Javascript实现格式化输出，比如输入999999999，输出为999,999,999

解题：

看到这个题目，是不是首先想到了正则表达式？（然而本人之前并不会正则表达式，然而下定决心恶补了一番…忧伤）。但是并不是任何时候使用正则都是最好的（当然可能是我正则写的不够好）。真相是，此题有3种解法：

解法一：

使用正则表达式：

var f = '99999999999'.replace(/\d{1,3}(?=(\d{3})+$)/g, '$&,');  
console.log(f);  
//99,999,999,999

该正则解释：

首先，需要将正则分成2部分来看，/\d{1,3} 与 (?=(\d{3})+$)/g, '$&,') ；

(?=) 断言组：检测内容的是否满足匹配，但不实际进行匹配，所以这只是一个条件。

(?=exp) 　 (?=exp)也叫**零宽度正预测先行断言**　　　 匹配exp前面的位置

那么，分开来看：/\d{1,3}/g，而该断言组只是是条件，断言组的意思是：\d{3}+是一个整体，意思也很明显，即3n（n>=1）个数字，然后在此基础上$，即，从后面数起的3n（n>=1）个数字。

给个栗子：” *I'm singing while you're dancing.”* *.match(/\b\w+(?=ing\b)/g)*

这样会匹配到sing和danc

那么，这里匹配的是一个断字符开头的单词【\b\w+】，并且这个单词是以ing和断字符结尾的【img\b】，而且只匹配这个单词ing以前的字符【(?=)】，而且会匹配全部选项【g】。

现在回来看我们的答案：

匹配3/2/1个数字，且这些数字后面的接着的3n（n>=1）个数字且这些数字是从后面开始数起的【$】。

好了，这个正则就是这么解释的。（所以一旦加上小数点这个正则就会出错）

另外解释replace中的第二个参数：

$1(2,3,4…………)，指的是与 regexp 中的第 1 到第 99 个子表达式相匹配的文本。，请注意理解子表达式，子表达式可能是子正则表达式，也可 以是分组表达式。如\w:,\w+，这是俩。

$&所有匹配的字串。Doe, John

$`匹配的字串的左侧的文本 Doe, 和空格

$^匹配的字串右侧的文本 , John 和空格

name = "Doe, John";

name.replace(/(\w+)\s\*, \s\*(\w+)/, "$2 $1");// John Doe



相关资料：

<http://www.w3school.com.cn/jsref/jsref_replace.asp>

https://developer.mozilla.org/zhCN/docs/Web/JavaScript/Reference/Global\_Objects/String/replace#Specifying\_a\_string\_as\_a\_parameter

<http://c.biancheng.net/cpp/html/1414.html>

(?=exp)也叫**零宽度正预测先行断言**，它断言自身出现的位置的后面能匹配表达式exp。比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)，如查找*I'm singing while you're dancing.*时，它会匹配sing和danc。

(?<=exp)也叫**零宽度正回顾后发断言**，它断言自身出现的位置的前面能匹配表达式exp。比如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找*reading a book*时，它匹配ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\d)\d{3})\*\b，用它对*1234567890*进行查找时结果是234567890。

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)。

解法二：

看到这么多的正则你是不是觉得这个题很难很恶心？那臣妾就是不会正则怎么办，那就没办法了吗？其实不然，字符串拼接也是可以写的！

思路是：每隔三位数字加一个逗号，就是要隔开三个数字。

ok，那么回想一下取一个数字的十位上的数字用什么办法，就是number%10！

那每取三位数不就是不断的去掉千位一下的数字么，所以，

function geshihua(number){

if(typeof number !== "number") {

alert("please enter number");

} else {

var arr = [];

while( number > 1 ){

arr.unshift(number%1000);

number = Number.parseInt(number / 1000);//js中所有数字都是浮点数，8个字节

console.log(number)

}

return arr.join(",");

}

}

这个是昱哥写的方法……

n = 123456789

a = []

while (n > 0) {

a.push(n % 1000)

n = n / 1000 | 0 //通过位运算符做取整操作 n >> 0 也可以做取整

}

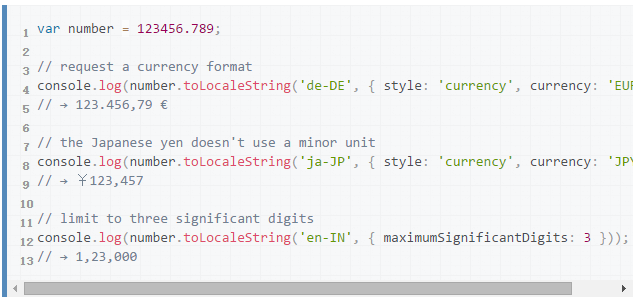
1. reverse().join(',')

方法三

好吧，其实用那个方法还是很难，如果你看到了这里，太难了……这里有一种取巧的方法：

一句话就完事噢！！！

numberObject.toLocalString('en-US')

具体的资料：

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toLocaleString>

正则表达式学习总结：

正则创建方法：

1、新建对象new RegExp();

2、正字字面量/ …… /

<http://www.nowamagic.net/librarys/veda/detail/1283>

**javascript中的正则表达式**

其实上面已经在开始讲了javascript对正则表达式的实现方式了，只定义了正则表达式，但是如何在javascript中真正使用正则表达式呢？在javascript中RegExp和String对象都有处理正则表达式的方法。

* **test** -- RegExp的test方法用来测试字符串是否匹配给出的匹配模式，返回布尔值；
* **exec** -- RegExp的exec方法返回包含第一个匹配的的数组或null；
* **match** -- String的match方法返回包含所有匹配子字符串的数组；
* **replace** -- String的replace方法完成string的替换操作，支持正则表达式；
* **search** -- 与String的indexof方法类似，不同的是search支持正则表达式，而不仅仅是字符串；
* **split** -- 按照一定规则拆分字符串并将子字符串存储到数组中的String方法。

关于这些函数的具体使用方法，可以参阅JS的相关函数手册。

一个实例对象除了方法当然还有属性，一个正则表达式有以下属性：

* **global** -- 布尔值，若全局选项g已设置则返回true，否则返回false；
* **ignoreCase** -- 布尔值，若忽略大小写选项i已设置则返回true，否则返回false；
* **lastIndex** -- 整数，使用exec或test方法时被填入，表示下次匹配将会从哪个字符位置开始；
* **multiline** -- 布尔值，表示多行模式选项m是否设置，若设置则返回true，否则返回false；
* **source** -- 正则表达式的元字符串形式。/\\/的source将返回”\\“。

#### 元字符

在正则表达式中有一些特殊的字符符号我们是不能直接使用的，必须对其进行转义后才能使用。如“\”，因为这些字符在正则表达式中有特殊的语法含义，这类字符被称为元字符，正则表达式中的元字符有：

|  |  |
| --- | --- |
| 1 | .  \ /  \* ? +  [ ( ) ] { }  ^ $ | |

可能不太好记忆，当无法确定某个字符是否是元字符的时候就勇敢的对其进行转义是没有错的，对不是元字符的字符进行转义是不会出什么问题的，但是如果不对元字符转义就会有意想不到的错误产生了。

#### 范围匹配 []

#### 取非匹配 [^]

#### 特殊字符

常见的特殊字符有：

* \d　任何一个数字字符，等价于[0-9]
* \D　任何一个非数字字符，等价于[^0-9]
* \w　任何一个字母数字或下划线字符，等价于[a-zA-Z\_]
* \W　任何一个非字母数字和下划线字符，等价于[^a-zA-Z\_]
* \s　任何一个空白字符，包括换页符、换行符、回车符、制表符和垂直制表符，等价于[\f\n\r\t\v]
* \S　任何一个非空白字符，等价于[^\f\n\r\t\v]
* .　换行和回车以外的任何单个字符，等价于[^\n\r]

**贪婪算法**

“+”表示字符出现一次或多次，至少出现一次。这个正则表达式其实并不能匹配所有合法的email地址，后面我们继续完善。

除了“+”可以指定至少匹配一次外，还有很多其他的可以指定匹配次数的方式。

* ?　出现零次或一次，最多一次
* \*　出现任意次（零次、一次、多次）
* +　出现一次或多次，至少一次
* {n}　能且只能出现n次
* {n,m}　至少出现n次，最多出现m次

#### 非贪婪算法 ？

* \* –> \*?
* + –> +?
* {n,} –> {n,}?

#### 位置匹配

* ^　用来匹配字符串开头
* $　用来匹配字符串结尾
* 又如经常被扩展的string方法trim：

|  |  |
| --- | --- |
| 1 | function trim(s){ |
| 2 | return s.replace(/(^\s\*)|(\s\*$)/g,""); |

|  |  |
| --- | --- |
| 3 | } |