);
```

二进制运算符

二进制运算符做为选讲内容

二进制运算符也叫位运算符，它按数值的位来进行运算，在 ECMAScript 的系统里面，所有的数据都是用64位存储，32位运算，如果进行二进制的位运算，需要先将原来的数值转换成2进制操作

位操作符用于在最基本的层次上，即按内存中表示数值的位来操作数值。ECMAScript 中的所有数值都以 IEEE-754 64 位格式存储，但位操作符并不直接操作 64 位的值。而是先将 64 位的值转换成 32 位的整数，然后执行操作，最后再将结果转换回 64 位。对于开发人员来说，由于 64 位存储格式是透明的，因此整个过程就像是只存在 32 位的整数一样。

按位非 (NOT)

按位非操作符由一个波浪线 (~) 表示，执行按位非的结果就是返回数值的反码。按位非是 ECMAScript 操作符中少数几个与二进制计算有关的操作符之一。

```
var num1 = 25;
var num2 = ~num1;
```

现在来看它的计算过程

第一步：先将这个数转2进制

```
num1.toString(2); //11001
```

第二步：补码

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25的二进制的补码	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	

第三步：取反

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25的二进制的补码	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
反码	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0

取反以后，我们发现它的最高位31位是一个数字 1 ,这就说明这个数是一个负数，**负数的二进制是需要进行转码的**

第三步：将负数的二进制转码

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	I	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	
位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
25的二进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	
反码	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	
减1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	
保留符号再取反	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0

将负数的二进制转码以后，得到的结果是 11010 ,但是它的符号位是 1 说明它是一个负数

```
parseInt("11010", 2);      //26  
//因为符号位是负数，所以最终的结果是-26
```

通过上面计算，其实我们发现二进制的取反过程它就是将原来的数乘以 -1 再减去1就可以

按位与 (AND)

按位与操作符使用的符号是 &

```
var result = 18 & 3;
```

这个操作过程也很简单

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
18的二进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	
3的二进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
按位与	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

第一个数值的位		第二个数值的位		结 果
1		1		1
1		0		0
0		1		0
0		0		0

按位或 (OR)

按位或操作符由一个竖线符号 (|) 表示，同样也有两个操作数。按位或操作遵循下面这个真值表

第一个数值的位	第二个数值的位	结 果
1	1	1
1	0	1
0	1	1
0	0	0

```
var result = 23 | 71;
```

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
23的二进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1		
71的二进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1		
按位或	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1		

得到结果87

小练习

```
var result1 = ~-17;           //16  
  
var result2 = -16 & 8;        //0  
  
var result3 = -22 & -18;      //-22  
  
var result4 = -4 | 5;  
  
var result5 = -5 | -6;
```

上面的练习里面有负数，对于负数的二进制要以补码的形式去完成

负数同样以二进制码存储，但使用的格式是二进制补码。计算一个数值的二进制补码，需要经过下列 3 个步骤：

图灵社区会员 StinkBC(StinkBC@gmail.com) 专享 尊重版权

10 第3章 基本概念

- (1) 求这个数值绝对值的二进制码（例如，要求-18 的二进制补码，先求 18 的二进制码）；
- (2) 求二进制反码，即将 0 替换为 1，将 1 替换为 0；
- (3) 得到的二进制反码加 1。

要根据这 3 个步骤求得-18 的二进制码，首先就要求得 18 的二进制码，即：

按位异或 (XOR)

按位异或操作符由一个插入符号 (^) 表示，也有两个操作数。以下是按位异或的真值表。

第一个数值的位	第二个数值的位	结 果
1	1	0
1	0	1
0	1	1
0	0	0

```
var result = 25 ^ 3;           //26
```

左移

按位左移运算符使用 << 来表示

```
var a = 2;
var b = a<<5;           //64
```

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2的2进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		
左移5位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0		

有符号的右移

有符号的右移操作符由两个大于号 (`>>`) 表示，这个操作符会将数值向右移动，但保留符号位（即正负号标记）。有符号的右移操作与左移操作恰好相反

```
var a = 64;  
var b = a>>5;
```

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
64的2进制	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
右移5位	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

无符号右移

无符号右移操作符由 3 个大于号 (`>>>`) 表示，这个操作符会将数值的所有 32 位都向右移动。对正数来说，无符号右移的结果与有符号右移相同，但对于负数来说，这个结果是会有影响的，因为负数的最高位是 1

```
var a = -64;  
var b = a>>>5; //134,217,726
```

位数	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
负64的2进制码	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	
无符号右移	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

注意事项

- 所有二进制的操作符都不对小数进行操作，后期我们可以复用这个特点来快速的将一个数值取整

```
var a = 3.14;  
var b = ~~a; //3;
```

- 左移与右移运算符不得超过原有的计算机位数，如果超过了，将返回一个毫无意义的数

JavaScript语句

语句是构成编程系统当中的最小单位，它是用于完成某些特定的逻辑功能，也是为了完成代码的流程控制的，所以语句也叫流程控制

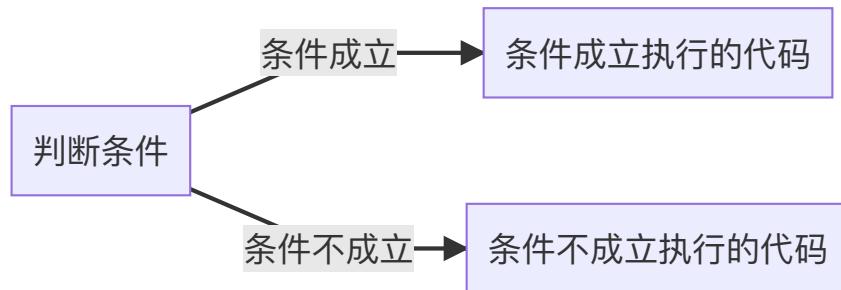
语句大体上来说可以分为以下几类

- 分支语句

2. 循环语句
3. 选择语句

所有的语句都应该是由一个或多个关键字来完成的

if...else



if...else 是我们的语句当中的分支语句，它用于做条件判断，来控制代码的执行流程，它的语法格式如下

```
if(条件1){  
    //符合条件1的时候  
}  
else if(条件2){  
    //后面还有好多条件....  
}  
else{  
    //不满足条件的时候  
}
```

如下所示

```
var a = 10;  
var b = 20;  
// 如果a>b则输出"你好"，否则就输出“你不好”  
// console.log()  
if (a > b) {  
    console.log("你好");  
}  
else{  
    console.log("你不好");  
}
```

上面的代码是一个非常简单的流程控制，我们只有一个条件。但是这里仍然有几个注意事项给大家说一下

```
var a = 10;
var b = 20;
if(a>b)
    console.log("你好");
else
    console.log("你不好");
```

当代码体只有一行的时候，我们可以省略花括号【不建议这么支操作，很容易形成岐义，也很容易出错】

请看下面的要求及代码

```
var score = 88;
//如果分数在90或以后，就输入优秀
//如果分数在80或以后，就输入良好
//如果分数在70或以后，就输入中等
//如果分数在60或以后，就输入及格
//否则就输出不及格
```

第一种情况下的代码

```
if(score>=90){
    console.log("优秀");
}
if(score>=80){
    console.log("良好");
}
if(score>=70){
    console.log("中等");
}
if(score>=69){
    console.log("及格");
}
else{
    console.log("不及格");
}
```

上面的代码写法是有问题的，它所有的if都是一个并行条件，它会把所有的 if 条件都测一遍，只要是符合要求了，就执行了。

最终打印的结果就是“良了， 中等， 及格”

第二种情况下的代码

```
if(score>=90){  
    console.log("优秀");  
}  
else if(score>=80){  
    console.log("良好");  
}  
else if(score>=70){  
    console.log("中等");  
}  
else if(score>=60){  
    console.log("及格");  
}  
else{  
    console.log("不及格");  
}
```

上面的代码最终打印输出“良好”，它所有的条件都是串行的条件，它要符合了一个要求，就会中断语句，提前结果

这里一共有4条语句
所以score的条件会被
判断4次 ,217,726

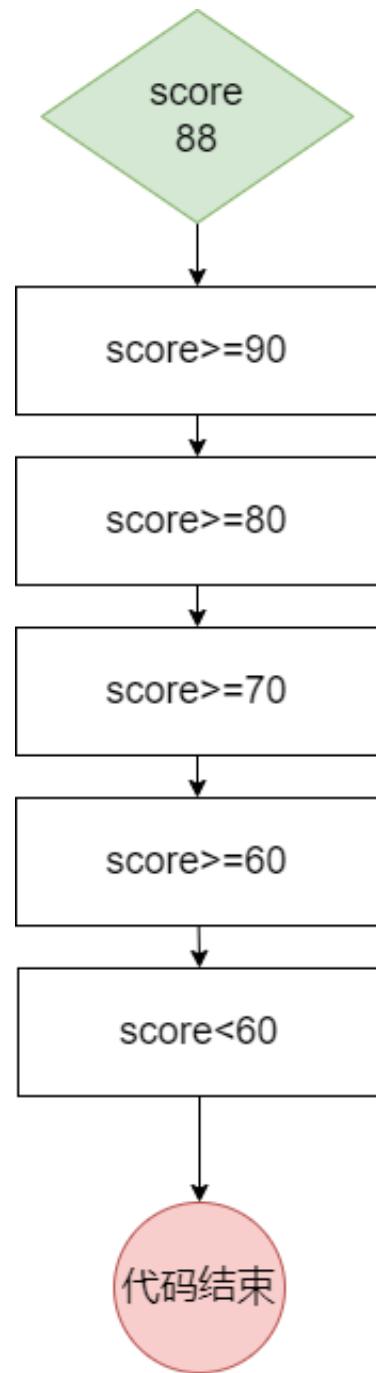
从上到下，这是一
整条语句

语句的结束

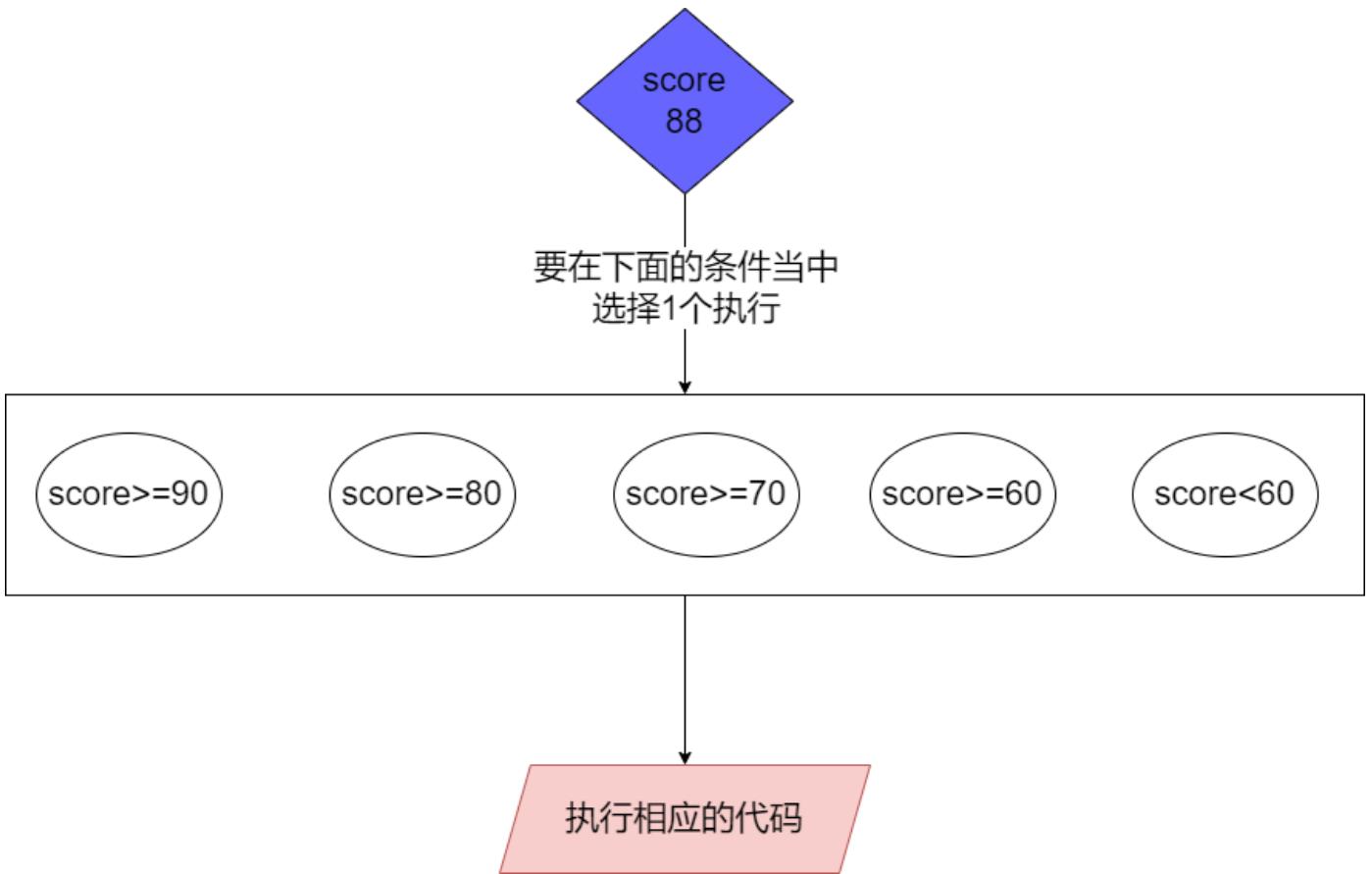
```
12 12  
13 </body> 13 </body>  
14 <script> 14 <script>  
15     var score = 88; 15     var score = 88;  
16     //如果分数在90或以后，就输入优秀 16     //如果分数在90或以后，就输入优秀  
17     //如果分数在80或以后，就输入良好 17     //如果分数在80或以后，就输入良好  
18     //如果分数在70或以后，就输入中等 18     //如果分数在70或以后，就输入中等  
19     //如果分数在60或以后，就输入及格 19     //如果分数在60或以后，就输入及格  
20     //否则就输出不及格 20     //否则就输出不及格  
21     if(score>=90){ 21     if(score>=90){  
22         console.log("优秀"); 22         console.log("优秀");  
23     } 23     }  
24     if(score>=80){ 24     else if(score>=80){  
25         console.log("良好"); 25         console.log("良好");  
26     } 26     }  
27     if(score>=70){ 27     else if(score>=70){  
28         console.log("中等"); 28         console.log("中等");  
29     } 29     }  
30     if(score>=60){ 30     else if(score>=60){  
31         console.log("及格"); 31         console.log("及格");  
32     } 32     }  
33     else{ 33     else{  
34         console.log("不及格"); 34         console.log("不及格");  
35     } 35     }  
36 </script> 36 </script>  
37 </html> 37 </html>
```

现在我们分别用2张图来说明这个情况

多个if的情况



if...else的情况



在 `if` 与 `else ...if` 里面，一共定要弄清楚它们的区别点是什么

`if` 是对条件进行判断的时候成立的时候执行的语句，那么如果 `if` 的条件的值不是一个 `boolean` 类型怎么办，如下所示

```

var a = 123;
var b = 0;

if(a){
    console.log("你好");
}
else{
    console.log("世界");
}

if(b){
    console.log("hello");
}
else {
    console.log("world");
}
  
```

其中的 condition (条件) 可以是任意表达式；而且对这个表达式求值的结果不一定是布尔值。ECMAScript 会自动调用 Boolean() 转换函数将这个表达式的结果转换为一个布尔值。如果对 condition 求值的结果是 true，则执行 statement1(语句 1)，如果对 condition 求值的结果是 false，则执行 statement2(语句 2)。而且这两个语句既可以是一行代码，也可以是一个代码块(以一对花括号括起来的多行代码)。

一定要弄清楚之前给大家讲过了JS里面6个明确的false条件

for循环语句

for 语句也是一种，它会循环的执行代码体里面的代码，它的语法格式如下

```
for(初始值;循环前测试条件;自变量){  
    //代码体  
}
```

现在我们先从生活当中的最基本的例子来讲起，请看以下场景

场景一：标哥有钱了，办了一个砖厂，颜一鸣同学在标哥这里在搬砖，今天标哥交给颜一鸣的任务就是要搬10块砖。因为颜一鸣的个子比较小，所以一次只能搬动1块砖。那么请用代码来说明一下颜一鸣搬砖的过程

针对上面的地场景，我们要弄清楚几个问题

1. 颜一鸣从第1块开始搬砖
2. 颜一鸣要搬10块
3. 颜一鸣每次搬1块

首先，我们就使用搬砖的思维方式来考虑这个问题

```
for (var count = 1; count <= 10; count++) {  
    console.log("颜一鸣从在搬" + count + "块砖");  
}
```

场景二：颜一鸣经过多年的搬砖生活以后，肌肉变得发达了，现在可以一次搬2块砖了，标哥很欣慰，看到了颜一鸣的成长，今天又给颜一鸣派了一个新的任务，让颜一鸣还是搬10块砖，颜一鸣听后非常高兴说道：“小意思啦，我现在是大力士了，可以一次搬2块，来，看我的表演”

现在请根据上面的场景，来使用编程思维解决

1. 颜一鸣从第1块开始搬砖
2. 颜一鸣要搬10块

3. 颜一鸣每次搬2块

```
for (var count=1; count<=10; count+=2) {  
    //代码体  
    console.log("颜一鸣从在搬" + count + "块砖");  
}
```

场景三：功夫不负有心人，颜一鸣经过了这么长时间在标哥这里的打拼，小有资本，它偷偷的搬了4块砖藏起来了。今天看到标哥，格外的神气一些，标哥不舒服了。说道：“来，小伙子，给你个光荣而又艰巨的任务，今天要搬20块砖，随便你怎么搬”。颜一鸣听后，无所谓的道：“没事，可以的，我可以一次搬2块砖”，同时颜一鸣心底还乐道：“嘿嘿，我之前还有4块砖藏着，可以直接使用，这样我就可以从第5块直接开始了”

现在请根据上面的场景来使用编程的思维解决

1. 颜一鸣从第5块开始
2. 颜一鸣要搬20块
3. 颜一鸣每次搬2块

```
for(var count = 5;count<=20;count+=2){  
    console.log("颜一鸣从在搬"+count+"块砖");  
}
```

在上面的3个场景里面，我们可以看到，for循环里面的三个条件我们是可以根据实际的场景去做出相应的改变的，这个三个条件代表的是什么意思，同学样一定要知道

循环的本质

for循环的本质其实是指for循环代码的执行过程是什么样式，我们现在通过断点调试的方式来迸行

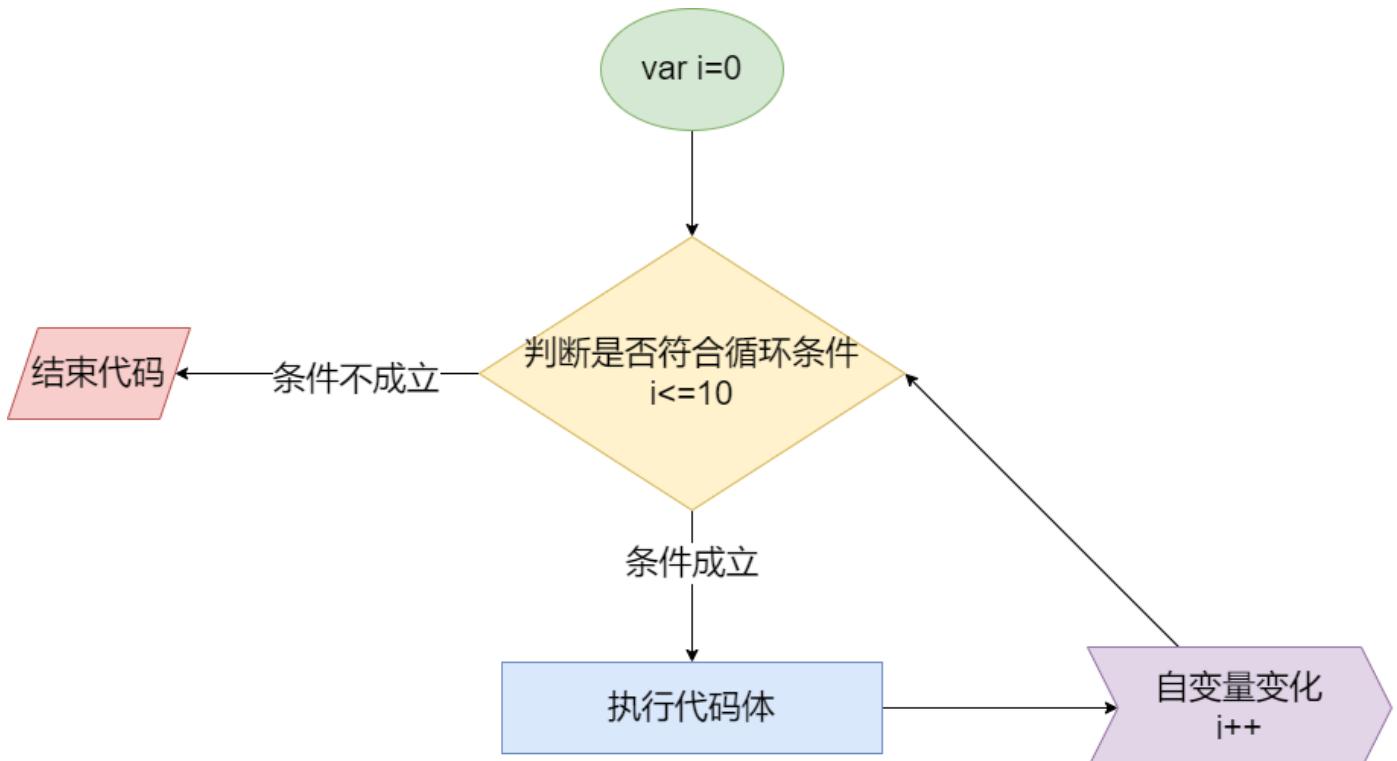
```
debugger;  
for (var i = 1; i <= 10; i++) {  
    console.log(i);  
}
```

当我们在代码里面添加了 **debugger** 这个关键字以后，代码在运行的时候，就会在这个地方暂停下来

```

debugger;
for (var i = 1; i <= 10, i++) {
    console.log(i);
}

```



通过上面的图，我们可以知道，循环的本质就是在不停的去执行三个地方的代码

1. 判断循环条件是否成立
2. 执行代码体
3. 自变量要变化

循环的本质，我们已经知道了，现在我们弄清楚一个点，这三个条件是否都是必须的

第一种情况，如果没有初始变量，怎么办

```

var count = 1;          //循环的初始变量是可以写在外边的
for (; count <= 10; count++) {
    console.log(count);
}

```

第二种情况，省略循环的判断条件

```
for(var count=1; ;count++){
    console.log(count);
}

//省略了循环的结束判断条件，那么这个循环就会一直进行，它就是一个死循环
```

第三种情况，省略自变量

```
for(var count=1;count<=10;){      //省略了自变量，它也会变成一个死循环
    console.log(count);
}
```

综上所述，for里面的三个东西，是都可以省略了

```
for(;){
    console.log("hello world");
}
```

上面的语法是没有错的，但是它是一个死循环

小练习

1. 请打印1~100之间的偶数

```
/**
 * 开始条件: 2
 * 结束条件: 100
 * 自变量: 2
 */
for (var i = 2; i <= 100; i = i + 2) {
    console.log(i);
}
```

还有一种写法

```
/**  
 * 开始条件: 1  
 * 结束条件: 100  
 * 自变量: 1  
 */  
for (var i = 1; i <= 100; i++) {  
    // 下面打印这句话, 不是必须的, 它有条件执行  
    // 条件成立我就执行, 条件不成立, 我就不执行  
    if (i % 2 === 0) {  
        console.log(i);  
    }  
}
```

2. 计算1+2+3+...+10的总和

```
/**  
 * 初始值: 1  
 * 结束值: 10  
 * 自变量: 1  
 */  
var sum = 0;  
for (var i = 1; i <= 10; i++) {  
    sum = sum + i;  
}  
console.log(sum);  
// 第一次循环 sum = 0 + 1;  
// 第二次循环 sum = 0 + 1 + 2;  
// 第三次循环 sum = 0 + 1 + 2 + 3;  
// 第十次循环 sum = 0 + 1+ .....+10
```

while循环语句

while循环语句与for循环的语句的本质是一样的，都是一样前测试循环语句，它的语法格式如下

```
while(循环条件){  
    //代码体  
}
```

while是当循环条件成立的时候会一直执行的代码，这个和我们之前所学习的for循环是一样的。我们之前在讲for循环的时候讲到过循环的三个条件

1. 初始值
2. 循环条件
3. 自变量

```
for(var i=1;i<=10;i++){
    console.log(i);
}
```

上面的 **for** 循环可以写成下面的样式

```
//之前我们都讲过，循环里面的三个部分，是可以拆分来写的
var i=1;
for(;i<=10;){
    console.log(i);
    i++;
}
```

现在我们就将上面的 **for** 转写成 **while** 循环

```
var i=1;
while(i<=10){
    console.log(i);
    i++;
}
```

do...while循环语句

do...while循环语句它是一个后测试循环语句，先执行代码体，再判断条件是否成立

语法格式

```
//初始值
do{
    //代码体
    //自变量
}while(循环条件)
```

通俗来讲就是“先上车，后买票”

```
// 想打印1~10之间的数
// for(var i=1;i<=10;i++){
//     console.log(i);
// }

// var i = 1;
// while (i <= 10) {
//     console.log(i);
//     i++;
// }

// do...while的写法
var i = 1;
do {
    console.log(i);
    i++;
} while (i <= 10);
```

通过上面的代码对比，我们发现，三种循环最终得到的结果是一样的，那么为什么还需要 `do...while` 呢

我们将上面的代码的初始条件更改一出，我们就发现区别

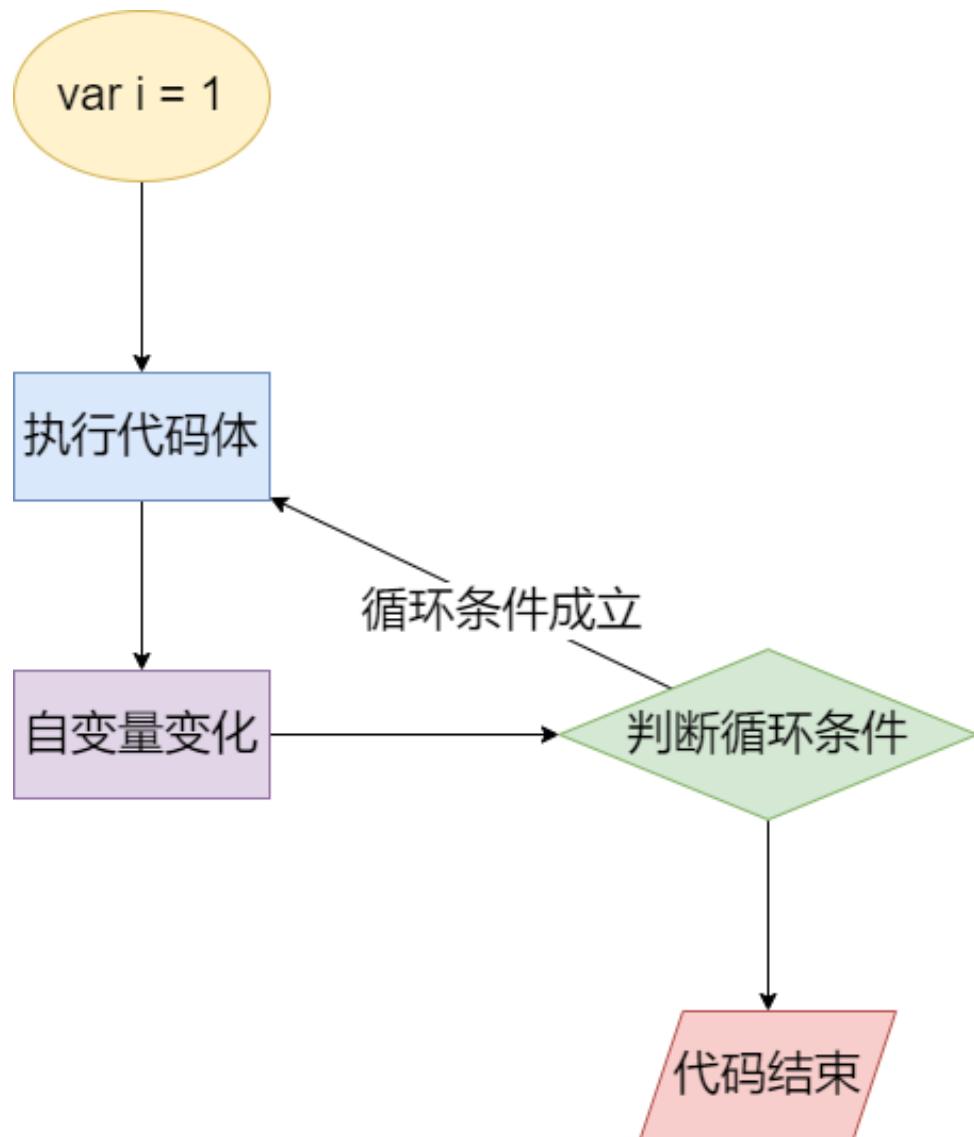
```
// 想打印1~10之间的数
// for (var i = 11; i <= 10; i++) {
//     console.log(i);
// }

// var i = 11;
// while (i <= 10) {
//     console.log(i);
//     i++;
// }

// do...while的写法
var i = 11;
do {
    console.log(i);
    i++;
} while (i <= 10);
```

这个时候我们可以看到，因为 `for` 与 `while` 都是前测试循环语句，所以在最初的时候循环条件就不成立，它就不会执行，而 `do...while` 是后测试循环语句，它会先执行一遍代码然后再判断条件是否成立，所以 `do...while` 至少会执行一次【因为即使后期条件不成立，我也执行了一次】

```
var a = 10;  
do {  
    console.log("我要牵女孩子的手");  
    a++;  
} while (a < 10);
```



break与continue

在上面的章节讲循环的时候，我们提到了循环的次数是由3个条件共同决定的，但是在某些特定的条件下，它会提前结束循环

正常情况下，我们的循环代码当条件不成立以后会自动退出循环，但是在JS里面，仍然有2个关键字会让循环中途退出，这2个关键字就是 **break** 与 **continue**

- **break** 有中断循环的意思，提前结束循环，后面的循环不再进行了【半途而废】
- **continue** 的意思就是跳过本次循环，继续执行下一次循环【浪子回头】

在多重for循环的嵌套里面，**break** 与 **continue** 执行的是就近原则，它会的对当前这个循环起作用

场景一：假设颜一鸣给标哥搬10块砖，但是颜一鸣在搬到第5块砖的时候就接到他爸爸的电话，说：“儿啊，你爸我中了彩票，5000万，你要做富二代了”，颜一鸣听了这个话以后，非常兴奋，直接把手上的砖一丢，说：“去TMD，我也是富二代了”

面对上面的场景，我们怎么办呢？

```
/*
 初始值：1
 结束值：10
 自变量：1
 */
for(var count=1;count<=10;count++){
    //现在颜一鸣要他爸电话
    if(count==5){
        break;
    }
    //因为是break,所以提前结束了循环，后面的次数也不再进行了
    console.log("颜一鸣正在搬"+count+"块砖");
}
```

场景二：颜一鸣拿着他爸爸给了5000，怀揣着梦想去了一个心仪已久的城市---东莞，结果被骗了，身心疲惫的回到了标哥的身边继续搬砖，突然之间它觉得，在这个炎炎夏日里，还是只有标哥的心是最温暖的，只有标哥对自己好！下定决心的颜一鸣决定要在标哥这里好好奋斗，做一个最强的搬砖人。标哥给颜一鸣派发了又一次的任务，搬10块砖，从第1块开始，因为天气太热，所以就1次只搬一次，这个时候正当颜一鸣开心的搬砖的时候，在耳边双听到了标哥的声音：“一鸣，你爸电话”

颜一鸣听到标哥的声音以后，身体一震，把手上的砖一丢，带着期望的眼神走向了办公室去接听电话，颤抖的声音说道：“爸呀，你又买彩票了？”

只见电话那头许久未曾出声，接着，一个声音慢悠悠的说道：“儿啊，你爸我失业了，给爸打点钱”

标哥在旁边看了许久，也不知道电话那头具体说了些什么，只见颜一鸣听完电话以后，神低落的走出了办公室，嘴里还喃喃的说道：“其它那些砖也不那么烫手了，我要继续搬砖，争取今天搬到100块，拿到今天的饭钱，这样我就可以请我的爸吃饭了”

针对上面的场景，我们又应该怎么样使用编程的思维去解决呢？

```
for (var count = 1; count <= 10; count++) {  
    // 当搬到第5块砖的时候，颜一鸣接电话去了  
    if(count==5){  
        //接电话  
        continue;  
    }  
    console.log("颜一鸣正在搬" + count + "块砖");  
}
```

嵌套的循环

循环也是可以嵌套的，如多个for循环是可以嵌套在一起的

```
for(var a=1;a<=10;a++){  
    for(var b = 1;b<=10;b++){  
        //代码体  
    }  
}
```

在上面的代码里面，我们可以看到，循环是可以进行嵌套的，嵌套的循环也可以有多层的嵌套关系，只要循环的语法是正确的，并且不要形成死循环就好

现在请根据下面的图片打印九九乘法表，这里会使用到2层的嵌套循环

1x1=1									
1x2=2	2x2=4								
1x3=3	2x3=6	3x3=9							
1x4=4	2x4=8	3x4=12	4x4=16						
1x5=5	2x5=10	3x5=15	4x5=20	5x5=25					
1x6=6	2x6=12	3x6=18	4x6=24	5x6=30	6x6=36				
1x7=7	2x7=14	3x7=21	4x7=28	5x7=35	6x7=42	7x7=49			
1x8=8	2x8=16	3x8=24	4x8=32	5x8=40	6x8=48	7x8=56	8x8=64		
1x9=9	2x9=18	3x9=27	4x9=36	5x9=45	6x9=54	7x9=63	8x9=72	9x9=81	

```
for (var row = 1; row <= 9; row++) {
    var str = "";
    for (var col = 1; col <= row; col++) {
        str += col + "*" + row + "=" + col * row + "\t";
    }
    console.log(str);
}
```

label语句

label语句是一个非常简单语句，在这一个简单的标记里面，它主要是用于配置嵌套的循环来使用的，用于表明 `break` 或 `continue` 跳到哪一层去

```
for (var row = 1; row <= 9; row++) {
    var str = "";
    for (col = 1; col <= row; col++) {
        str += row + "*" + col + "=" + row * col + "\t";
        if (col == 5) {
            break;
        }
    }
    console.log(str);
}
```

在上面的代码里面，我们在第二个循环里面使用了 `break`，这个时候得到的结果如下所示

1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45

>

从上面的图里面，我们可以看出，`for` 里面的 `break` 中断是内部的循环，这是因为 `break` 与 `continue` 执行的是就近原则，有没有什么办法让 `break/continue` 去指定的循环那里跳出或中断呢，这个时候我们就需要使用 `label`

```
aaa:for (var row = 1; row <= 9; row++) {  
    var str = "";  
    bbb:for (col = 1; col <= row; col++) {  
        str += row + "*" + col + "=" + row * col + "\t";  
        if (col == 5) {  
            break aaa; //在这里通过label来跳出了指定的for循环  
        }  
    }  
    console.log(str);  
}
```

switch语句

switch语句也叫选择语句，它会选择符合条件的语句去执行，它与我们的 **if** 语句关系是最为亲密

switch 语句应该与 **case** 语句结合在一起使用，它的语法格式如下

```
switch(条件的值){  
    case 条件1:  
        //条件1的代码体  
        break;  
    case 条件2:  
        //条件2的代码体  
        break;  
    //这里可能还有很多个case  
    default:  
        //所有的条件都不符合要求的时候  
        break;  
}
```

上面的场景与我们之前所学习的if语句是非常相像的，它是根据一个条件来选择某一个部分的代码去执行，所以在某些情况下， **if** 和 **switch** 两种语句是可以互相转换的

案例：今天的天气很好，同学们都想出去玩，但是桃子说要看下行的天气情况再决定，她说如果下雨，我们就在教室自习，如果下雪，我们就出去打雪仗，如果起风了，我们就去放风筝，如果是晴天，我们就去郊游，否则我们就在教室里面上课！

现在我们先使用 **if** 语句来完成

```
var weather = "晴天";  
if (weather == "下雨") {
```

```
    console.log("自习");
}
else if (weather == "下雪") {
    console.log("打雪仗");
}
else if (weather == "起风") {
    console.log("放风筝");
}
else if (weather == "晴天") {
    console.log("郊游");
}
else {
    console.log("上课");
}
```

上面的代码里面，我们使用了很多个 `if` 去完成，`if`太多对我们的阅读代码并不友好，所以我们可以换一种方式来完成这种业务逻辑

接下来，我们使用 `switch...case` 来完成

```
var weather = "下雪";
//switch会根据weather的值来选择某一个
//符合要求的条件去执行
switch (weather) {
    case "下雨":
        console.log("自习");
        break;
    case "下雪":
        console.log("打雪仗");
        break;
    case "起风":
        console.log("放风筝");
        break;
    case "晴天":
        console.log("郊游");
        break;
    default:
        console.log("上课");
        break;
}
```

`switch`的注意事项一：当没有`break`的时候

```

var weather = "下雪";
switch (weather) {
    case "下雨":
        console.log("自习");
    case "下雪":
        console.log("打雪仗");
    case "起风":
        console.log("放风筝");
    case "晴天":
        console.log("郊游");
    default:
        console.log("上课");
}

```

这个时候我们可以看到，上面的代码没有了 `break`，当 `case` 的一个条件 执行完毕以后，它不会退出，它会继续选择并执行后面的 `case` 的代码，直到碰到了 `break` 或程序退出

所以程序最终输出的结果应该就是"打雪仗， 放风筝， 郊游， 上课"

switch注意事项二： switch...case 在执行判断的时候使用的是 === 全等操作

```

4 <head>
5     <meta charset="UTF-8">
6     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7     <meta name="viewport" content="width=device-width">
8     <title>switch注意事项2</title>
9     </head>
10    <body>
11    <script>
12        var a = "1";
13        switch (a) {
14            case 0:
15                console.log("A");
16                break;
17            case 1:
18                console.log("B");
19                break;
20            case 2:
21                console.log("C");
22                break;
23            default:
24                console.log("D");
25                break;
26        }
27    </script>
28    <div>switch注意事项二:</div>
29 </body>
30 </html>

```



```

4 <head>
5     <meta charset="UTF-8">
6     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7     <meta name="viewport" content="width=device-width">
8     <title>switch注意事项2</title>
9     </head>
10    <body>
11    <script>
12        var a = "1";
13        if(a==0){
14            console.log("A");
15        }
16        else if(a==1){
17            console.log("B");
18        }
19        else if(a==2){
20            console.log("C");
21        }
22        else{
23            console.log("D");
24        }
25    </script>
26    <div>switch注意事项二:</div>
27 </body>
28 </html>

```

在上面的代码里面，我们可以很清楚的就看互它们的区别，左边打印的结果是 D， 右边的打印结果是 B

原因： `switch...case` 在做条件选择的时候，它使用的是强判断全等操作 `==`，而右边的 `if` 语句里面我们使用的是 `==` 的相等操作，在相等操作里面 “`1`”`==``1` 这是成立的，而全等里面则不相等



`switch` 语句在比较值时使用的是全等操作符，因此不会发生类型转换（例如，字符串“`10`”不等于数值 `10`）。

`switch...case`注意事项三：当有多个 `case` 条件符合要求的时候，优先选择第一个执行

```
var a = 1;
switch (a) {
    case 1:
        console.log("A");
        break;
    case 2:
        console.log("B");
        break;
    case 1:
        console.log("D");
        break;
    default:
        console.log("F");
        break;
}
```

案例一：现有一个学生成绩的变量 `score`，要根据学生成绩来定等级。等级划分如下：
100~90为优秀，80~89为良好，70~79为中等，60~69为及格，否则就不及格，请使用 `switch` 语句去完成代码

```
var score = 86;
switch (true) {
    case score >= 90:
        console.log("优秀");
        break;
    case score >= 80:
        console.log("良好");
        break;
    case score >= 70:
        console.log("中等");
        break;
    case score >= 60:
```

```
    console.log("及格");
    break;
default:
    console.log("不及格");
    break;
}
```

下面还有一种思路

```
var score = 100;
var n = ~~(score / 10);
switch (n) {
    case 10:
    case 9:
        console.log("优秀");
        break;
    case 8:
        console.log("良好");
        break;
    case 7:
        console.log("中等");
        break;
    case 6:
        console.log("及格");
        break;
default:
    console.log("不及格");
    break;
}
```

练习

1. 一张纸的厚度是0.0001米，将纸对折，对折多少次厚度超过珠峰高度8848米。

```
/**  
 * 实始值:厚度0.0001米  
 * 终止值:厚度8848米  
 * 自变量: 在原来的上面x2  
 */  
var count = 0; //次数  
for (var height = 0.0001; height <= 8848; height *= 2) {  
    count++; //每次对折以后，次数都要加1  
}  
console.log(count);
```

上面的代码其实我更愿意以下面的方式来完成

```
var count = 0;  
var height = 0.0001;  
for (; height <= 8848;) {  
    count++;  
    height *= 2;  
}  
console.log(count);
```

在上面的场景里面，我们可以把 `for` 换成 `while`，这样更好理解一些

```
var height = 0.0001;  
var count = 1;  
  
while(height<=8848){  
    height *= 2;  
    count++;  
}  
console.log(count);
```

2. 有一篮球从5米高处自由落下，每次弹起的高度是上一次的1/3，当篮球弹起的高度小于0.1米以后就不再弹起了，请问，这个篮球会弹起多少次？

```

// 结束条件: 小于0.1
// 初始值: 5m
var height = 5;
var count = 0;
// 一直弹, 直到小于0.1
while (height >= 0.1) {
    height = height / 3;
    count++;
}
console.log(count);

```

3. 打印出所有的水仙花数(提示: 水仙花数的范围在111~999之间) 水仙花是指一个三位数, 它的每个位上的数字的3次幂之和等于它本身 (例如: $1^3 + 5^3 + 3^3 = 153$)

第一种解法

```

for (var num = 111; num <= 999; num++) {
    // 百位
    var a = ~~(num / 100);
    // 十位
    var b = ~~(num % 100 / 10);
    // 个位
    var c = num % 10;
    if (num === a * a * a + b * b * b + c * c * c) {
        console.log(num);
    }
}

```

第二种解法

```

// 假设百位是a, 十位是b, 个位是c
for (var a = 1; a <= 9; a++) {
    for (var b = 0; b <= 9; b++) {
        for (var c = 0; c <= 9; c++) {
            // a:1,b:5,c:3
            /*
            if (a * a * a + b * b * b + c * c * c === a * 100 + b * 10
+ c) {
                console.log(a * 100 + b * 10 + c);
            }
            */
            if (a * a * a + b * b * b + c * c * c === +( " " + a + b +
c)) {

```

```

        console.log(a * 100 + b * 10 + c);
    }
}
}
}

```

4. 有一个台阶，如果一次跨2个，则最后剩下1阶，如果一次跨3个，则最后剩下2阶，如果一次跨5个，则最后剩下4阶，如果一次跨6个，则最后剩下5阶，如果一次跨7个，则刚刚好跨完，请问这个台阶最少为多少阶？

```

// 我要一个一个的去找这个数，直到找到符合要求的为止
var num = 1;
while (true) {
    if (num % 2 === 1 && num % 3 === 2 && num % 5 === 4 && num % 6 ===
5 && num % 7 === 0) {
        console.log(num);
        break;
    }
    num++;
}

```

5. 羽毛球拍15元，球3元，水2元。200元每种至少一个，有多少可能？

```

// 球拍是a，球是b，水是c
// 全部买球拍 200/15          1
// 全部买球 200/3             1
// 全部买水 200/2             1
for (var a = 1; a <= 200 / 15; a++) {
    for (var b = 1; b <= 200 / 3; b++) {
        for (var c = 1; c <= 200 / 2; c++) {
            if (a * 15 + b * 3 + c * 2 === 200) {
                console.log(a, b, c);
            }
        }
    }
}

```

6. 百马百担问题，有100匹马，驮100担货，大马驮3担，中马驮2担，2匹小马驮1担，求大、中、小各多少匹？

/*

设大马是a，中马是b，小马是c，现在列出如下的方程式

```

a+b+c=100;
a*3+b*2+c*0.5=100;

a取值范围 0<=a<=100/3;
b的取值范围 0<=b<=100/2;
c的取值范围 0<=c<=100

*/
for (var a = 0; a <= 100 / 3; a++) { //大马
    for (var b = 0; b <= 100 / 2; b++) { //中马
        for (var c = 0; c <= 100; c++) { //小马
            if (a + b + c === 100 && a * 3 + b * 2 + c * 0.5 === 100)
{
                console.log("大马: " + a + ", 中马: " + b + ", 小马: " +
c);
            }
        }
    }
}

```

7. 括号里面只能放加或减，如果要使等式成立，括号里面应该放什么运算符

```

12 () 34 () 56 () 78 () 9 = 59
/*

```

在上面的代码里面，我们可以分析

12+34可以理解为 $12 + 34 * 1$

12-34可以理解为 $12 + 34 * -1$

所以每个括号里面，我们可以认为是 $-1 \sim 1$ 这两个数

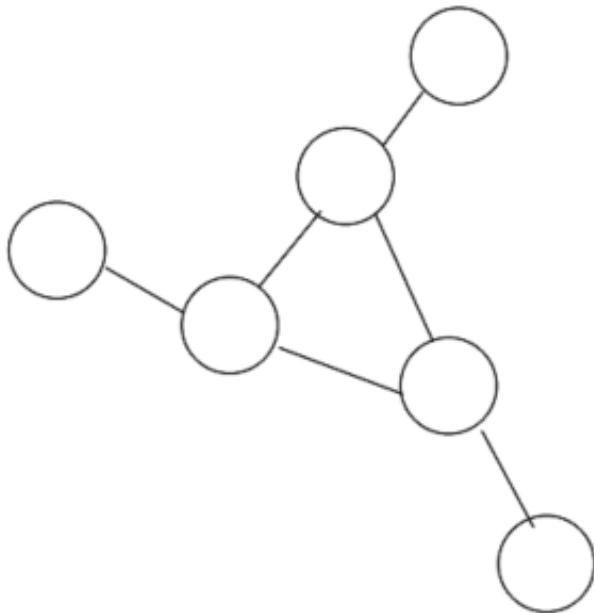
现在我们把每个括号里面的东西用 a, b, c, d 去表示，则 a, b, c, d 应该都是 $-1 \sim 1$

```

*/
outer:for (var a = -1; a <= 1; a += 2) {
    for (var b = -1; b <= 1; b += 2) {
        for (var c = -1; c <= 1; c += 2) {
            for (var d = -1; d <= 1; d += 2) {
                if (12 + 34 * a + 56 * b + 78 * c + 9 * d === 59) {
                    console.log(a,b,c,d);
                    break outer;
                }
            }
        }
    }
}

```

8. 请完成下面的题目



把1~6的数填入圆中，每条线上的数相加为10
使用编程思维解决

```
/**  
 * 把小圆用abcdef6个变量去代替就可以了  
 */  
for (var a = 1; a <= 6; a++) {  
    for (var b = 1; b <= 6; b++) {  
        for (var c = 1; c <= 6; c++) {  
            for (var d = 1; d <= 6; d++) {  
                for (var e = 1; e <= 6; e++) {  
                    for (var f = 1; f <= 6; f++) {  
                        if (a + b + c === 10 && b + e + f === 10 && d  
+ c + e == 10 && a + b + c + d + e + f === 21 && a * b * c * d * e * f  
== 720) {  
                            console.log(a, b, c, d, e);  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

9. 打印出100以内的素数。

素数：除了1与自身以外不能被其它的数整除的数叫素数

```

// 除了1和自身以外，不能被其它的数整数
// 2~100之间，素数是从2开始的
//2<=x<=100

for (var x = 2; x <= 100; x++) {
    //假设型思维
    //我先假设这个数是素数，然后再反推
    var flag = true;
    //开始反推验证
    for (var n = 2; n < x; n++) {
        if (x % n === 0) {
            flag = false;
            //上面的假设已经被推翻了，就不再对这个数进行后面的验证了
            break;
        }
    }
    //在这里看一下假设的结果
    if(flag){
        console.log(x);
    }
}

```

10. 题目：日本某地发生了一起谋杀案，警察通过排查确定杀人凶手为4个嫌疑犯的一个，以下为四个嫌疑犯的供词

A说：不是我

B说：是C

C说：是D

D说：C在胡说

已知三个人说了真话，一个人说了假话，请编程找出凶手

```

/*
1.四个人里面只有1个凶手
2.四个人里面有3个说了真话

```

假设条件：凶手我们用1表示，不是凶手就用0表示

$a + b + c + d == 1;$

12. 有一对幼兔，幼兔1个月后长成小兔，小兔1个月后长成成兔并生下一对幼兔，问8个月后有多少对兔子，幼兔、小兔、成兔对数分别是多少？

```
var yt = 1;
var xt = 0;
var ct = 0;
for (var month = 1; month <= 8; month++) {
    ct = xt + ct;
    xt = yt;
    yt = ct;
    console.log("第" + month + "月有成图" + ct + "只" + xt + "只" + yt + "只");
}
```

13. 猴子吃桃问题：猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个，第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想再吃时，见只剩下一个桃子了。求第一天共摘了多少？（提示：采用逆向思维的方向，从后向前推算）

```
var peach = 1;
for (var day = 9; day >= 1; day--) {
    peach = (peach + 1) * 2
}
console.log(peach);
```

终级题

1. 猜测产品质量评奖 5家工厂的产品在一次评比中分获1，2，3，4，5，在公布结果前，已知E厂产品肯定不是第二、三名，五厂代表猜测评比结果，
 A厂的代表说：E厂一定能获得第一名。
 B厂的代表说：我厂的产品可能获第二名。
 C厂的代表说：A厂的产品质量最次。
 D厂的代表说：C厂的产品不是最好的。
 E厂的代表说：D厂的产品会获得第一名。

公布结果后，证明只有产品获第一名和第二名的两个厂的代表猜对了。求5个厂产品各获第几名。

```

for (var a = 1; a <= 5; a++) {
  for (var b = 1; b <= 5; b++) {
    for (var c = 1; c <= 5; c++) {
      for (var d = 1; d <= 5; d++) {
        for (var e = 1; e <= 5; e++) {
          if (a + b + c + d + e === 15 && a * b * c * d * e
          === 120) {
            if (e != 2 && e !== 3) {
              // 判断说的话
              if ((e == 1) + (b == 2) + (a == 5) + (c !=
              1) + (d == 1) == 2) {
                var aSay = a <= 2 && e == 1;
                var bSay = b <= 2 && b == 2;
                var cSay = c <= 2 && a == 5;
                var dSay = d <= 2 && c != 1;
                var eSay = e <= 2 && d == 1;
                if(aSay+bSay+cSay+dSay+eSay === 2){
                  console.log(a,b,c,d,e);
                }
              }
            }
          }
        }
      }
    }
  }
}

```

2. 填写数字 设有算式如图所示,求出口中的数字,并打印出完整的算式

$$\begin{array}{r}
 & 8 & 0 & 9 \\
 & \boxed{\square \square} & & \\
 & \square & \square & \square \\
 & \hline
 & \square & \square & \square \\
 & \hline
 & \square & \square & \square \\
 & \hline
 & & & 1
 \end{array}$$

```

for (var x = 10; x <= 99; x++) {
    for (var y = 1000; y <= 9999; y++) {
        if (y % x === 1 && ~~(y / x) === 809) {
            console.log(y, x);
        }
    }
}

```

3. 海滩上有一堆桃子，五只猴子来分，第一只猴子把这堆桃子平均分为五份，多了一个，这只猴子把多的一个扔入海中，拿走了一份；第二只猴子把剩下的桃子又分成了五份，多了一个，它同样把多的一个扔入海中，拿走一份，第三、四、五只猴子都是这么做的，问海滩上原来最少有多少个桃子？
4. 题目：有一分数序列： $\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \dots$ 求出这个数列的前10项之和。

```

var fz = 2;
var fm = 1;
var sum = 0;
for (var i = 1; i <= 10; i++) {
    sum += fz / fm;
    var temp = fz; //temp就是原来的fz
    // 新的数的分子就应该是 原来的分母 + 原来的分子,
    fz = fm + fz;
    // 新的分母就应该是原来的分子,
    fm = temp;
}
console.log(sum);

```

5. 分书问题

有A、B、C、D、E五本书，要分给张、王、李、赵、钱五位同学，每人只能选一本，事先让每人把自己喜爱的书填于下表，编程找出让每个人都满意的方案。

	A	B	C	D	E
张			✓	✓	
王	✓	✓			✓
李		✓	✓		
赵	✓	✓		✓	
钱		✓			✓

```
//ABCDE分别当中1,2,3,4,5
//张王李赵钱每人都有可能是1~5之间
//1<=张<=5;
//1<=王<=5;
//1<=李<=5;
//1<=赵<=5;
//1<=钱<=5
for (var zhang = 1; zhang <= 5; zhang++) {
    for (var wang = 1; wang <= 5; wang++) {
        for (var li = 1; li <= 5; li++) {
            for (var zhao = 1; zhao <= 5; zhao++) {
                for (var qian = 1; qian <= 5; qian++) {
                    if (zhang + wang + li + zhao + qian == 15 && zhang
* wang * li * zhao * qian == 120) {
                        var result1 = zhang == 3 || zhang == 4;
                        var result2 = wang == 1 || wang == 2 || wang
== 5;
                        var result3 = li == 2 || li == 3;
                        var result4 = zhao == 1 || zhao == 2 || zhao
== 4;
                        var result5 = qian == 2 || qian == 5;
                        //我要同时满足这5个人的条件
                        if (result1 && result2 && result3 && result4
&& result5) {
                            console.log(zhang, wang, li, zhao, qian);
                        }
                    }
                }
            }
        }
    }
}
```

函数

函数对于编程语言来说是一个核心的概念，通过函数我们可以封装 任意多条语句，在任何地方任何时候调用执行

1. 任意多条语句
2. 任意地方任何时候
3. 调用执行

在ECMAScript里面，函数是通过关键字来定义 **function**

在学习函数之前，有些概念要先跟大家理清一下

函数也叫方法，在面向过程的编程语言里面它叫函数，在面向对象的编程语言里面，它叫方法。在JS当中你可以认为函数与方法就是一个东西

在学习这个章节之前，我们先要弄清楚一个问题，为什么需要函数，如果没有函数会有什么问题？

```
console.log("-----");
console.log("颜一鸣说他是我们班最帅的男孩子...");
console.log("-----");
```

// 再次打印一次

```
console.log("-----");
console.log("颜一鸣说他是我们班最帅的男孩子...");
console.log("-----");
```

如果上面的代码我们要不停的打印，或不固定的时间和地址点去打印，这样每次在使用之前我们都要去书写那三行代码，这个会很麻烦

同学们就想一下，有没有什么办法将这三条语句组合起来，就像我们之前在讲CSS的时候，如果多个页面有相同的样式，我们可以把样式写在公共的文件里面，然后通过 **link** 标签引入就可以了

JS也可以使用类似的原理，我们把公共的代码封装起来，要的时候直接调用就可以了

函数的定义

目前来说函数的定义方式是多种多样的，我们只学习ECMAScript中的标准定义方式，其它的定义方式我们到后面再学习

函数的定义使用的是关键字 `function`，它的语法格式如下

第一种定义方式

```
function 函数名(参数...?) {  
    // 函数的代码体  
}
```

```
// 封装了我们的第一个函数  
function sayHello() {  
    console.log("-----");  
    console.log("颜一鸣说他是我们班最帅的男孩子...");  
    console.log("-----");  
}
```

第二种定义方式

函数的定义其实与变量的定义是一样的，它也可以通过变量赋值的方式来完成

```
var 函数名 = function(参数...?) {  
    // 函数代码  
}
```

这一种定义方式其实就是将一个函数赋值给了一个变量

上面的代码我们可以同时使用下面的方式来完成

```
// 第二种定义方式  
var sayHello = function(){  
    console.log("-----");  
    console.log("颜一鸣说他是我们班最帅的男孩子...");  
    console.log("-----");  
}
```

 **警告：**两种定义方式最终得到的效果是一样的，唯一的区别就在于它们的调用方式，同学样在本阶段（没有进入ES6之前）先只使用第一种方式来定义

两种函数定义的区别

```
aaa(); //在定义之前调用不会报错
function aaa(){
    console.log("我是函数aaa");
}

bbb(); //通过var定义的函数在定义之前调用就会报错
var bbb = function(){
    console.log("我是函数bbb");
}
```

通过 `function` 关键字定义的函数是可以在任何地方调用执行的，但是通过 `var` 定义的函数只能在定义之后再调用执行（原因：后面会讲到一个东西叫执行上下文Execute Context）

函数的调用执行

一个函数封装完成以后，它不是自己执行，它只会调用以后才执行（所以函数也叫调用执行）

函数的调用执行方式非常简单，只需要通过 `函数名()` 就可以了

```
//这里是函数的封装
function sayHello() {
    console.log("-----");
    console.log("颜一鸣说他是我们班最帅的男孩子... ");
    console.log("-----");
}

// 函数名+()来调用执行
// 上面的函数名是sayHello, 我们调用就可以通过下面的方式来进行
sayHello()
```

函数的检测

函数也可以看成是一个变量，变量就会有数据类型，如果想检测一个变量是否是函数，我们可以通过关键字 `typeof` 来完成

```
var aaa = function(){
}

function bbb(){
}

console.log(typeof aaa);           // "function"
console.log(typeof bbb);           // "function"
```

函数作用域【重点】

之前我们在讲解变量的时候，我们只是讲解了变量的定义，并没有讲解作用域的概念，如下所示

```
{
    var a = 123;
}

console.log(a);
```

在上面的代码当中会正常的打印输出 123。这是因为在JS里面，普通的花括号是形成不了作用域，形成不了作用域，那么我们就把这些变量全都叫做**全局变量**

```
if(true){
    var a = 123;
}
console.log(a); //123
```

总结一：在ECMAScript里面，普通的花括号，如 `if, else, while, for, do` 等一系列的，都形成不了作用域，所以这些作用里面的变量全都是全局变量

只有通过函数的花括号里面定义变量才是**局部变量**

```
function bgg(){
    var b = 456;           //这里定义的b是局部变量，只能在bgg这个函数里面使用
}

console.log(b); //报错
```

总结二：函数体的花括号会形成作用域

函数的参数

函数的参数是一种特殊范围的变量，它可以实现函数外部的赋值，同时函数的内部也可以使用

请看下面的场景

```
function sayHello() {  
    console.log("-----");  
    console.log("颜一鸣说他是我们班最帅的男孩子...");  
    console.log("-----");  
}  
sayHello();  
  
function sayHello2(){  
    console.log("-----");  
    console.log("洪延军说他是我们班最帅的男孩子...");  
    console.log("-----");  
}  
sayHello2();  
/**  
 * 在上面的代码里，我们可以很明显的感觉到一点，代码的冗余量太高了  
 * 同时我们发现了一个点，.代码里面，只有姓名不一样，其它的都一样  
 * 我们有没有什么办法去解决呢  
 */
```

尝试第一次改造：失败

```
function sayHello() {  
    var stuName = "颜一鸣";  
    console.log("-----");  
    console.log(stuName+"说他是我们班最帅的男孩子...");  
    console.log("-----");  
}  
sayHello();
```

```
// 洪延军 他要说话了  
// 他必须将里面的stuName的值换成"洪延军"  
// 但是stuName定义在函数体里面，它是一个局部变量，访问不了  
// 最终失败  
sayHello();
```

尝试第二次改造：失败

```
// 全局变量
var stuName = "";
function sayHello() {
    console.log("-----");
    console.log(stuName+"说他是我们班最帅的男孩子..."); // stuName是全局变量，风险很大
    console.log("-----");
}

// 颜一鸣
stuName="颜一鸣";
sayHello();

// 洪延军
stuName="洪延军";
sayHello();

/*
在每次调用函数之前都要进行变量赋值
stuName是全局变量，风险很大，后期容易被其它人覆盖
如果用户在调用sayHello的方法之前忘记变量赋值也会出现问题
*/
```

上面的第二次改造没有错误，但是不完美

尝试第三次改造：成功

```
function sayHello(stuName) {
    // 这个时候的stuName就是一个特殊范围的变量
    // 这种特殊的变量叫参数
    // 参数是可以实现外部赋值，内部使用了
    console.log("-----");
    console.log(stuName+"说他是我们班最帅的男孩子..."); // stuName是局部变量
    console.log("-----");
}

// 调用函数
sayHello("颜一鸣");

sayHello("洪延军");
```

在第三次改造里面，我们就带出了函数的一个重要概念：参数

参数的应用点：什么情况下会使用参数？

当我们在封装函数的时候，如果发现某些值需要通过外部来设定的时候，我们就会使用参数

参数的注意事项：参数里面的值会随着函数体代码运行结束而自动销毁

```
function aaa(userName){  
    console.log("-----");  
    console.log(userName);  
    console.log("-----");  
}  
  
aaa("陈韩家"); //当代码运行结束以后，参数就销毁了  
aaa(); //这一次我们没有传值，userName上一次的值被销毁，这一次没有值，它就是  
undefined
```

形参与实参

形参：形式参数，指函数在定义的时候花括号里面的参数【说得通俗一点就是变量名】

实参：实际参数，指调用函数的时候的括号里面的值（它是一个具体的值，可以认为就是变量的值）

通俗来说，形参就是变量名，实参就是变量的值

```
function sayHello(stuName, age) {  
    console.log("大家好，我叫" + stuName + "，我今年年龄" + age);  
}  
sayHello("颜一鸣", 18);
```

上面的 `stuName` 与 `age` 代表的就是形参（具体的变量名），下面的"颜一鸣、18"就是实参（变量值）

一定一定要注意，实参的值传给形参，相当于变量的值赋值给了变量

同时还请看一下下面的代码

```

function sayHello(stuName, age) {
    console.log("大家好, 我叫" + stuName + ", 我今年年龄" + age);
}

sayHello("颜一鸣", 18); //var stuName = "颜一鸣"; var age = 18;

sayHello("李心悦"); //var stuName = "李心悦"; var age;

// 请思考下面的代码是否报错
sayHello("陈怡静", 19, "活泼开朗");
//var stuName = "陈怡静"; var age = 19; "活泼开朗"

```

在上面的代码里面，我们可以看到，函数里面的参数是可以一一对应的。在第二次调用 `sayHello` 的时候，我们的实参只有1个，而形参有2个，第三次调用的时候，我们有3个实参，这样也不会报错

实参与形参总结

1. 实参向形参赋值
2. 实参与形参的个数不需要一一对应

上面第2点与其它编程语言是完全不同的，其它的编程里面言里同，形参与实参必须相一一对应



函数的重载

重载：多个函数的名字相同，但是它们的参数的类型或参数个数不相同，这种现象我们就叫函数的重载，英文名叫 `overload`

其它的强类型的编程语言面，如C、C++、C#、JAVA等都有这种概念，但是JS里面没有重载

下面是JAVA的代码

```

public class Test {
    //第一个方法
    public void sayHello(String stuName, int age){
        System.out.println("大家好, 我叫" + stuName + ", 我的年龄是" + age);
    }
}

```

```

//第二个方法
public void sayHello(){
    System.out.println("我在给你打招呼啊... ");
}

//第三个方法
public void sayHello(String stuName){
    System.out.println("我是女生，我叫" + stuName + "，但是我不告诉你的我年龄");
}

public static void main(String[] args) {
    Test t = new Test();
    t.sayHello();
    t.sayHello("李心悦");
    t.sayHello("颜一鸣", 18);
}
}

```

在上面的代码里面，我们发现有三个函数都叫 `sayHello`，只是它们的参数的个数不相同，像这种现象，我们就叫重载

JavaScript是不具备重载的特性的

```

// 第一个
function sayHello() {
    console.log("我在给你打招呼啊... ");
}

// 第二个
function sayHello(stuName) {
    console.log("我是女生，我叫" + stuName + "，但是我不告诉你的我年龄");
}

// 第三个
function sayHello(stuName, age){
    console.log("大家好，我叫" + stuName + "，我的年龄是" + age);
}

```

在上面的JS代码里面，我们可以看到定义了3个同名函数，只是参数的个数不一样，这么做是不对的，JS没有重载的特性，所以最后一次出现的函数会将上面的函数全部覆盖掉，只会保留最后一次出现的

问题就在于：为什么JS没有重载的概念

原因一：之前已经讲过了，函数的形参与实参是不需要一一对应的，所以根据参数的个数来区分来函数是不行的（不能通过参数的个数来区分）

原因二：JS是弱类型语句，所有的变量都是通过 var 定义的，在没有赋值之前是不存在数据类型的说法的，所以也不能通过参数的类型去区分

正是因为没有重载的特性，所以函数名在定义的时候千万不能重复，否则就会是后面的函数覆盖前面的的函数

函数的返回值

返回值与参数其实是一个相对的过程

- 参数：函数外边的值赋值给函数里面使用
- 返回值：函数里面的值拿到外边来使用

函数的返回值是通过 return 关键字来实现的，每一个函数都可以设置返回值

```
function aaa(){  
    var x = "颜一鸣";  
    return x;  
}  
  
//当aaa()函数运行结束以后，最后返回x到外边，将这值在外边赋值给了变量stuName  
var stuName = aaa();  
console.log(stuName);
```

什么场景下面会使用返回值？

当一个函数运行结束以后，我们希望还要把某个值重新给外边去使用，这个时候就可以使用 return 将这个值返回到外边

```

+-----+
5 // 这个函数只将两个数相加，得到结果以后，我再返回到外边
6 function sum(a, b) {
7     var x = a + b;
8     return x;
9 }  

10 // 返回值，由内向外传递
11 var first = sum(11, 12);  

12 console.log(first);
13 </script>
14

```

参数，由外向内传递

返回值，由内向外传递

函数运行结束以后，接收返回值x，也就是23

函数的本质【重点】

今天我们讲了函数的定义，函数的参数，函数的返回值，那么试想一下我们在什么地方会需要函数？

要弄清楚这个问题，我们先从系统当中的函数讲起。在JS里面，有很多内容的函数

1. `Number()` 转换成数值类型
2. `Boolean()` 转换成布尔类型
3. `String()` 转字符串类型
4. `parseInt()` 转整数
5. `parseFloat()` 转小数

试想一下，你们在什么场景下面会使用到上面的函数？

①函数的本质就是为了完成某一个功能而将多个代码组合在一起

```

/**
 * 编写一个函数sum
 * 参数有2个
 * 作用：将2个参数相加，将结果返回到外边
 */
function sum(a, b) {
    var c = a + b;
    return c;
}

var x = sum(11, 12);
console.log(x);

```

在上面的代码里面，我们可以看到以下几

1. 定义了一个函数，取名为 `sum`

2. 传递了2个参数 `a, b`
3. 在函数的内部将两个参数相加，得到了结果 `c`
4. 返回了结果 `c`

②本质： 函数其实就是为了实现某些功能而存在的，所以我们把函数也叫方法，既然
是方法那么就可以拿过来使用，例如上面的方法 `sum` 就是对两个数求和

了解了本质点可以让我们明白后期在什么场景下面会需要使用到函数

递归函数

递归函数其实就是一个普通函数，只是它的调用方式比较特殊，这个函数在内部又调用了
自己，这种函数就是递归函数（套娃）

一般情况下，递归会与循环会结合的很密切

请看下边的案例：要求实现1~10之间每个数的打印

这个问题如果我们使用循环去解决会非常快

```
for (var i = 1; i <= 10; i++) {  
    console.log(i);  
}
```

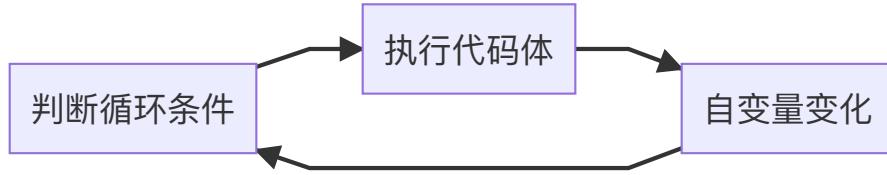
现在我们来分析一下这个代码，来列表一下这个循环的三个条件

1. 初始条件：`i=1`
2. 结束条件：`i<=10`
3. 自变量：`i++`

我们再来看一下这个循环的本质

1. 判断循环条件是否成立 `i<=10`
2. 执行循环体的代码 `console.log(i)`
3. 自变量的变化 `i++`

我们如果也能够让这三部分的代码循环执行，这样也是可以了



现在请看好，我们就用函数去完成

```

var i = 1;

// 完成一个功能，打印1~10
function abc() {
    console.log(i);      //代码体
    i++;                 //自变量变化
    if (i <= 10) {
        //console.log(i);  //代码体
        //i++;            //自变量变化
        abc();
    }
}
abc();

```

要求：请同学们使用递归完成 $1+2+3+\dots+10$ 的和

提示：如果不是很熟练的使用递归的，可以先用for循环写一遍以后再改写

```

var sum = 0;
var i = 1;
function abc() {
    sum += i;
    i++;
    if (i <= 10) {
        abc();           //在这里形成了递归
    }
}
abc();
console.log(sum);

```

带参数与返回值的递归函数【难点】

现有一个斐波拉契数列，排列是这样的1,1,2,3,5,8,13,21,34.....，求这个数列的第9项是多少？

第一种思维方式：使用循环语句去完成

```
var a1 = 1;           //前一个数
var a2 = 1;           //前二个数
var result;          //当前项的结果
for (var x = 1; x <= 9; x++) {
    if (x == 1 || x == 2) {
        result = 1;
    } else{
        result = a1 + a2;
        a1 = a2;
        a2 = result;
    }
}
console.log(result);
```

第二种思维方式：使用递归去完成

这个题目如果使用递归去完成，那将会变得非常简单

```
function abc(x) {
    //这里的x是参数，代表第几项
    if (x == 1 || x == 2) {
        return 1;
    } else{
        // 如果不是第1项或第2项
        // 前2项相加
        var c = abc(x-1) + abc(x-2);
        return c;
    }
}

var result = abc(9);
console.log(result);
```

注意事项

1. 函数里面是否还可以再定义函数? 【可以】

```
function sayHello(){
    console.log("大家好, 我是标哥哥");

    function abc(){
        console.log("我是abc函数");
    }
    // abc();    这里是不会有问题是
}

sayHello();
abc();      //调用不到, 因为在sayHello函数内部定义的, 有作用域, 只能在内部调用
```

2. 关于函数的返回值 return

函数不是因为结束的时候才return, 是因为有了 return 才会结束, 所有的函数都会有一个 return

当一个函数的内部一旦碰到了 return 它就会结束

```
function abc(){
    console.log("111111111111")
    return "标哥哥";          //函数运行到这里就结束了
    console.log("222222222222");
}

var x= abc();
```

为什么说函数内部都会有一个return

```

function abc() {
    console.log("11111");
    return;
}

function def() {
    console.log("22222");
    // 这里没有return 你有一个看不见的return
}

var x = abc();           //undefined
var y = def();           //undefined

```

练习

- 5个人坐在一起，问第五个人多少岁？他说比第4个人大2岁。问第4个人岁数，他说比第3个人大2岁。问第3个人，又说比第2人大两岁。问第2个人，说比第一个人大两岁。最后问第1个人，他说是10岁。请问第五个人多大？

第一种解决：使用for循环解决

```

// 循环执行了4次。
var age = 10;
for (var i = 1; i <= 4; i++) {
    age = age + 2;
}
console.log(age);

```

第二种解法：使用递归去完成

```

//递归肯定是一个函数。函数就是为了完成某一个功能而存在的
//我给你一个一个东西，你还我一个东西
//我给你一个人的序号，你还我这个人的年龄
    // 我给你一个人序号，你还我这个人的年龄。
    //我给你的东西叫参数，你还我的东西叫返回值
function getAge(i) {
    if (i == 1) {
        //第1个人的年龄是10岁
        return 10;
    }
    else {
        //i是当前这个人，前一个人就是i-1
    }
}

```

```

    //当前人的年龄应该比前一个人大2岁
    var age = getAge(i - 1) + 2;
    return age;
}
}

```

2. 编写函数,输入两个正整数参数m和n, (返回) 求其最大公约数和最小公倍数。

```

/*
我给你2个参数, 你返回我一个结果

*/
//最小公倍数, 我给你2个数, 你还我1个最小公倍数
function gbs(m, n) {
    //第一步: 先找m, n当中比较大的一个数
    var max = m > n ? m : n;
    for (var x = max; ; x++) {
        if (x % m == 0 && x % n == 0) {
            break;
        }
    }
    return x;
}

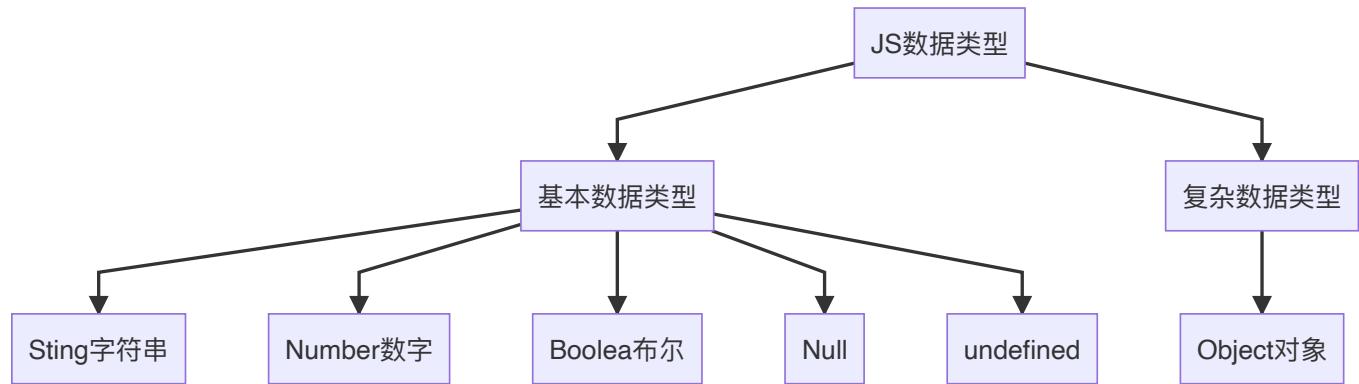
//最大公约数, 我给你2个数, 你还我1个最大公约数
function gys(m, n) {
    //第一步: 先找m, n当中比较小的一个数
    var min = m < n ? m : n;
    //第二步: 使用循环去找
    for (var x = min; ; x--) {
        if (m % x == 0 && n % x == 0) {
            break;
        }
    }
    return x;
}

```

提示一下: 可以先网上查询一下什么是最小公倍数以及最大公约数的求取方法

JavaScript数组

数组在大多数的编程语言里面都是会有的一个东西，在不同语言对数组的定义都是不同的，数组不是一个基本数据类型，我在前面讲解JavaScript数据类型的讲过



数组是我们学到第一个复杂数据类型，也是我所学习的第一个内置对象

数组的概念

JavaScript与其他的编程语言相对比，数组的概念是完全不一样的

- JavaScript里面的数组：一系列数据的集合
- 其他编程语言的数组：一系列相同数据类型的固定长度的数据集合

在上面的概念中，我们发现JS的数组没有限定数据的类型，也没有限定数组的长度

数组的定义

数组的定义也叫数组的创建，他有很多种定义方式

通过 `new Array()` 的方式来创建

第一种方式：直接通过 `new Array()` 的方式来创建数组

```
var arr = new Array();
// 定义了一个数组，当前这个数组里面什么也没有，这数组的长度为0
```

在上面的代码里面我们第一次使用到了 `new` 关键字，这个关键字在后期面向对象里面会具体讲解

第二种方式：创建数组的时候直接定义数组的长度 `new Array(arrayLength?:number):[]`

我们看上面的语法可以知道，在定义数组的时候可以直接确定数组的长度

```
var arr = new Array(10); // 定义了一个数组，且当前的数组长度为10
```

第三种方式：直接静态初始化，直接给多个元素的值

```
var arr = new Array("a", "b", "c", "d");
var arr2 = new Array("10", 10, true, null, undefined, NaN); // 定义了一个数组
```

这个时候我们发现，不管在数组内存入的数据是相同类型还是不同类型，这里都不会报错，这也应证了前面JavaScript数组的定于——一系列数据的集合

但是在定义数组的时候我们传入参数的时候要注意一些特殊情况

注意事项

我们看下面的代码

```
var a = new Array("2") // "2"代表元素，因为他是一个字符串类型
var b = new Array(2) // 2 代表数组的长度，因为他是一个数字类型

var c = new Array(-1) // -1 是数字类型他表数组的长度，但是他是一个负数而数组的长度不能为一个负值这里就会报错
var d = new Array(3.5) // 在这里3.5也是一个number类型因此会被系统识别为数组的长度但是3.5是一个小数他也不是一个合法的数组长度

var e = new Array(-1, -2) // -1, -2就都是数组的元素，相当于静态初始化
```

通过上面的代码，我们发现在定义数组的时候，如果只传一个参数并且是负数或者小数的时候，就会报错。一般情况下，这种定义方式很少使用。

思考：如何定义一个数组，该数组只有一个元素，并且是一个正整数，我们该怎么办？

通过 [] 字面量来定义

在上面的定义方式里面，我们可以看到，使用 `new Array()` 的方式来定义数组得到的结果就是一个 `[]`。所以我们现在就是用这个 `[]` 来定义数组，这种定义方式也叫字面量定义法

```
var a = new Array()      // 定义了一个空数组
var b = []              // 定义了一个空数组
var c = [1, 2, 3, 4, 5] // 静态初始化
var d = [2]              // 静态初始化
```

通过这种方式我们可以直接在 `[]` 里面给初始化数组，并且也不会像第一种定义方式那样受到参数的限制，这种定义方式才是我们经常使用的，但是这种方式也有一个缺点，他不能初始化数组的长度

注意事项：

看下面代码

```
var a = [1, 2, 3, 4, 5] // 这个数组的长度是多少? 5
var b = [1, , 3, 4, 5] // 这个数组的长度又是多少? 6
```

通过上面的代码我们可以知道，在JS里面的数组并不一定需要数据一一相连



数组的检测

在之前学习数据类型检测的时候，我们学习了 `typeof` 这个关键字，他可以用来检测数据类型，那么他是否也能用来检测数组呢？

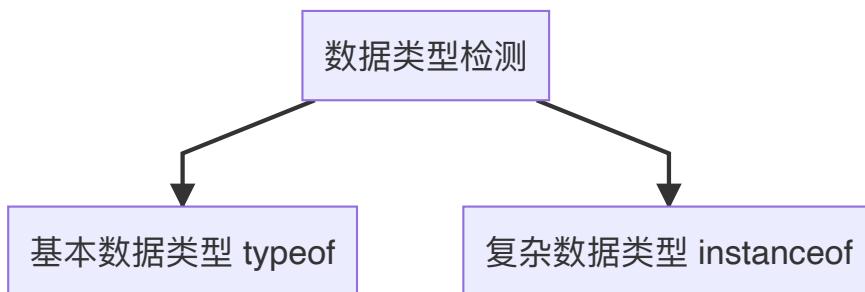
```
var arr = [1, 2, 3]
console.log(typeof arr); // "object"
```

看到当使用 `typeof` 关键字来检测数组的时候，得到的结果是 `object`。这样是不行的，因为 `object` 是对象，在JS里面所有的东西其实可以看成是一个对象

```
var day = new Date(); // 得到日期  
console.log(typeof day); // "object"  
  
var str = new String("Hello"); // 得到字符串  
console.log(typeof str); // "object"
```

在上面的代码里面我们看到使用 `typeof` 去检测的时候得到的结果都是 `object`，这并不是一个准确的结果，这个时候我们应该怎么办？

注意： `typeof` 关键字只适用于基本数据类型的检测，对于复杂数据类型使用 `typeof` 去检测时他给我们的全都是 `object`，为保证数据监测的准确性我们就需要使用另外一个关键字 `instanceof` 完成



通过 `instanceof` 关键字来完成检测

数组都是通过 `new Array()` 这一种方式来完成的，所以每个数组都应该是 `Array` 对象的实例

```
var arr = [1, 2, 3, 4, 5] //这里相当于 new Array(1, 2, 3, 4, 5)  
console.log(arr instanceof Array); // true  
  
var arr2 = new Array(1, 2, 3, 4, 5)  
console.log(arr2 instanceof Array); // true
```

所以只要通过 `instanceof Array` 得到的结果为 `true`，那么它就是一个数组

通过上面的代码，我们也可以反推出下面的代码

```
var a = new Array();  
var b = new Date();
```

针对上面的a, b这两个变量我们对它们去进行数据类型检测

```
var a = new Array();
var b = new Date();

console.log(a instanceof Array);      // true
console.log(a instanceof Date);       // false
console.log(b instanceof Array);      // false
console.log(b instanceof Date);       // true
```

通过 `Array.isArray()` 来检测

`Array.isArray()` 这个方法是数组对象里面专门用于检测数组的方法，如果结果为 `true` 则说明它是一个数组，否则就不是

```
var a = new Array();
var b = [1, 23, 4];
console.log(Array.isArray(a));    // true
console.log(Array.isArray(b));    // true
var c = new Date()
console.log(Array.isArray(c));    // false
```

数组的取值与赋值

数组是一系列数据的集合，我们如果吧数据放到数组内部后，怎么去取值与赋值呢？

数组的取值与赋值是通过数组下标（索引）来完成的，索引从0开始，通过 `数组名[索引]` 这种方式来完成取值与赋值的过程。

数组里面的只也叫数组的元素。

```
var arr = [1, 2, 3, 4, 5]

/**
 * 在数组里面元素的取值与赋值都是通过下标来完成的
 * 注意： 数组的下标是从0开始的
 */

var num = arr[2]      // 在这里我通过数组下标拿到了数组中的第三个元素

arr[2] = 10 // 这里我们就完成了赋值的过程将10赋值给数组的第三个元素
```

```
var aaa = arr[10]
/*
undefined 即使在取值的过程中我们超过了数组的长度在JavaScript也不会报错
这里与其他强类型语言是不同的
*/
arr[20] = "a"
/* 这里也不会报错 JS数组没有固定长度,
如果赋值的时候超过了原来的长度则会改变原数组的长度
JS数组也没有类型限制
*/
*/
```

JS数组与其他语言数组的区别

1. JS数组的数据类型可以不用保持一致，而其他强类型语言必须是一样的
2. JS数组里面的下标是可以越界访问的【越界的值为undefined】，但是其他类型语言会报错
3. JS数组长度是有出现的最大下标所决定的，而其他编程语言由长度来决定最大下标



数组最多可以包含 4 294 967 295 个项，这几乎已经能够满足任何编程需求了。如果想添加的项数超过这个上限值，就会发生异常。而创建一个初始大小与这个上限值接近的数组，则可能会导致运行时间超长的脚本错误。

数组的遍历

遍历就是指将数组中的每一个元素都拿出来走一遍

```
var arr = [0,1,2,3,4,5,6,7,8,9]
```

如果现在我们希望将上面的数组每一个元素都拿出来，打印一边，怎么办？

我们可以通过索引去拿到每一个元素，现在我们知道数组初始索引为0，结束索引为9

```
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for (var i = 0; i <= 9; i++) {
    console.log(arr[i]);
}
```

通过上的代码我们就完成了一个数组的遍历，但是这种写法有一个隐患，我们怎么知道数组的结束索引就是 9 呢？

在前面讲过，JS数组的长度是不固定的，所以这个地方的长度10是会发生变化的，这个时候我们就应该用一个变化的值去替代我们的9

在数组里面有一个东西叫做 `length`，他是数组对象的属性，用于描述当前数组的长度。所以上面的代码我们就可以改写成下面的样子

```
for (var i = 0; i < arr.length; i++) {  
    // i 代表每个元素的索引  
    console.log(arr[i]);  
}
```

数组的常用属性及方法

属性：用于描述对象事物的特征的我们叫属性

方法：在面向对象里面，方法也叫函数。所以方法可以理解为我们之前所学习的函数

属性是用于描述的，方法是用于使用的，学习数组里面的方法与属性可以让我们更加清楚的了解数组的特征，并使用数组

1. `length` 属性，该属性用于获取或设置数组的长度

```
var arr1 = ["张三", "李四"];  
console.log(arr1.length);  
arr1.length = 10;           //重新设置数组的长度
```

2. `push(...item):number` 方法，该方法是向数组的最后面追加新的元素，并返回当前数组的长度

```
var arr1 = ["张三", "李四"];  
//在数组的最后面追加了“王五”，并返回数组的长度3赋值给x  
var x = arr1.push("王五");      // 这里的x就是3  
  
//它还可以同时添加多个元素  
var y = arr1.push("赵六", "田七");  
//这个时候arr1就向数组后面同时添加了2个元素， 返回了新数组的长度为5
```

3. `pop()` 方法，从数组的最后面移出一个元素，并返回这个移出的元素

```
var arr1 = ["张三", "李四", "王五"];  
//移除数组的最后一个元素，并返回了这个移出的元素  
var x = arr1.pop();          //x的值就是王五，arr1就变成了["张三", "李四"]
```

4. `unshift(...item):number` 从数组的最前面追加新元素，并返回数组的新的长度

这一个方法与 `push` 方法是相对的

```
var arr1 = ["张三", "李四", "王五"];
arr1.unshift("赵六");      //["赵六", "张三", "李四", "王五"]

//同样，它也可以追加多个
arr1.unshift("a", "b");
//["a", "b", "赵六", "张三", "李四", "王五"]
```

5. `shift()` 方法，移除数组最前面的元素，并返回这个移除的元素

```
var arr1 = ["张三", "李四", "王五"];
var x = arr1.shift();
//数组会变成["李四", "王五"], x的值为"张三"
```

在上面的4个方法里面，同学们要注意它是可以在数组的最前面和最后面新增或移除元素的，它会涉及到后期我们讲解数据结构里面的相关操作【队列与栈的操作】

6. `concat()` 方法，将多个数组拼接成一个新的数组，返回这个新的数组的，原数组不变

```
var arr1 = ["a", "b", "c"];
var arr2 = ["张三", "李四"];
var arr3 = ["王五", "赵六"];

//将arr1与arr2接在一起，形成result1
var result1 = arr1.concat(arr2);
//['a', 'b', 'c', '张三', '李四']

//将arr1, arr2, arr3接在一起，形成result2
var result2 = arr1.concat(arr2, arr3);
//['a', 'b', 'c', '张三', '李四', '王五', '赵六']
```

如果要将多个数组拼接，还可以使用下面的方法来进行

```
var arr1 = ["a", "b", "c"];
var arr2 = ["张三", "李四"];
var arr3 = ["王五", "赵六"];

//将arr1,arr2,arr3接在一起，形成result3 还有没有其它的方法？

var result1 = arr1.concat(arr2); //['a', 'b', 'c', '张三', '李四']
//现在我们只用将上面的结果再接上arr3就可以了
var result2 = result1.concat(arr3); //['a', 'b', 'c', '张三', '李四',
'王五', '赵六']
```

现在再将上面的代码做精简

```
var result2 = arr1.concat(arr2).concat(arr3);
//相当于我们之前所学过的arr1.concat(arr2,arr3);
```

? 小问题：请同学们看下面的代码

```
var arr1 = ["颜一鸣", "李心悦", "李康"];
// 现在的arr1与arr2是相同的
// 相当于把这个数组复制一遍
var arr2 = arr1;
```

这个时候，上面的代码 `arr1` 与 `arr2` 是一样的元素，相当于复制了一份数组，但是这种复制方式，我们叫浅拷贝，它们之间是会存在问题的

```
> arr1
< ► (3) ['颜一鸣', '李心悦', '李康']
> arr2
< ► (3) ['颜一鸣', '李心悦', '李康']
```

```
arr1[0] = "颜二鸣";
console.log(arr2[0]); //颜二鸣
```

当我们去更改数组里面的某一个元素的时候，我们就会发现另一个数组也发生了改变，这样就是相互影响（浅拷贝就会相互影响，这点在后期的面向对象里面给大家讲解）

```
> arr1
< ▶ (3) ['颜二鸣', '李心悦', '李康']
> arr2
< ▶ (3) ['颜二鸣', '李心悦', '李康']
> |
```

如果我们想得到1个互不影响的但是又相同元素的数组，怎么办呢？

```
var arr1 = ["颜一鸣", "李心悦", "李康"];
// 新数组与原数组属于2个不同的数组
var arr2 = arr1.concat();
//现在再去改变里面的某个元素
arr1[1] = "我心悦";
console.log(arr2[1]);           //李心悦
```

如果想得到2个互不影响的数组，我们可以通过 **concat** 去实现

```
> arr1
< ▶ (3) ['颜一鸣', '我心悦', '李康']
> arr2
< ▶ (3) ['颜一鸣', '李心悦', '李康']
```

7. **slice(start?:number, end?:number):Array** 截取数组中的某些元素，形成一个新的数组，原数组不变

它会从 **start** 的位置开始截取，一直到 **end** 的前一个结束【开始位置不能在结束位置的后面】

```

var arr1 = ["a", "b", "c", "d", "e", "f", "g"];
// slice从开始位置开始，到结束的前一个

// ["b", "c", "d", "e"];
var arr2 = arr1.slice(1, 5);

//省略了end结束，则默认会在到最后
var arr3 = arr1.slice(2);
//[ 'c', 'd', 'e', 'f', 'g']

//这里省略了开始与结束，这样会从第1个开始，截取到最后一个
//相当于把原来的数组复制了一份
var arr4 = arr1.slice();

```

现在请看几种特殊场景

```

var arr1 = ["a", "b", "c", "d", "e", "f", "g"];

var arr2 = arr1.slice(3, 1);    //开始位置不能在结束位置后面 []

var arr3 = arr1.slice(1, -1);   //从第1个开始，到倒数第1个
//[ 'b', 'c', 'd', 'e', 'f']

var arr4 = arr1.slice(-4, -1);
//[ 'd', 'e', 'f']

//如果不是数字，则进行隐式类型转换
var arr5 = arr1.slice(1, "4");
//[ 'b', 'c', 'd']

//转换以后得到NaN，错误，返回空数组 []
var arr6 = arr1.slice(0, "a");

```

最后再说一点，`slice` 也可以实现数组的复制

8. `reverse():Array` 方法，将原数组里面的元素进行反转，形成一个新的数组，原数组也会改变

```
var arr1 = ["a", "b", "c", "d"];
// arr2就是arr1反转以后的数组
var arr2 = arr1.reverse();           //['d', 'c', 'b', 'a']

arr1 === arr2;        //true 它们是一样的东西，原数组也变了
```

9. `toString():string` 方法，该方法会将数组里面的元素使用逗号隔开以后，然后形成字符串

```
var arr1 = ["颜一鸣", "李心悦", "李康"];
var str = arr1.toString();
 //'颜一鸣,李心悦,李康'
```

10. `join(separator?:string):string` 使用指定的字符串将数组的元素隔开，然后形成字符串

`join` 方法其实就是 `toString()` 方法的高级版

```
var arr1 = ["颜一鸣", "李心悦", "李康"];
var str1 = arr1.join("#");          //'颜一鸣#李心悦#李康'
var str2 = arr1.join("~");          //'颜一鸣~李心悦~李康'
var str3 = arr1.join();             //'颜一鸣,李心悦,李康'
//如果没有指定分隔字符，则相当于toString()，默认会使用逗号隔开
```

11. `splice(start:number, deleteCount?:number, ...items?):Array` 方法

从指定的位置开始，删除指定个数的元素，并放入新的数组元素，删除的元素会形成一个数组

`start` 代表开始删除的位置

`deleteCount` 代表要删除的元素的个数

第一种场景，只删除，不新增，这一种场景就可以实现在数组的某个位置去删除元素

```
// 只删除，不新增
var arr = ["颜一鸣", "李心悦", "李康", "贺锐", "左小龙"];
var result1 = arr.splice(1, 2);
//arr的结果： ['颜一鸣', '贺锐', '左小龙']
//result1就是删除的元素，会形成一个新的数组['李心悦', '李康']
```

第二种场景，只新增，不删除，这一种场景就可以实现数组的某个新增去插入元素

```
var arr = ["颜一鸣", "李心悦", "李康", "贺锐", "左小龙"];
var result2 = arr.splice(3, 0, "桃子");
// arr最终的结果 : ['颜一鸣', '李心悦', '李康', '桃子', '贺锐', '左小龙']
// 因为没有删除的元素, 所以result2就是[ ]空数组
```

同时还请同学们注意一下，在新增的时候可以同时新增多个

```
arr.splice(3, 0, "桃子", "花花");
// arr的结果 ['颜一鸣', '李心悦', '李康', '桃子', '花花', '贺锐', '左小龙']
```

第三种场景：既删除，也新增，这种操作相当于替换

```
var arr = ["颜一鸣", "李心悦", "李康", "贺锐", "左小龙"];
var result3 = arr.splice(1, 1, "母牛");
//arr的结果 ['颜一鸣', '母牛', '李康', '贺锐', '左小龙']
// 替换出来的result3 结果 ['李心悦']
```

特殊场景，特殊对待

```
var arr = ["a", "b", "c", "d", "e", "f", "g"];

// var arr1 = arr.splice(1); //["a"];
// var arr2 = arr.splice(-2); // ['a', 'b', 'c', 'd', 'e']

// var arr3 = arr.splice(1,-1); //无效 不删除任何元素

// var arr4 = arr.splice(1,"a"); //无效 不删除任何元素

var arr5 = arr.splice(1, "2"); //['a', 'd', 'e', 'f', 'g']
```

12. `index0f()` 在当前数组中查询某个元素的索引值，如果找到这个元素就返回这个元素的索引，找不到就返回 `-1`

它的语法格式如下

```
index0f(searchElement:string|boolean|number, fromIndex:number?):number;
```

我们现在就来试一下这个方法

```
var arr = ["颜一鸣", "李心悦", "李康", "贺锐", "左小龙"];
// 查询李康的位置
var index = arr.indexOf("李康"); //2
var index2 = arr.indexOf("桃子"); //找不到就返回-1
```

特殊情况一：在查找的时候，如果出现了相同的元素，则我们就要返回第一次查询的索引结果

```
var arr = ["颜一鸣", "李心悦", "李康", "贺锐", "李心悦", "左小龙"];
var index1 = arr.indexOf("李心悦"); //1 只返回第一次查询的结果
```

特殊情况二：查询的时候它做的是强判断 `==` 全等判断操作？

```
var arr = ["标哥", true, 1, 2, 3, 4, "5", 6, 7];
var index1 = arr.indexOf(5); // -1
```

特殊情况三：查询的时候不是从索引0的位置开始的

```
var arr = ["a", "b", "a", "c", "b", 100, "true", false];
var index1 = arr.indexOf("b"); //1

var index2 = arr.indexOf("b", 2); //4
// 它会去找"b"，便是找到索引如果小于fromIndex则忽略，再继续找
```

后期这个方法的使用面会非常广，我们可以通过这个方法来判断某个元素是否在这个数组里面

13. `lastIndexOf(searchElement:boolean|string|number, fromIndex?:number):number` 查找某一个元素最后一次出现的位置

这里的 `fromIndex` 标哥更希望你们理解成 `endIndex`

```
var arr = ["a", "b", "a", "c", "b", "a", 100, "true", false];
var index1 = arr.lastIndexOf("a"); //5

var index2 = arr.lastIndexOf("a", 4); //
```

在上面的知识点里面，我们看到了12个方法

课堂练习

- 现有一数组，求数组中和奇数的和

```
var arr = [11, 23, 56, 78, 90, 45, 22, 31, 66];
var sum = 0;
for (var i = 0; i < arr.length; i++) {
    if (arr[i] % 2 != 0) {
        sum += arr[i];
    }
}
console.log(sum);
```

这个练习主要使用的就是如何对数组里面的元素取值和赋值，然后使用 `length` 属性来获取数组的长度，最后遍历数组

- 还是上面这个数组，请求出数组当中最大的一个值

```
var arr = [11, 23, 56, 78, 90, 45, 22, 31, 66];
/**
 * 假设型思维
 * 假设第1个数是最大的数，然后剩下的每一个都跟它比较
 * 比武招亲的思维
 */
var max = arr[0]; //假设第0个元素是最大值
//剩下的所有的人都要和站在台上的人比较，这个舞台就是max
for (var i = 1; i < arr.length; i++) {
    if (max < arr[i]) {
        max = arr[i];
    }
}
```

- 还是上面的数组，请删除上面数组里面奇数值的项，保留偶数

```
var arr = [11, 23, 56, 78, 90, 45, 22, 31, 66];
//[56,78,90,22,66]
for (var i = arr.length - 1; i >= 0; i--) {
    if (arr[i] % 2 != 0) {
        arr.splice(i, 1);
    }
}
```

这个题目本身难度并不是很大，关键点在于要注意数组操作里面的沙漏效应（也叫队列效果）。一般情况下，在删除数组元素的时候，我们要使用倒序遍历

4. 现给定一个数组，元素如下

```
var arr = [11, 23, 56, 78, 90, 45, 22, 31, 66];
```

请编写代码，找出原数组当中最大的这个值，把这个值放到一个新数组里面去，然后再回到原来的数组当中删除这个元素，依次执行上面的的叫什么，这样直到原数组的长度为0，这样新数组就会是一个从大到小排列的数组

```
var arr = [11, 23, 56, 78, 90, 45, 22, 31, 66];
var newArr = [];

//编写一个方法，用于求数组当中的最大值
//我给你一个数组，你还我这个数组中最大的值
function getMax(arr1) {
    //在操作之前，要先检测一下，你是否是数组
    if (Array.isArray(arr1)) {
        var max = arr1[0];
        for (var i = 1; i < arr1.length; i++) {
            if (max < arr1[i]) {
                max = arr1[i];
            }
        }
        return max;
    }
    else {
        return NaN;
    }
}

while (arr.length > 0) {
    //第一步：求原数组最大值
    var max = getMax(arr);
    //第二步：将这个值放到新数组中
    newArr.push(max);
    //第三步：在原数组中去删除
    var index = arr.indexOf(max);
    arr.splice(index, 1);
}
console.log(newArr);
```

JavaScript函数（二）

之前已经讲过了函数的基础，并且也在数组里面运行了函数，函数在面向对象的语言里面叫方法，它是为了完成某一引起功能封装的一个集体

函数的定义

之前已经学过了函数的定义，现在我们再来补充一下

第一种定义方式：通过 `function` 的关键字来进行定义

```
function sayHello(){
    console.log("大家好");
}
```

第二种定义方式：这一种方式叫函数表达式

```
var sayHello = function(){
    console.log("我是标哥哥");
}
```

第三种定义方式：通过 `Function` 来定义，这一种完全不建议使用，了解就行了

```
var abc = new Function("abc");
```

从技术角度讲，这是一个函数表达式。但是，我们不推荐读者使用这种方法定义函数，因为这种语法会导致解析两次代码（第一次是解析常规 ECMAScript 代码，第二次是解析传入构造函数中的字符串），从而影响性能。不过，这种语法对于理解“函数是对象，函数名是指针”的概念倒是非常直观的。

函数的参数

函数的参数分为2种

- 形参：函数在定义的时候的参数，是一个形式上面的参数，相当于变量名
- 实参：调用函数的时候的那个参数，它是一个实际的值，相当于变明不明白值

函数的在调用的时候，是实参向形参赋值

```
function sayHello(userName) {  
    console.log("大家好，我叫" + userName);  
}  
sayHello("标哥哥");
```

在上面的代码里面，“标哥哥”变量的值赋值给了 `userName`

问题就在这里，我们说过一个点，函数的形参与实参没有必要形成一一的对应关系

```
/**  
 * 编写一个函数，求参数相加的和  
 */  
function add(a, b) {  
    var sum = a + b;  
    return sum;  
}  
  
var x = add(11, 12);           //23 实参数与形参相等  
var y = add(11, 12, 13);       //23 实参数大于形参  
var z = add(11);              //NaN 实参数小于形参
```

ECMAScript里面的函数与其它语言里面的函数是不一样，它的参数不需要一一对应，指导书主上面的 `add` 在定义的时候有2个形参，但是在调用的时候不需要遵守

问题：请问 `add(11, 12, 13)`，第3个参数的13给了谁？？？？

参数不同情况的说明

1. 当实参数与形参数相同的时候，一对一对赋值
2. 当实参数小于形参数，没有接收到值的形参它就是 `undefined`
3. 当实参数大于形参数，前面的一对一赋值，多的值会放在一个特殊的地方，叫实参数组，这个实参数组叫 `arguments` 【特殊说明，后期我要更正这句话】

arguments实参数组

根据上面的理解，我们可以知道，函数里面所有的实参最终都会在 `arguments` 里面

```
function add(a, b) {  
    console.log(a);  
    console.log(b);  
    // 所有的实参都会在arguments里面  
    console.log(arguments);  
}  
  
add("颜一鸣", "李心悦", "李康");
```

在上面的代码里面，我们打印了 `arguments`，结果如下

[04实参数组.html:56](#)

```
Arguments(3) [ '颜一鸣', '李心悦', '李康', callee: f, Symbol(Symbol.iterator):  
f] ⓘ  
 0: "颜一鸣"  
 1: "李心悦"  
 2: "李康"  
▶ callee: f add(a, b)  
  length: 3  
▶ Symbol(Symbol.iterator): f values()  
▶ [[Prototype]]: Object
```

在上面的结果当中我们可以看到，所有传入到函数里面的实参都在 `arguments` 里面

你传递了多少个实参进去，那么 `arguments` 里面就会有多少个元素，如果不给实参，它就是空的

arguments解析

在上面已经得知，`arguments`就是所有实参的集合，它是一个什么东西呢？

在上面的图片里面，我们得到了一个点，它有索引，它也有 `length` 属性，长得像一个我们之前学习的数组。所以我们把 `arguments` 叫实参数组

```
function add(a, b) {  
    console.log(arguments);  
    console.log(arguments[0]);          //通过索引来取值  
    console.log(arguments.length);      //也可以获取长度  
    console.log(Array.isArray(arguments)); //false  
}  
  
add("颜一鸣", "李心悦", "李康", "贺锐", "左小龙");
```

在上面的代码里面，我们对 `arguments` 像数组一样的取值，然后也可以获取长度，但是通过数组的检测时发现，它不是一个数组，这是为什么呢？

arguments	Array
<pre>▼ Arguments(5) ['颜一鸣', '李心悦', '李康', '贺锐', '左小龙' symbol.iterator): f] ⓘ 0: "颜一鸣" 1: "李心悦" 2: "李康" 3: "贺锐" 4: "左小龙" ► callee: f add(a, b) length: 5 ► Symbol(Symbol.iterator): f values() ► [[Prototype]]: Object</pre>	<pre>▼ (5) ['颜一鸣', '李心悦', '李康', '贺锐', '左小龙'] ⓘ 0: "颜一鸣" 1: "李心悦" 2: "李康" 3: "贺锐" 4: "左小龙" length: 5 ► [[Prototype]]: Array(0)</pre>

在上面的对比里面，左边是 `arguments`，右边的数组，我们发现有以下几个点

1. 两边的对象上面都有索引，以及 `length` 属性，所以它们都可以通过索引来取值，也可以通过 `length` 来获取长度 【相同点】
2. `arguments` 只具备数组的特征，但是不具备数组的 `push/pop/shift/unshift` 等一系列方法 【不同点】

总结：在 `JavaScript` 里面，如果有一个对象具备数组的特征（索引与长度），但是又不具备数组的那些方法，我们就把这些对象叫**类数组（伪数组）**

所以上面的 `arguments` 并不是一个真正的数组，它是一个类数组

思考 想一想，这里的 `arguments` 有什么应用点

```
//我想编写一个方法，将所有的参数求和以后，将结果返回出去
/*
function add(a, b) {
    var sum = a + b;
    return sum;
}
var x = add(11, 12);
var y = add(11, 12, 13);
var z = add(11);

*/
function add() {
    //所有的参数都会在arguments里面，所以我们直接从这里找就行了
    // console.log(arguments);
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
```

```
        sum += arguments[i];
    }
    return sum;
}

var x = add(11, 12);
var y = add(11, 12, 13);
var z = add(11);
```

函数的调用

之前已讲过函数的执行方式，它是通过 **函数名+()** 来实现调用的，除了这一种调用方式以外，还有其它的几种调用方式我们来了解一下

常规调用

```
function sayHello(){
    console.log("我是标哥哥");
}
sayHello();
```

立即执行函数

当我们定义好一个函数以后，希望立即就执行这个函数，这个时候就可以使用下面的方式来完成

```
!function sayHello() {
    console.log("我是标哥哥");
}();
```

代码分析：感叹号是取非操作符，而取非是一定要得到结果才能取非，所以它会把后面的函数执行以后再取非，而后面的函数执行就代表要调用后面的函数

前面的 **!** 也并不固定的，它可以换成其它的运算符，如 **+** 也可以

下面是带参数的立即执行函数

```
function sayHello(userName) {
    console.log("大家好，我叫" + userName);
}
sayHello("标哥哥");
```

改成成立即执行函数

```
!function sayHello(userName) {  
    console.log("大家好，我叫" + userName);  
}("标哥哥");
```

函数表达式的执行

之前已经学过了函数表达式的定义，如下

```
// 函数表达式的执行  
var abc = function () {  
    console.log("我是一个函数");  
    return 123;  
}  
abc();  
//typeof abc == "function"
```

我们如果希望定义好了函数以后立即执行怎么办？

```
var abc = function () {  
    console.log("我是一个函数");  
    return 123;  
}();  
//这里的abc就是123
```

只要在函数表达式的后面加上一个括号，则函数表达式就立即执行了

注意事项

1. 在上面的代码里面 `abc` 这是函数的名子，我们是将一个函数赋值给了变量 `abc`
2. 在下面的代码里面，`abc` 就是后面的函数表达式执行完毕以后的返回值 `123`

闭包调用

在这里，先把闭包的概念放在一边，先看调用方式

TODO：后面讲到执行上下文以后再具体探讨闭包

闭包函数的写法其实非常简单，本质上它也是一个立即执行函数

```
function abc(){
    console.log("我是abc函数");
}
abc();
```

在上面的代码里面，我们通过函数名 **abc** 去调用了这个函数，我们将代码演变一下

```
(function abc(){
    console.log("我是abc函数");
})();
```

同时函数的名子 **abc** 还可以省略，因为现在我们已经不需要函数来调用了

```
(function(){
    console.log("我是abc函数");
})();
```

上面的函数我们就称之为闭包函数的写法，闭包函数的本质其实就是匿名的立即执行函数

闭包函数也是可以添加参数与返回值的

```
function add(a, b) {
    var x = a + b;
    return x;
}
var result = add(1, 2);
```

上面的函数通过闭包调用的形式去写

```
var result = (function(a, b) {
    var x = a + b;
    return x;
})(1,2)
```

匿名函数

匿名函数顾名思义就是没有名子的函数。什么情况下的函数是没有名子的

我们先来看一下之前的立即执行函数

```
!function sayHello(userName) {
    console.log("大家好，我叫" + userName);
}("标哥哥");
```

函数名的作用是用来找到这个函数，并且调用

在上面的立即执行函数里面，我们定义好这个函数以后就立即执行了这个函数，这个函数名的作用就消失了。这个时候就可以不需要函数名了

```
!function (userName) {  
    console.log("大家好，我叫" + userName);  
}("标哥哥");
```

回调函数【重点】

回调函数就是回调方法，它指的是将函数的当成参数传递到另一个函数当中去。在学习回调函数之前一定先弄清楚2个点

1. 函数的参数分为形参与实参
2. 函数是需要经过调用以后才会执行

回调函数的使用场景

1. 当一个函数的返回值无法返回的时候
2. 当某些功能需要分段执行的时候（俗称流水线操作）

请看下面代码

```
function abc(x, y) {  
    var a = x + 10;  
    var b = y + 20;  
    //如果我希望将a, b这2个值返回到外边，怎么办?  
}
```

我们在上面定义了一个函数，希望在函数运行结果以后向外边返回2个值 **a, b**，怎么办呢？

虽然说我们之前学过了 **return** 关键字，它可以在函数运行结束的时候把值返回到外边，但是只能返回一个，现在我们返回2个怎么办呢？

现在有2种方法，第一种是直接将2个数放在一个数组里面，然后把这个数组返回

```

function abc(x, y) {
    var a = x + 10;
    var b = y + 20;

    //我们现在是把a,b放在了一个数组里面，然后返回了这个数组
    var arr = [a, b];
    return arr;
}

var result = abc(11, 12);
console.log("第一个值: " + result[0]);
console.log("第二个值: " + result[1]);

```

上面这种方式虽然可以实现，但是比较麻烦，我们需要将结果封装成数组以后再返回

现在我们使用回调函数的方式去执行

The screenshot shows a browser's developer tools console. On the left, there is a code editor with the following JavaScript code:

```

<body>
</body>
<script>
    function abc(x, y, callBack) {
        var a = x + 10;
        var b = y + 20;

        //函数名+()就调用了函数
        callBack();
    }

    function def() {
        console.log("第一个值: ");
        console.log("第二个值: ");
    }

```

On the right, the console output is shown in a red-bordered box:

```

第一个值:
第二个值:

```

A red arrow points from the text "这就是def函数执行的结果" to the console output.

上面的代码当中，**def** 就是回调函数，我们把一个函数当成参数传递到了另一个函数里面，现在 **def** 函数仍然没有拿到 **a, b** 的值，怎么办呢？

The screenshot shows a browser's developer tools console. On the left, there is a code editor with the same code as before, plus an additional line at the bottom:

```

abc(11, 12, def);

```

On the right, the console output is shown in a red-bordered box:

```

Arguments(2) [21, 32, callee: f, caller: f]
第一个值:21
第二个值:32

```

Two purple arrows point from the word "arguments" in the code editor to the "callee" and "caller" fields in the console output. Another purple arrow points from the "def" parameter in the code editor to the "callee" field in the output.

回调函数最大的作用就让代码分段执行，也就是像流水线一样去执行

先看下面的场景

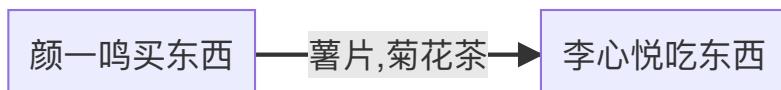
第一阶段，颜一鸣买东西
food, drink

第二阶段，李心悦吃东西
food,drink

我们现在如果想实现上面的场景，我们先采用普通的函数调用

```
function buy() {  
    var food = "薯片";  
    var drink = "菊花茶";  
    console.log("颜一鸣买了" + food);  
    console.log("颜一鸣买了" + drink);  
    lixinyue(food, drink); //在这里直接调用了李心悦  
}  
  
function lixinyue(_food, _drink) {  
    console.log("李心悦在吃" + _food);  
    console.log("李心悦在喝" + _drink);  
}  
  
function likuang(_food, _drink){  
    console.log("李康在吃" + _food);  
    console.log("李康在喝" + _drink);  
}  
buy();
```

上面的代码看起来会没有问题，但是不完美，因为buy函数内部固定的调用了 lixinyue，这就说明颜一鸣买完了东西以后只能给李心悦，而李康如果想吃东西不行的，因为代码在 buy 写死了



所以上面的写法并不好，按照我们程序开发的思维，每一个函数应该只负责完成一件事情，将多个函数组和就可以完成更多复杂的事件

现在我们将上面的代码进行改装

```
// 买东西
function buy(callBack) {
    var food = "薯片";
    var drink = "菊花茶";
    console.log("颜一鸣买了" + food);
    console.log("颜一鸣买了" + drink);
    // 颜一鸣买完东西以后应该就不再管，它要把东西交给下道工序
    if (typeof callBack === "function") {
        //说明callBack是一个函数，函数就可以调用
        //在调用函数的过程当中，我们可以传递参数
        callBack(food, drink);
    }
}

//李心悦吃东西
function lixinyue(_food, _drink) {
    console.log("李心悦在吃" + _food);
    console.log("李心悦在喝" + _drink);
}

//李康吃东西
function likuang(_food, _drink) {
    console.log("李康在吃" + _food);
    console.log("李康在喝" + _drink);
}

//buy();
//我希望颜一鸣买了东西以后，交给李心悦去吃
//buy(lixinyue);
//颜一鸣买了以后给李康吃
buy(likuang);
```

匿名函数的回调

```
// 买东西
function buy(callBack) {
    var food = "薯片";
    var drink = "菊花茶";
    console.log("颜一鸣买了" + food);
    console.log("颜一鸣买了" + drink);
    // 颜一鸣买完东西以后应该就不再管，它要把东西交给下道工序
    if (typeof callBack === "function") {
        //说明callBack是一个函数，函数就可以调用
        //在调用函数的过程当中，我们可以传递参数
        callBack(food, drink);
    }
}

//李心悦买东西
buy(function(_food, _drink) {
    console.log("李心悦在吃" + _food);
    console.log("李心悦在喝" + _drink);
});
```

直接在原来的代码上面使用匿名函数做了回调了，这样更方便

```

.5 // 买东西
.6 function buy(callBack) { 1 // 买东西
.7     var food = "薯片"; 2 function buy(callBack) {
.8     var drink = "菊花茶"; 3     var food = "薯片";
.9     console.log("颜一鸣买了" + food); 4     k = "菊花茶";
10    console.log("颜一鸣买了" + drink); 5     console.log("颜一鸣买了" + food);
11    // 颜一鸣买完东西以后应该就不再管，它要把东西交给下道工序 6     console.log("颜一鸣买了" + drink);
12    if (typeof callBack === "function") { 7         callBack(food,drink);
13        //说明callBack是一个函数，函数就可以调用 8     }
14        //在调用函数的过程当中，我们可以传递参数 9     callBack === "function") {
15        callBack(food,drink); 10    //说明callBack是一个函数，函数就可以调用
16    } 11    //在调用函数的过程当中，我们可以传递参数
17 } 12    }
18 //李心悦买东西
19 /*
20  * 13
21  buy(function(_food, _drink) { 14
22      console.log("李心悦在吃" + _food); 15     buy(function(_food, _drink) {
23      console.log("李心悦在喝" + _drink); 16         console.log("李心悦在吃" + _food);
24  }); 17     console.log("李心悦在喝" + _drink);
25  */
26
27 buy(function(_food, _drink){ 18
28     console.log("标哥哥在质检"+_food); 19     console.log("标哥哥在质检"+_drink);
29     console.log("标哥哥在质检"+_drink); 20 });
30
31 
```

直接在原来的代码上面使用匿名函数做了回调了，这样更方便

数组的高级方法

在之前学习数组的时候，我们已经学习了数组的12个方法

1. **push()** 向数组的后面追加新元素
2. **pop()** 从数组的后面移除一个元素
3. **unshift()** 从数组的前面追加新元素
4. **shift()** 从数组的前面移除元素
5. **concat()** 将多个数拼接，形成新数组
6. **slice()** 提取数组中的元素形成新数组
7. **toString()** 将元素用逗号隔开变成字符串
8. **join()** 使用指定的字符隔开变成字符串
9. **splice()** 替换元素
10. **indexOf()** 查找元素的索引
11. **lastIndexOf()** 查找元素最后一次出现的索引
12. **reverse()** 将数组里面的元素返回

迭代方法

迭代方法也叫遍历方法，迭代是把集合里面的元素依次的一个一个的拿出来

ECMAScript 5 为数组定义了 5 个迭代方法。每个方法都接收两个参数：要在每一项上运行的函数和（可选的）运行该函数的作用域对象——影响 this 的值。传入这些方法中的函数会接收三个参数：数组项的值、该项在数组中的位置和数组对象本身。

如果要学习迭代的方法，我们就先回顾一下之前我们是怎么遍历数组的

最原始的方式

```
var arr = ["a", "b", "c", "d", "e"];
// 我们采用最原始的方式来完成迭代
for (var i = 0; i < arr.length; i++) {
    console.log(i, arr[i]);
}
```

forEach方法

数组提供了很多个迭代的方法，其中 **forEach** 就是最基本的迭代方法，它会将数组里面的元素依次一个一个的拿出来，但是它只负责拿，拿出来以后给谁它不管

arr.forEach只负责拿

item,index,_arr

**这里就负责接收就可以了，
至于是谁拿的，我也不管**

bgg

两部分的函数各司其职，互不影响

```
var arr = ["a", "b", "c", "d", "e"];
// forEach就是其中的一种迭代方法
// 迭代就是把集合里面的值依次的一个一个的拿出来
arr.forEach(bgg);

//函数会接收三个参数：数组项的值、该项在数组中的位置和数组对象本身
function bgg(item, index, _arr){
    console.log(item, index);
}
```

上面的代码就是 `forEach` 最典型的迭代方法的使用，同样，我们还可以把上面的函数写成匿名回调的形式

```
var arr = ["a", "b", "c", "d", "e"];
// forEach就是其中的一种迭代方法
arr.forEach(function (item, index, _arr) {
    console.log(item, index);
});
```

map方法

它也是数组里面的一个遍历方法，它与 `forEach` 很相似，但是这个方法可以接回调函数的返回值，它会将回调函数的返回值组成一个新的数组

```
var arr = [11, 12, 13, 14, 15];
// 希望将上面的数组中的每一个元素都乘以2，然后放到一个新的数组里面去
var newArr = [];
/*
for (var i = 0; i < arr.length; i++) {
    // console.log(arr[i]*2);
    newArr.push(arr[i] * 2);
}
*/
arr.forEach(function(item, index, _arr){
    newArr.push(item*2);
});
console.log(newArr);
```

上面的2种方式都可以实现这种需求，但是我们的 `map` 会更好一些

```
var arr = [11, 12, 13, 14, 15];
var newArr = arr.map(function (item, index, _arr) {
    var x = item * 2;
    return x;
})
```

区别： `map` 方法与 `forEach` 方法是非常相似的，只是 `map` 可以将回调函数里面的返回值再构成一个新的数组

filter方法

`filter` 英文单词有过滤的意思，它会根据指定的条件在原数组当中过滤符合要求的元素（为true就会保留），再将这些符合要求的元素放在一个新数组里面，最后将这个数组返回

```
var arr = [1, 5, 7, 9, 2, 4, 6, 34, 21];
// 请将arr里的偶数提取出来，放在一个新的数组里面
var newArr = [];
arr.forEach(function (item, index, _arr) {
    if (item % 2 == 0) {
        newArr.push(item);
    }
});
```

在上面的代码里面，我们将原数组中的偶数拿出来，放在了新的数组 `newArr` 里面，判断这个数是否是偶数，我们使用了 `item%2==0` 这个条件

```
var arr = [1, 5, 7, 9, 2, 4, 6, 34, 21];
var newArr = arr.filter(function (item, index, _arr) {
    // 我要偶数
    return item % 2 == 0;
    // filter返回的是一个条件，它会根据这个条件自动判断
    // 如果成立就保留这个元素，如果不成立就不保留
});
```

some方法

这个方法相当于一真即真的操作，对数组中的每一项运行给定函数，如果该函数对任一项返回 `true`，则返回 `true`

```
var arr = [1, 3, 5, 9, 7, 4, 11];
// 请问，上面的arr当中有偶数吗？
//假设法，我假设上面没有偶数
// flag为false代表没有偶数，false为true代表有偶数
var flag = false;
for (var i = 0; i < arr.length; i++) {
    if (arr[i] % 2 == 0) {
        flag = true;
        break;
    }
}
console.log(flag?"有偶数":"没偶数");
```

这是我们以前的写法，现在我们要使用 `some` 的方式去完成

```
var arr = [1, 3, 5, 9, 7, 4, 11];
// 请问，上面的arr当中有偶数吗？
var flag = arr.some(function (item, index, _arr) {
    return item % 2 == 0;
});
// false||false||false||false||false||true||false
console.log(flag);
//上面的代码里面，它是将所有的结果执行或运算，最终的结果就是true
```

every方法

这个方法与 `some` 的方法是相对的，它执行的是一假即假的原则，它执行逻辑与的操作

```
var arr = [1, 3, 5, 9, 7, 4, 11];
//在上面数组里，每个元素都是奇数吗？
var flag = true;
// 先假设都是奇数
for (var i = 0; i < arr.length; i++) {
    if (arr[i] % 2 == 0) {
        flag = false;
        break;
    }
}
console.log(flag);
```

上面的代码我们仍然使用了假设法去完成，现在在数组里面，我们使用 `every` 去完成

```

var arr = [1, 3, 5, 9, 7, 4, 11];
// 问上面的数组中的元素是否全都是奇数
var flag = arr.every(function (item, index, _arr) {
    return item % 2 == 1;
});
//true&&true&&true&&true&&false&&true;
console.log(flag);
//every要对所有的结果做逻辑与判断，一假即假

```

- `every()`: 对数组中的每一项运行给定函数，如果该函数对每一项都返回 `true`，则返回 `true`。
- `filter()`: 对数组中的每一项运行给定函数，返回该函数会返回 `true` 的项组成的数组。
- `forEach()`: 对数组中的每一项运行给定函数。这个方法没有返回值。
- `map()`: 对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。
- `some()`: 对数组中的每一项运行给定函数，如果该函数对任一项返回 `true`，则返回 `true`。

以上方法都不会修改数组中的包含的值。

注意事项

上面的5个迭代方法都有一些注意事项，如下

1. 不能使用 `break` 与 `continue` 关键字来中断迭代
2. 所有的迭代方法执行的都是静态遍历，也是正向遍历，它在遍历开始的时候就已经定好了要遍历的次数，最终迭代的次数只能小于或等于最初定好的次数
3. 正是因为所有的迭代方法都是静态遍历，所以我们建议在使用迭代方法的时候去改变原数组的长度【也不要对数组进行新增，删除，替换的操作】

归并方法

reduce方法

它的语法格式如下

```

var arr = [10, 20, 30, 40, 50, 60];
var result = arr.reduce(bgg);

function bgg(prev, current, index, _arr){
    console.log(prev, current);
}

```

- `prev` 代表前一个回调函数的返回值【这一次回调函数的返回值会做为下一次回调函数的参数 `prev` 来使用】
- `current` 代表当前遍历的这一项

- `index` 代表当前遍历的索引
- `_arr` 代表当前遍历的数组
- `result` 接收的是整个归并函数的返回值

理解一下 `prev` 参数

```
var arr = [10, 20, 30, 40, 50, 60];
var result = arr.reduce(function(prev, current, index, _arr){
  console.log(prev, current);
  return "标哥哥" + index;
});

/***
  prev          current          index      返回值
  10            20              1          "标哥哥1"
  "标哥哥1"     30              2          "标哥哥2"
  "标哥哥2"     40              3          "标哥哥3"
  "标哥哥3"     50              4          "标哥哥4"
  "标哥哥4"     60              5          "标哥哥5"
*/
//最后一次的回调函数的返回值给了整个归并函数的结果result
```

通过上面的代码，我们可以发现几个点

1. 归并函数默认是从第2项开始的
2. 当前回调函数的返回值会做为下一次回调函数的 `prev` 参数使用
3. 最后一次回调函数的返回值给了整个归并函数 `result`

理解一下 `current` 参数

在上面的代码里面，我们可以看到，默认情况下的 `current` 是从第2项开始的，因为第1项给了 `prev` 参数

思考： `current` 有没有可能从第1项开始？

```
var arr = [10, 20, 30, 40, 50, 60];
var result = arr.reduce(function(prev, current, index, _arr){
  console.log(prev, current);
  return "标哥哥" + index;
  //这个时候 "颜一鸣" 就坐做为第一次回调函数的prev参数
}, "颜一鸣");
```

颜一鸣 10

标哥哥0 20

标哥哥1 30

标哥哥2 40

标哥哥3 50

标哥哥4 60

通过上面的例子我们可以得到，如果在归并的时候我们手动的指定了第一次回调函数的 `prev` 参数，则 `current` 就会从数组的第1项（索引为0的地方）开始

场景一：利用 `reduce` 来进行数组求最大值

```
var arr = [12, 4, 5, 98, 22, 56];
// 求上面数组的最大值
var max = arr.reduce(function (prev, current, index, _arr) {
  console.log(prev, current);
  return prev > current ? prev : current;
});
/***
    prev      current      返回值
    12        4            12
    12        5            12
    12        98           98
    98        22           98
    98        56           98
*/
```

场景二：利用 `reduce` 求数组的和

```
var arr = [12, 4, 5, 98, 22, 56];
// reduce
var sum = arr.reduce(function (prev, current, index, _arr) {
  console.log(prev, current);
  return prev + current;
});
/***
```

prev	current	返回值
12	4	16
16	5	21
21	98	119
119	22	141
141	56	197

*/

reduceRight方法

这个方法与 `reduce` 方法一致，它只是从右边开始遍历

```
var arr = [12, 4, 5, 98, 22, 56];
arr.reduceRight(function (prev, current, index, _arr) {
  console.log(prev, current);
});
```

排序方法

在数里面，我们经常会讲到排序，在其它的编程语言里面可能会还有很多个排序的算法，但是JS里面不需要，因为JS自带一从此排序的方法叫 `sort()`

`sort()` 方法会将当前方法进行排序，然后返回一个排序以后的新数组，新数组与旧数组相同

```
var arr = [1, 4, 5, 6, 3, 7, 9, 8, 2];
//现在希望将上面的数组排序
arr.sort();
//[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

上面的方式是数组提供给我们的默认的排序方式，但是这种方式有隐患

在默认情况下，`sort()` 方法按升序排列数组项——即最小的值位于最前面，最大的值排在最后面。为了实现排序，`sort()` 方法会调用每个数组项的 `toString()` 转型方法，然后比较得到的字符串，以确定如何排序。即使数组中的每一项都是数值，`sort()` 方法比较的也是字符串，如下所示。

它的隐患条件就是 `sort` 会将所有的元素转换成字符串以后再去比较大小

```
var arr = [1, 4, 5, 6, 3, 10, 7, 9, 8, 2];
//现在希望将上面的数组排序
arr.sort();
// [1, 10, 2, 3, 4, 5, 6, 7, 8, 9]
```

出现上面问题的原因就是因为 "10" < "2"

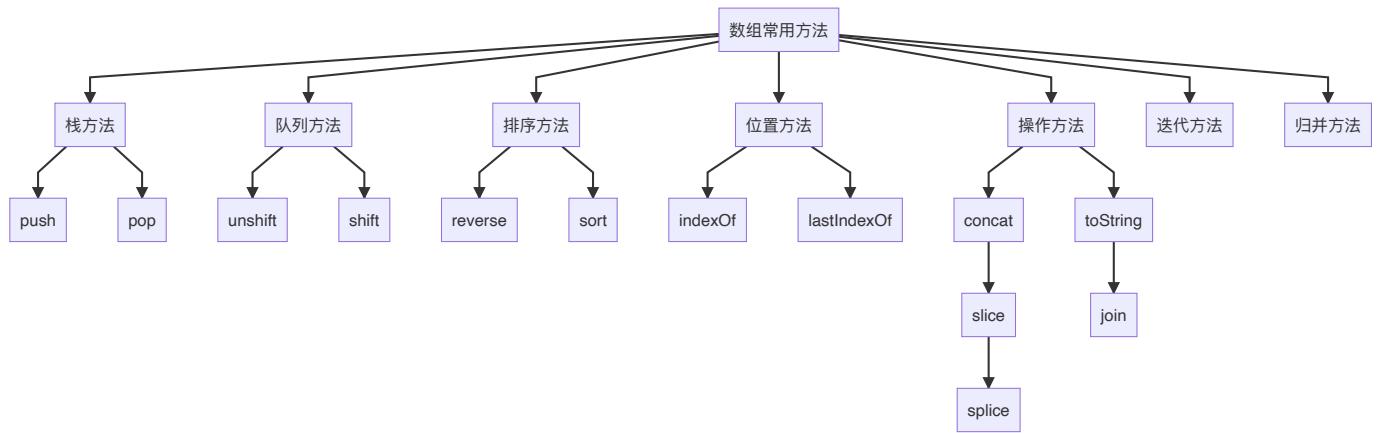
数组提供给我们的默认的排序方法是有问题的，所以我们要使用自己的排序方法

比较函数接收两个参数，如果第一个参数应该位于第二个之前则返回一个负数，如果两个参数相等则返回 0，如果第一个参数应该位于第二个之后则返回一个正数。以下就是一个简单的比较函数：

```
var arr = [1, 4, 5, 6, 3, 10, 7, 9, 8, 2];
//现在希望将上面的数组排序
arr.sort(function (a, b) {
    //这个回调函数就是我们自己定义的排序规则
    //参数a,b代表数组中的任意2项
    if (a < b) {
        return -1;
    }
    else if (a == b) {
        return 0;
    }
    else {
        return 1;
    }
});
//第1个数要放在第2个数的前面，则返回负数
//第1个数要放在第2个数的后面，则返回正数
//如果位置保持不变，则返回0
//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

在上面的函数里面，我们手动的向 `sort()` 里面传递了一个回调函数，回调函数里面的 `a, b` 代表两个要比较的值，我们只用指定这2个值的规则就可以了

数组方法的总结



课堂练习

1. 使用 `filter` 来过滤符合要求的元素

```

var arr = [1, "a", "2", 123, true, NaN, false, ""];
//请找出arr里在的number类型,放在一个新的数组里面

var newArr = arr.filter(function (item, index, _arr) {
  return typeof item == "number";
})
  
```

2. 使用 `filter` 来过滤符合要要求的元素

```

var arr = [5.1, 2, "3.14", true, "", NaN, null, "1", 9];
// 请找出arr当中的整数,放在新数组newArr里面

var newArr = arr.filter(function (item, index, _arr) {
  return item % 1 === 0 && typeof item === "number";
})
  
```

第二种解法

```
var newArr = arr.filter(function (item, index, _arr) {
    // return parseInt(item) === item;
    return ~~item === item;
});
```

课后练习

1. 编写一个方法，输入一个年份，判断这个年份是平年还是润年

```
function isLeapYear(year) {
    //代码体
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        //润年
        return true;
    }
    else {
        //平年
        return false;
    }
}
```

2. 编写一个方法，输入某年某月某日，返回这这一天是这一年的第几天

```
function isLeapYear(year) {
    //代码体
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        //润年
        return true;
    }
    else {
        //平年
        return false;
    }
}

function getDate(year, month, day) {
    var arr = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];
    //第一步：把当前这个月份之前的所有天数加起来
    for (var i = 0; i < month - 1; i++) {
        day += arr[i];
    }
}
```

```

// 是润年，同时月分还大于2，则需要加一天
if (isLeapYear(year) && month > 2) {
    day++;
}
return day;
}

```

3. 两个乒乓球队进行比赛，各出三人。甲队为a,b,c三人，乙队为x,y,z三人。以抽签决定比赛名单。有人向队员打听比赛的名单。a说他不和x比，c说他不和x,z比，请编程序找出三队赛手的名单。

```

//我们把x,y,z分别当成1, 2, 3
//a, b, c的出场顺序也有可能是1, 2, 3
outer:for (var a = 1; a <= 3; a++) {
    for (var b = 1; b <= 3; b++) {
        for (var c = 1; c <= 3; c++) {
            var aSay = a != 1;
            var cSay = c != 1 && c != 3;
            if (aSay && cSay) {
                if (a + b + c === 6 && a * b * c == 6) {
                    console.log(a, b, c);
                    break outer;
                }
            }
        }
    }
}
//x,y,z
//b,c,a

```

数组的解法

```

var arr1 = ["a", "b", "c"];
var arr2 = ["x", "y", "z"];
for (var i = 0; i < arr1.length; i++) {
    for (var j = 0; j < arr2.length; j++) {
        var item1 = arr1[i];           //第一个队出场的人
        var item2 = arr2[j];           //第二个队出场的人
        if (item1 == "a" && item2 == "x") {
            continue;
        }
    }
}

```

```

        else if (item1 == "c" && (item2 == "x" || item2 == "z")) {
            continue;
        }
        else if (item2 == "y" && (item1 == "a" || item1 == "b")) {
            continue;
        }
        else if (item1 == "b" && item2 != "x") {
            continue;
        }
        else {
            console.log(item1 + " vs " + item2);
        }
    }
}

```

4. 颜一鸣现在有1头母牛，母牛在长成4岁的时候生一头小母牛（3岁长成4岁的时候立即生一头小母牛），以后每年都生一头小母牛，每头母牛在10岁的时候就会死亡（由9岁长到10岁的时候就生一头小母牛，然后这头老母牛立即死亡）
请问在20年以后颜一鸣一共有多少头母牛？

```

var arr = [0];
for (var year = 1; year <= 20; year++) {
    //每一年开始的时候，我都重进定义一个数组，用于放新生的牛
    var newArr = [];
    for (var i = 0; i < arr.length; i++) {
        //如果当前这头牛的年龄3~9，就牛一头小母牛
        if (arr[i] >= 3 && arr[i] <= 9) {
            newArr.push(0);
        }

        //牛的年龄+1
        arr[i] = arr[i] + 1;
    }

    arr = arr.concat(newArr);
}

//排队年龄大于等于10岁的牛
var result = arr.filter(function (item, index, _arr) {
    return item < 10;
});

```

二维数组

- 在严格意思上面来讲，JavaScript没有二维数组的概念，它只有多维数组的概念
- 二维数组指的是当一个数组当中的元素又是一个数组的时候，它就变在了二维数组

二维的数组的定义

```
var teachers = ["标哥", "飞哥", "桃哥"];      //将老师放在了一个数组里
var stus = ["颜一鸣", "李心悦", "李康"];        //将学生也放在了一个数组里
var person = [teachers, stus];                //这个时候数组里面的元素又是一个数组
```

上面的 `person` 当中的 `teachers` 以及 `stus` 都是一个数组，这种情况我们就把 `person` 叫二维数组

```
var arr1 = new Array(3);
var arr2 = new Array(3);
var arr = new Array(arr1, arr2);
```

上面的代码还可以简化成下面的写法

```
var arr = new Array(new Array(3), new Array(3));
```

还可以直接静态初始化

```
var arr = new Array(new Array("a", "b", "c"), new Array("d", "e", "f"));
```

我们之前在学习一维数组的时候我们也知道可以使用 `[]` 来创建数组，所以上面的代码可以直接写成下面的写法

```
var arr = [
  ["a", "b", "c"],
  ["d", "e", "f"]
];
```

二维数组的定义方式是有多种多样的，只需要参照原来的一维数组进行就可以了

二维数组的特殊情况

首先我们先要弄清楚一， JS当中的数组与其它语言的数是不一样

1. 不限定数据类型
2. 不限定数组的长度

正是因为有了上面这两种特殊情况， 所以我们先来看一下下面的几种方式的二维数组

第一种特殊情况

```
var arr = [
    ["颜一鸣", "李心悦", "李康"],
    ["陈韩家", "许才奇", "何祥宇", "张恒"]
];
```

上面的代码是合法的， 因为JS的数组没有限定长度

第二种特殊情况

```
var arr =[ 
    ["颜一鸣", "李心悦", "李康"],
    "陈韩家",
    10
]
```

上面的代码也是合法的， 因为JS的数组没有固定数据类型， 里面的元素可以是任何数据类型

第三种特殊情况

```
var arr = [
    ["颜一鸣", "李心悦", "李康"],
    [
        ["标哥", "飞哥"],
        ["桃子"]
    ],
    "陈韩家",
    10
]
```

上面的代码也是合法的， 因为JS没有长度的限制， 也没有数据类型的限制

二维数组的取值与赋值

在之前讲一维数组的时候我们讲到过数组的取值是通过下标（索引）来完成的，同理，二维数组也是通过下标来完成的

```
var arr = [
  ["颜一鸣", "李心悦", "李康", "贺锐"],
  ["陈韩家", "许才奇", "何祥宇", "张恒"]
];
```

我们现在先在内存当中看一下这一个二维数组是如何表现

行	列	0	1	2	3
0	颜一鸣	李心悦	李康	贺锐	
1	陈韩家	许才奇	何祥宇	张恒	

二维数组的取值可以像表一样去操作

```
var row0 = arr[0];          //这相当于取了第0行的数据
var s1 = row0[1];           //这相当于取这一行的第一个数取
```

上在的操作是一步一步进行的，现在合起来一起进行

```
var s1 = arr[0][1];        //李心悦
var s2 = arr[1][2];         //何祥宇
```

赋值也是一样的，现在把张恒换成“标哥哥”

```
arr[1][3] = "标哥哥";
```

总结：多维数组的取值一层一层向下找就可以了

```
var arr = [
    ["颜一鸣", "李心悦", "李康"],
    [
        ["标哥", "飞哥"],
        ["桃子"]
    ],
    "陈韩家",
    10
];
```

在上面的数组里面，我们要找出"飞哥"

```
arr[1][0][1];
```

二维数组的应用点

二维数组在日常开发与工作当中用于展示表的信息是最好的，同时也是用于展示平面信息是最好的

操作说明：玩家1：wasd上左下右，space射击；玩家2：方向键，



上面的游戏地图就是一个平面信息，这个地图我们可以使用二维数组的方式去完成

同时，电影购票里面的坐位信息也是可以使用二维数组去完成的。



立

可选 空位

1排	<input type="text"/>	<input type="button" value="x"/>						
2排	<input type="text"/>	<input type="button" value="x"/>						
3排	<input type="text"/>	<input type="button" value="x"/>						

案例与练习

1. 现在有以下的二维数组，请将下面的二维数组中每一个元素遍历出来打印

```
var arr = [
    ["颜一鸣", "李心悦", "李康", "贺锐"],
    ["陈韩家", "许才奇", "何祥宇", "张恒"]
];
for (var i = 0; i < arr.length; i++) {
    // console.log(arr[i]);
    // 它又是一个数组，它又要遍历
    for (var j = 0; j < arr[i].length; j++) {
        console.log(arr[i][j]);
    }
}
```

上面的代码使用 `forEach` 的方式也是可以的。

```
arr.forEach(function (item) {  
    //item是当前的数组，我们要把这个数组再去做二次遍历  
    item.forEach(function(item2){  
        console.log(item2);  
    });  
});
```

2. 现在有以下的二维数组，请将下面的二维数组中每一个元素遍历出来打印

```
var arr = [
    ["颜一鸣", "李心悦", "李康", "贺锐"],
    ["陈韩家", "许才奇", "何祥宇", "张恒"],
    "陈韩家",
    10
];
```

```
arr.forEach(function (item) {
    //判断item是不是数组
    //如果是数组就二次遍历
    if (Array.isArray(item)) {
        item.forEach(function (item2) {
            console.log(item2);
        });
    }
    else {
        console.log(item);
    }
    //如果不是数组就直接打印
});
```

3. 现在有以下的多维数组，请将下面的多维数组中每一个元素遍历出来打印

```
var arr = [
    ["颜一鸣", "李心悦", ["李康", "贺锐"]],
    ["陈韩家", ["许才奇", "何祥宇"], "张恒"],
    "陈韩家",
    [10, [20, [30, 40]]]
];
```

上面的题目很明显是一个多维数组了，我们不能再手动的去打印了，要考虑另外的办法

提示：可以使用递归

```
function printArray(aaa){
    if(Array.isArray(aaa)){
        //首先就判断你是不是数组
        aaa.forEach(function(item){
            printArray(item);
        })
    }
    else{
        console.log(aaa);
    }
}
```

4. 打印杨辉三角的前10行

提示：使用二维数组去完成

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1
1   7   21  35  35  21  7   1
1   8   28  56  70  56  28  8   1
1   9   36  84  126 126 84  36  9   1
1  10  45  120 210 252 210 120 45  10  1
```

5. 100个人手拉手围成一个圆，开始数数，数到3的倍数的人退出，最后剩下一个人的是之前的第几个人 【约瑟夫环】

提示：使用一维数组完成

```
// 这里是100个人，我们创建一个长度为100的数组，每个人最初都用0表示
//第一步：创建长度为100的数组
var arr = new Array(100);
//第二步：在长度为100的数组里面全部放0，代表人
for (var i = 0; i < arr.length; i++) {
    arr[i] = 0;
}
//第三步：开始数数，数到3的倍数的人退出
//如果这个人数到3了，我们就把这个人的0换成1，1就表示这个人已经退出了
var count = 0;
while (true) {
    for (var i = 0; i < arr.length; i++) {
        if (arr[i] == 1) {
            //说明这个已经退出了，不用喊了
            continue;
        }
        count++;
        if (count == 3) {
            arr[i] = 1;
            count = 0;
        }
    }
}
```

```
//在什么情况下它就不数123了，只有剩下唯一的1的个人，用0表示，  
if (arr.indexOf(0) == arr.lastIndexOf(0)) {  
    break;  
}  
  
//第四步：最终理想的状态就是arr数组里面只剩下1个0  
//说明当前这个数组arr里面只能1个0  
console.log(arr.indexOf(0) + 1);
```

面向对象（一）

对象的概念

上面是通过语义的角度来理解什么是面向对象

如果它是一个对象，那么它应该具备以下几个特点

1. 对象具备属性

属性就是用于描述当前对象的特征的，可以理解为上面的【数据】

2. 对象具备方法

方法就是一个对象的能力，例如它可以做什么事情。可以理解为上面的【能力】

3. 对象应该是可以继承的

父级对象里面的某些方法或属性可以在子级对象里面，继续使用

 **思考：**为什么需要对象，如果没有对象会怎么办？

现在我想描述我们班同学的信息，应该怎么办呢

```
//颜一鸣同学的相关信息
var stuName = "颜一鸣";
var sex = "男";
var age = 18;
var hobby = "看书, 睡觉";

//现在我想描述一下另一位同学 曹慧的信息 怎么办
var stuName2 = "曹慧";
var sex2 = "女";
var age2 = 18;
var hobby2 = "做饭, 逛街";
```

在没有面向对象的情况下，如果我们想形容一些对象的数据【属性】我们必须要定义大量的变量，我们现在迫切需要有一种方式来进行集中式的数据管理

对象创建

对于同学们来说对象同学们已经接触过了，之前的数组就是对象。在JavaScript里面，对象的创建也叫对象的定义，它有很多种方式

通过 **Object** 来创建

```
//创建了一个对象
var obj1 = new Object();
//我现在反这个对象当成颜一鸣，去描述颜一鸣的数据
obj1.stuName = "颜一鸣";
obj1.sex = "男";
obj1.age = 18;
obj1.hobby = "看书, 睡觉";
```

这一种方式去创建对象是非常简单的，它直接通过 `new Object()` 就可以得到一个空的对象，然后在这个空的对象上面赋值属性就可以了

通过字面量 **{}** 来创建

这一种创建方式与之前数组的创建方式很相似，我们在讲数组的时候我们说过 `new Array()` 就相当于 `[]`，所以 `new Object()` 就相当于 `{}`

语法格式

```
var 对象名 = {  
    属性名1:属性值1,  
    属性名2:属性值2  
}
```

通过上面的语法，可以得到下面的结果

```
var obj1 = new Object();  
obj1.stuName = "颜一鸣";  
//等价于  
var obj2 = {};  
obj2.stuName = "曹慧";
```

使用字面量创建的时候还可以使用下面的方式创建

```
var obj1 = new Object();  
obj1.stuName = "颜一鸣";  
obj1.sex = "男";  
obj1.age = 18;  
obj1.hobby = "看书, 睡觉";  
  
//我们可以直接在花括号里面把所需要的属性写进去  
var obj2 = {  
    stuName: "曹慧",  
    sex: "女",  
    age: 18  
};  
obj2.hobby = "做饭, 逛街";
```

对象属性的调用方式

通过上面的学习，我们知道对象里面有数据，这些数据叫属性，一个对象如何去调用呢。这里有2种方式给大家介绍一下

通过 . 的方式来调用

正常情况下，通过 **对象.属性** 就可以进行调用了，如下所示

```
var obj1 = {
    stuName: "颜一鸣",
    sex: "男",
    age: 18,
    hobby: "看书, 睡觉"
}

//打印obj1的姓名
console.log(obj1.stuName);
console.log(obj1.sex);
console.log(obj1.age);
console.log(obj1.hobby);
```

注意：这一种方式去调用属性的时候只能调用常规范的属性，对于特殊的属性则不能通过`.`的方式来调用，要使用`[]`来调用

通过`[]`的方式来调用

中括号的调用方式其实同学们之前已经接触过了，就是数组里面的索引，数组里面的索引其实也算是一个属性，所以我们在调用数字的属性的时候我们使用了`[]`

对于特殊的属性，我们是不能够使用`.`的方式来调用的，如下所示

```
var obj1 = {
    "stu-name": "颜一鸣",
    sex: "男",
    age: 18,
    0:"hello"
}

//所有的属性都可以通过[]来调用
console.log(obj1.sex);
console.log(obj1["sex"]);
// console.log(obj1.0); //报错
console.log(obj1[0]);
console.log(obj1["stu-name"]);
```

```
> obj1
< {0: 'hello', stu-name: '颜一鸣', sex: '男', ag
  e: 18} i
    0: "hello"
    age: 18
    sex: "男"
    stu-name: "颜一鸣"
▶ [[Prototype]]: Object
```

说明：[]中括号调用属性的方式其实是可以调用所有的属性的

总结：所有的属性都可以通过[]来进行调用，.点这种方式只能调用常规属性

变量做为属性名

```
var a = "hello";
var obj1 = {
  stu: a          //上面的a就是一个变量a
}

var obj2 = {
  a:"world"       //这里的a就是一个普通的字符串a
}

//有没有一种办法把a用变量的形式放在属性当中
var obj3 = {
  [a]:"world"     //这里的a加了中括号，代表的就是变量a的值
}
```

```
> obj1
<  ► {stu: 'hello'}
> obj2
<  ► {a: 'world'}
> obj3
<  ► {hello: 'world'}
```

在上面的学习的2种方式里面，我们都已经知道了怎么样去创建对象，对象可以把某些数据集中管理。但是我们仍然面对一个问题，如果我现在需要创建很多个对象，怎么办？我现在想把我们班65位同学们的信息全部都用对象表示，这又怎么办？如果我们还是使用原来的方式去创建，这样我们就要把所有的学生的属性都一一的输入一遍，这样非常不好，效率太低了。我们现在需要一种技术，快速的创建对象。

使用工厂模式创建对象

一说起工厂，脑海之中不自觉的就会想起批量生产，并且生产的东西应该都是相同的。工厂模式其实就是批量生产对象，并且生产相同的对象。

```
// 现在我要批量的创建相同类型的对象
function createStudent(_name,_sex,_age) {
    var obj = {
        name: _name,
        sex: _sex,
        age: _age
    }
    return obj;
}

var stu1 = createStudent("李心悦","女",17);
var stu2 = createStudent("颜一鸣","男",20);
```

在上面的工厂模式里面，我们可以快速的去创建对象，并且对象的格式也是一样的，这样就解决了我们需要创建重复对象的问题。

函数 `createPerson()` 能够根据接受的参数来构建一个包含所有必要信息的 `Person` 对象。可以无数次地调用这个函数，而每次它都会返回一个包含三个属性一个方法的对象。**工厂模式虽然解决了创建多个相似对象的问题，但却没有解决对象识别的问题（即怎样知道一个对象的类型）。**随着 JavaScript 的发展，又一个新模式出现了。

```
// 现在我要批量的创建相同类型的对象
//专门生产学生的工厂
function createStudent(_name, _sex, _age) {
    var obj = {
        name: _name,
        sex: _sex,
        age: _age
    }
    return obj;
}

var stu1 = createStudent("李心悦", "女", 17);
var stu2 = createStudent("颜一鸣", "男", 20);

//专门生产牛的工厂
function createCow(_name, _sex, _age) {
    var obj = {
        name: _name,
        sex: _sex,
        age: _age
    }
    return obj;
}

var cow1 = createCow("小牛1号", "母", 1);
var cow2 = createCow("公牛2号", "公", 2);
```

在上面的代码里面，我们有2个工厂，一个是专门生产学生的对象的，一个是专门生产牛的对象的，但是这两个对象最终所表现出来的类型竟然是相同的

```
> cow1
< ▶ {name: '小牛1号', sex: '母', age: 1}
> cow2
< ▶ {name: '公牛2号', sex: '公', age: 2}
> stu1
< ▶ {name: '李心悦', sex: '女', age: 17}
> stu2
< ▶ {name: '颜一鸣', sex: '男', age: 20}
```

```
typeof stu1;      // "object"
typeof cow1;      // "object"
//之前也给大家讲过，基本数据类型才使用`typeof`，而复杂数据类型使用`instanceof`
//现在我们使用`instanceof`来检测
stu1 instanceof Object;      // true
cow1 instanceof Object;      // true
```

通过上面的代码我们可以看到，我们不能够去识别工厂模式创建的对象的类型，无论是通过 `typeof` 或 `instanceof` 都不能区分

使用构造函数创建对象【重点】

重要的事情说三遍，这个章节是重点，重点，重点

要弄清楚构造函数就首先先弄清楚函数

函数之前我们已经学地了，就是一个通过 `function` 去定义的东西，同时我们也学习了函数的调用方式，函数是通过函数名`+()`的形式去调用的

现在我们来讲函数的另一种调用方式

```
var arr = new Array();
// 这会创建一个数组对象
var obj = new Object();
// 为什么new一个Object也会得到对象？
```

在上面的代码里面，我们就要思考一下，为什么 `new` 一个东西就会得到对象，同时 `new` 的这个东西又是什么？

现在我们就来看一下 `Array` 与 `Object` 到底是个什么东西

```
console.log(typeof Array);          // function
console.log(typeof Object);         // function
```

我们现在可以看到 `Array` 与 `Object` 都是一个 `function` 函数，所以我们知道了一点 `new` 一个函数就会得到一个对象，`Array` 与 `Object` 就是系统当中内置的构造函数

所谓的构造其实就是通过 `function` 关键字定义的普通函数，只是它的调用方式不一样而已。如果一个函数使用了 `new` 关键字去调用，那么这个函数就叫构造函数

```
function abc(){
    console.log("abc");
    return 123;
}

var x = abc();           //普通函数    返回值x 123
var y = new abc();       //new调用的,构造函数  返回了一个对象
```

通过上面的代码我们可以发现，`new` 应该就是调用一个函数，然后再拿到对象

```
var 对象 = new 函数();
```

? 问题：为什么用 `new` 去调用的函数就叫构造函数

如果我们现在想通过上面的 `new 函数()` 的方式来创建对象，那么我们必须要知道构造函数是怎么调用的，而构造函数的调用主要点还是在 `new` 上面，所以我们需知道 `new` 到底干了什么事情

要创建 `Person` 的新实例，必须使用 `new` 操作符。以这种方式调用构造函数实际上会经历以下 4 个步骤：

- (1) 创建一个新对象；
- (2) 将构造函数的作用域赋给新对象（因此 `this` 就指向了这个新对象）；
- (3) 执行构造函数中的代码（为这个新对象添加属性）；
- (4) 返回新对象。

```
// 现在我要创建一个学生的对象
function Student(_name, _age) {
    // console.log(this);
    // this就是新创建的对象stu1
    this.name = _name;
    this.age = _age;
    // return this;
}

var stu1 = new Student("颜一鸣", 18);      //第一个对象
var stu2 = new Student("李心悦", 20);      //第二个对象
```

通过上面的构造函数，我们也可以创建出相同格式的对象

注意：我们之前的工厂模式也能快速创建对象，但是不能识别对象，这个时候的构造函数是否可以识别对象呢

```
// 学生的构造函数
```

```

function Student(_age) {
    this.age = _age;
}

// 牛的构造函数
function Cow(_age) {
    this.age = _age;
}

var stu1 = new Student(18);
var cow1 = new Cow(2);

console.log(stu1 instanceof Object);           //true
console.log(cow1 instanceof Object);           //true
console.log("-----");

//现在开始真正的识别
console.log(stu1 instanceof Student);          //true
console.log(cow1 instanceof Cow);               //true
console.log(stu1 instanceof Cow);                //false
console.log(cow1 instanceof Student);            //false

```

The screenshot shows a browser window with several tabs at the top. The active tab is '12构造函数创建的对象识别问题.html'. On the right, the developer tools' '控制台' (Console) tab is open, displaying the following output:

```

> stu1
< ▶ Student {age: 18}
> cow1
< ▶ Cow {age: 2}      这就是对象的类型
>

```

The 'stu1' and 'cow1' objects are shown with their properties and types. A red box highlights the 'Cow' object, and the text '这就是对象的类型' (This is the type of the object) is displayed next to it.

The left side of the screen shows the source code of the '12构造函数创建的对象识别问题.html' file. It contains JavaScript code defining two constructor functions, `Student` and `Cow`, and creating instances `stu1` and `cow1`. The code also includes comments explaining the execution flow and the results of the `instanceof` operator.

这个时候我们再去使用 `instanceof` 去检测的时候就可以识别对象的类型了，就可以把2个对象区分开了

构造函数与普通函数的区别

构造函数与普通函数在定义上面是没有任何区别的，关键是看他的调用形式

1. 一个函数如果以 `new` 去调用那么它就是构造函数，如果只是通过 `函数名+()` 这种形式调用它就是普通函数

2. 普通函数的返回值是通过 `return` 来完成的，而构造函数的返回值是自动返回的

构造函数的返回值如果内部返回的是一个基本数据类型则不生效，它会自动返回构造函数创建的对象

构造函数的返回值如果内部返回的是一个对象，则放弃构造函数自动创建的对象，以 `return` 为主

```
function Student(name){  
    this.name = name;  
    return 123;          //123是基本数据类型，所以构造函数不要  
}  
  
var x = Student("张三");           //普通函数调用，所以x接收的就是return返回的结果  
var y = new Student("李四");      //构造函数调用，默认返回当前创建的对象
```

注意：这里有个天大的坑

```
function Student(name) {  
    this.name = name;  
    var arr = [ "a", "b", "c" ];  
    //arr是一个数组，数组是对象  
    return arr;        //这里返回的是一个对象  
}  
  
var x = Student("张三");           //普通函数调用    接收return的结果 arr  
var y = new Student("李四");      //构造函数调用    它到底接收什么
```

在上面的代码里面，因为 `Student` 的函数内部返回的是一个对象 `arr` 数组，所以在构造函数进行 `new` 的返回的时候，它会放弃自己创建的对象，而返回 `arr` 给外边

构造函数并不会接收 `return` 的基本数据类型的返回值，但是如果你返回的是一个对象，它那我就要了（这个时候就不再返回默认的构造函数创建的对象）

3. 构造函数里面的 `this` 指向了当前构造函数所创建的对象，而普通函数里面的 `this` 指向了浏览器的全局对象 `window` 【这个知识点在后面的DOM里面会讲到】

4. 普通函数在调用的时候是需要通过 **函数名+()** 来调用，而构造函数如果不需要传递参数，则可以省略括号 **()**

```
var arr = new Array;  
var obj = new Object;  
  
function Student(){  
    this.name = "张三";  
}  
  
Student();      //普通函数  
var stu1 = new Student;           //构造函数调用
```

在构造函数里面，它的 **()** 主要是为了传递参数的，如果我们不需要传递参数，则这个小括号 **()** 可以省略不写

约定俗成：构造函数的函数名首字母大写，而普通函数的首字母是小写的，这不是规范，但这是约定。在系统当中如果你要是发现一个函数名大小了，则它肯定是构造函数

保证函数以构造的方式执行

```
//我们想把下面的函数定义成构造函数  
function Student(name) {  
    // 如何保证当前的函数只能被当成构造函数调用  
    if (new.target === Student) {  
        this.name = name;  
    }  
    else{  
        console.error("当前的函数只能以构造函数执行");  
    }  
}  
//Student("张三");          //普通限制  
var x = new Student("李四"); //构造函数调用
```

对象中的方法

每个对象上面或多或少的都具备一些能力，这些能力其实说法是对象上面的方法（方法就是函数，在面向过程的编程里面函数，在面向对象里面的叫方法）

Object的方式

```
var stu1 = new Object();
stu1.name = "江海丽";
stu1.sayHello = function(){
    console.log("你好啊, 我叫: " + this.name);
}
```

字面量的方式

```
var stu2 = {
    name:"曹慧",
    sayHello:function(){
        console.log("hello world! My name is " + this.name);
        // console.log(this === stu2);
    }
}
```

构造函数创建对象里面的方法

```
function Student(name) {
    this.name = name;
    // this指向新创建的对象
    this.sayHello = function(){
        console.log("我是一个曹同学, 我的姓名是: " + this.name);
    }
}

var stu3 = new Student("曹方");
```

💡 小技巧：在对象方法的内部，`this` 是指向当前这个对象的，如果想在方法里拿到自己的某一个属性，则可以通过 `this` 来调用

基础篇总结

1. 面向对象的概念是什么？

面向结果的编程方式，集中式的数据管理方式，高内聚的特征，集中数据的这个过程就是封装对象的过程

2. 对象创建的几方式是什么？最常见的方式有哪些？

- `new Object()` 创建对象【不常用】
- 通过 `{}` 花括号创建 【很常见】

上面的两种试都只是适合创建少量的对象，如果批量则不适用了

- 工厂模式【不常用】
 - 构造函数的创建【很常用】
3. 构造函数与普通函数的区别在哪里?
- 调用方式的不同, 特别是 `new`
 - 返回值的情况不同
 - `this` 指向会不同
 - 通过 `new` 调用的构造函数可以省略 `()`
4. 对象调用属性或方法的方式

```
//[ ]是可以调用任何属性
var stu1 = {
    userName: "江海丽",
    sayHello: function () {
        console.log("大家好,我叫" + this.userName);
    }
}
stu1.sayHello();

//stu1.sayHello === stu1["sayHello"];
stu1["sayHello"]();\n\n
var arr = ["标哥哥"];
arr.push("江海丽");

arr["push"]("曹慧");
//arr["push"] === arr.push
//arr["push"]("曹慧")===arr.push("曹慧")
```

5. `new` 到底干了什么事情? 【`new`干的4件事情】
6. 记住 `new.target` 保证函数以构造函数方式执行

面向对象 (二)

之前我们已经讲过了对象的封装 (也就是对象的创建过程), 那么有没有一种可能对象创建以后会发生变化

```
var obj1 = {  
    stuName: "曹方",  
    sex: "女",  
    age: 18,  
    height: 168,  
    weight: 50  
}
```

问题：

1. 当这个对象创建好了以后，这个对象里面有些属性是不能公开，如我们希望 `age` 这个属性不要，怎么办？
2. 当这个对象创建好了以后，我们希望 `sex` 这个属性的值不能更改，怎么办呢？

针对上面的问题，如果我们还使用之前的方式去创建对象就会有问题

delete关键字

如果我们希望某一个属性不要了，就可以使用 `delete` 把这个属性把它删掉

```
delete obj1.age; //删除obj1对象上面的age属性
```

上面的操作会返回一个布尔值，如果返回的是 `true` 就代表这个属性删除成功，如果返回 `false` 就代表这个属性删除失败

扩展

```
var arr = ["a", "b", "c", "d", "e", "f", "g";  
//我想删除"c"怎么办  
delete arr[2];
```

▼ (7) ['a', 'b', 空, 'd', 'e', 'f', 'g'] i

0: "a"
1: "b"
3: "d"
4: "e"
5: "f"
6: "g"
length: 7

► [[Prototype]]: Array(0)

因为 `delete` 是可以删除属性的，而数组的索引也是一个属性，也就可以附表和 `delete` 去删除，不过我也发现了通过 `delete` 删除以后数组是不会呈现队列效应（沙漏效应）的

? 思考一下：为什么我们执行 `delete arr.length` 的时候会得到 `false`，也就是删除失败？

如果想定义一个不能被删除的属性，则必须要使用特殊的方式来定义属性

Object.defineProperty来定义属性

如果想定义一些特殊的属性，我们就需要使用刚刚说的这个方法，它可以定义一些不能删除的属性，不能更改的属性等一系列操作

所有的属性其实都可以分为2类

1. 数据属性
2. 访问器属性

数据属性

1. 数据属性

数据属性包含一个数据值的位置。在这个位置可以读取和写入值。数据属性有 4 个描述其行为的特性。

- `[[Configurable]]`: 表示能否通过 `delete` 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 `true`。
- `[[Enumerable]]`: 表示能否通过 `for-in` 循环返回属性。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 `true`。
- `[[Writable]]`: 表示能否修改属性的值。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 `true`。
- `[[Value]]`: 包含这个属性的数据值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。这个特性的默认值为 `undefined`。

在定义属性的时候，我们可以通过上面的4个设置来定义特殊的属性，现在我们来定义第一个特殊的对象的属性

```
var stu1 = {  
    age: 18  
}  
  
//现在，我们希望`stu1`对象上面有一个属性stuName，不能被删除  
//如果想定义一个特殊的属性，要使用Object.defineProperty来完成  
Object.defineProperty(stu1, "stuName", {  
    //这一个对象就描述了属性stuName它的特征  
    value: "张三",  
    configurable: false           //决定当前属性不可以被删除  
});  
  
//现在我想定义一个性别sex的属性，但是这个属性不可以更改它的值  
Object.defineProperty(stu1, "sex", {  
    value: "男",  
    writable: false             //当前属性值不可以被更改，相当于只读  
})
```

访问器属性

2. 访问器属性

访问器属性不包含数据值；它们包含一对儿 getter 和 setter 函数（不过，这两个函数都不是必需的）。在读取访问器属性时，会调用 getter 函数，这个函数负责返回有效的值；在写入访问器属性时，会调用 setter 函数并传入新值，这个函数负责决定如何处理数据。访问器属性有如下 4 个特性。

- [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的默认值为 true。
- [[Enumerable]]：表示能否通过 for-in 循环返回属性。对于直接在对象上定义的属性，这个特性的默认值为 true。
- [[Get]]：在读取属性时调用的函数。默认值为 undefined。
- [[Set]]：在写入属性时调用的函数。默认值为 undefined。

```
// 这就是一个普通的对象
var obj1 = {
  birthday: "2002-1-10"
}

//我们现在定义一个年龄的属性
Object.defineProperty(obj1, "age", {
  configurable: false,           //不可以被删除
  get: function () {
    //在获取这个属性值的时候自动调用
    console.log("我要取age属性的值");
  },
  set: function () {
    //在设置这个属性的值的时候自动调用
    console.log("你现在是不是要对age属性赋值")
  }
});

obj1.age;                      //在调用obj1的age属性的时候, get方法会被调用
obj1.age = 18;                  //在设置obj1的age属性的值的时候, set方法会被调用
```

上面的 age 属性就是一个访问器属性

1. get 函数与 set 函数是在对 age 属性进行访问或设置的时候自动调用的
2. age 属性是没有 value，所以它没有存储具体的值

问题：访问器属性的用处到底在哪里？

```
var obj1 = {
  // 这是一个身份证号
  IDCard: "420984199009081014"
```

```

}

//现在定义一个属性sex,这个sex的值是根据当前的身份证号来决定的
//字符串可以像数组一样操作
Object.defineProperty(obj1, "sex", {
  configurable: false,
  get: function () {
    //取值的时候调用
    return this.IDCard[16] % 2 == 0 ? "女" : "男";
  }
});

console.log(obj1.sex); //男

obj1.IDCard = "420984199009081082";
console.log(obj1.sex); //女

```

通过上面的访问器属性，我们可以看到，当我们设置了一个对象的身份证号属性 `IDCard` 以后，我们再去获取 `sex` 的时候，它会自动调用 `get` 函数，而 `get` 函数会根据 `IDCard` 来计算性别，这样会非常方便

有一句话是这么说的，访问器属性也叫联动属性

访问器的优点：

1. 通过 `get` 函数我们可以让一个属性值以另一个属性值为参照
2. 我们可以通过 `get/set` 来决定属性的取值与赋值操作。在上面的代码里面，我们没有设置 `set` 方法，就说明 `sex` 这个属性不能被赋值，只能被取值

现在我们来看一下访问器属性里面的 `set` 的函数的用法。

```

var obj1 = {
  // 定义了姓的属性
  firstName: "杨",
  // 定义了名的属性
  lastName: "标"
}

//现在我们想定义一个`userName`的访问器属性
Object.defineProperty(obj1, "userName", {
  configurable: false,
  get: function () {
    return this.firstName + this.lastName;
  }
})

```

```

    },
    set: function (v) {
        //在赋值的时候会自动调用set函数
        console.log("赋值");
        // console.log(v);
        this.firstName = v[0];           //张
        this.lastName = v[1];           //三
    }
});

obj1.userName = "张三";

```

```

4 <script>
5     var obj1 = {
6         // 定义了姓的属性
7         firstName: "杨",
8         //定义了名的属性
9         lastName: "标"
0     }
1
2 //现在我们想定义一个`userName`的访问器属性
3 Object.defineProperty(obj1, "userName", {
4     configurable: false,
5     get: function () {
6         return this.firstName + this.lastName;
7     },
8     set: function (v) {           这里要注意，参数v就是要赋的值
9         //在赋值的时候会自动调用set函数
0         console.log("赋值");
1         // console.log(v);
2         this.firstName = v[0];           //张
3         this.lastName = v[1];           //三
4     }
5 );
6
7     obj1.userName = "张三";
8 </script>

```

通过上面的上面的代码我们可以很清楚的看到一点，访问器属性本身是没有任何值存储的，它所有的值都依赖于其它的属性，`get` 负责取值的操作，`set` 负责赋值的操作，它们都是自动的

场景：通过访问器属性来控制属性值的设置与读取

```

// 想定义一个年龄age，这年龄可以取值，但是赋值的时候只能大于或等于18
var stu1 = {

```

```

    userName: "张三",
    _aaa: 18           //这是一个普通的属性，可以存数据
}

//访问器属性本身不存储任何数据，它要依赖于其它属性`_aaa`、
Object.defineProperty(stu1, "age", {
  configurable: false,
  get: function () {
    //get函数负责取值，在调用age属性的时候返回了`_aaa`的值
    return this._aaa;
  },
  set: function (v) {
    //在赋值的时候 v就是要赋的值，如果这个值大于等于18.就正常赋值
    //同时因为访问器属性本身不存储任何数据，所以它只能依赖于其它属性
    //最终这值是赋到了`_aaa`上面
    if (v >= 18) {
      //正常赋值
      this._aaa = v;
    }
  }
});

```

Object.defineProperties来定义属性

之前我们已经使用 `Object.defineProperty` 来实现特殊属性的定义，但是这个方法只能一次性的定义一个属性，如果我们要定义多个特殊的属性则要多次的调用这个方法，比较麻烦

`Object.defineProperties` 这个方法就可以一次性地定义多个特殊的属性，它的语法格式如下

```

Object.defineProperties(对象, {
  属性名1:{
    //相关的特性
  },
  属性名2:{
    //相关的特性
  }
});

```

通过上面的方式我们就可以同时定义多个特殊属性

```
var obj1 = {
```

```

        userName: "张三",
    }
//obj1上面有一个age属性，这个属性不能被删除 默认值是18
//obj1上面有一个sex属性，这属性不能被修改【只读】 默认值是男

Object.defineProperties(obj1, {
    age: {
        configurable: false,
        value: 18
    },
    sex: {
        value: "男",
        writable: false
    }
});

```

对象特殊属性定义



遍历对象的属性

其实这个东西早看应该讲了，只是时机不成熟

遍历对象的属性就是把对象当中的所有属性名都拿出来

for...in遍历对象

3.6.5 for-in语句

for-in语句是一种精准的迭代语句，可以用来枚举对象的属性。以下是for-in语句的语法：

```
for (property in expression) statement
```

下面是一个示例：

```
for (var propName in window) {  
    document.write(propName);  
}
```

[ForInStatementExample01.htm](#)

在这个例子中，我们使用for-in循环来显示了BOM中window对象的所有属性。每次执行循环时，都会将window对象中存在的一个属性名赋值给变量propName。这个过程会一直持续到对象中的所有属性都被枚举一遍为止。与for语句类似，这里控制语句中的var操作符也不是必需的。但是，为了保证使用局部变量，我们推荐上面例子中的这种做法。

ECMAScript对象的属性没有顺序。因此，通过for-in循环输出的属性名的顺序是不可预测的。

具体来讲，所有属性都会被返回一次，但返回的先后次序可能会因浏览器而异。

但是，如果表示要迭代的对象的变量值为null或undefined，for-in语句会抛出错误。ECMAScript 5更正了这一行为；对这种情况不再抛出错误，而只是不执行循环体。为了保证最大限度的

上面的话是“红宝书”当中的话，如果想遍历对象里面的属性我们需要使用**for...in**

```
var obj1 = {  
    userName: "张三",  
    age: 18,  
    sex: "男"  
}  
  
//现在我希望把上面所有的属性都拿出来打印一遍，怎么办?  
for(var i in obj1){  
    console.log(i); //这个时候的i就是所有的属性名  
    //userName, age, sex三个打印结果  
}
```

for...in是我们第一种遍历对象属性的方式，只这个属性的**enumerable**不为**false**则可以正常的遍历出来

```
var obj1 = {  
    userName: "张三",  
    age: 18  
}  
  
//单独的去定义一个特殊属性sex  
Object.defineProperty(obj1, "sex", {
```

```
    value:"男",
    enumerable:false //这里因为设置false, 所以就不能被for...in遍历
  })

for(var i in obj1){
  console.log(i);
  //userName, age
}
```

扩展

```
var arr = ["a", "b", "c", "d", "e", "f"];
// 遍历上面的数组
/*
  for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
  }
*/

/*
arr.forEach(function (item, index, _arr) {
  console.log(item);
})
*/

for (var i in arr) {
  console.log(arr[i]);
}
```

通过Object.keys()遍历对象

在 JavaScript 的内部，有一个方法叫 `Object.keys()`

之前我们在学习对象的创建的时候有一种字面量创建法，如下

```
var obj1 = {  
    属性名:属性值  
}  
//其实也有另一种叫法  
var obj2 = {  
    键:值  
}  
//这种写法叫 键值对 , 键的英文单词是key, 值的英文单词是value  
//属性名也叫key
```

顾名思义 `Object.keys()` 就是获取所有的属性名，但是它只能获取 `enumerable:true` 的属性名

```
var obj1 = {  
    userName: "张三",  
    age: 18  
}  
  
//单独的去定义一个特殊属性sex  
Object.defineProperty(obj1, "sex", {  
    value: "男",  
    enumerable: false //不能被遍历  
});  
  
var arr = Object.keys(obj1);  
//这个时候的arr就会得到`obj1`对象里面所有`enumerable:true`的属性名  
["userName", "age"]  
arr.forEach(function(item){  
    console.log(item, obj1[item]);  
    //如果想打印属性值, 怎么办?  
})
```

`Object.keys(对象)` 它可以把当前这个对象里面所有 `enumerable:true` 的属性拿过来组成一数组，然后我们再遍历这个属性数组就相当于遍历了整个对象

注意：上面的2种方式只能遍历 `enumerable:true` 的属性

思考：有没有什么办法可以遍历对象当中的所有属性？【把 `enumerable:false` 的也遍历出来】

Object.getOwnPropertyNames()方法遍历

这一个方法可以获取某一个对象上面的所有的属性名，它会忽略掉 enumerable 这个特性

```
var obj1 = {
  userName: "张三",
  age: 18
}

//单独的去定义一个特殊属性sex
Object.defineProperty(obj1, "sex", {
  value: "男",
  enumerable: false           //不能被遍历
});

//for...in
//Object.keys(obj1)

//这个方法会获取`obj1`对象上面自身所有的属性
var arr = Object.getOwnPropertyNames(obj1);
// ['userName', 'age', 'sex']
arr.forEach(function(item, index, _arr){
  console.log(item);
})
```

对象的遍历

1 for...in遍历

2 Object.keys()遍历

3 Object.getOwnPropertyNames()遍历

只能遍历enumerable:true的属性

可以遍历所有的属性

构造函数与特殊属性结合

构造函数是可以帮我们快速的创建对象，而 Object.defineProperty 可以帮我们来创建一些特殊的属性，那么这两个东西是可以结合的

场景：现在我希望创建我们班所有学生的对象【65人】，每个学生都具备 `userNmae, sex, age, birthday` 这4个属性，其中 `sex` 属性一旦设置就不可更改，`age` 是一个仅仅只可以访问的属性，并且 `age` 属性的值要通过 `birthday` 来决定，`birthday` 这个属性一定设置也不可以更改

对于上面的场景需要，我们应该怎么办呢？

```
//先不考虑特殊的属性，只考虑普通的属性
function Student(userName, sex, birthday) {
    this.userName = userName;
    this.sex = sex;
    this.birthday = birthday;
}

var s1 = new Student("颜一鸣", "男", "2000-12-1");
```

上面的代码肯定是不符合我们的要求的，怎么办呢？

现在我们已经知道构造函数所创建的对象是一个普通属性的对象，如果想包含特殊的属性只能使用 `Object.defineProperty` 去完成。现在关键问题就是如果将2者结合

要完成上面的问题先思考一下，我们的构造函数内部主要干了什么事情？

1. 创建了一个新的对象
2. `this` 指向了这个新的对象
3. 执行构造函数中的代码（为这个新对象添加属性）；
4. 返回这个新对象

问题就在第3步，我要对这个新对象添加特殊属性

```
function Student(userName, sex, birthday) {
    this.userName = userName;
    Object.defineProperties(this, {
        sex: {
            value: sex,
            writable: false
        },
        birthday: {
            value: birthday,
            writable: false
        },
        age: {
            get: function () {
```

```
// 希望通过生日来获取年龄
// 用当前日期 - 生日 = 年龄
var now = new Date();
var birthdayDate = new Date(this.birthday);
var times = now - birthdayDate;
var yearCount = ~~(times / 1000 / 60 / 60 / 24 / 365);
return yearCount;
}
}
});
}

var s1 = new Student("颜一鸣", "男", "2000-12-1");
```

1. `age` 是一个访问器属性通过 `get` 函数来计算年龄，它依赖于 `birthday` 属性（这个计算过程没有学，可以先不管）
2. 我们将构造函数与特殊属性进行了结合
3. 像上面这种方式我们平常在工作中经常使用

获取对象属性的描述信息

获取对象属性的描述信息也叫获取对象属性的特征，当我们得到一个对象以后，我们想知道某些属性是否可以更改，某一个是否可以被删除，怎么办呢？

在 JS 里面，有一个方法是可以获取对象属性的描述信息的

```
var 描述信息 = Object.getOwnPropertyDescriptor(对象, 属性);
```

```
var arr = ["a", "b", "c", "d", "e"];
//length可以遍历吗?
//length可以删除吗?
// 如果想获取某珍上属性的特征，是可以通过一个方法的
```

```
var lengthDesc = Object.getOwnPropertyDescriptor(arr, "length");
```

```
> lengthDesc
< {value: 5, writable: true, enumerable: false, configurable: false} i
  configurable: false
  enumerable: false
  value: 5
  writable: true
▶ [[Prototype]]: Object
```

看到了上面的描述信息，我们就知道了 `length` 属性不能被删除，也不能被 `for...in` 遍历，但是可以更改长度

判断对象是否具备某一个属性

假设现在我们有一个对象，我们需要判断某一个对象是否具备某一个属性，怎么办呢？

```
var stu1 = {
  userName: "李心悦",
  sex: "女"
}
Object.defineProperty(stu1, "age", {
  value: 18,
  enumerable: false, //是否可遍历
  configurable: false,
  writable: false
});
```

当判断某一个对象是否具备某一个属性的时候一定要注意我们之前所学习的对象的属性的遍历 `Object.keys()` 以及 `for...in` 都只能够将 `enumerable:true` 的属性遍历出来，所以在判断一个对象是否具备一个属性的时候是不能够使用这2个方法的

通过 `in` 关键字来检测

判断某一个对象是否具备某一从此属性我们可以使用关键字 `in` 来完成，它的语法格式如下

```
属性 in 对象；
```

如果上面这个结果是 `true` 就代表这个对象具备这个属性，如果是 `false` 就代表这个对象不具备这个属性

```
console.log("userName" in stu1);          //true
console.log("sex" in stu1);                //true
console.log("age" in stu1);                //true
console.log("a" in stu1);                  //false
```

通过 `hasOwnProperty()` 来完成检测

在每个对象的内部，都会有一个方法叫 `hasOwnProperty("属性名")` 用来检测当前这个对象是否包含这个属性，如果得到的结果是 `true` 就代表当前这个对象包含这个属性，`false` 则不包含

```
stu1.hasOwnProperty("userName");          //true
stu1.hasOwnProperty("sex");                //true
stu1.hasOwnProperty("age");                //true
stu1.hasOwnProperty("aaa");                //false
```

在上面的2种判断方法里面，我们都可以完成一个对象对于属性是否存在检测。**对于是否具备某一个属性应该忽略掉 `enumerable:false` 这个条件，因为这个条件只是用来控制遍历的，不是用于来控制是否存在**

区别：

1. `in` 关键字去检测属性是否存在时候它会到父级对象去检测
2. `hasOwnProperty()` 只是用于检测自己有不有这个属性，不会到父级对象上面去检测

```
var arr = ["a", "b", "c", "d", "e"];

console.log("length" in arr);          //true
console.log(arr.hasOwnProperty("length")); //true

console.log("push" in arr);            //true
console.log(arr.hasOwnProperty("push")); //false 自身没有，push方法是在父级的
```

练习

1. 根据要求创建对象

- 定义一构造函数Person,它有5个属性，分别是姓名(userName)，性别(sex)，年龄(age)，生日(birthday)，身份证号(IDCard)，
- 用户在创建对象的时候，这个构造函数接收二个参数，分别是姓名(userName)，和身份证号 (IDCard)
- 所有的属性都不能被delete,也不能更改它的初始值
- 所有的属性都可以被 for...in 遍历出来
- 根据身份证号来得到用户的性别
- 根据身份证号来得到用户的生日（身份证号里面是包含生日的，如420984199905041016）。
- 根据生日来得到年龄（这里可以参考标哥之前的代码）
- 根据上面定义好的构造函数去创建一个对象，然后再去遍历这个对象，打印这个对象里面的属性名与当前属性的值

提示：字符串可以通过索引取到里面的某个值 `var str = "abc"; str[0]==="a"`，字符串也可以像数组一样去操作，有 `slice` 提取的方法

```
function Person(userName, IDCard) {  
    Object.defineProperties(this, {  
        userName: {  
            configurable: false,  
            writable: false,  
            value: userName,  
            enumerable: true  
        },  
        IDCard: {  
            configurable: false,  
            writable: false,  
            value: IDCard,  
            enumerable: true  
        },  
        sex: {  
            configurable: false,  
            enumerable: true,  
            get: function () {  
                // return ["女","男"][this.IDCard[16] % 2];  
                return this.IDCard[16] % 2 == 0 ? "女" : "男";  
            }  
        },  
    },  
}
```

```

        birthday: {
            configurable: false,
            enumerable: true,
            get: function () {
                var year = this.IDCard.slice(6, 10);
                var month = this.IDCard.slice(10, 12);
                var day = this.IDCard.slice(12, 14);
                // return year + "-" + month + "-" + day;
                return [year, month, day].join("-");
            }
        },
        age: {
            configurable: false,
            enumerable: true,
            get: function () {
                var now = new Date();
                var birthdayDate = new Date(this.birthday);
                var times = now - birthdayDate;
                var yearCount = ~~(times / 1000 / 60 / 60 / 24 / 365);
                return yearCount;
            }
        }
    });

var p1 = new Person("张珊", "420984199801021019");
for (var i in p1) {
    console.log(i, p1[i]);
}

```

2. 请从下面的对象当中遍历所有的属性，如果这个属性是一个方法就调用这个方法
同时，在遍历的过程当中要把所有 `enumerable:false` 的属性标记出来放在一个数组里面

```

var obj1 = {
    userName: "张珊"
}

Object.defineProperties(obj1, {
    sex: {
        enumerable: false,
        value: "男"
    }
})

```

```
        },
        age: {
            enumerable:false,
            get: function () {
                return 19;
            }
        },
        sayHello: {
            enumerable: false,
            value: function () {
                console.log("大家好，我叫" + this.userName);
            }
        }
    )
}
```

3. 请将下面对象当中所有的属性值全部打印出来

```
var teacherInfo = {
    teacherName: "标哥哥",
    age: 19,
    sex: "男",
    money: 1000
}
var stu1 = {
    userName: "张珊",
    sex: "女",
    age: 18,
    aaa: teacherInfo,
    hobby: ["看书", "睡觉"]
}

//请将stu1所有的属性值全部打印一遍
//张珊 女 18 标哥哥 19 男 1000 看书 睡觉
```

提示： `stu1.aaa.teacherName` 通过这样的方式就可以拿到 `标哥哥`。这题可以使用递归去完成

JavaScript面向对象（三）

this关键字

在之前讲构造函数的时候我们已经初步的接触过了this，他是一个指针，指向谁就是谁

this 可以理解为他就是一个指针，具体表现形式要看在调用时他指向了谁

对象中的this

之前面向对象基础的时候，我们说过在面向对象中会经常使用 this 关键字，这个 this 在对象中到底只想谁？

```
var name = "张三";
var stu1 = {
  name: "李四",
  sayHello: function () {
    console.log(this); // this----> stu1
    console.log("大家好，我叫" + this.name);
  }
}
console.log(this); // this----> window
console.log(this.name); // 张三
stu1.sayHello() // 李四
```

在上面的代码里面我们看到如果在全局环境下面 this 指向了 window，在 stu1 对象的内部我们使用 this 他又指向了 stu1，那么 this 到底只想谁？

1. 在全局环境下，使用 this 他就指向 window

> console.log(this)

VM139:1

▶ Window {window: Window, self: Window, document: document, name: '张三', location: Location, ...}

直接在控制台打印 this，我们可以看到 this 指向了 window 全局对象

2. 在自定义对象里面使用 this，他就执行当前调用它的这个对象

> stu1.sayHello()

▶ {name: '李四', sayHello: f}

01.this.html:56

大家好，我叫李四

01.this.html:57

在上面的代码里面我们的 `sayHello` 这一方法被 `stu1` 这个对象在调用，所以 `sayHello` 这一方法的内部使用 `this` 就指向了 `stu1`，方法里面的 `this.name` 就相当于 `stu1.name`

根据上面代码的特点，全局环境的 `this` 指向 `window` 这个特点我们扩展一个点

之前在讲变量的时候，我们提到了全局变量与局部变量，全局变量就是在方法外使用 `var` 关键字来定义的变量，局部变量就是在方法内部定义的变量

以面向对象的角度去理解：全局变量就是在全局环境下定义的变量，全局环境就是 `window`，所以可以认为全局变量就是在 `window` 对象下面定义的变量

在JavaScript里面，一切皆对象，所以我们可以这个理解，`window` 就是一个浏览器的全局对象（最高对象），所以我们如果在 `window` 全局环境下使用 `var` 关键字定义的对象就是相当于是在 `window` 对象下面扩展了一个属性

```
var userName = "张三";
```

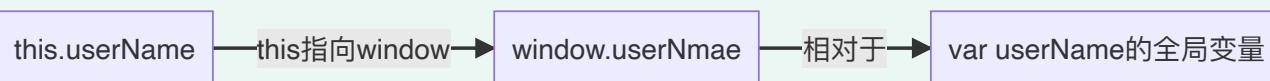
上面的代码在全局环境下定义了一个 `userName` 变量，等价于下面的代码

```
var userName = "张三";
/**
 * 定义了一个全局变量 userName
 * 相当于在window对象下面扩展了一个userName属性
 */
console.log(userName);          // 张三

console.log(window.userName);   // 张三

// 全局环境下面this 指向window
console.log(this.userName);    // 张三
```

这就是我们能够通过 `this.userName` 打印“张三”的原因



```
// function abc() {
//   console.log("我是方法abc");
//   console.log(this);
// }
// 上面的方法也可以通过下面的函数表达式的方式来定义
```

```
// var abc = function () {  
//   console.log("我是方法abc");  
//   console.log(this);  
// }  
// abcs 一个全局的变量，全局变量相当于在window对象下面新增一个属性
```

// 所以上面的代码又可以用下面这种方式来定义

```
window.abc = function () {  
  console.log("我是方法abc");  
  console.log(this);  
}
```

// 在这里调用的时候 我们发现当前的this指向的是window

```
// abc()  
// 这种调用方式也可以理解为下面的调用方式  
window.abc()
```

由上面的代码我们可以得出 `this` 总是指向调用它的对象也就是上面说到的两点：

1. 普通函数调用环境为 `window` 所以 `this` 指向 `window`
2. 对象里面的方法调用，是通过对象 `.` 方法名来调用所以 `this` 指向调用它的这个对象

注意：在使用 `window` 全局对象来调用方法的时候我们可以把 `window` 省略

小练习

```
var userName = "标哥";  
  
var obj1 = {  
  userName: "赛罗奥特曼",  
  sayHello: function () {  
    console.log(this.userName)  
  }  
}  
  
obj1.sayHello() // 打印什么?      "赛罗奥特曼"  
// 把sayHello的方法赋值给了aaa  
var aaa = obj1.sayHello;  
  
aaa()           // 打印什么?      "标哥"
```

```
/*
 普通函数调用 相当于window.aaa()
 这时候的this就是window
 this.userName ----> window.userName
 -----> var userName
 */
```

构造函数里面的 this

在之前讲构造函数的时候，我们就说过构造函数在调用的时候，对象当中的 `this` 就指向了当前的这个对象

```
function Person(userName, age) {
    this.userName = userName;
    this.age = age;
    this.sayHello = function () {
        console.log("大家好，我叫" + this.userName);
    }
}

var p1 = new Person('张三', 20)

console.log(p1.userName);
console.log(p1.age);
p1.sayHello();
// p1这个对象是构造函数创建的对象 在构造函数内部的this就会指向当前对象
// 这里的this.userName 等价于 p1.userName
```

函数调用方式的不同决定this指向

之前我们讲过了函数的调用形式

- 1. 方法名+()调用
- 2. new 函数名 构造函数的调用
- 3. call()/apply()调用

- 1. 在全局环境下调用方法 方法名+()
 - 这种方式的 `this` 指向全局对象 `window`
- 2. 对象内部的方法 对象名.方法名()
 - 这种方式的 `this` 指向当前调用它的对象

- 3. 构造函数调用 `new 方法名()`
 - 这种方式的 `this` 指向当前创建的对象
- 4. `call()/apply()`
 - 这种方式的 `this` 指向参数传递的那个对象

补充

在之前我们已经讲过了方法的调用方式，在JS里在，函数的调用是有很多种方式

- 1. `函数名()` 这种方式来调用
- 2. `new 函数名()` 来调用

除了以上两种方式外，还有两种调用方式

通过 `call()` 的方式来调用

这是一种最新的调用方式，具体我们看代码

```
function a() {  
    console.log("我是a方法");  
}  
  
// a() // 以前的调用方式  
  
a.call()
```

在上面的代码中我们看到函数还可以通过 `call()` 的方式来调用，它的语法格式如下：

```
函数名.call(this指向?, ...原函数参数?);
```

在前面我们讲到了 `this` 的指向问题，普通函数在调用的时候 `this` 执行全局 `window` 对象，在构造函数通过 `new` 去调用的时候内部的 `this` 指向当前创建的这个对象，在对象的调用自己的方法时，对象内部的 `this` 指向当前调用自己的这个对象

```

15 var stu1 = {
16   // ...
17   userName: "颜一鸣"
18 }
19
20 function a() {
21   console.log("我是方法a");
22   console.log(this);
23 }
24
25 function b() {
26   console.log("我是方法b");
27   console.log(this);
28 }
29
30 a()
31 b.call(stu1)
32
33 </script>

```

代码分析：

- 在上面的代码里面，我们通过 `a()` 的方式是普通函数的调用，所以它的 `this` 就指向了全局对象 `window`
- 在 `b.call(stu1)` 是通过 `call()` 方法来调用，这样里面的 `this` 就指向了 `stu1` 这个对象

通过上面的代码我们可以充分说明，`call` 是可以改变 `this` 指向的

同时函数在通过 `call` 去调用的时候也是可以传递参数的

```

var obj = {
  name: "张三"
}
function sum(x, y) {
  return x + y
}

var x = sum(1, 2)      // 普通函数调用
console.log(x);

var y = sum.call(obj, 1, 2)    // call调用 第一个参数是this指向, 后面跟原函数参数
console.log(y);

```

通过 `apply()` 的方式来调用

函数通过 `apply()` 调用其实与 `call()` 调用作用是一样的，语法如下：

```
函数名.apply(this指向?, [...原函数参数]?)
```

```
var stu1 = {
  userName: "张三"
}

function a() {
  console.log("我是方法a");
  console.log(this);
}

a()
a.call(stu1) // call调用改变this指向 stu1
a.apply(stu1) // apply调用改变this指向 stu1
```

上面的代码可以看到 `call` 与 `apply` 调用效果是一模一样的，都可以改变 `this` 的指向，它们唯一的区别就是后面的参数传递不一样

```
var obj = {
  name: "aa"
}

function sum(x, y) {
  return x + y
}

var a = sum(1, 2)
console.log(a);

var b = sum.call(obj, 1, 2) // call 是将原函数的参数一个一个传进去
console.log(b);

var c = sum.apply(obj, [1, 2]) // apply 是讲原函数的参数放到一个数组里面在传进去
console.log(c);
```

通过上面的代码我们发现 `call` 是向原函数内部一个一个的传递参数，而 `apply` 是吧所有的参数都放到一个数组里面然后再去传递

练习

1. 阅读以下代码写出执行结果

```
var a = 123;
var obj = {
  a: 456,
  def: function () {
    console.log(this.a);
  }
}
function abc() {
  console.log(this.a);
}

abc() // 打印什么
abc.call(obj) // 打印什么
obj.def() // 打印什么
obj.def.call(window) // 打印什么
```

2. 阅读以下代码写出执行结果

```
var a = 2;
var obj = {
  a: 5,
  def: function (x,y) {
    console.log(this.a);
    return x+y;
  }
}
function abc(x,y) {
  console.log(this.a);
  return x*y;
}

var result1 = abc.call(obj,a,obj.a);
var result2 = obj.def(this.a,obj.a);
var result3 = obj.def.apply(window,[obj.a,a]);
```

```
// 求result1, result2, result3的值分别是多少同时执行上面代码的同时控制台会打印什么s
```

面向对象 (四)

主题：继承

继承是一种通俗的概念，子级对象是可以默认使用父级对象的东西的

之前我们已经学习过了怎么样封装一个对象（封装对象也叫创建对象），在封装的过程当中难免会出现一些问题

场景一：现在有一个学生对象，他有姓名userName，性别sex，年龄age，学号sid四个属性，而又有一个人老师对象，他有姓名userName，性别sex，年龄age，它还有tid工号，那么，对于这样的对象，我们怎么处理呢？

场景二：现在有一群男生和女生，它们的属性不相同，但是他们都有一个共同的方法叫sayHello，对于这种场景，我们又怎么办？

在平常我们接触对象的时候其实我们已经接触过继承的特点了

通过 call/apply 来继承

这一种方式的继承只是实现了代码上面的继承关系，并没有实现父子的表现关系

```
function Student(userName, sex, age, sid) {
    this.userName = userName;
    this.sex = sex;
    this.age = age;
    this.sid = sid;
}

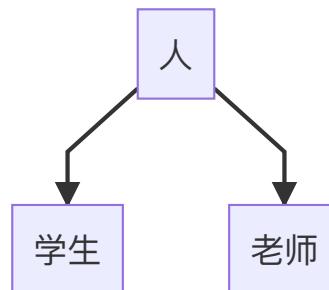
function Teacher(userName, sex, age, tid) {
    this.userName = userName;
    this.sex = sex;
    this.age = age;
    this.tid = tid;
}
```

```

var s1 = new Student("张珊", "女", 18, "H22040001");
var t1 = new Teacher("桃子", "女", 20, "T01");

```

在上面的代码里面，我们已经创建了两个构造，并通过2个构造函数来实现了对象的创建过程，但是问题就在于这个代码它的冗余量非常大。学生 `Student` 与 `Teacher` 老师有3个共同的属性，我们能否将这3个共同的属性提取出来



```

//学生与老师公共的东西就是人
function Person(userName, sex, age) {
    this.userName = userName;
    this.sex = sex;
    this.age = age;
}

function Student(userName, sex, age, sid) {
    this.sid = sid;
    //构造函数地this指向当前新创建的对象
    // Person(userName, sex, age); 如果是以这种方式去调用，则this的指向就会指向
    // window, 而没有指向当前调用的对象
    Person.call(this, userName, sex, age);
}

function Teacher(userName, sex, age, tid) {
    this.tid = tid;
    Person.call(this, userName, sex, age);
}

var s1 = new Student("张珊", "女", 18, "H22040001");
var t1 = new Teacher("桃子", "女", 20, "T01");

```

上面的 `call` 是可以变成 `apply`

现在请通过上面的方式来完成一个小小的练习

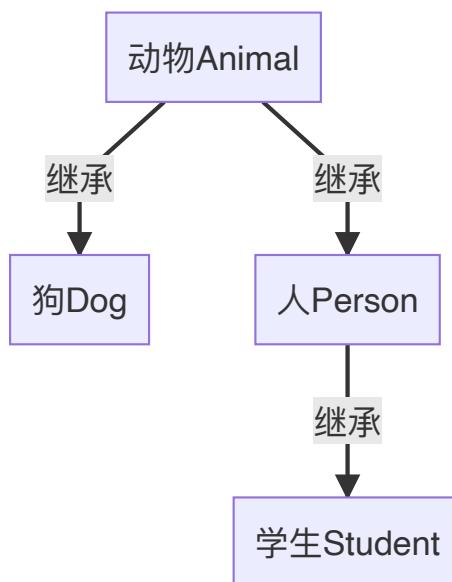
现有一个动画对象 `Animal`，它有名称 `name` 与年龄 `age`，所有的动画都有一个 `sayHello` 的方法，这个方法默认打印输出“hello world”

现有小狗对象 `Dog`，它有性别 `sex`

现有的人的对象 `Person`，它有爱好 `hobby`，人有自己的 `sayHello` 方法，这个方法打印输出“你好，世界”

现有学生对象 `Student`，它有学号 `sid`

请根据继承关系图来完成操作



```
//动物
function Animal(name, age) {
    this.name = name;
    this.age = age;
    this.sayHello=function(){
        console.log("hello world");
    }
}
//狗
function Dog(name, age, sex) {
    Animal.call(this, name, age);
    this.sex = sex;
}

//人
function Person(name, age, hobby) {
    Animal.call(this, name, age);
```

```

this.hobby = hobby;
this.sayHello=function(){
    console.log("你好，世界");
}
}

//学生
function Student(name, age, hobby, sid) {
    Person.call(this, name, age, hobby);
    this.sid = sid;
}

var s1 = new Student("小方", "女", "吃饭,睡觉,打豆豆", "H22040001");
var d1 = new Dog("大黄", 4, "公");
var p1 = new Person("黄某人", 20, "看书,睡觉");

```

存在的问题

虽然说我们通过 `call/apply` 实现了代码的继承关系，然后我们在控制台也打印对象

```

> p1
< ▼ Person {name: '黄某人', age: 20, hobby: '看书,睡觉', sayHello: f} ⓘ
  age: 20
  hobby: "看书,睡觉"
  name: "黄某人"
  ▶ sayHello: f ()
  ▶ [[Prototype]]: Object

> d1
< ▼ Dog {name: '大黄', age: 4, sex: '公', sayHello: f} ⓘ
  age: 4
  name: "大黄"
  ▶ sayHello: f ()
  sex: "公"
  ▶ [[Prototype]]: Object

```

通过上面的图我们可以看到一个很明显的一个点，我们根据就无法区分哪属性是子级对象的，哪些属性是父级对象的。通过这一种方式实现的继承它会把所有的属性都放在子级对象上面，并没有严格的构成父子关系，`call/apply` 只是将代码的属性进行了混合

通过原型继承【重点，难点】

要弄清楚这一点，我们就必须先弄清楚什么是“原型”

理解原型

之前我们讲过 `in` 关键字是可以检测一个对象是否具体某一个属性，所以现在我们先测试如下

```
var stu1 = {
    userName: "张三",
    age: 18
}

"userName" in stu1;           //true
"age" in stu1;               //true
"aaa" in stu1;               //false
"toString" in stu1;          //true      问题就在这里
```

`stu1` 的上面明明是没有 `toString` 的方法的，为什么通过关键字 `in` 去检测的结果又是 `true`（说明：`stu1` 对象本身没有这个属性，但是它的父级有）

```
> stu1
< ▼ {userName: '张三', age: 18} ⓘ
  age: 18
  userName: "张三"
  ▼ [[Prototype]]: Object
    ► constructor: f Object()
    ► hasOwnProperty: f hasOwnProperty()
    ► isPrototypeOf: f isPrototypeOf()
    ► propertyIsEnumerable: f propertyIsEnumerable()
    ► toLocaleString: f toLocaleString()
    ► toString: f toString()      prototype原型的意思
    ► valueOf: f valueOf()
    ► __defineGetter__: f __defineGetter__()
    ► __defineSetter__: f __defineSetter__()
    ► __lookupGetter__: f __LookupGetter__()
    ► __lookupSetter__: f __LookupSetter__()
    ► __proto__: ...
    ► get __proto__: f __proto__()
    ► set __proto__: f __proto__()

> "toString" in stu1
< true
```

什么是原型？原型也叫 `prototype`，它是所有对象默认都具备的东西，可以理解为父级对象的意思。在以前 `prototype` 有另外一个名子叫 `__proto__`。它们两个名子不一样，但都是同样的一个

```
"toString" in stu1;
```

在我们去执行上面的检测的时候，`stu1` 这个对象是没有 `toString` 的这个方法，但是它的父级对象（原型）上面有这个，所以最终检测出来的结果就是 `true`

现在我们是不是可以得到一个点，在JS里面如果想成为一个对象的爹，你就必须是它的原型，你想是它的原型你就必须是一个叫做 `__proto__` 的东西，如果你想是 `__proto__` 现在叫 `prototype`

```
//人
var p1 = {
    sex:"男"
}

//学生
var stu1 = {
    userName: "张三",
    age: 18
}

//想让人变成学生的爹，怎么办？
//p1必须是stu1的原型
//__proto__
stu1.__proto__ = p1;           //stu1的原型就是p1, stu1对象的父级就是p1

"userName" in stu1;          //true;
"age" in stu1;               //true
"sex" in stu1;               //true

stu1.sex === stu1.__proto__.sex;
//子级对象没有， 默认找父级对象
```

通过上面的代码，我们知道，如果想让一个对象继承自另一个对象，最简单的方法就是设置它的原型 `__proto__`

现在各位同学已经有了原型的概念，那么，我们再继续的看一下下面的代码

```
var arr = ["a", "b", "c", "d"];
arr.push("e"); //向数组的最后面添加1个"e"

arr.push === arr.__proto__.push; //true
```

请问： `push` 这个方法是 `arr` 的还是 `arr` 的爹（原型）的

`arr` 本身是没有 `push` 的方法的，所以它五个人是它爹的方法，这就说明了一点，子级元素是默认可以使用父级元素的东西

总结

1. 原型就是 `__proto__`
2. 原型也是 `prototype`
3. 原型也可以理解为一个对象的爹
4. 当一个对象调用属性或方法时，当前对象如果不存在，默认就会向原型（爹）上面找
5. 每一个对象默认都会有原型

通过原型继承(简单版)

场景：现有一个对象 `a1`,这个对象上面有一个 `sayHello` 方法，有一个 `sleep` 的方法，有一个 `eat` 的方法，现在所有的动画对象都应该有这个三个方法，怎么办呢

```
var a1 = {
  sayHello: function () {
    console.log("大家好");
  },
  sleep: function () {
    console.log("睡觉");
  },
  eat: function () {
    console.log("吃东西");
  }
}

var s1 = {
  userName: "张三",
  sex: "男"
}
var s2 = {
  userName: "李四",
  sex: "男"
}
```

```
var s3 = {
    userName: "桃子",
    sex: "女"
}
// 现在所有的学生对象都应该有这个三个方法，怎么办呢
s1.__proto__ = a1;
s2.__proto__ = a1;
s3.__proto__ = a1;
```

看上面的代码，思考问题：如果现在我们创建65个学生对象，请问，咋办？这个时候我们发现上面的 `userName`, `sex` 这两部分的代码就会重复进行，怎么简化呢？

构造函数的原型继承【过渡】

(不建议使用，知道就行)

```
var a1 = {
    sayHello: function () {
        console.log("大家好");
    },
    sleep: function () {
        console.log("睡觉");
    },
    eat: function () {
        console.log("吃东西");
    }
}
//如果要创建很多个格式相同的对象，最好的办法就是先创建构造函数，再通过构造函数创建对象
function Student(userName, sex) {
    this.userName = userName;
    this.sex = sex;
}

var s1 = new Student("张珊", "女");
s1.__proto__ = a1;
var s2 = new Student("李四", "男");
s2.__proto__ = a1;
var s3 = new Student("标哥", "男");
s3.__proto__ = a1;
```

在上面的代码里面，我们发现，构造函数虽然帮我们解决了快速创建对象的问题，但是在创建完对象以后，我们仍然要手动的去指定原型 `__proto__`，这样也会非常麻烦

在JS的对象里面，一个对象的`__proto__`应该等于这个对象的构造函数的`prototype`属性

```
var a1 = {
    sayHello: function () {
        console.log("大家好");
    },
    sleep: function () {
        console.log("睡觉");
    },
    eat: function () {
        console.log("吃东西");
    }
}

//如果要创建很多个格式相同的对象，最好的办法就是先创建构造函数，再通过构造函数创建对象
function Student(userName, sex) {
    this.userName = userName;
    this.sex = sex;
}

//第一步推断：一个对象的__proto__等于这个对象的构造函数的prototype
//第二步推断：一个对象的父级对象（爹）等于这个对象的构造函数的prototype

// s1.__proto__ === Student.prototype;      //true
Student.prototype = a1;
var s1 = new Student("张珊", "女");
var s2 = new Student("李四", "男");
var s3 = new Student("标哥", "男");
```

构造函数的原型继承2【过渡】

```
//人的构造函数
function Person() {
    this.sayHello = function () {
        console.log("大家好");
    }
    this.sleep = function () {
        console.log("睡觉");
    }
    this.eat = function () {
        console.log("吃东西");
    }
}
```

```
}

//学生的构造函数
function Student(userName, sex) {
    this.userName = userName;
    this.sex = sex;
}
```

现在的要求非常简单，我们希望 `Student` 构造出来的对象要具备 `Person` 构造出来的对象里面的3个方法

```
var s1 = new Student("张珊", "女");
//希望s1具备 sayHello/sleep/eat三个方法，怎么办？
```

```
//人的构造函数
function Person() {
    this.sayHello = function () {
        console.log("大家好");
    }
    this.sleep = function () {
        console.log("睡觉");
    }
    this.eat = function () {
        console.log("吃东西");
    }
}

//学生的构造函数
function Student(userName, sex) {
    this.userName = userName;
    this.sex = sex;
}

// var p1 = new Person();
// Student.prototype = p1;

//Student所构造出来的对象，它的爹是Person构造出来的对象
Student.prototype = new Person();

var s1 = new Student("张珊", "女");
//希望s1具备 sayHello/sleep/eat三个方法，怎么办？
```

```
s1 instanceof Student;           //true
s1 instanceof Person;            //true
```

构造函数的原型继承3【标哥推荐版】

这个做法，仍然有一点小小的遗憾，它会破坏原型。但是这种方法是目前主流的方法

```
/**
 * 人：姓名，性别，sayHello方法
 * 学生：姓名，性别，学号，sayHello方法
 */

//人的构造函数
function Person(userName, sex) {
    this.userName = userName;
    this.sex = sex;
    this.sayHello = function () {
        console.log("hello world");
    }
}

//学生的构造函数
function Student(userName, sex, sid) {
    this.userName = userName;
    this.sex = sex;
    this.sayHello = function () {
        console.log("hello world");
    }
    this.sid = sid;
}
```

对于上面的代码，我们怎么样让学生Student继承了Person以后，简化我们的代码

```

<body>
<script>
    /**
     * 人: 姓名, 性别, sayHello方法
     * 学生: 姓名, 性别, 学号, sayHello方法
    */

    //人的构造函数
    function Person(userName, sex) {
        this.userName = userName;
        this.sex = sex;
        this.sayHello = function () {
            console.log("hello world");
        }
    }

    //学生的构造函数
    function Student(userName, sex, sid) {
        this.sid = sid;
    }

    //我们现在按照之前的办法去设置原型继承
    //现在Person需要2个参数，我们强有
    Student.prototype = new Person();

    var s1 = new Student("张三", "男", "H20040001");
    var s2 = new Student("李四", "男", "H20040002");
</script>
</body>

```

如果仍然通过原来的原型继承来实现的，这个时候我们就发现，如果父级要接收参数就接收不到了

上面问题的根本就在于当我们去继承的时候 `Student.prototype = new Person()` 我们还不知道学生的信息，这个时候的 `Person` 在接收参数的时候就会接收不到

```

    /**
     * 人: 姓名, 性别, sayHello方法
     * 学生: 姓名, 性别, 学号, sayHello方法
    */

    //人的构造函数
    function Person(userName, sex) {
        this.userName = userName;
        this.sex = sex;
        this.sayHello = function () {
            console.log("hello world");
        }
    }

    //学生的构造函数
    function Student(userName, sex, sid) {
        this.sid = sid;
        // Student.prototype = new Person(userName, sex);
    }

```

```

//this.__proto__ == Student.prototype;
this.__proto__ = new Person(userName, sex);
}

//依据：一个对象的__proto__等于它的构造函数的prototype
//我们现在按照之前的办法去设置原型继承
//现在Person需要2个参数，我们没有
// Student.prototype = new Person();

//s1.__proto__ === Student.prototype

var s1 = new Student("张三", "男", "H20040001");
var s2 = new Student("李四", "男", "H20040002");

```

面向对象的补充点

- 为什么所有的数据在 `instanceof Object` 的时候都会是 `true`

在JS里面，除 `null, undefined` 在外，所以对象最终都会继承自 `Object`

- 那么 `Object` 对象里面有2个的特殊的方法

- `toString()` 方法
- `valueOf()` 方法

关于 `toString()` 方法

`toString()` 方法是所有对象都具备的方法，而我们之前在讲运算符的时候有一个加法运算符执行的是“字符串优先”原则

```

var obj1 = {
  age: 19,
  money: 1000
}
var obj2 = {
  age: 50,
  money: 10
}
var arr1 = [123];

//加法：执行字符串优先原则，如果是对象，则会调用对象的toString方法

//obj1.toString();           //'[object Object]'

```

```
//obj2.toString();           //'[object Object]'  
var result1 = obj1 + obj2;   //'[object Object][object Object]'
```

在上面的代码里面，它就会把两个对象 `toString()` 以后再相加

```
//arr1.toString()    '123';  
var result2 = obj1 + arr1;      //'[object Object]123'
```

再看下面的代码

```
// 继承一个对象以后就可以使用这个对象上面所有的方法与属性  
// 自己有就用自己的，自己没有就找父级  
var obj1 = {  
    age: 19,  
    money: 1000,  
    toString: function () {  
        return this.age;  
    }  
}  
var obj2 = {  
    age: 50,  
    money: 10,  
    toString: function () {  
        return this.age;  
    }  
}  
  
//obj1.toString();          //19  
//obj2.toString();          //50  
var result3 = obj1 + obj2;    //69
```

关于 `valueOf()` 方法

- 所有执行以数字优先的相关操作的都调用 `valueOf`
- 如果 `valueOf` 调用以后得不到结果或得到了没有用的结果就调用 `toString()`

```
// 继承一个对象以后就可以使用这个对象上面所有的方法与属性  
// 自己有就用自己的，自己没有就找父级  
var obj1 = {  
    age: 19,  
    money: 1000,  
    toString: function () {
```

```

        return this.age;
    }
}

var obj2 = {
    age: 50,
    money: 10,
    toString: function () {
        return this.age;
    }
}

//obj1.valueOf();           //调用失败,转而toString() 19
//obj2.valueOf();           //调用失败,转而toString() 50

var result2 = obj1 - obj2; // -31

```

现在我们手动的再指定 `valueOf` 方法

```

var obj1 = {
    age: 19,
    money: 1000,
    toString: function () {
        return this.age;
    },
    valueOf: function () {
        return this.money;
    }
}

var obj2 = {
    age: 50,
    money: 10,
    toString: function () {
        return this.age;
    },
    valueOf: function () {
        return this.money;
    }
}

//obj1.valueOf();           //调用成功 1000
//obj2.valueOf();           //调用成功 10
var result2 = obj1 - obj2; // 990

```

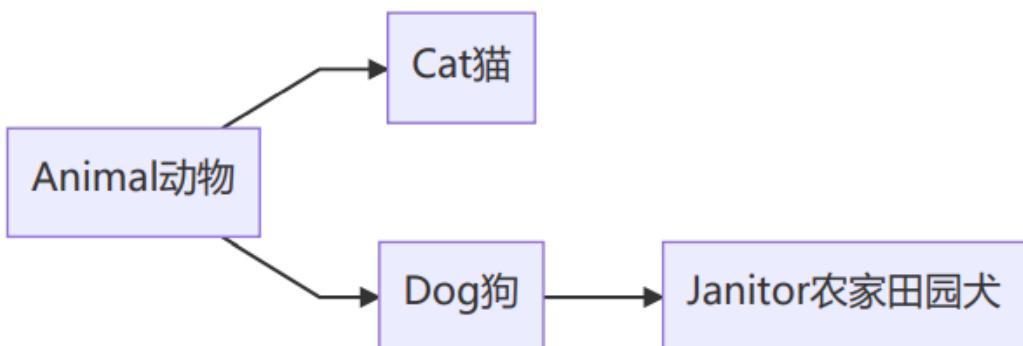
```
var result3 = obj1 > obj2;      //true
```

练习

1. 现在几个动物分别小猫Cat和小狗Dog,Cat构造函数里面有性别（sex），年龄（age），昵称（nickName）和体重（weight）四个属性，而Dog构造函数里面有性别，年龄，昵称，和身高（height）四个属性，猫与狗有一个共同的方法叫sleep睡觉，但是猫有一个方法 `miao` ,狗有一个方法叫 `wang`
 - 现在请列举出Cat与Dog的构造函数，并且提取公共部分使用继承
 - 突然之间又有一个小动物，农家田园犬Pastoral，它也是属于狗的类别，也具备Dog所有的属性，但是它还有一个方法是看门 `janitor`

现在请创建 Pastoral 的构造函数，并实现继承关系

既然是使用构造函数，那么肯定是使用原型继承



```
function Animal(sex, age, nickName) {  
    this.sex = sex;  
    this.age = age;  
    this.nickName = nickName;  
    this.sleep = function () {  
        console.log("我在睡觉");  
    }  
}  
  
//猫的构造函数  
function Cat(sex, age, nickName, weight) {  
    this.weight = weight;
```

```

this.miao = function () {
    console.log("喵喵喵...") ;
}
this.__proto__ = Cat.prototype = new Animal(sex, age, nickName);
}

//猫的构造函数
function Dog(sex, age, nickName, height) {
    this.height = height;
    this.wang = function () {
        console.log("旺旺旺...") ;
    }
    this.__proto__ = Dog.prototype = new Animal(sex, age, nickName);
}

//家家田园犬
function Pastoral(sex, age, nickName, height) {
    this.janitor = function () {
        console.log("我在看门....") ;
    }
    this.__proto__ = Pastoral.prototype = new Dog(sex, age, nickName,
height);
}

var c1 = new Cat("公", 3, "小花", 5);
var d1 = new Dog("母", 1, "阿黄", 10);
var p1 = new Pastoral("公", 2, "科基", 8);

```

上面是我们使用了标哥的方式去完成的

2. 还是上面的题目，请使用组合继承（寄生继承）的方式来写一遍

- 先只完成 Dog、Cat 继承 Animal
- 完成了上面的操作以后，再让 Pastoral 继承 Dog
- 修复原型链

```

function Animal(sex, age, nickName) {
    this.sex = sex;
    this.age = age;
    this.nickName = nickName;
    this.sleep = function () {
        console.log("睡觉吧...") ;
    }
}

```

```
this.__proto__ = Animal.prototype;
}

//猫的构造函数
function Cat(sex, age, nickName, weight) {
    this.weight = weight;
    this.miao = function () {
        console.log("喵喵喵...");
    }
    // this.__proto__;
    // Cat.prototype
    // 寄生继承

    // this.__proto__.sex = sex;
    // this.__proto__.age = age;
    // this.__proto__.nickName = nickName;
    Animal.call(Cat.prototype, sex, age, nickName);
    this.__proto__ = Cat.prototype;

}

//狗的构造函数
function Dog(sex, age, nickName, height) {
    this.height = height;
    this.wang = function () {
        console.log("旺旺旺...");
    }
    // 开始组合继承
    Animal.call(Dog.prototype, sex, age, nickName);
    //修复原型
    this.__proto__ = Dog.prototype;
}

//田园犬
function Pastoral(sex, age, nickName, height) {
    this.janitor = function () {
        console.log("我在辛苦的看家");
    }
    // 开始组合继承
    Dog.call(Pastoral.prototype, sex, age, nickName, height);
    //修复原型
    this.__proto__ = Pastoral.prototype;
```

```
}

var c1 = new Cat("母", 3, "小花", 2);
var d1 = new Dog("公", 2, "大黄", 1);
var p1 = new Pastoral("公", 4, "小田田", 3);
```

对象在内存当中的存储

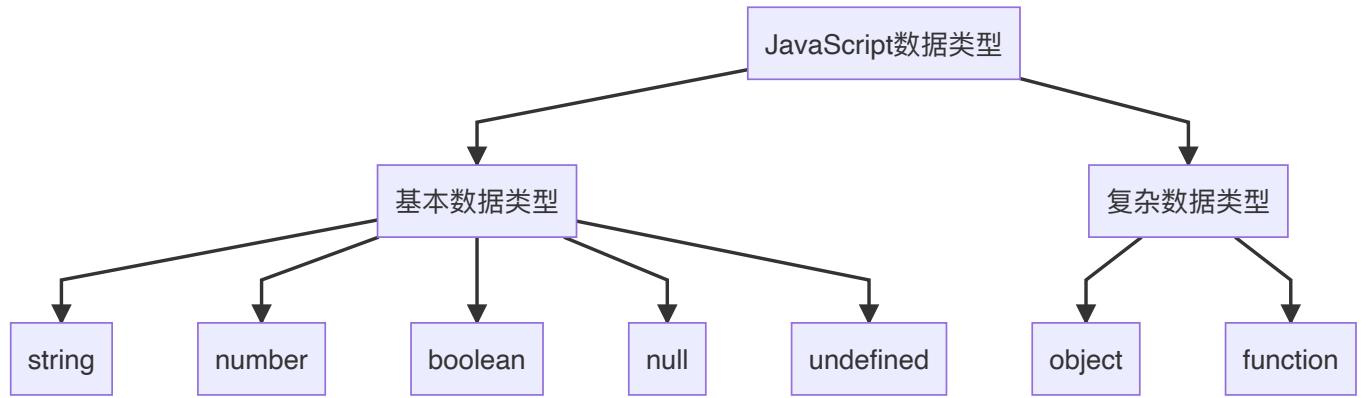
这个章节也叫内存的堆栈原理

在之前面向对象的章节当中我们讲了对象的封装（创建）,继承，讲完这些东西以后，我们对对象应该有了一个初步的认识

1. 对象是一个复杂数据类型

在JavaScript里面，数据类型分2大类，分别是基本数据类型和复杂数据类型（复杂数据类型指的就是对象）

1. 基本数据类型分为5大类，复杂数据类型就是一个 `object`，基本数据类型使用 `typeof` 的关键字来检测，复杂数据类型使用 `instanceof` 去检测
2. 复杂数据类型也有另一种叫法叫“引用类型”



从上面这个图里面，我们可以很清楚的看到JS把数据类型分为了2大类，这是为什么呢？

JS将数据类型分为2大类的原因还是与计算的内存有关系，因为它们在内存当中的存储形式是不一样的，我们先通过一个简单的例子来说明

```
var x = "标哥";
var y = x;
x = "颜一鸣";
console.log(y);
//结论: x, y 2个变量互不影响
```

在上面的代码里面我们应该可以看到一个点，2个变量之间应该是互不影响的，`x`, `y`应该是相互独立的（这其实就是基本数据类型的特点）

现在再看第2个例子

```
var obj1 ={
  userName: "张三"
}
var obj2 = obj1;
obj1.userName = "李四";
console.log(obj2.userName);
// 结论: obj1, obj2 2个变量互相影响
```

通过上面的代码我们可以发现，2个变量之间又相互影响了（这就是复杂数据类型的特点）

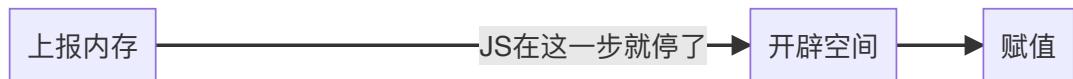
要弄清楚基本数据类型与复杂数据类型的根本特点，我们一定要深入到内存当中去看

数据的内存储存

内存是负责存储数据的，在存储的时候数据结构主要分为4大部分，分别是“堆，栈，链，表”。在目前主要的编程语言里面（如JS, java）只使用到了“堆”和“栈”这两大内存结构，**我们的基本数据类型是保存在内存的栈空间里面，我们的复杂数据类型是保存在内存的堆空间里面**

```
//这是JS代码
var a = 10;
var b;
```

在编译系统里面，所有的变量的初始化都分为3个步骤，无论是何种语言



JS是一个弱类型的语言，它的空间大小由后面的值决定

如果说是 **Java** 的强类型语言

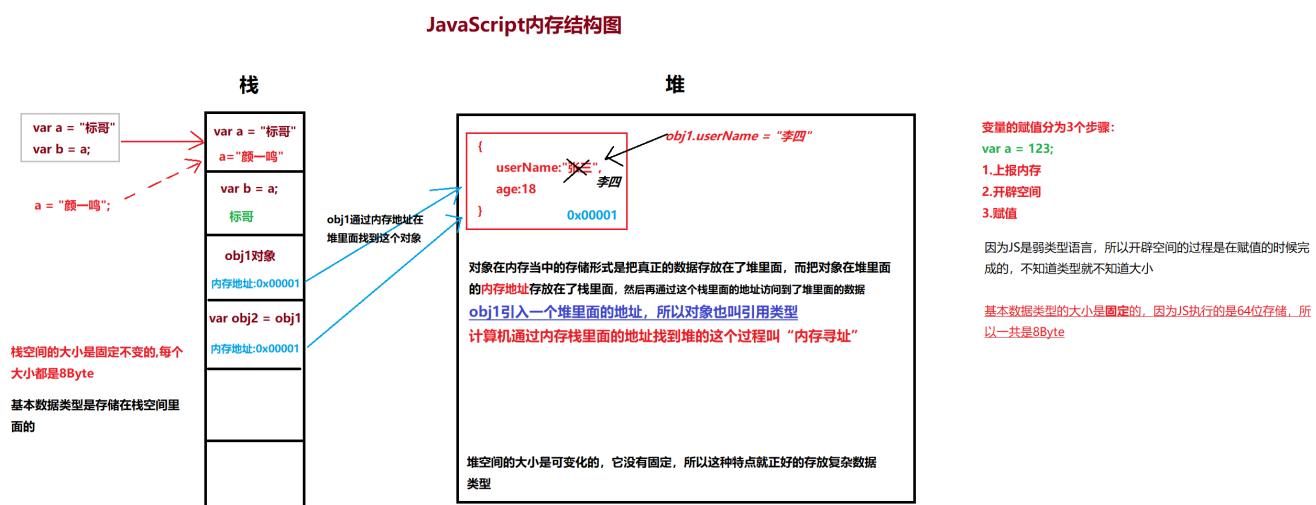
```
//这是Java代码
int a = 10;
int b; //即使在没有赋值的情况下，我也知道b变量是int类型，占4个Byte
```



从上面的图当中我们可以看到，无论是什么编程语言的定义变量的时候都会有一个开辟空间的过程，只要关注这个空间到底在内存的什么地方开辟，在不同的地方开辟所表现出来的特点是不一样的

栈：栈空间的大小是固定不变的，它是8Byte，编程语言只能直接操作栈空间

堆：堆空间的大小是可变化的，编程语言不能直接操作堆空间（可以间接操作），**C/C++** 除外



在上图当中我们可以看到基本数据类型的存储方式与复杂数据类型的存储方式是不一样的，基本数据类型因为固定了大小所以是放在了栈里面，而复杂数据类型因为没有固定大小，所以在内存的堆当中存储，然后把这个堆的地址又放到了栈里面

复杂数据类型通过这个栈里面的地址去堆里面寻找内存的某一个空间，相当于栈里面引入了堆里面的某一个内存地址，所以也常常将复杂数据类型叫引用类型

- `var obj2 = obj1` 本质上面是把 `obj1` 里面存放的内存地址给了 `obj2`，这样两个内存地址指向的是同一个内存空间，像这种传递地址的过程我们叫地址传递
- `var b = a` 这个过程是把 `a` 在栈里面存放的数据拷贝了一份给 `b`，这样的两个栈的空间是相互隔离的，互不影响，这个传递过程我们叫值传递

总结：基本数据类型采用值传递，复杂数据类型采用地址传递

场景一

```
function abc(x){  
    x = 20;  
}  
  
var a = 10;  
abc(a);  
console.log(a);
```

代码分析：

在上面的场景当中，最终打印的结果仍然是10。原因就在于a是一个基本数据型，在传递的时候使用值传递，形参x在接收的时候接收到的是值10，a与x就是两个互不影响的，x后面在函数里面变成了20以后对a没有任何影响

场景二

```
function def(_arr){  
    _arr[0] = "标哥";  
}  
  
var arr = ["a", "b", "c"]  
def(arr);  
console.log(arr);
```

代码分析

`arr` 是一个数组，它是一个对象也就是一个复杂数据类型，复数数据类型是存放在内存的堆里面，然后堆里在帧地址又放在了栈里面，在进行传递的时候，它是把栈里的地址传递出去了，最终通过地址所找到的对象是同一个对象，当`arr`的实参传递给形参`_arr`的时候，它们的地址是同一个地址，这样通过`_arr`改变一个值以后，外边的`arr`也被改变了

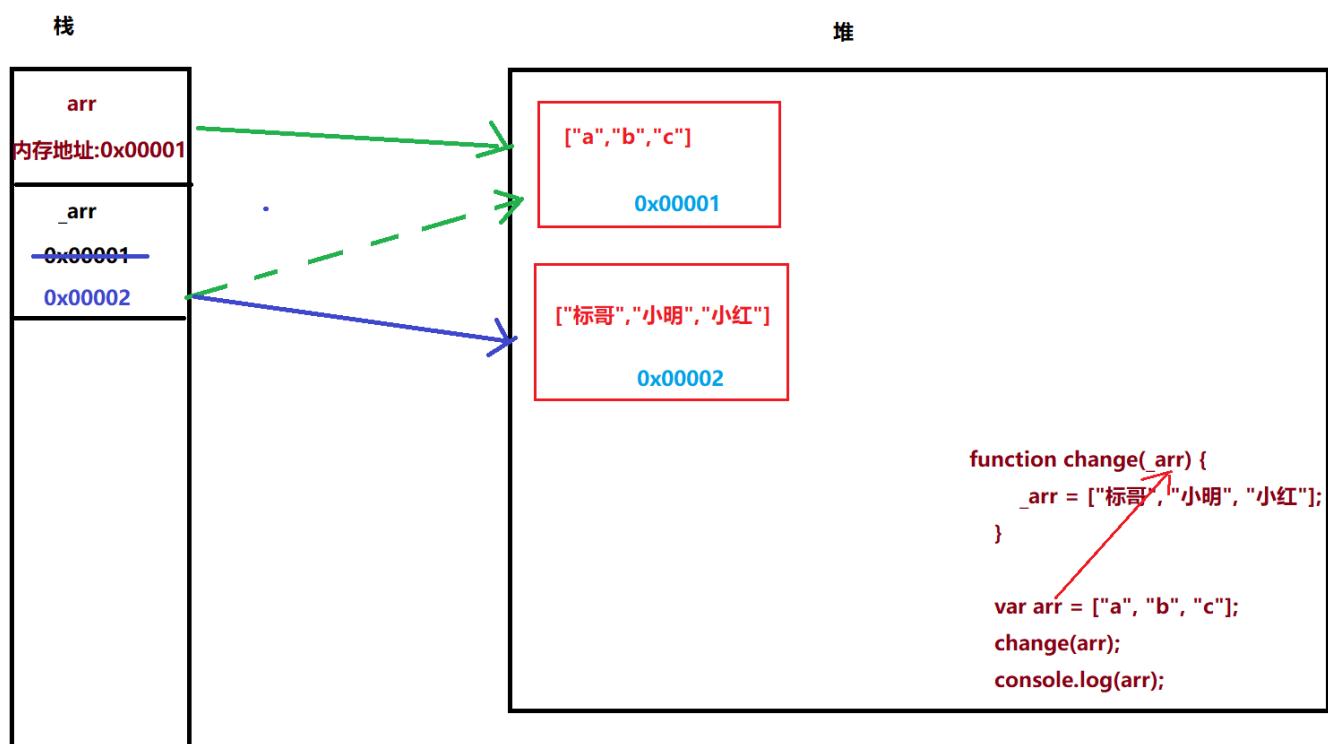
场景三

```
function change(_arr) {  
    _arr = ["标哥", "小明", "小红"];  
}  
  
var arr = ["a", "b", "c"];  
change(arr);  
console.log(arr);
```

代码分析

这一种场景虽然前期在`arr`传递给`_arr`的时候仍然传递的是地址，但是后面的`_arr`又重新的赋值了一个新的数组，这个`arr`与`_arr`所表示的地址就不一样的，最终两个人就不是不相同的数组

JavaScript内存结构图



场景四

```
var arr = ["a", "b", "c", { userName: "标哥" }, "e", "f"];
// arr = ["a", "b", "c", 0x00001, "e", "f"];

//0x00002
var obj = {
    userName: "标哥"
}

var index1 = arr.indexOf("c");           //2
var index2 = arr.indexOf(obj);          //-1
```

代码分析

对象在通过 `indexOf` 去查找的时候，它找的是地址所以会存在找不到的情况，同时这个问题也可以解释以下的问题

`[]==[]` 结果是 `false` , `{ }=={ }` 结果也是 `false`

对象的拷贝

对象在进行传递的时候执行的是地址传递，它的拷贝现象会有两种情况

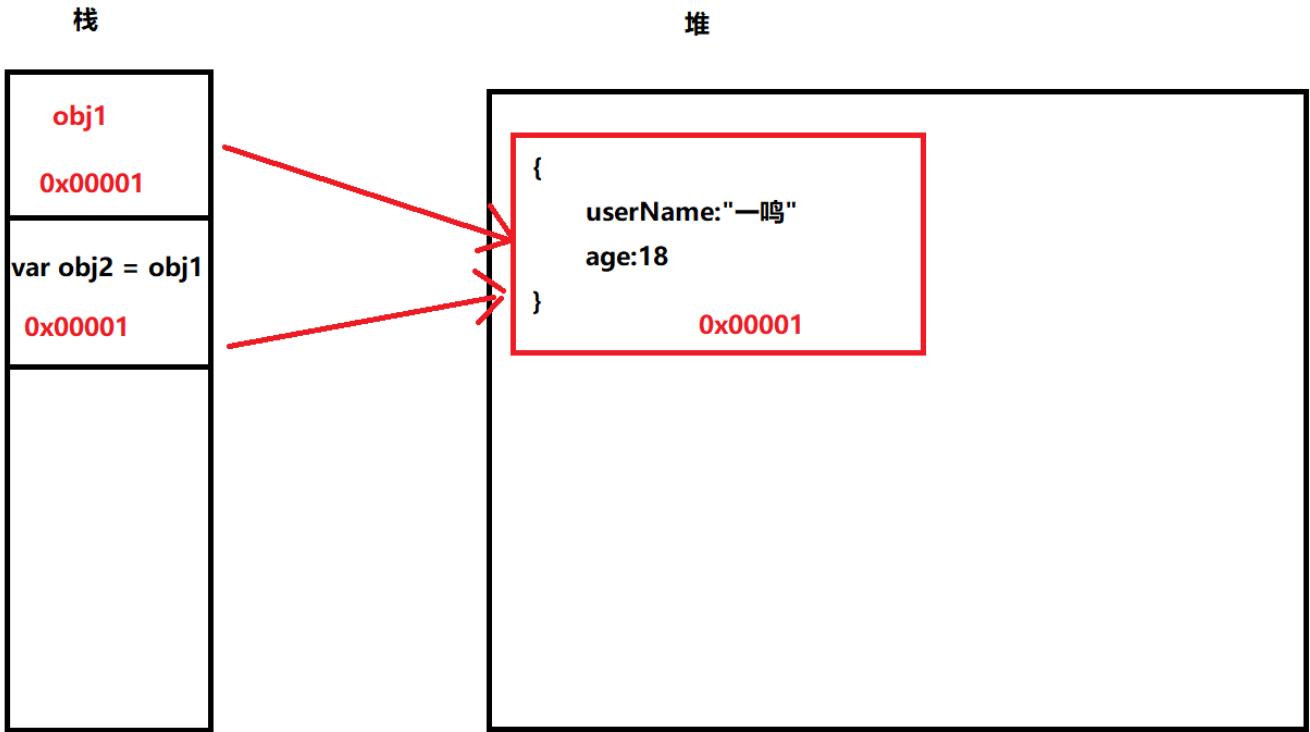
1. 对象的浅拷贝
2. 对象的深拷贝

对象的浅拷贝

对象的浅拷贝就是只拷贝栈里面的数据

在对象的浅拷贝里面，它栈里面的地址是相同的

```
var obj1 = {
    userName:"一鸣",
    age:18
}
var obj2 = obj1;      //这一种现象我们称之为浅拷贝
                     //这一种现象只拷贝了栈里面的地址，这个地址指向了同一个对象
```



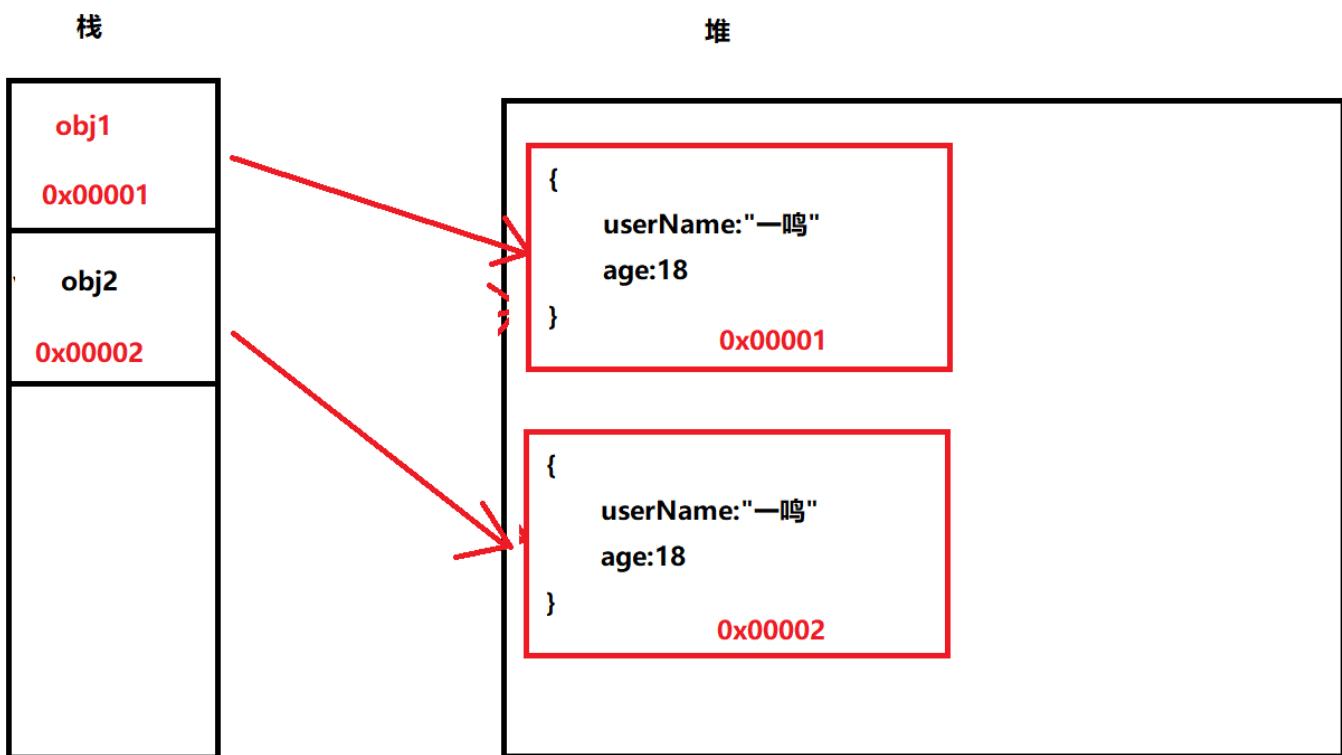
在上在的图当中我们可以的看到这一种现象就是浅拷贝，浅拷贝最大的问题就是只拷贝了栈里面的地址，这样造成的后果就是两个对象指向了同一个堆的地址，两个变量之间就会相互影响

```
obj1.userName = "张珊";
console.log(obj2.userName); //张珊
```

针对上现的现象，如果我们希望得到一个互不影响的两个对象，怎么办呢？这个时候就要需要有一种新的技术叫对象深拷贝

对象深拷贝

简而言而，对象的深拷贝就是深入到内存的堆里面去拷贝一份出来，这样两个对象指向的堆的地址就不是同一个地址



在上面的图里面，我们可以看到在堆里面我们得到了一模一样的对象，`obj1` 与 `obj2` 两个对象之间互不影响

简单对象的深拷贝 【1】

```

var obj1 = {
  userName: "张珊",
  age: 18
}

Object.defineProperty(obj1, "sex", {
  configurable: false,
  enumerable: false,
  writable: false,
  value: "女"
});

//现在想得到obj2的对象，然后与obj1完全相同，但又互不影响
//第一步：先创建一个对象
var obj2 = {} //JS会在内存的堆里面创建一个新对象
//第二步：获取原对象的属性
var arr = Object.getOwnPropertyNames(obj1); //['userName', 'age', 'sex']
//第三步：将原对象的属性值遍历出来，放在新的对象上面
arr.forEach(function (item) {

```

```
//item代表遍历出来的每一项，也就是属性名  
obj2[item] = obj1[item];  
})
```

在上面的代码里面，我们使用了一个最简单的对象深拷贝过程，但是要注意这里不能使用 `for...in` 也不能使用 `Object.keys()` 来完成，因为这两个方法获取不到 `enumerable:false` 的属性，只有 `Object.getOwnPropertyNames()` 这个方法才可以获取所有的属性

简单数组深拷贝【2】

数组也是一个对象，如果想实现数组的深拷贝，也是有办法的

```
var arr = ["a", "b", "c", "d", "e"];  
//相得到一份相同的数组，但是又互不影响  
  
//第一种办法，concat  
var arr1 = arr.concat();  
  
//第二种办法，slice  
var arr2 = arr.slice();  
  
//第三种方法  
var arr3 = arr.map(function(item){  
    return item;  
});
```

通过 `Object.assign()` 拷贝简单对象【3】

这一个方法是JS专门提供给我们做对象的深拷贝的方法，它的语法格式如下

```
var 目标对象 = Object.assign(目标对象, 源对象...);
```

上面的语法格式说得非常清楚，它会把源对象拷贝到目标对象里面同时再返回这个目标对象

```
var obj1 = {
  userName: "张珊",
  age: 18,
  sex: "女"
}

//请创建对象obj2，实现obj1的所有属性，但是又互不影响
//第一步：创建一个新对象
var obj2 = {};
obj2 = Object.assign(obj2, obj1);
```

`Object.assign()` 除了可以实现对象的拷贝以外还可以实现对象的合并

```
var obj1 = {
  userName: "张珊",
  age: 18,
  sex: "女"
}

var obj2 = {
  height: 170,
  weight: 50,
  age: 20
}

var obj3 = {} //新对象
Object.assign(obj3, obj1, obj2);
//obj1先拷贝给obj3，再把obj2拷贝给obj3
```

注意事项：`Object.assign()` 在拷贝对象的时候它会拿不到 `enumerable:false` 的属性

复杂对象的深拷贝【终极】

上面的三个方法讲解了对象的深拷贝，其实都是假的（标哥说它们是假的深拷贝），当面对对象复杂的对象的时候（也就是对象里面又包含对象的时候就会出问题）

推翻第1点

```
var obj1 = {
  userName: "张三",
  age: 18,
  teacherInfo: {
    name: "标哥哥",
    height: 170,
```

```

        weight: 65
    }
}

//现在想创建一个obj2,与obj1完全相同，但又互不影响
// 第一步：创建新对象
var obj2 = {} ; //JS在内存的堆里又创建一个对象，与之前的obj1不是同一个对象
//第二步：获取所有的属性名
var arr = Object.getOwnPropertyNames(obj1);
//第三步：遍历旧对象的属性，赋值到新对象上面
arr.forEach(function(item){
    //item代表当前的属性
    obj2[item] = obj1[item];
});

```

我们还是采用了之前的方法去进行，但是就发现有问题了

```

obj1.teacherInfo.name = "第一帅哥";
console.log(obj2.teacherInfo.name); //第一帅哥

```

我们现在就发现这两个对象还没有完全的隔开

推翻第2点

```

var arr = [
    {userName:"张三", age:18},
    {userName:"李四", age:19},
    {userName:"王五", age:20},
    {userName:"赵六", age:21},
    {userName:"田七", age:22}
]

```

```

//第一种方式
var arr1 = arr.concat();

```

```

//第二种方式
var arr2 = arr.slice();

```

```

//现在我们去改变arr里面的第0项
arr[0].userName="韩宏扬";
//这里我们发现arr1与arr2都发生了变化

```

之前我们所学习的数组里面的拷贝也存在了问题，如果数组里面放基本数据类型没有问题，但是如果数组里面放的是对象就不行了

推翻第3点

```
var obj1 = {  
    userName: "张三",  
    age: 18,  
    teacherInfo: {  
        name: "标哥哥",  
        height: 170,  
        weight: 65  
    }  
}  
  
//创建了新对象  
var obj2 = {};  
Object.assign(obj2,obj1);  
  
obj1.teacherInfo.name = "帅哥哥哥哥";  
console.log(obj2.teacherInfo.name); //帅哥哥哥哥
```

在这里我们也发现 `Object.assign()` 也只能实现简单对象的拷贝对象复杂对象的拷贝仍然无能为力

系统到目前为止是没有提供任何有效的方法完全的实现对象的深拷贝的，如果想完全实现对象的深拷贝一定要手写深拷贝

标哥的手写深拷贝

```
var obj1 = {  
    userName: "张珊",  
    sex: "男",  
    computer: {  
        type: "国产",  
        price: 3999,  
        name: "RedmiBook16"  
    },  
    boyFriends: ["男朋友1号", "男朋友2号", "备胎1号", "2号鱼"]  
}  
  
//现在想对上面的对象实现完整的拷贝，怎么办呢?
```

```

//我给你一个对象，你还我一个互不影响 的相同对象
function deepCopy(oldObj) {
    if (typeof oldObj != "object" || oldObj == null) {
        //说明不是对象，或是null，直接返回
        return oldObj;
    }
    //第一步：创建对象
    var newObj = Array.isArray(oldObj) ? [] : {};
    var arr = Object.getOwnPropertyNames(oldObj);
    arr.forEach(function (item) {
        newObj[item] = deepCopy(oldObj[item]);
    });
    return newObj;
}

var obj2 = deepCopy(obj1);

```

跨对象调用方法【重点】

之前我们已经讲过了在对象里面调用方法的方式，还有函数的调用调用

```

function aaa(userName) {
    console.log(this);
    console.log("你好," + userName);
}

var obj1 = {
    age:18
}

window.aaa("张珊");           //aaa被window调用，this指向window
window.aaa.call(obj1, "李四"); //aaa呼叫obj1调用，this指向obj1
window.aaa.apply(obj1, ["王五"]); //aaa申请obj1来调用，this指向obj1

```

这些 **call/apply** 调用方法的方式我们之前已经接触过了。现在我们来看一下 **call/apply** 还有什么其它的用法

call 有呼叫的意思，**方法.call()** 有呼叫来调用这个方法的意思

apply 有申请的意思，**方法.apply** 有申请谁来叫用这个方法的意思

所以通过 `call/apply` 去调用的时候，里面的 `this` 就指向了当前申请或呼叫的这个人，本质上其实还是方法的 `this` 指向了当前调用这个方法的对象

现在请同学们看下面的代码

```
var stu1 = {
    userName: "颜一鸣",
    doHomeWork: function () {
        console.log(this.userName);
    }
}

var stu2 = {
    userName: "袁池康"
}

stu1.doHomeWork();

//在上面的代码里面，我们的stu1与stu2都有userName，stu1有一个方法doHomeWork
//现在stu2没有这个方法，请问袁池康有没有办法调用stu1的doHomeWork方法呢？
```

现在的问题就在于能否让 `stu2` 也去调用 `stu1` 的 `doHomeWork` 方法

换句话来说 `stu2` 就是要借用 `stu1` 的 `doHomeWork` 的方法

第一种办法

```
stu2.__proto__ = stu1;

stu2.doHomeWork === stu1.doHomeWork;           //true
stu2.doHomeWork === stu2.__proto__.doHomeWork; //true

stu2.doHomeWork();                //袁池康
stu2.__proto__.doHomeWork();     //颜一鸣
```

在这一种方法里面，我们只要让一个对象继承了另一个对象不就可以使用这个对象的方法了呢，所以我们让 `stu2` 继承了 `stu1`，这样就可以实现我们的场景需求

`stu2.doHomeWork()` 和 `stu2.__proto__.doHomeWork()` 在调用方法的时候，因为调用者发生了变化，所以方法里面的 `this` 也发生了变化，这一种要注意

第二种办法

```
stu1.doHomeWork.call(stu2);      //袁池康      //stu1.doHomeWork呼叫stu2过来调用
stu1.doHomeWork.apply(stu2);      //袁池康      //stu1.doHomeWork申请stu2过来调用
```

这一种情况就是跨对象调用方法的应用场景

案例一：求数组的最大值

```
var arr = [1, 5, 7, 9, 2, 3, 8];
```

现在有上面的数组，我们要求这个数组的最大值怎么办呢。

解决这个问题我们已经有了很多个思路，原始的方法我们可以使用比武招亲的思维，数组里面也提供了归并方法

```
//第一种思路：比武招亲
var max = arr[0];
for (var i = 1; i < arr.length; i++) {
    if (max < arr[i]) {
        max = arr[i];
    }
}
```

```
//第二种：数组归并
var max = arr.reduce(function (prev, current, index, _arr) {
    return prev > current ? prev : current;
})
```

上面的2种方式仍然是我们在这个地方之前所学过的试试，现在我来教一个新的方法给大家

在后面的学习当中我们有一个对象叫 **Math** 内置对象，这个对象下面有一个 **Max** 这个方法，它可以求一系列数的最大值。我们现在就要借用一下系统的这个方法去使用

```
var arr = [1, 5, 7, 9, 2, 3, 8];
var max = Math.max(1, 5, 7, 9, 2, 3, 8);
console.log(max);      //9
```

上面的 **Math.max** 虽然帮我们找到了最大值，但是它需要我们将所有的数组元素1个1个的放进去，这样做非常麻烦，怎么办呢

我们知道方法除了使用 **方法名+()** 来调用以外，还可以通过 **call/apply** 来调用

```
//var max = Math.max.call(Math,1, 5, 7, 9, 2, 3, 8);
// var max = Math.max.apply(Math,[1, 5, 7, 9, 2, 3, 8]);
var max = Math.max.apply(Math,arr);
```

说明一下：虽然说跨对象调用，但是这里没有跨对象调用，因为最后还是 **Math** 在调用

案例二：将类数组转换为数组

类数组：也叫伪数组，具备数组的特点（通过索引来取值赋值，通过length来获取长度），但是不具备数组的方法

```
var obj = {
  0:"a",
  1:"b",
  2:"c",
  3:"d",
  4:"e",
  5:"f",
  length:6
}
```

上面是一个对象，它是一个类数组，它有数组的特性（有索引，有长度），但是不具备数组的方法，所以如何将上面的这个类数组转换成数组 `["a", "b", "c", "d", "e", "f"]`

推导过程

```
var arr = ["a", "b", "c", "d", "e", "f"];

var arr1 = arr.slice();
//slice的结果一定是一个数组
//能不能让obj也借用一下数组的slice方法呢?
arr.slice === arr.__proto__.slice; //true

var arr2 = arr.__proto__.slice(); //出错了
//它得到一个空的数组，没有元素
//原因arr.slice中slice里面的this指向提arr
//而arr.__proto__.slice方法中的slice的this指向arr.__proto__
//所以我们要让arr.__proto__.slice中的this指向arr

var arr3 = arr.__proto__.slice.call(arr);
```

经过上面的推理以后，我们发现 `arr.slice()` 就相当于

```
arr.__proto__.slice.call(arr)
```

现在我们来回顾一个点之前我们讲过一个对象的 `__proto__` 应该等于它的构造函数的 `prototype`

`arr` 是一个数组，它的构造函数是 `Array`，
`var arr = new Array()`

```
arr.__proto__ === Array.prototype;           //true
arr.__proto__.slice === Array.prototype.slice; //true
var arr4 = Array.prototype.slice.call(arr);    //又得到了一个数组
```

```
> arr.slice()
< ▶ (6) ['a', 'b', 'c', 'd', 'e', 'f']
> arr.__proto__.slice.call(arr)
< ▶ (6) ['a', 'b', 'c', 'd', 'e', 'f']
> Array.prototype.slice.call(arr)
< ▶ (6) ['a', 'b', 'c', 'd', 'e', 'f']
\
```

上面的三种方法本质上其实都是一样的

现在思考一下，如果其它对象也可以调用 `slice`，是不是也可以得到一个新数组

结论

```
var obj = {
  0: "a",
  1: "b",
  2: "c",
  3: "d",
  4: "e",
  5: "f",
  length: 6
}

//var arr = obj.slice(); //但是obj没有slice方法，所以会报错
//所以这里就代用Array数组的slice方法
var arr1 = Array.prototype.slice.call(obj);
```

同理，我们也可以使用数组的其它方法来试一下

```
var str = Array.prototype.join.call(obj, "#"); // 'a#b#c#d#e#f'
```

执行上下文栈

什么是执行上下文

我们可以将执行上下文看作代码当前运行的环境。代码的运行环境分为三种

- 全局级别的代码 - 这个是默认的代码运行环境，一旦代码被载入，js引擎最先进入的就是这个环境
全局执行环境可以看成是浏览器里面的 **script** 标签,一个页面的多个 **script** 会构成一个全局执行环境
- 函数级别的代码 - 当执行一个函数时，运行函数体中的代码（局部环境）
- ~~Eval的代码 - 在Eval函数内运行的代码（这个不常使用，也不推荐使用，故不作了解）~~

其实，主要就是全局执行上下文和函数执行上下文。下面举一个简单的例子：

```
// global context

var sayHello = 'Hello';

function person() {           // execution context
    var first = 'David',
        last = 'Shariff';

    function firstName() { // execution context
        return first;
    }

    function lastName() { // execution context
        return last;
    }

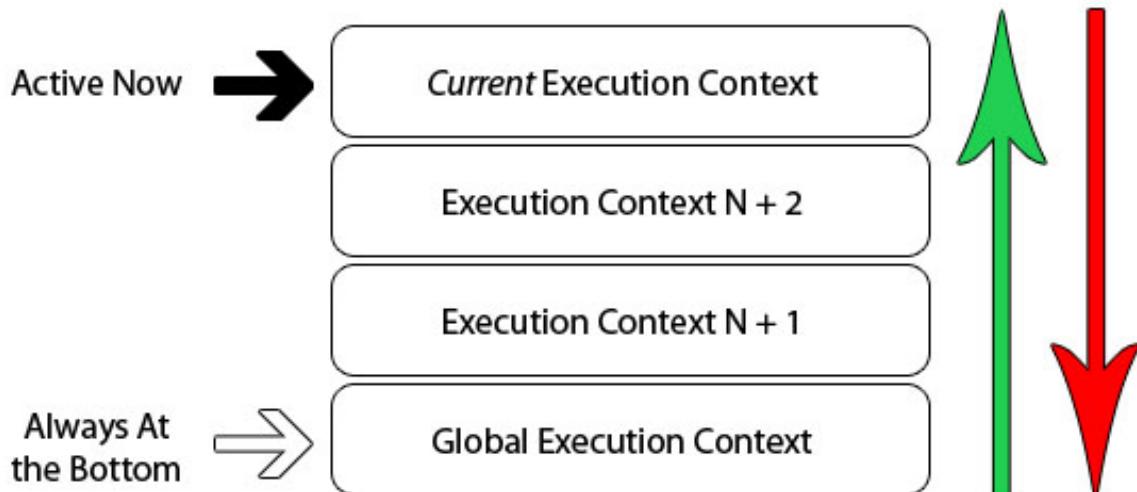
    alert(sayHello + firstName() + ' ' + lastName());
}
```

上图中，一共用4个执行上下文，紫色边框括起来的部分代表全局的上下文；绿色边框括起来的部分代表 `person` 函数内的上下文；蓝色边框括起来的部分代表 `person` 函数内的 `firstname` 函数的上下文；橙色边框括起来的部分代表 `person` 函数内的 `lastname` 函数的上下文。

注意：不管什么情况下，只存在一个全局的上下文，该上下文能被任何其它的上下文所访问到。函数上下文的个数是没有任何限制的，每到调用执行一个函数时，JavaScript 引擎就会自动新建出一个函数上下文。在外部的上下文中是无法直接访问到其内部上下文里的变量的，但是内部上下文可以访问到外部上下文中的的变量。（外边的不能访问里面的，里面的可以访问外边的）。其实一块就解释了我们的变量里面的全局变量和局部变量的区别

执行上下文堆栈

在浏览器中，JavaScript引擎的工作方式是单线程的（单线程就是排队执行,也就是说，某一时刻只有唯一的一个事件是被激活处理的）其它的事件被放入队列中，等待被处理。下面的示例图描述了这样的一个堆栈：



我们已经知道，当JavaScript代码文件被浏览器载入后，默认最先进入的是一个全局的执行上下文。当在全局上下文中调用执行一个函数时，程序流就进入该被调用函数内，此时引擎就会为该函数创建一个新的执行上下文，并且将其压入到执行上下文堆栈的顶部。浏览器总是执行当前在堆栈顶部的上下文，一旦执行完毕，该上下文就会从堆栈顶部被弹出，然后，进入其下的上下文执行代码。这样，堆栈中的上下文就会被依次执行并且弹出堆栈，直到回到全局的上下文。

执行上下文的建立过程

我们现在已经知道，每当调用一个函数时，一个新的执行上下文就会被创建出来。然而，在JavaScript引擎内部，这个上下文的创建过程具体分为两个阶段：

1. 代码的建立阶段

- 建立函数，变量
- 建立作用域
- 初步确定this的值

2. 代码的执行阶段

- 变量赋值，执行其它代码，最终确定this的值

实际上，可以把执行上下文看做一个对象，其下包含了以上3个属性

```
variableObject: { /* 函数中的参数对象并给参数赋值，内部的变量以及函数声明 */ },
scopeChain: { /* variableObject 以及所有父执行上下文中的variableObject */ },
this: {}
```

上面的variableObject我们一般叫变量对象，有时候简称VO，下面的scopeChain我们叫作用域，有时候也叫活动域，简称AO

执行上下文建立阶段以及代码执行阶段

上述第一个阶段即执行上下文建立阶段的具体过程如下：

当代码一旦开始进入的时候这个环境就会创建，环境的创建有2个阶段，一个是建立阶段，一个是执行阶段



1. 第一个阶段-建立阶段：

- 建立VariableObject对象（简称VO）
 - 建立arguments对象（全局环境里面没有），检查当前上下文中的参数，建立该对象下的属性以及属性值

- 检查当前上下文中的 **function** 函数声明。每找到一个函数声明，就在 VariableObject下面用函数名建立一个属性，属性值就是指向该函数在内存中的地址的一个引用,如果上述函数名已经存在于VariableObject下，那么则忽略后面的函数声明。

```

sayHello();           //执行阶段运行的代码

//在全局环境里面定义了一个方法
//执行环境在建立阶段去检查一下你有没有 function定义的函数，如果有，就直接给你建立好
function sayHello(){
    console.log("大家好才是真的好");
}

/*
建立阶段：
建立了一个方法sayHello

执行阶段：
调用执行了这个方法sayHello
*/

```

- 检查当前上下文中的变量声明。每找到一个变量声明，就在VariableObject下面用变量名建立一个属性，该属性值为undefined。如果变量属性名和函数属性名重复，则不建立新的变量。

```

var a = "hello";
//这段分为两个部分
//第一部分：声明变量var a ;
//第二部分：a = "hello";

```

其实第一部分发生在环境的建立阶段，第二部分发生在环境的执行阶段

```

sayHello();
//这个时候代码在这里会报错，这个时候的sayHello还是一个undefined

//这是另一种定义方法的方式
var sayHello = function2
    console.log("hello world");
}
/*
建立阶段：
定义变量sayHello，默认值是undefined

```

执行阶段：

1. 调用执行sayHello() 报错
2. 对sayHello赋值为一个 function

*/

```
var sayHello = function(){  
    console.log("第一个");  
}  
function sayHello (){  
    console.log("第二个");  
}  
  
sayHello();
```

/*

建立阶段：

1. function sayHello()一气呵成
2. 扫描var关键字，找到变量sayHello，上报内存，结果内存当中已经有了，直接忽略上报

执行阶段：

1. 将第一个函数function赋值给了sayHello
2. 执行调用sayHello()，打印结果 “第一个”

*/

- 初始化作用域
- 初步确定上下文中this的指向对象

代码在建立阶段，你是看不到代码的运行，因为阶段发生在你写代码的时候，你代码写好了，建立阶段也好了

2. 第二个阶段-执行阶段

- 执行阶段就是一行一行的执行代码
- 最终确定上下文中的this指向对象

下面来看个具体的示例

```
function foo(i) {
    var a = 'hello';
    var b = function() {
        };
    function c() {
        }
}
foo(22);
```

如果要判断一段代码的执行顺序，就请根据代码的建立阶段与执行阶段去判断

作用域

首先，提到作用域，要了解一个常识

JavaScript中没有块级作用域，只有函数作用域和全局作用域。

什么是块级作用域？块级作用域就是定义在{}之内的范围，比如if(){}
或者for(){}
里那个{}里的范围就叫做块级作用域。

如下代码所示

```
console.log(userName); //undefined
if(0 < 2)
{
    var userName = "标哥";
}
console.log(userName); //标哥
/*
```

1. 没有块级作用域，所以 if{}形成不了作用域

2. 代码在在建立阶段，创建好了变量userName，并且是在全局创建的，默认值是undefined

3. 执行阶段

第一次打印userName，因为还没有赋值，所以应该是undefined

后面执行了userName = "标哥"；

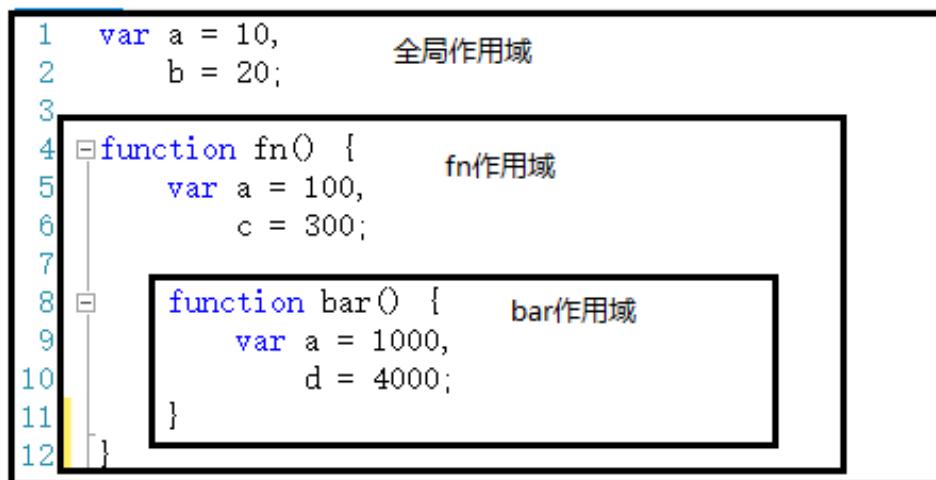
第二次打印userName，因为变量已经赋值了，所以应该是"标哥"

*/

```
for(var i=0;i<9;i++)
{
}
alert(i); //9
```

所以，在声明变量时，我们要做到，全局变量要在代码前端声明，函数中的变量要在函数体一开始的地方就声明好。除了这两个地方，其他地方都不要出现变量声明。这样就防止了在调用这个变量之前还没有赋值

下面来说作用域，简单的说，作用域相当于一个区域，就是为了说明这个区域有多大，而不管这个区域里有什么东西。这个区域里有什么东西恰恰就是这个作用域所对应的执行上下文所要说明的东西。如下图所示



作用域有上下级的关系，上下级关系的确定就看函数是在哪个作用域下创建的。例如，`fn`作用域下创建了`bar`函数，那么“`fn`作用域”就是“`bar`作用域”的上级。(也可以说父子关系)

我们知道，作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

另外，我们还需牢记于心的是，作用域是在函数创建的时候就已经确定了，而不是函数调用的时候。

函数创建和函数调用可是两个不同的概念，函数创建就是定义一个函数，函数调用是在某处调用一个已经定义好的函数。下面举一个例子来说明

```
function foo() { //函数创建，定义了foo函数，而foo函数的作用域也在此时确定了
  alert("buddy!");
}
alert ("hey!");
foo(); //调用函数，这里调用foo函数
```

自由变量与作用域链

概念：凡是跨了自己的作用域的变量都叫自由变量

```
var x=10;
function foo(){
    var b=20;
    console.log(x + b); //这里x就是自由变量
}
foo(); //30
```

说明：在调用foo()函数时，函数体中第3行。取b的值就直接可以在foo作用域中取，因为b就是在这里定义的。而取x的值时，就需要到另一个作用域中取。到哪个作用域中取呢？

这时你可能就说，到上一级作用域去取值啊，（到上一级作用域里取值就涉及到了作用域链的概念），但是这种说法是有歧义的，下面的例子就说明了这个歧义（为什么说是歧义呢？）

```
var age = 22;
function a(){
    console.log(age);
}
function b(fn){
    var age = 11;
    fn(); //这个fn就是a
/*
    建立：
    1.建立fn参数
    2.建立变量age
执行：
    1.执行age=11
    2.执行fn()
*/
}
b(a);
```

说明：按照一般思路来讲，到上一级作用域取值，结果应该是11，为什么结果是22，而不是11呢？这就是这个歧义，不是说到上一级作用域取值有问题，而是到哪一个上一级作用域取值？

注意了，在上一章的作用域的笔记当中，我们提到过很重要的作用域的特点 作用域是在函数创建的时候就已经确定了，而不是函数调用的时候。

所以，在上面的例子中函数a()的上一级作用域是全局作用域，而不是函数b()的作用域，故到上一级作用域取到的变量age的值是22，而非11。

所以，我们要铭记：要到创建函数a()的那个作用域中取——而不管函数a()是在哪里调用。

标识符的查找

上面描述的只是上一级作用域就找到了变量，但是如果上一级作用域没有找到了，那就再上一级，如果再没有，就再上一级，直到全局作用域，如果全局作用域没有，那就真的没有了，并且，每上一级作用域取值都是到创建这个函数的作用域去取值。（标识符的查找）

所以我们可以看到，作用域链是因为自由变量才存在的，也是因为自由变量，作用域才有意义。

作用域与执行上下文

在上图的代码当中，我们看一下代码的执行顺序

```
1  var a = 10,
2      d = 20;
3
4  全局作用域
5
6
7  function fn(x) {
8      var a = 100,
9          c = 300;
10
11
12  fn作用域
13
14  function bar(x) {
15      var a = 1000,
16          d = 4000;
17  }
18
19
20
21  bar(100);
22  bar(200);
23
24
25  }
26
27  fn(10);
```

全局上下文环境	
a	10
d	20
其他

第一步，在加载程序时，已经确定了全局上下文环境，并随着程序的执行而对变量就行赋值，

在这里提一句，函数的执行上下文是在调用函数后建立的，执行上下文的第一阶段（建立阶段）是在调用函数后，执行具体代码前进行，执行上下文的第二阶段（执行阶段）是执行具体代码时进行的。

第二步，程序执行到第27行，调用fn(10)，此时会生成此次调用fn函数时的上下文环境，压栈，并将此上下文环境设置为活动状态

The diagram illustrates the execution flow and the creation of context environments across three nested scopes:

- Global Scope (Line 1-6):** Contains declarations for variables `a` and `d`.

全局上下文环境	
a	10
d	20
其他
- fn Scope (Line 7-12):** Contains a function `fn(x)` which declares variables `x`, `a`, and `c`.

fn(10) 上下文环境	
x	10
a	100
c	300
其他
- bar Scope (Line 14-18):** Nested within the fn scope, it contains a function `bar(x)` which declares variables `x`, `a`, and `d`.

bar作用域	
x	100
a	1000
d	4000
其他

Execution flow: The program starts at line 1. At line 27, it reaches the call `fn(10)`, which creates the `fn(10) 上下文环境` (context environment for `fn(10)`) and pushes it onto the stack, setting it as the active context.

1	var	a	=	10,
2		d	=	20;
3				
4				全局作用域
5				
6				
7	function	fn	(x)	{
8	var	a	=	100,
9	c	=	300;	
10				
11				fn作用域
12				
13	function	bar	(x)	{
14	var	a	=	1000,
15	d	=	4000;	
16	}			
17				
18				bar作用域
19				
20				
21				
22				
23	bar	(100);		
24	bar	(200);		
25	}			
26				
27	fn	(10);		

第三步，执行到第23行时，调用bar(100)，生成此次调用的上下文环境，压栈，并设置为活动状态

```
1 var a = 10,
2     d = 20;
3
4 全局作用域
5
6
7 function fn(x) {
8     var a = 100,
9         c = 300;
10
11 fn作用域
12
13 function bar(x) {
14     var a = 1000,
15         d = 4000;
16 }
17
18 bar作用域
19
20
21 bar(100);
22 bar(200);
23
24 }
25
26
27 fn(10);
```

全局上下文环境

a	10
d	20
其他

fn(10)上下文环境

x	10
a	100
c	300
其他

bar(100)上下文环境

x	100
a	1000
d	4000
其他

第四步，执行完第23行，`bar(100)`调用完成。则`bar(100)`上下文环境被销毁。接着执行第24行，调用`bar(200)`，则又生成`bar(200)`的上下文环境，压栈，设置为活动状态。

```
1 var a = 10,
2     d = 20;
3
4 全局作用域
5
6
7 function fn(x) {
8     var a = 100,
9         c = 300;
10
11 fn作用域
12
13 function bar(x) {
14     var a = 1000,
15         d = 4000;
16 }
17
18 bar作用域
19
20
21 bar(100);
22 bar(200);
23
24 }
25
26
27 fn(10);
```

全局上下文环境

a	10
d	20
其他

fn(10)上下文环境

x	10
a	100
c	300
其他

bar(200)上下文环境

x	200
a	1000
d	4000
其他

第五步，执行完第24行，则bar(200)调用结束，其上下文环境被销毁。此时会回到fn(10)上下文环境，变为活动状态

```
1 var a = 10,
2     d = 20;
3
4 全局作用域
5
6
7 function fn(x) {
8     var a = 100,
9         c = 300;
10
11 fn作用域
12
13
14     function bar(x) {
15         var a = 1000,
16             d = 4000;
17     }
18
19
20
21 bar作用域
22
23 bar(100);
24 bar(200);
25
26
27 fn(10);
```

全局上下文环境

	全局上下文环境
a	10
d	20
其他

fn(10)上下文环境

	fn(10)上下文环境
x	10
a	100
c	300
其他

第六步，执行完第27行代码，fn(10)执行完成之后，fn(10)上下文环境被销毁，全局上下文环境又回到活动状态

```

1  var a = 10,
2      d = 20;
3
4
5  全局作用域
6
7  function fn(x) {
8      var a = 100,
9          c = 300;
10
11
12  fn作用域
13
14  function bar(x) {
15      var a = 1000,
16          d = 4000;
17  }
18
19
20
21  bar作用域
22
23  bar(100);
24  bar(200);
25
26
27  fn(10);

```

全局上下文环境	
a	10
d	20
其他

所有过程到此结束。

我们可以看出，作用域只是一个“区域”，一个抽象的概念，其中没有变量。要通过作用域对应的执行上下文环境来获取变量的值。同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。

所以，如果要查找一个作用域下某个变量的值，就需要找到这个作用域对应的执行上下文环境，再在其中寻找变量的值。

案例

现在下面有一段代码，请分析一下代码的执行结果

```

function a(){
    var age=21;
    var height=178;
    var weight=70;
    function b(){
        console.log(age);
        console.log(height);
    }
}

```

```
var age=25;
height=180;
console.log(age);
console.log(height);
}
b();
}
a();
```

目前阶段有两个点一定要注意

1. 所有的环境代码应该都有两个阶段，分别是代码的建立阶段，与代码的执行阶段
2. 建立阶段的运行我们是看不到的，它主要干两件事情
 - 第一件事情就是把所有的实参都接收到，把所有的方法找到，建立好，然后把所有的变量找出来，定义好，默认为undefined
参数先于函数，先于变量
 - 第二件事情就是建立当前环境的作用域

变量会随着执行环境而建立吗

是不是所有的变量都是在代码的建立阶段进行的？答案不是的！有一种变量很特殊，它没有经过var去定义

```
//这一段代码它没有var，所以没有建立阶段
userName = "张三";
console.log(userName);
```

关于变量提权

变量提权使用的是定义变量的时候，不使用关键var，这个时候，当代码执行到某一个环境的时候，它这个变量变没有建立阶段，那么在这个个执行环境里，就不会建立这个变量

而进入到代码的执行过程以后，这个时候再去建立刚刚的变量，它会在全局状态下面去建立

```
function a(){
  userName = "张三";
}
a();
console.log(userName);
```

思考：下面的代码的运行结果是什么

```
function b(){
    function c(){
        userName = "张三";
    }
    c();
    console.log(userName);
}
b();
console.log(userName);
```

问：什么是执行上下文？

答：执行上下文就是代码的执行环境，代码会在这个环境当中按照规范进行与运行，所以的环境都经过2个阶段，分别是建立阶段与执行阶段

问：建立阶段干了什么

答：建立阶段主要干了3个事情

1. 上报内存
2. 确定作用域
3. 初步确定this的值

问：上报内存的顺序

答：它有三个顺序，先建立arguments, 再进行function的上报，最后是var的变量上报

问：作用域是如何确定的

答：作用域是在函数建立的时候就已经确定了，也就是在你编写这个函数的时候作用域就已经确定了

```
var age = 22;
function a(){
    console.log(age);
}
function b(){
    var age = 11;
    a();
}
b();
```

再看下面

```
var age = 22;

function b(){
    var age = 11;
    function a(){
        console.log(age);
    }
    a();
}
b();
```

常用内置对象

之前在讲面向对象的时候都是我们自己去创建对象，其实在 **JavaScript** 的内部有很多内置对象，系统提供的内置对象是可以直接使用的，这些对象我们叫JS常用对象

Math对象

它是一个全局的数学对象，是一个系统已经定义好的对象，不需要我们通过 **new** 的关键来构造就可以直接使用，它主要用于我们的数据运算

这个数学对象里面主要包含了2个东西，一个是属性【常数】，一个是方法

常数

常数数学或自然界当中一些固定的数，如 **π** 它就是一个常数

- **Math.PI** 圆周率，约等于3.14
- **Math.E** 自然对数的底数，约等于2.718
- **Math.LN10** 10的自然对数值，约等于2.302
- **Math.LN2** 2的自然对数值，约等于0.693
- **Math.SQRT1_2** 0.5的平方根
- **Math.SQRT2** 2的平方根

上面这些东西都不用去记，用的时候拿出来看一下就行了，它是固定的不会变。

方法

方法才是数学对象的重点

1. 绝对值 `Math.abs()` 方法

```
Math.abs(3);      //3
Math.abs(-3);     //3
```

2. 四舍五入 `Math.round()` 方法

```
Math.round(3.4);    //3
Math.round(9.8);    //10
Math.round(-10.2);   //-10
Math.round(-10.8);   //-11;
Math.round(-10.5);   //-10
```

注意：上面的 `Math.round()` 可以实现整数位的四舍五入，但是如果想多保留几位小数，怎么办呢？

```
var a = 3.1415926;
//我希望保存3位小数以后再四舍五入，怎么办？
var b = Math.round(a * 1000) / 1000;
console.log(b);
```

3. `Math.floor()` 向下取整，返回这小于或等于这个数的最大整数

```
Math.floor(88.2);    //88
Math.floor(24.7);    //24
Math.floor(23);       //23
Math.floor(-12.4);   //-13
```

4. `Math.ceil()` 向上取整，返回大于或等于这个数最小整数

```
Math.ceil(11.1);    //12
Math.ceil(22.7);    //23
Math.ceil(34);       //34
Math.ceil(-34.5);   //-34
```

到目前为止，我们处理数据的方式就有很多种了，如 `parseInt()`, `Math.round()`, `Math.floor()`, `Math.ceil()`。现在我们来将这些方法做一个对比

参数	parseInt()	Math.ceil()	Math.floor()	Math.round()
10.25	10	11	10	10
10.75	10	11	10	11
10.5	10	11	10	11
-10.25	-10	-10	-11	-10
-10.75	-10	-10	-11	-11
-10.5	-10	-10	-11	-10

在负数执行 `Math.round()` 操作的时候基本可以理解为“五舍六入”就行了

5. `Math.pow(x, y)` 函数。求取x的y次方的结果

```
Math.pow(100,3);           //1000000
Math.pow(2,8);             //256
```

6. `Math.max()` 方法用于求一系列数当中的最大值

```
Math.max(111,23,120);     //120;
```

之前在讲跨对象调用的时候我们已经用过了这个方法，通过这个方法来求取了数组当中的最大值

```
var arr = [100,87,65,34,209];
var max = Math.max.apply(Math,arr);      //209
```

7. `Math.min()` 方法用于求一系列数当中最小值

```
Math.min(111,23,120);       //23

var arr = [100,87,65,34,209];
var min = Math.min.apply(Math,arr);    //34
```

8. `Math.sqrt(x)` 求x的平方根

```
Math.sqrt(9);              //3
```

9. `Math.random()` 获取0~1之间的随机数

```
Math.random();             //0.5808373583762474
```

思考：上面的方法是通过0~1之间的随机数，如果我们想获取0~99之间的随机整数怎么办呢？

```
var x = ~~(Math.random() * 100)
```

小技巧：如果想获取某一个范围的随机整数，我们可以使用下面的公式

```
var x = ~~(Math.random() * 范围值); //最后生成的随机数不包含这个范围值
```

10. **Math.sin()** 方法，求一个弧度的正弦值

要注意，这里弧度，不是角度

弧度与角度有一个计算公式，不要弄混了，这个与我们这前所学习的数学里面的角度不一样 $1^\circ = \pi/180^\circ$, $1\text{rad} = 180^\circ/\pi$ 。

度	0°	30°	45°	60°	90°	120°	135°	150°	180°	270°	360°
弧度	0	$\pi/6$	$\pi/4$	$\pi/3$	$\pi/2$	$2\pi/3$	$3\pi/4$	$5\pi/6$	π	$3\pi/2$	2π

如果我现在想求90度的正弦值，应该怎么算

```
//第一步：将90度的角度转换成弧度
var rad = 90 * Math.PI / 180;
var x = Math.sin(rad); //1
```

11. **Math.cos()** 方法，求一个弧度的余弦值

12. **Math.tan()** 方法，求一个弧度的正切值

Date对象

在JavaScript里面Date对象是用于表述日期的对象，它可以获取系统当前的日期与时间，也可以手动的设置日期与时间

Date 是一个构造函数，如果要使用我们就要 **new** 一个对象出来（这一点就与我们上面的 **Math** 对象不一样的，， **Math**已经是一个对象了，我们可以直接使用）。如果我们要使用日期对象就要先创建对象

```
var d = new Date(); //得到了一个当前日期的对象
// Thu Aug 18 2022 09:43:17 GMT+0800 (中国标准时间)
```

当我们使用构造函数去创建对象的时候，默认是以当前日期为主，如果我们需要手动的去得到一个时间的日期对象，可以在构造函数里面传递参数

```
var d1 = new Date("2022-5-1 12:12:33");
//Sun May 01 2022 12:12:33 GMT+0800 (中国标准时间)

var d2 = new Date(2022, 5, 1, 12, 33);
//Wed Jun 01 2022 12:12:33 GMT+0800 (中国标准时间)
```

这里有个坑要注意，在使用第2种方法创建Date的时候，它的月份是从0开始的

方法

1. `Date.now()` 方法，这个方法不需要 `new` 就可以使用，它返回前的时间戳，它是一串数字，这个数字代表从 `1970-1-1` 到现在的毫秒数

```
var d1 = Date.now();      //1660787467301
```

上面的这个数字代表的就是当前的时间，我们可以通过这一串数字来知道这个时间到底是什么时候，在后期学习数据库的时候如果我们要在数据库当中存储一个时间，一般都是存这样的一个数组

现在我们就通过这个数字来得到时间

```
var currentDate = new Date(1660787467301);

//1630033003696
```

2. `getFullYear()` 获取年份的方法

```
var d = new Date();
d.getFullYear();      //2022
```

3. `getMonth()` 获取月份的方法

一年有12个月，但是月份是从0开始的

```
d.getMonth();        //7
```

4. `getDate()` 获取日期的方法

```
d.getDate();        //18
```

5. `getDay()` 获取星期几

一个星期的第一天是星期天，但是星期天是0

```
d.getDay(); //4
```

6. `getHours()` 获取小时数

```
d.getHours(); //10
```

7. `getMinutes()` 获取分钟数

```
d.getMinutes(); //32
```

8. `getSeconds()` 获取秒钟

```
d.getSeconds(); //23
```

9. `getMillSeconds()` 获取毫秒数

```
d.getMillSeconds(); //747
```

上面的几个方法我们全都是通过 `get` 来获取的，与之相对应的还有 `set` 的方法，它们是赋值方法

如果我们想对某一个日期对象重新赋值，可以使用 `set` 的方式进行

```
var d = new Date(); //当前时间  
//现在想重新赋值，改变它的年份  
d.setFullYear(1921);  
//现在去改变它的月份  
d.setMonth(5); //现在把月份设置为了6月，因为月份是从0开始的，如果大于  
11则会向前进位  
//同理，日期也是一样的，天数如果大于当前月份的最大天数也就会向前进位
```

10. `toString()` 方法

日期对象也可以调用这个方法将日期对象转换成日期的字符串，但是在转换过程当中，它有以下几种情况

- `toString()`
- `toDateString()`

- `toTimeString()`

```
d.toString();
//Thu Aug 18 2022 10:41:02 GMT+0800 (中国标准时间)
```

```
d.toDateString();      //Thu Aug 18 2022
```

```
d.toTimeString();      //10:41:02 GMT+0800 (中国标准时间)
```

11. `toLocaleString()` 转换成本地时间，也就是你电脑右下角的时间

- `toLocaleString()`

- `toLocaleDateString()`

- `toLocaleTimeString()`

```
d.toLocaleString();      //2022/8/18 10:44:40
```

```
d.toLocaleDateString(); //2022/8/18
```

```
d.toLocaleTimeString(); //10:44:40
```

12. `toGMTString()` 将当前的时间以GMT字符串显示

GMT时间叫格林宁治时间，也就是0时区的时间

```
var d = new Date();
d.toGMTString();      //Thu, 18 Aug 2022 02:49:20 GMT
```

上面的12个方法都是我们处理日期对象的时候所使用的方法，有一个方法比较特殊，我要单独的去讲一下

扩展：现在假设有2个时间，我们要将2个时间进行比较，或相减，针对这一种业务场景，我们应该怎么处理呢？

- 第一种场景：如果要计算2个时间相隔多久
- 第二种场景：我想知道哪一个时间在前面，哪一个时间在后面（比较大小）

```
var d1 = new Date("2022-4-6 12:00:00");
var d2 = new Date("2022-8-18 12:00:00");
```

现在上面有2个日期对象，我们想知道两个日期对象当中有以下操作，怎么办

- 我想知道谁哪个时间在前面，哪个时间在后面
- 我想知道两个日期间隔了多少天，怎么办呢？

如果我们的日期对象需要去做上面的操作，怎么办呢？这个时间一定要注意，所以的日期操作应该全部都以时间戳去操作

如果一个日期对象想要转换成时间戳应该使用 `getTime()` 这个方法

```
d1.getTime();           //1649217600000  
d2.getTime();           //1660795200000
```

有了这个时间戳的具体数值以后，我们就可以比较大小，也可以进行相减相加操作

```
var d1Time = d1.getTime();           //时间戳  
var d2Time = d2.getTime();           //时间戳  
  
if (d1Time < d2Time) {  
    console.log("d1在d2之前");  
}  
else {  
    console.log("d1在d2之后");  
}  
  
//要计算两个时间相隔多久  
var x = d2Time - d1Time;           //这样会得到一个时间的差值，单位是毫秒  
  
//两个时间相差多少天  
  
console.log(x/1000/60/60/24);
```

这里还有一个小细节要注意，在 `Date` 对象下面有 `valueOf` 的方法，这个方法也会返回时间戳

```
d1.valueOf() === d1.getTime();
```

延时调用与循环调用

延时调用相当于一次性定时器，循环调用相当于循环定时器

延时调用

延时调用相当于一次性的定时器，定时器可以把它理解为闹钟，我现在定义了一个闹钟，这个闹钟的时间到了以后就会通知我来做一些事情

在我们以前编程的时候，我们写的代码都会立即执行

```
function sayHello(){
    console.log("你好啊")
}

sayHello(); //这个代码会立即执行
```

现在我们希望 `sayHello` 这个代码不要立即执行，应该是等一段时间以后再执行，这一种情况我们就叫延时调用

```
var timeId = setTimeout(sayHello, 5000);
// 上面的代码就设置了一个定时器
// 第一个参数：代表时间到了以后要干什么事情
// 第二个参数：代表设置延时的时间，以毫秒为单位
// 这个方法会返回定时器的编号，后期我们可以通过这个编号来取消这个定时器
```

现在我们再来看下面的代码

```
function sayHello(userName, age) {
    console.log("你好啊，我叫" + userName + "，我的年龄是：" + age);
}

var timeId = setTimeout(sayHello, 5000, "标哥哥", 18);
// 当我们的函数里面需要参数的时候，我们就把原来函数的参数写在 `setTimeout` 的后面就可以了
```

```
<script>
    function sayHello(userName, age) {
        console.log("你好啊，我叫" + userName + "，我的年龄是：" + age);
    }

    var timeId = setTimeout(sayHello, 5000, "标哥哥", 18);

    // 上面的代码就设置了一个定时器

```

通常情况下，我们在工作当中的时候，我们都会将上面的回调函数 `sayHello` 写成一个匿名函数

```
var timeId = setTimeout(function (userName, age) {
    console.log("你好啊，我叫" + userName + "，我的年龄是：" + age);
}, 5000, "标哥哥", 18);
```

取消延时调用

当一个延时调用设置好了以后，它会返回这个定时器的编号，如果我们后期想取消这个定时器，就通过这个编号来取消，只需要调用 `clearTimeout(编号)` 就可以了

```
function sayHello(userName, age) {
    console.log("你好啊, 我叫" + userName + ", 我的年龄是:" + age);
}

var timeId = setTimeout(sayHello, 5000, "标哥哥", 18);
//clearTimeout(timeId); //定时器就取消了
```

循环调用

循环调用就是循环定时器，它会每隔一段时间就会自己执行一次（可以理解为手机上面的一个起床闹钟，它每天早上7:30起床，隔24小时执行一次）

它的语法与上面的延时调用的语一致

```
var 定时器编号 = setInterval(时间到了以后要做的事, 间隔的时间);
```

```
function sayHello() {
    console.log("你好啊, 现在时间是:" + new Date().toLocaleString());
}

var timeId = setInterval(sayHello, 1000);
```

循环定时器在设置以后，它会根据间隔时间去进行相关的自动调用操作

同理 `setInterval` 也可以接收参数

```
function sayHello(userName) {
    console.log("你好啊, 我叫" + userName + ", 现在时间是:" + new
Date().toLocaleString());
}

var timeId = setInterval(sayHello, 1000, "帅气的标哥哥");
```

取消循环调用

在设置循环定时器的时候我们会得一个编号，有了这个编号以后，我们就可以在后期取消这个循环定时器

```
 clearInterval(定时器编号);
```

正则表达式对象

等进入DOM再讲

包装对象

包装对象也是系统的内置对象

我们之前在讲数据类型的时候 我们已经学习了5种基本数据类型

- `string` 字符串类型
- `number` 数字类型
- `boolean` 布尔类型
- `null` 类型
- `undefined` 类型

所谓的包装对象其实就是为 `string/number/boolean` 基本数据类型服务的

```
var str = "hello world";
console.log(str.length);
console.log(str.toUpperCase());

var flag = true;
console.log(flag.toString());

var num = 123;
console.log(num.toString(2));
```

`str` 明白只是一个字符串的基本数据类型，为什么它可以像对象一样有属性，有方法？

`flag` 明白只是一个布尔类的基本数据类型，为什么也可以调用方法？

`num` 也是一个数字的基本数据类型，为什么它也有方法？

基本数据类型为什么会有属性和方法，而方法与属性不是对象才有的吗？

为了便于操作基本类型值，ECMAScript 还提供了 3 个特殊的引用类型：Boolean、Number 和 String。这些类型与本章介绍的其他引用类型相似，但同时也具有与各自的基本类型相应的特殊行为。实际上，每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，从而让我们能够调用一些方法来操作这些数据。来看下面的例子。

每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象

var str = "hello world"
String对象

var flag = true
Boolean对象

var num = 123
Number对象

每当我们去使用基本数据类型的时候，其它是一个对象把它包裹起来，这样的对象人们叫包装对象

```
var str = "hello world";
//其实JS的后面是这么操作
//相当于
var str = new String("hello world");

var flag = true;
//相当于
var flag = new Boolean(true);

var num = 123;
//相当于
var num = new Number(123);
```

根据上面的解释，再结合我们的面向对象的理解（一个对象的 `__proto__` 等于它构造函数的 `prototype`），我们可以得到一个结论

```
var str = "hello world";           //new String("hello world")
var flag = true;
var num = 123;

str.__proto__ === String.prototype; //true
flag.__proto__ === Boolean.prototype; //true
num.__proto__ === Number.prototype;
```

通过上面的分析，我们也就知道了所谓的包装对象就是为了让我们更方便的操作 `string/number/boolean` 的。这样我们只用掌握每个包装对象上面的方法就掌握了三个基本数据类型的操作方法

String对象

String字符串对象是我们在JavaScript最常见的一种对象，他是用于包装 `string` 基本数据类型型

我们可以将字符串理解为一个特殊的字符数组，数组具备的大部分方法字符串都有

1. `length` 属性

这个属性用于获取字符串的长度，它是一个只读的属性，不能手动的修改

2. 索引取值

之前在讲数组的时候就用过了，数组是通过索引取值的，所以我们的字符串也可以通过索引取值

```
var str = "abcdefg";
console.log(str[0]);      //a
```

3. `at()` 方法，通过索引来取字符串的某个字符

```
var str = "abcdefg";
str.at(0);          // "a"
str.at(-1);        // "g"    倒数第1个字符
```

4. `concat()` 拼接字符串，它可以将多个字符串拼接，形成一个新字符串，旧的字符串不变

```
var str = "abcdefg";
var str2 = "hijklmn";
var str3 = str.concat(str2);
```

这个用法与之前数组里面的用法是一样的，但是一般情况我也不用，因为字符串的拼接我们使用 `+` 会更好一些

5. `slice(start?, end?)` 方法，截取原字符串中的字符形成一个新字符串，原字符不变 这个方法与数组的方法也保持了一致

```
var str = "abcdefg";
str.slice(1,4);           // "bcd";
str.slice(2,-1);          // 'cdef'
str.slice(-5,-1);         // 'cdef'
str.slice(1);              // 'bcdefg'
str.slice();                // 'abcdefg'
```

6. `substring(start,end?)` 方法， 载取字符串，它的能力比 `slice` 差一些

```
var str = "abcdefg";
str.substring(1,5);        // 'bcde'
str.substring(1);          // 'bcdefg'
```

注意：

- `substring` 的开始位置不能省略，而 `slice` 是可以省略
- `substring` 不要以负数做为参数，而 `slice` 是可以的
- `substring` 与 `slice` 在截取的时候都不包含 `end` 这个元素

7. `substr(start,length?)` 截取字符串，从 `start` 的位置开始，截取长度为 `length` 的字符串

这一个方法与前面的2个方法完全不一样，这个方法的第2个参数是截取字符串的长度

```
var str = "abcdefg";
str.substr(1,5);           // 'bcdef'
str.substr(3);              // 'defg'
```

8. `indexOf/lastIndexOf` 查询匹配的字符串的位置，如果找到了就返回索引，找不到就返回 `-1`

```
var str = "暮晓春来迟;先于百花知;岁岁种桃树;开在断肠时";
str.indexOf("百");          // 8
str.indexOf("钱");          // -1
str.indexOf(";",8);         // 11
```

9. `split()` 方法，将字符串使用指定的方式隔开，变成数组

```
var str = "hello-world";
str.split("-");           //["hello", "world"];
str.split("");            //['h', 'e', 'l', 'l', 'o', '-', 'w', 'o', 'r',
'l', 'd']
```

这个方法正好与数组里面的 `join()` 方法执行相反的操作

```
var arr = ["hello", "world"];
arr.join("-");           //"hello-world";

var arr2 = ['h', 'e', 'l', 'l', 'o', '-', 'w', 'o', 'r', 'l', 'd'];
arr2.join("");           //"hello-world";
```

后期在开发工过程当中这两个方法经常会合起来一起使用

10. `charAt()` 这个方法根据一个索引来获取指定的字符，相当于通过索引取字符

```
var str = "hello-world";
str[0];                //"h"
str.at(0);              //"h"
str.charAt(0);          //"h";
//charAt不支持负数，而at支持负数
```

11. `charCodeAt()` 方法，通过索引来取字符串里面的某一个字符的 `unicode` 编码，如果是英文字符串则返回 `ascii` 码

```
var str = "我爱你";
str.charCodeAt(0);        //25105
str.charCodeAt(2);        //20320
"我" > "你"             //true 中文在比较的时候比较的是unicode码
var str2 = "abc";
str2.charCodeAt(0);       //97    这里得到的是ascii码
```

12. `String.fromCharCode()` 根据一个编码反向的得到一个字符串

```
String.fromCharCode(25105);    //"我"
String.fromCharCode(97);      //"a";
String.fromCharCode(65);      //"A";
```

13. `startsWith` 判断某个字符串是否以什么开始

```
var str = "标哥哥是一个大帅哥";
str.startsWith("标");           //true
str.startsWith("标哥");        //true
str.startsWith("桃");          //false

var url1 = "https://www.softteam.xin";
url1.startsWith("https");      //true
```

14. `endsWith()` 判断某个字符串是否以什么结束

```
var str = "标哥哥是一个大帅哥";
str.endsWith("哥");           //true
str.endsWith("帅哥");        //true;
str.endsWith("姐");          //false

var url1 = "https://www.softteam.xin";
url1.endsWith("com");        //false
```

15. `includes()` 判断某个字符串是否包含某个字符

```
var str = "标哥哥是一个大帅哥";
str.includes("帅");          //true;
```

这个操作其实在 `indexOf/lastIndexOf` 也可以实现

```
str.indexOf("帅");           //只要不是-1就说明包含
```

16. `trim/trimLeft/trimRight` 去除字符串左边的，右边的或两边的空格

- `trimLeft()` 去除左边的空格
- `trimRight()` 去除右边的空格
- `trim()` 去除左右的空格

```
var str = "    标哥是一个大帅小伙    ";
str.trimLeft();      //"标哥是一个大帅小伙    ";
str.trimRight();     //"    标哥是一个大帅小伙";
str.trim();          //"标哥是一个大帅小伙"
```

17. `replace()` 方法替换，可以将字符串中的字符进行替换，该方法不会改变原字符串，它会返回新字符串

```
var str = "我爱你";
//现在我们把“爱”换成“恨”
var str1 = str.replace("爱", "恨");           // "我恨你"

var str2 = "我爱你，你爱我吗？";
var str3 = str2.replace("爱", "恨");           // "我恨你，你爱我吗？"
```

这个时候我们会发现一个问题，`replace()` 只替换了第一次查找到的值，后面的没有替换，如果要进行全局的替换，我们需要使用正则表达式

```
var str6 = str2.replace(/爱/g, "恨");
```

18. `anchor` 锚方法，没什么用，知道就行了

```
var str = "http://www.softteam.xin:8090";

var str1 = str.anchor("bbb");
console.log(str1);      //<a
name="bbb">http://www.softteam.xin:8090</a>
```

19. `small/big` 方法

```
var str = "标哥哥";
var str3 = str2.big();           // '<big>标哥哥</big>'
var str4 = str2.small();        // '<small>标哥哥</small>'
```

20. `padStart(maxLength, fillString)` 方法，该方法将字符串以指定的长度补齐在左边

```
var a = "1";
var b = "23";
var c = "456";
a.padStart(4,0);           // "0001"    总长度为4, 不够应 左边补0
b.padStart(3,0);           // "023"     总长度为3, 不够应 左边补0
c.padStart(3,0);           // "456"     总长度为3, 够, 不用补
c.padStart(2,0);           // "456"     超过了, 不用补
```

21. `padEnd(maxLength, fillString)` 方法，该方法将字符串补齐到指定长度在右边

```
var a = "1";
var b = "23";
var c = "456";
a.padEnd(4,0);           // "1000"
b.padEnd(3,0);           // "230"
c.padEnd(3,0);           // "456"
c.padEnd(2,0);           // "45"
```

22. `repeat()` 方法，将字符串重复多少次

```
var x = "*";
var a = x.repeat(4);      // '****'
```

23. `toUpperCase() /toLowerCase()` 将英文的转换成大小写，原字符串不变，返回新字符串

```
var str = "hello world";
var str1 = str.toUpperCase();    // 'HELLO WORLD'
```

Number对象

Number的包装对象主要是针对 `number` 数据类型的，它也有一些经常使用的方法

1. `toString(进制)` 方法，将数字以转换成目标进制的字符串

```
var a = 18;
var b = a.toString(2);        // "10010"
var c = a.toString();         // "10";
var x = 255;
var y = x.toString(16);       // "FF"
```

2. `Number.isNaN()` 方法，判断一个变量是否是 `NaN`

```
console.log(Number.isNaN(NaN));          // true
console.log(Number.isNaN(1));             // false
console.log(Number.isNaN("a"));           // false
console.log(Number.isNaN(1 % 0));          // true
```

注意：在 `window` 对象下面也有一个 `isNaN` 的方法，它们2个者之间是有区别

```
Number.isNaN(value:any);
window.isNaN(value:number);
```

现在我们根据上面的2种不同的判断来看下面的结果

```
Number.isNaN("a");      //false  
window.isNaN("a");     //true
```

`window.isNaN()` 这个判断方法它只接收 `number` 类型，如果我们传入的参数不是数值，它会默认调用 `Number` 方法做一次隐式类型转换，`Number("a")` 得到的结果 `Nan`，所以最后 `window.isNaN("a")` 检测的结果就是 `true`

3. `Number.isInteger()` 判断某一个数是否是整数

```
Number.isInteger(3);      //true  
Number.isInteger(3.14)    //false  
Number.isInteger("3");   //false;
```

4. `Number.isFinite()` 判断某个数值是否是有限数值（通俗一点说就是不能是无穷大）

```
Number.isFinite(Infinity); //false;
```

5. `toFixed()` 该方法保存有效位数，执行四舍六入五成双的算法

待定

Boolean对象

`Boolean` 是专门为操作 `boolean` 来实现的包装对象，里面没有方法，只有一个 `toString()` 与 `valueOf()` 的方法。这里就省略该对象的讲解

练习

- 现有一产品，它的生产日期是 `2022-1-18`，它的过期时间是 `2022-6-30`，请计算这个产品的保质期天数是多少？

```
var d1 = new Date(2022, 0, 18);  
var d2 = new Date("2022-6-30");  
//上面两个定义日期的方法都没有问题  
var x = d2 - d1;                  //两个时间的差值  
var days = ~~(x / 1000 / 60 / 60 / 24); //转换成天数  
console.log(days);
```

2. 现有一产品，它的生产日期是 2022-8-18，它的保持期是180天，请计算它的过期的时间是什么时候？

```
var d1 = new Date(2022, 7, 18);
//算一下180天的时间戳
var x = 180 * 24 * 60 * 60 * 1000;
var d2 = d1.getTime() + x;

//将上面的时间戳转换成日期
var d3 = new Date(d2);
console.log(d3.toLocaleString());
```

3. 现有一字符串 var str = "abcdefg"，请将这个字符串反转“gfedcba”

```
var str = "abcdefg"
var str2 = str.split("").reverse().join("");
```

4. 编写一个函数，实现将一个文件的后缀名去掉

```
//如歌a.jpg变成a    123.gif变成123  如abcdefg.exe就变成abcdefg, 如
abc.exe.conf变成abc.exe
function getFileName(oldFileName) {
    return oldFileName.split(".").slice(0, -1).join(".");
}

getFileName("a.jpg");           // "a"
getFileName("123.gif");        // "123"
getFileName("abcdefg.exe");    // "abcdefg"
getFileName("abc.exe.conf");   // "abc.exe"
```

5. 请随机生成4位数的验证码，验证码包含 a-z, A-Z, 0~9

```
//通过String.fromCharCode()方法来完成
```

6. 产生一个20~100(包含100)之间的随机数10个，把它放在一个数组里面，数组里面的随机数不重复
7. 现有一人从外地回小区，根据相关政策规定需要进行《新冠疫情医学隔离》，隔离时间是14天，它回来的日期是 2022年8月18日，请计算一下他什么时间可以解除隔离？【输出的日期格式必须是xxxx年xx月xx日】
8. 在第4题的基础上面做一次扩展，如果结束隔离的时间是星期六或星期天，则顺延到星期一

9. 已知有字符串 `get-element-by-id`,写一个 方法将其转化成 驼峰表示法
`getElementById`

```
function change(str){  
    //补齐代码  
    return  
}
```