# 一、[JavaScript](file:///C:\jump\super-jump\word%3fword=JavaScript)

## 原始值和引用值类型及区别

### 1.原始值（简单数据类型）：

存储在栈中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

包含五种简单数据类型：undefined、null、boolean、number、string、symbol、bigInt ；可以通过typeof 运算符来判断一个值是否在某种类型的范围内，如果它是原始类型，还可以判断它表示哪种原始类型。

变量的值存在栈中，传递变量时进行栈中内容的复制

#### （1）undefined

  undefined 的字面意思就是未定义的值，这个值的语义是，希望表示一个变量最原始的状态，而非人为操作的结果

**出现场景：**

（1）声明变量后未赋值

（2）访问对象上不存在的属性

（3）函数定义了形参，但没有传参

（4）使用void对表达式求值

#### （2）null

字面意思是空值 ，这个值的语义是，希望表示 一个对象被人为的重置为空对象，而非一个变量最原始的状态 。 在内存里的表示就是，栈中的变量没有指向堆中的内存对象. 如果需要释放某个对象，就将变量设置为null，即表示该对象已经被清空，目前无效状态。

#### （3）boolean

对于new Number(x)，其中的x:



注意：new Boolean()返回的是一个对象，对象的boolean值为true，因此不要放于if()中

#### （4）String

new String(x)

当x数据类型为：

1）数字：返回对应字符串

2）数组：用,拼接的字符串

3）对象：遇到对象后，变为字符串[object Object]

为什么基本数据类型的字符串可以调用方法？调用的时候，会自动创建String对象，用完方法后销毁。

#### （5）number

数字最大值：Number.MAX\_VALUE

new Number(x)

对于x：

1）null：0

2）true/false：1/0

3）字符串数字：对应数字

4）其他：NaN

#### （6）Symbol

每次创建的Symbol对象具有唯一性，因此用于做标记；

#### （7）bigInt

范围更大的整数，防止不精确

使用：

1）后面加n：13343535353n

2）使用函数：BigInt("23423423423")（内部是字符串）

### 2.引用值（复杂数据类型）：

存储在堆中的对象，放在变量的栈空间中的值是该对象存储在堆中的地址，也就是说，存储在变量处的值是一个指针（内存地址），指向存储对象的堆内存中。

包含：Object、function、array、RegExp、Data等。

将引用变量（内存地址）存储在栈中，它指向堆中的实际对象，传递变量时复制的是栈中的引用变量。

#### （1）object

new Object(x)

当x数据类型为：

1）对象类型（object\array)：返回本身

2）拥有对象方法的基本数据类型（string/number/boolean)：返回其对应对象

3）忽略，或者null、undefined：返回空对象

#### （2）RegExp

1）使用方法

reg = new RegExp('...'，'gi')

reg = /...../gi

其中：

i：忽略大小写

g：全文查找（重复调用会向后查找）

m：多行查找

2）匹配符

用于指定可以匹配的内容，用[]包裹；

特殊字符：

\d、\D：匹配任意数字、非数字

\w、\W：匹配任意单词、非单词

\s、\S：匹配空空格/制表符/换行符、非...

3）限定符

用于指定通配符的次数

{n,m}：至少n次，至多m次

{n}：指定n次

\*：0到多

+：1到多

?：0或1

4）定位符

用于规定查找位置

^：匹配输入字符串的开始位置（以\*\*\*开始）

$ ：匹配输入字符串的结束位置（以\*\*\*结束）

\b ：匹配一个单词边界（字符串开头、结尾、空格、逗号、点号等符号）

\B ：匹配非单词边界

5）转义符

遇到（） [ ] \* + ? . / \ ^ $等，需要用\进行转义

6）选择匹配符

选择多个组合方式

/hello(php|java)/gi

选择hello后是php或java的字符串

### 3.判断类型

**判断数据类型typeof、instanceof、Object.prototype.toString.call()、constructor**

#### 1.typeof

主要用于判断基本数据类型,返回字符串

Null代表空指针对象，由于 null 的所有机器码均为0，因此直接被当做了对象来看待，返回object

返回object的有：除Function/class外的引用类型、null

返回本身类型：除null外的基本数据类型（包括bigint/symbol)

返回number：NaN、Infinity

返回function：functino、class（指class构造函数，其实例是对象）

#### 2.instanceof

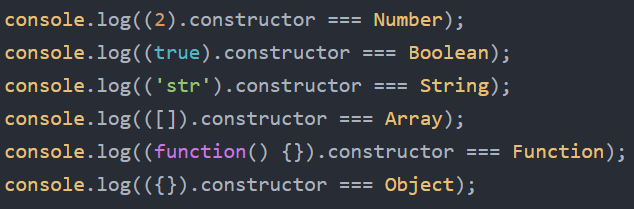
object instanceof constructor 用来测试一个对象object在其原型链中是否存在一个构造函数constructor的 prototype 属性

（1）基本数据类型number、string、boolean、symbol等不是对象，因此instanceof返回false

（2）基本数据类型实例化后可以返回true, new Number(3) new String('str')，但是symbol没有new Symbol,因此无法实例化

（3）null、undefined无法作为构造器constructor，用instanceof会报错

#### 3.constructor



constructor可以直接定位到元素的构造器上（RegExp、Array、Function等），但是构造器是可以被改变的。

另外null、undefined没有构造器，会报错

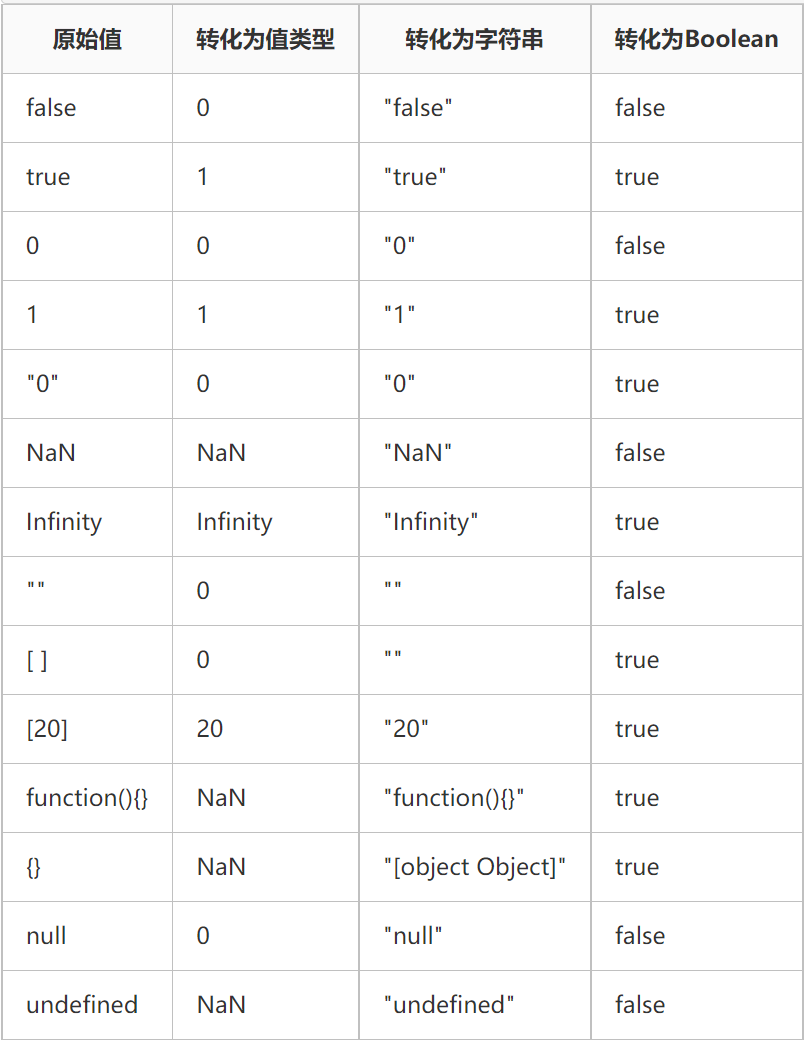
#### 4.Object.prototype.toString.call(obj)

（1）所有类都继承于Object，Object的原型拥有toString方法，它会返回[object 值类型] (Object.prototype.toString)

（2）很多类都重写了toString方法,因此我们使用call，令Object原型中toString方法的this指向obj，得到的就是obj的类型。（+call()）

（3）Object.prototype.toString遇到null、undefined会直接返回null、undefined

### 4.类型隐式转换



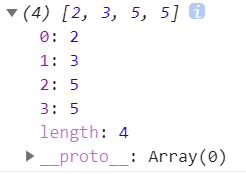
## 运算符

&&：用于数值运算时（不是条件运算），结果为true，返回第二个true对应值。结果为false，返回第一个false对应值。

||：用于数值运算时，结果为true，返回第一个true对应的值，结果为false，返回第二个false对应的值

## 类数组与数组的区别与转换

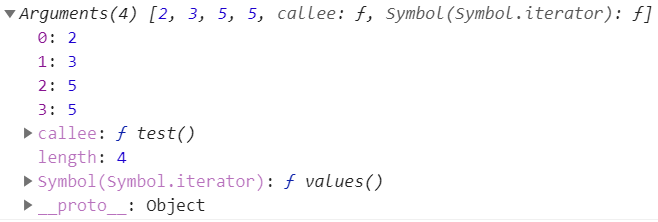
### 1.数组：



对象类型为Array

遍历时可以for in 、可以for

### 2.类数组：



（1）典型类数组比如：arguments对象、DOM方法的返回结果、jQuery对象

（2）类数组类型：Object

（3）类数组构成：字符串形式的索引+值：{'0':1, '1':2, '2':3}

（4）属性：length属性，callee属性。其他的没有。遍历时只能用for，不能for in

callee属性是一个指针，指向拥有这个arguments对象的函数test

可以通过索引访问值 arg[2]

### 3.数组转类数组：

**（1）赋值**

创建一个空对象obj，依次赋值。令obj[i]=str[i].

**（2）push、apply**，

利用apply参数是数组arr的特点，让空对象使用push，并传参arr。（push还会指定length）

[ ].push.apply(obj, arr);

### 4.类数组转数组

**（1）slice**

Array的slice方法不传参时会返回原数组的拷贝

var arr = Array.prototype.slice.call(arguments, 忽略参数)

**（2）ES6 from**

用于将类数组转换为数组，有length属性的对象都可以用此方法

var arr = Array.from(arguments)

**（3）扩展运算符**

var args = [...arguments]

**（4）jQuery $.makeArray()**

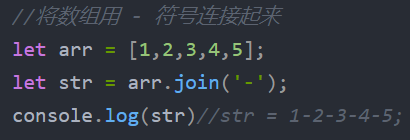
var arr = $.makeArray(arguments)

## 数组的常见API

### 1. join (原数组不受影响)

​ 将数组里的元素,通过指定的分隔符,以字符串的形式连接起来。

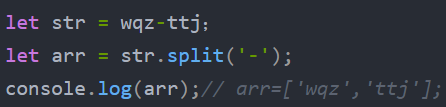
返回值:返回一个新的字符串



### 2. split (原数组不受影响)

​ 用指定的分隔符,将字符串分割成数组。

返回值:返回一个新的数组



### 3. push(改变原数组）

该方法在数组末尾添加一个或多个元素

arr.push(val)

返回值：数组新长度

### 4. pop(改变原数组）

该方法删除数组最后一个元素

arr.pop()

返回值：返回删除的元素

### 5. unshift(改变原数组）

该方法在数组最前端添加一个或者多个元素

arr.unshift(val)

返回值：返回数组新长度

### 6. shift(改变原数组）

该方法删除数组最前端的一个元素

arr.shift()

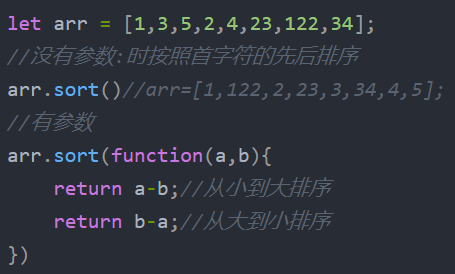
返回值：返回删除的元素

### 7. reverse(改变原数组）

arr.reverse()

### 8. sort (改变原数组）

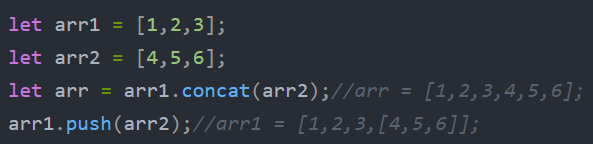
该方法可以对数组进行排序.



### 9. concat(不改变原数组）

该方法可以把两个数组里的元素拼接成一个新的数组

返回值: 返回拼接后的新数组

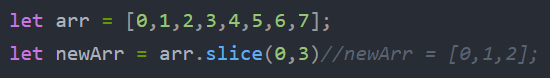


### 10. slice(不改变原数组)

该方法可以从数组中截取指定的字段,返回出来

返回值:返回截取出来的字段,放到新的数组中

结构1:arr.slice(start,end) ;从start下标开始截取,一直到end结束,不包括end



结构2:arr.slice(start) ;从start下标开始截取,一直到最后

结构3:arr.slice( ) ;全部截取

### 11. splice(改变原数组，没有返回值)

结构1: arr.splice(start,n,item1,item2, ...)

从start下标开始,删除n个，用之后的内容替换

### 12. indexOf

该方法用来查找元素在数组中第一次出现的位置，成功返回下标，失败返回-1。

结构: arr.indexOf(元素，开始位置）

### 13. lastIndexOf

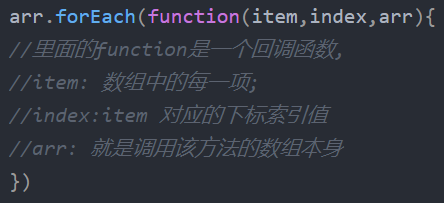
该方法用来查找元素最后一次在数组中出现的位置

### 14. 数组遍历

#### （1）forEach()

该方法等同于for循环,没有返回值，可迭代数据类型

用法:



#### （2）for in

遍历索引

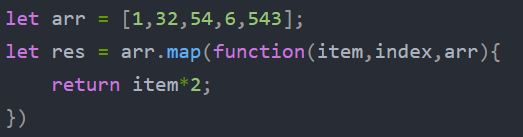
#### （3）for of

遍历值，需要是迭代器对象。（Object实例不行，没有迭代器）

#### （4）map()

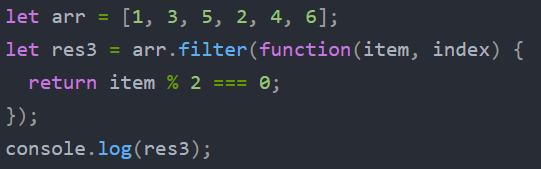
映射,该方法使用和forEach大致相同,但是该方法有返回值,返回一个新数组,新数组的长度和原数组长度相等

//里面的function是一个回调函数



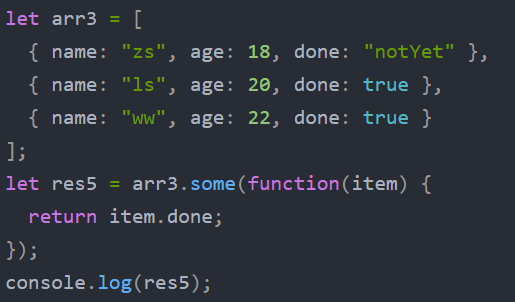
#### （5）filter()

filter方法: 有返回值, 过滤出符合条件的元素,返回数组



#### （6）some()

判断数组中有没有符合条件的项，返回布尔值(只要有,就返回true),如果一个都没有,才返回false



#### （7）every()

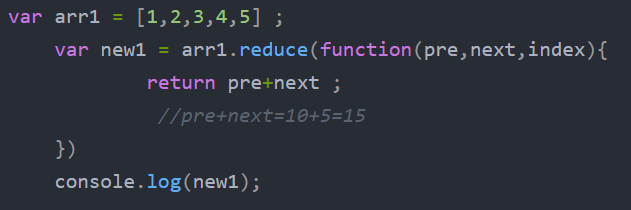
判断数组中所有的项是否满足要求,返回布尔值。如果全都满足,才返回true,否则返回false

#### （8）reduce

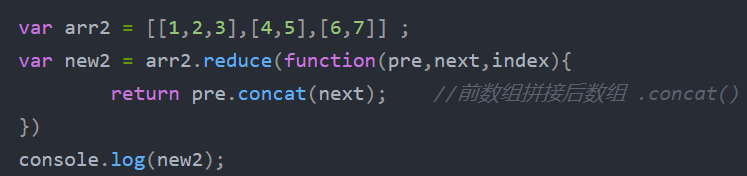
​reduce会将跟踪每次的执行位置



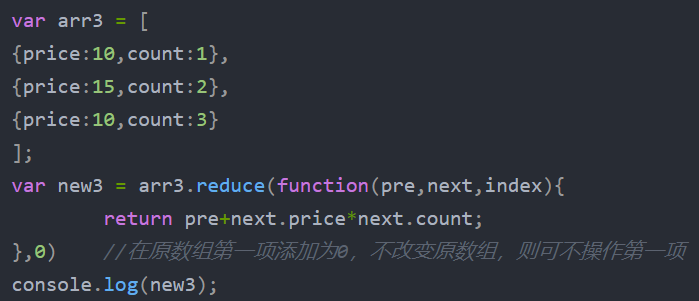
**1）求和计算**



**2）拼接数组**



**3）对象数组叠加**



#### （9）find

寻找符合条件的第一个值，没有就返回undefined，有返回其值

****

#### （10）对象要求、中断

任意对象：for in，可以break中断

要求迭代器对象：除for in 外所有，不可以break中断

## bind、call、apply区别

### 1.bind

语法：let f=fn.bind(obj,arr[0],arr[1])

使用bind时不执行fn，只进行绑定，使用时才执行fn

### 2.call

语法：push.apply(obj,arr[0],arr[1],arr[2])

obj使用push方法，其参数为arr[0]、arr[1]...l

临时改变this指向，直接执行

### 3.apply

语法：push.apply(obj,arr)

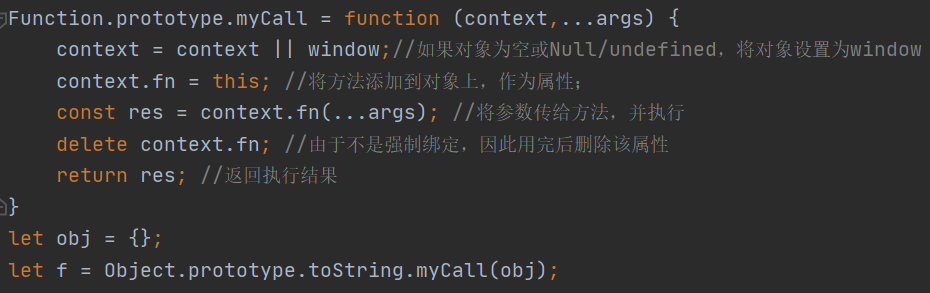
obj使用push方法，其参数是args: obj.push(args)

临时改变this指向，直接执行

### 4.实现

#### （1）call

给obj添加一个方法obj.fn，再执行它，之后删除



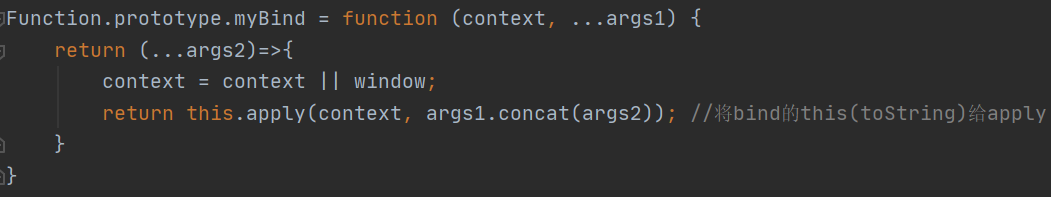
#### （2）array

同obj,但是需要对输入的数组进行处理

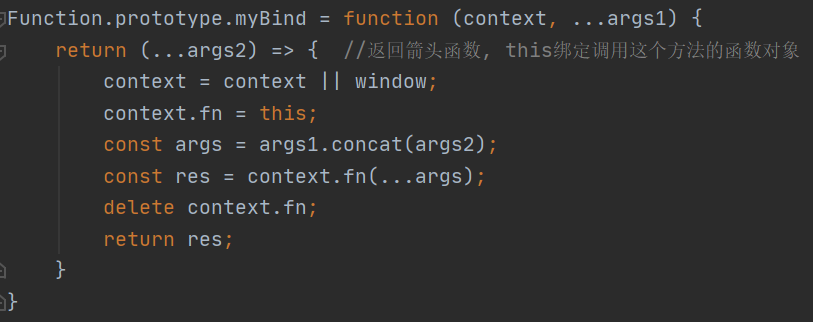


#### （3）bind

1）借用apply



2）不用apply



## new的原理

### 1.原理：

1.new关键字会首先创建一个空对象

2.将这个空对象的原型对象指向构造函数的原型属性，从而继承原型上的方法

3.将this指向这个空对象，执行构造函数中的代码，以构造其私有属性

4.如果构造函数返回了一个对象res，就将该返回值res返回，如果返回值不是对象（未返回、返回基本数据类型），就将创建的对象返回（继承基本数据类型的原型方法）。

### 2.实现

**Fn（构造函数）、args分开作为参数时：**

**function** **myNew**(Fn,...args){

*//1.创建一个空对象，并将对象的\_\_proto\_\_指向构造函数的prototype 这里我两步一起做了*

**const** obj=Object.create(Fn.prototype); //Object.create会创造一个对象并链接原型

*//2.将构造函数中的this指向obj，执行构造函数代码,获取返回值*

**const** res=Fn.apply(obj,args);

*//3.判断返回值类型*

**return** res **instanceof** Object ? res : obj

}

**args作为参数，Fn包含在内**

**function create**(args){

//定义空对象

**let** obj = {};

//取出参数列表的第一个参数（构造函数）

**let** constructor= [].shift.call(arguments);

//手动指正obj的构造函数为Con（链接原型）

obj.\_\_proto\_\_ = constructor.prototype;

//调用obj，改变this为obj，传入剩余参数arguments

**let** result = Con.apply(obj,arguments);

//考虑到obj函数中有return的原因，需要对result进行判断

**return** result instanceof Object ? result : obj

}

## 如何正确判断this？

### 1.定义：

this指向直接调用它所在作用域？的对象

### 2举例

this指向调用它的对象

//fn:()=>{console.log(this)} fn中有this

**直接调用：**

（1）obj.fn() //obj

（2）(obj.fn)() //obj

**改变this:**

call/apply/bind本质上是令obj新增了方法fn，然后obj.fn()，因此this指向obj

（1）fn.call(obj) //obj

（2）fn.bind(obj)() //obj

let f = fn.bind(obj); f(); //obj

**箭头函数：**

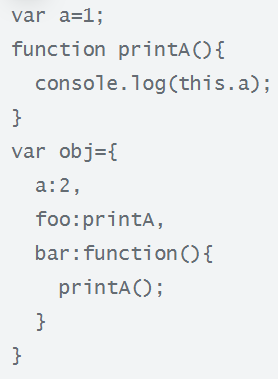
this是声明处外部的this（该this还应继续判断谁调用了它）

**没有被调用：**

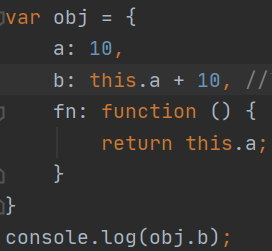
（1）fn() //window

（2）setTimeout(()=>{}) //window

（3）作用域内被直接执行（printA())，this指向window



（4）不在函数内

(this.a的作用域为obj）

（5）作为返回值调用

 //window

## 闭包及其作用

### 1.定义

有权访问另一个函数作用域中的变量的函数；

第一，闭包是一个函数，而且存在于另一个函数当中

第二，闭包可以访问到父级函数的变量，且该变量不会销毁

### 2.原理

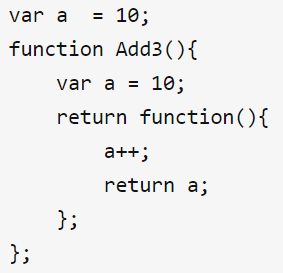
闭包的实现原理，其实是利用了作用域链的特性，我们都知道作用域链就是在当前执行环境下访问某个变量时，如果不存在就一直向外层寻找，最终寻找到最外层也就是全局作用域，这样就形成了一个链条。

### 3.作用

（1）隐藏变量，避免全局污染。

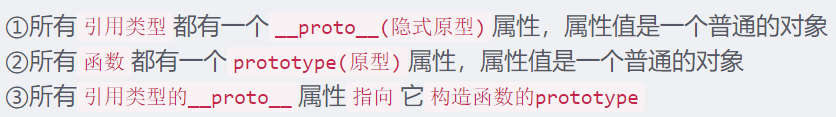
（2）读取函数内部变量。

### 4.实现



## 原型和原型链

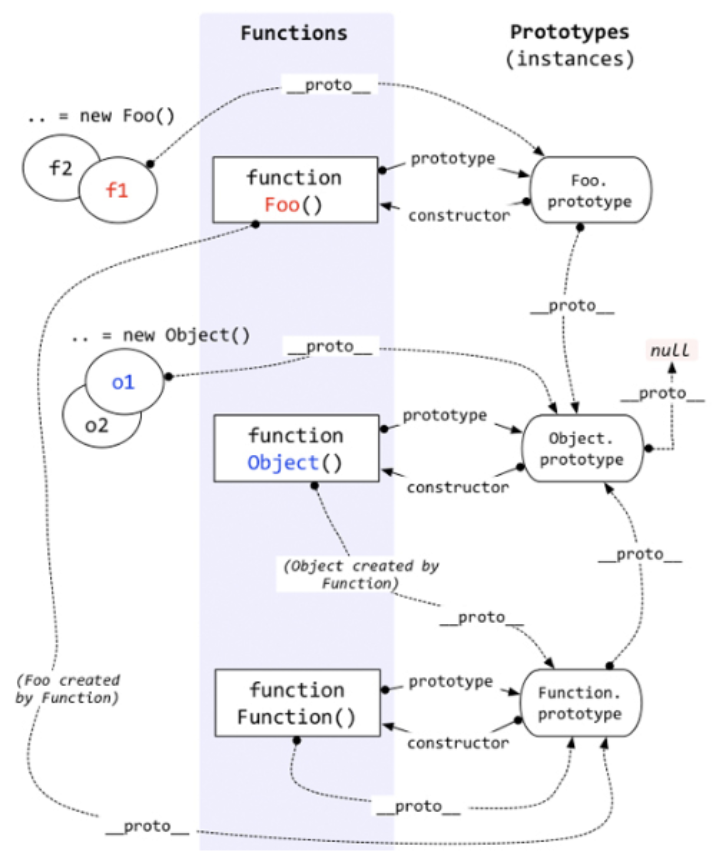
### 1.原型



原型在什么时候创建？

### 2.原型链





## prototype与\_\_proto\_\_的关系与区别

### 1.prototype显式原型

每一个函数在创建之后都会拥有一个名为prototype的属性，这个属性指向函数的原型对象。

**Note**：通过Function.prototype.bind方法构造出来的函数是个例外，它没有prototype属性。

显式原型的作用：用来实现基于原型的继承与属性的共享。

### 2.\_proto\_隐式原型

很方便地指向对象构造函数的原型。

隐式原型指向**创建**这个对象的函数(构造函数constructor)的原型prototype

**Note**: Object.prototype 这个对象是个例外，它的\_\_proto\_\_值为null

隐式原型的作用：构成原型链，同样用于实现基于原型的继承。

## 继承的实现方式及比较

### 1.原型链继承

#### （1）原理：

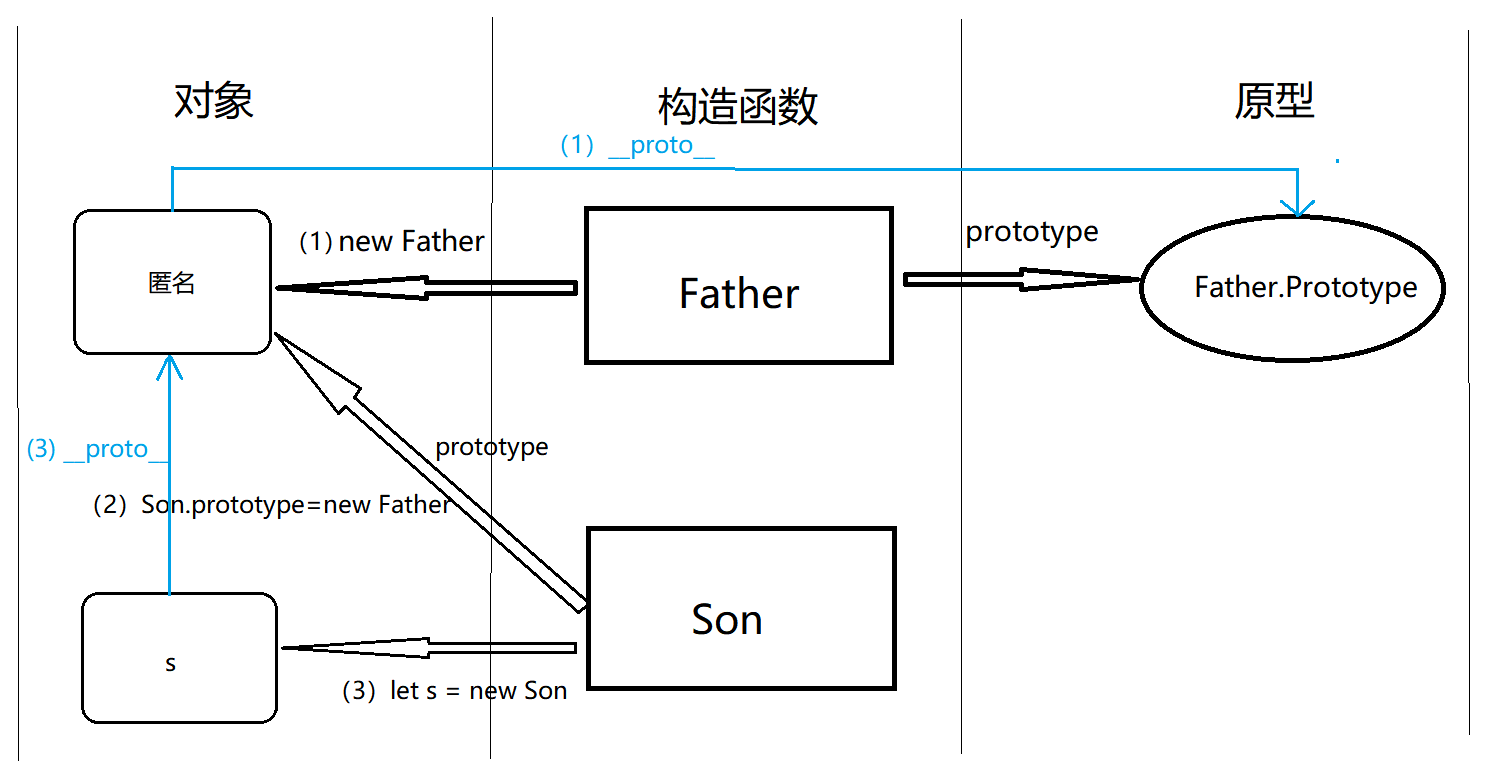
将父类的实例（返回值？）作为子类的原型。通过：子类 >> 子类原型 >> 父类 访问父类的方法

#### （2）步骤：

1）创造父类实例 Son.prototype = **new Father()**

2）让其成为子类的原型。**Son.prototype** = new Father()

3）创建子类对象。 let s = new Son()



#### （3）特点：

1. 纯粹的原型链继承
2. 父类原型方法/属性，子类都能访问到
3. 简单，易于实现

#### （4）缺点：

1. 要想为子类新增属性和方法，必须要在new father()这样的语句之后执行
2. 无法实现多继承
3. 来自原型对象的引用属性是所有实例共享的（this.arr=[ ]被所有对象共享）
4. 创建子类实例时，无法向父类构造函数传参

### 2.构造函数继承

#### （1）原理

使用父类的构造函数创建子类对象，相当于复制父类的实例属性给子类（没有用到原型）

#### （2）步骤

在子类构造函数中用call或apply使用父类构造函数:

function Son(){

Father.call(this)

}

#### （3）特点：

1. 解决了子类实例共享父类引用属性的问题
2. 创建子类实例时，可以向父类传递参数
3. 多继承（call多个父类对象）

#### （4）缺点：

1. 实例并不是父类的实例，只是子类的实例
2. 只能继承父类的构造函数的实例属性和方法，不能继承原型属性/方法
3. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能。（每次new新对象都会产生一模一样但是占不同内存的相同方法，如果放在原型里就能调用同一个了）

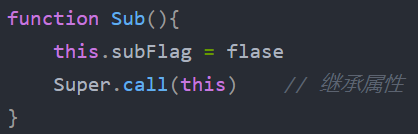
### 3.组合继承（原型链继承+构造继承）

#### （1）原理

利用构造函数继承可以传参的优点，

#### （2）步骤

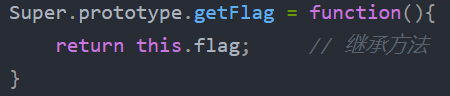
1）子类中调用父类构造函数，用来继承属性



2）使用原型链继承，继承父类原型上的方法



想添加方法时，只要在父类构造函数之后（先得有原型），在原型上添加方法即可：



3）由于sub.prototype = new Super()中，我们通过Super创建了对象，因此导致sub.prototype的构造函数constructor就变成了父类Super，因此最好将其修正为子类：



#### （3）特点：

　　1. 可以继承实例（构造函数？）属性/方法，也可以继承原型属性/方法

　　2. 既是子类的实例，也是父类的实例

　　3. 不存在引用属性共享问题

　　4. 可传参

　　5. 函数可复用

#### （4）缺点：

　　1. 利用原型链继承时，给子类（子类构造函数）的原型添加了一层父类属性。（子类原型其实不需要）

2. 利用构造函数继承时，给子类（子类构造函数）添加了一层父类属性。（覆盖了子类原型的属性）

### 4.寄生组合继承

#### （1）原理

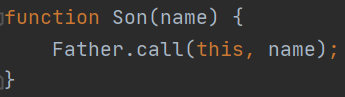
组合继承中父类构造函数被调用两次，因此可以在原型链继承时，让其只继承（指向父类原型）而不调用父类构造函数。

#### （2）步骤

1）子类中调用父类构造函数，用来继承属性

2）使用原型链继承，继承父类原型上的方法，但不执行父类构造函数：

<1>使用构造函数继承



<2>使用Object.create，建立原型链继承却不调用父类构造函数



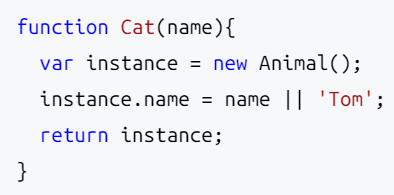
3）由于sub.prototype = new Super()中，我们通过Super创建了对象，因此导致sub.prototype的构造函数constructor就变成了父类Super，因此最好将其修正为子类：

### 5.实例继承

#### （1）原理

为父类实例添加新特征，作为子类实例返回

#### （2）步骤



#### （3）特点：

1) 不限制调用方式，不管是new Son()还是Son(),返回的对象具有相同的效果

#### （4）缺点：

1）实例是父类的实例，不是子类的实例

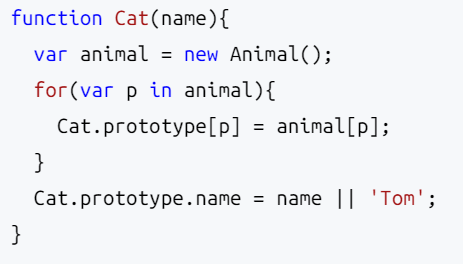
2）不支持多继承

### 6.拷贝继承

#### （1）原理

子类构造函数中创建一个父类实例，将其属性一一拷贝给子类原型

#### （2）步骤



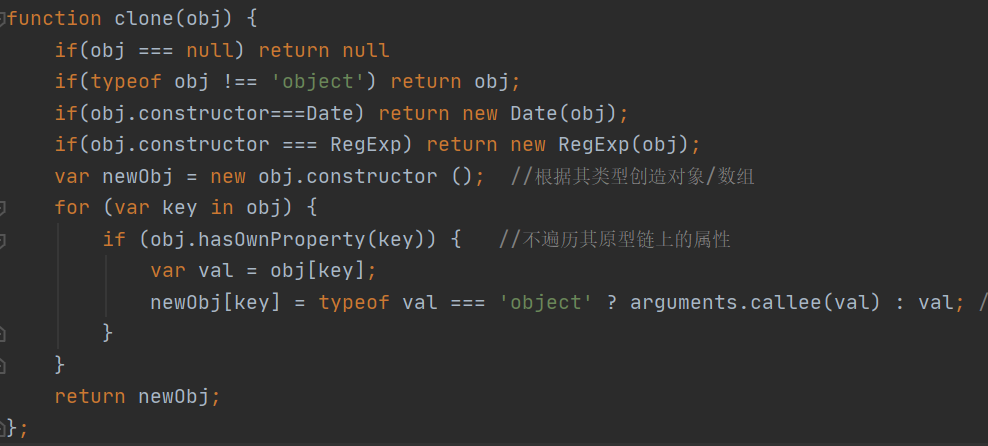
## 深拷贝与浅拷贝

### 1.浅拷贝

浅拷贝只复制单层属性，对于引用类型只复制其引用，指向的堆内存仍然是之前的。（只对堆内存元素进行复制？）

### 2.深拷贝

#### （1）递归实现



#### （2）JSON、拓展运算符、assign







#### （4）lodash函数库实现深拷贝

import lodash from 'lodash'

const form = lodash.cloneDeep(this.addForm)

## 防抖和节流

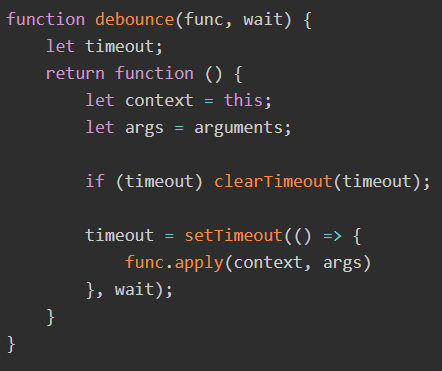
都在降低某个高频回调函数的执行频率

### 1.防抖

#### （1）定义

指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间，直到n秒后才可以执行下次事件。

#### （2）方法







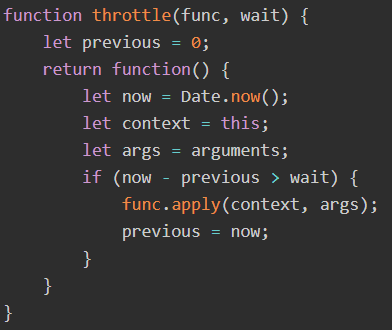
### 2.节流

#### （1）定义

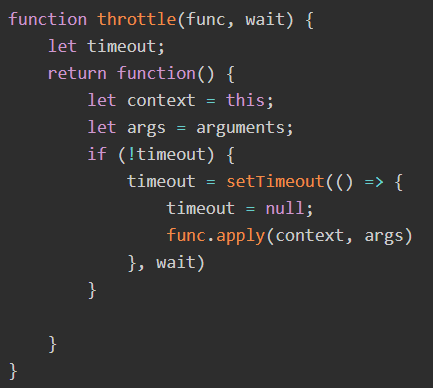
连续触发事件时，在 n 秒内只执行一次函数。

#### （2）方法

**1）时间戳**



**2）定时器**

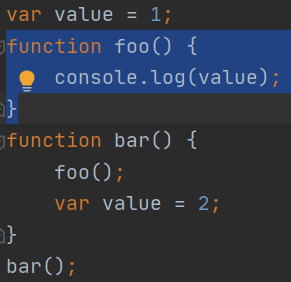


## 作用域和作用域链、执行期上下文

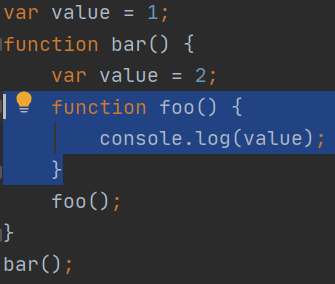
### 1.作用域类型

#### （1）静态作用域（词法作用域）：

词法作用域是指在词法分析阶段就确定了，不会改变。变量的作用域是在定义时决定而不是执行时决定，也就是说词法作用域取决于源码，通过静态分析就能确定，因此词法作用域也叫做静态作用域。



定义时位于外面，因此value=1;



定义时位于内部，因此value=2

#### （2）动态作用域：

动态作用域是在运行时根据程序的流程信息来动态确定的，而不是在写代码时进行静态确定的。 动态作用域并不关心函数和作用域是如何声明以及在何处声明的，只关心它们在何处调用。

### 2.作用域范围

#### （1）全局作用域

在代码中任何地方都能访问到的对象拥有全局作用域。全局作用域的变量是全局对象的属性，不论在什么函数中都可以直接访问，而不需要通过全局对象，但加上全局对象，可以提供搜索效率。

a、没有用var声明的变量（除去函数的参数）都具有全局作用域，成为全局变量，所以声明局部变量必须要用var。

b、window的所有属性都具有全局作用域

c、最外层函数体外声明的变量也具有全局作用域

#### （2）局部作用域

局部变量的优先级高于全局变量，仅限于函数内。

a、函数体内声明的变量具有局部作用域，成为局部变量

b、函数的参数也具有局部作用域

#### （3）块级作用域

一切大括号{} 内部使用let/const声明的变量（函数、if、{}）

局部作用域优先级 > 块级作用域 (在函数体大括号内部，无论使用什么关键字声明var/let/const都是局部作用域，在调试器中是local，而不是block)

块级作用域内的var声明会被提升到外围作用域，但是let\const会被限制在块级作用域

### 3.作用域链

执行流每进入一个执行环境，都会创建一个**作用域链**。

作用域链由执行环境的变量对象组成，作用域链的前端始终是当前执行环境的变量对象，下个变量对象来自外围环境，再下一个变量对象则来自下一个外围环境，一直延续到全局执行环境的变量对象。全局执行环境的变量对象始终都是作用域链中的最后一个变量对象。

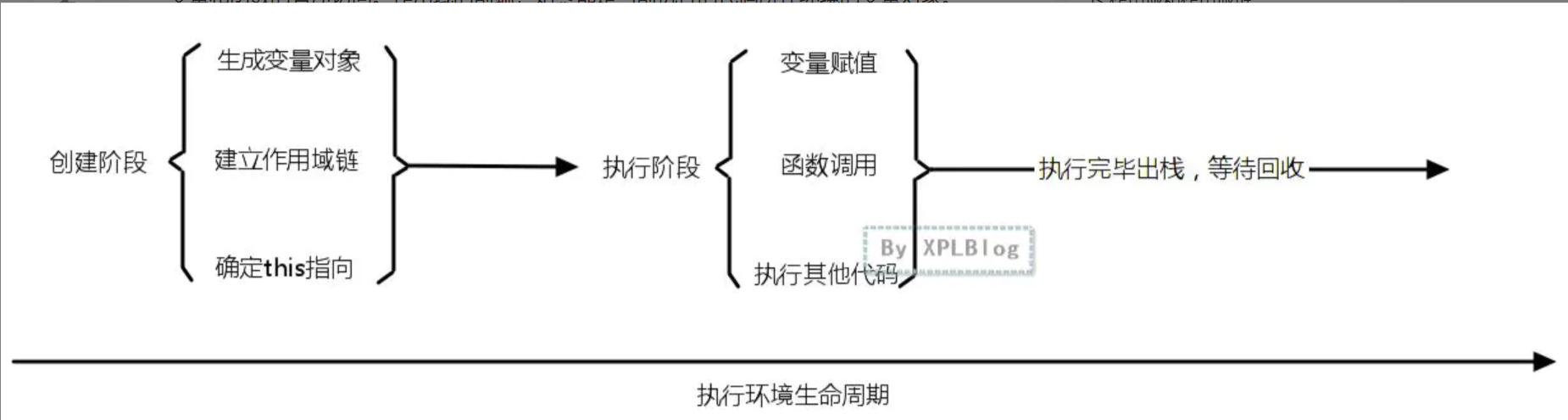
作用域链的用途是保证对执行环境有权访问的所有变量和函数的有序访问。

### 4.执行上下文

#### （1）定义

执行上下文(Execution Context): 函数执行前进行的准备工作（也称执行上下文环境、执行环境）

#### （2）步骤



**创建阶段：**当函数被调用时，尚未执行函数内部代码之前。

**1）**变量对象（VO, variable object）：变量对象即包含变量的对象，开发者无法直接访问。

<1.1>根据函数的参数，创建并初始化arguments对象。

<1.2>声明提升:

<2.1>扫描函数内部代码，查找函数声明。

查找所有函数声明将函数名和函数引用存入变量对象中，如果变量对象中已存在同名函数则覆盖。

<2.2>扫描函数内部代码查找变量声明

查找所有变量声明，将变量名存入变量对象中，初始化为undefined。如果变量名和已声明的形参或函数同名，则什么也不

（函数提升优先级大于变量提升，当两者同名时，变量声明但未赋值的话，函数优先；变量只要是赋值了，就变量优先；）

**2）**作用域属性：[[Scope]]属性时一个指向单向链表的头结点的指针

作用域即变量对象，作用域链是一个由变量对象组成的带头结点的单向链表，作用是用来进行变量查找。

**3）**this：指向一个环境对象

**执行阶段**：代码执行阶段

1）初始化变量对象：设置变量的值、函数的引用

2）解释并执行代码 这里将产生新的执行环境，推入执行栈递归执行上述步骤

**出栈等待回收**

#### （3）分类

**1）全局执行环境**

在浏览器环境中全局执行环境就是window对象，window对象是JS代码开始运行时的默认环境。全局执行环境的变量对象始终都是作用域中的最后一个对象。

**2）函数执行环境**

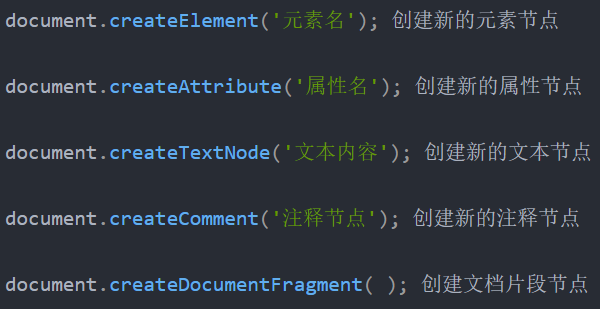
当某函数被调用时，首先会创建一个执行环境及相应的作用域链，然后使用arguments和其他命名参数的值来初始化执行环境的变量对象。

## DOM常见的操作方式

### 1.查找节点

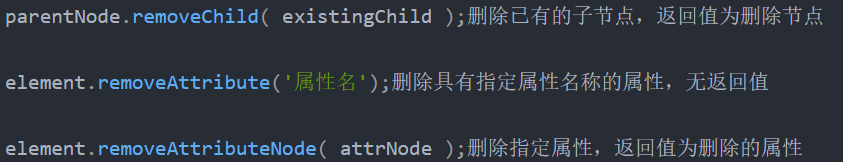


### 2.创建节点

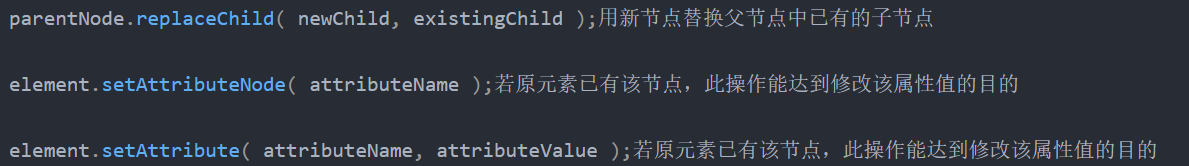


appendchild 移动某节点到当前节点

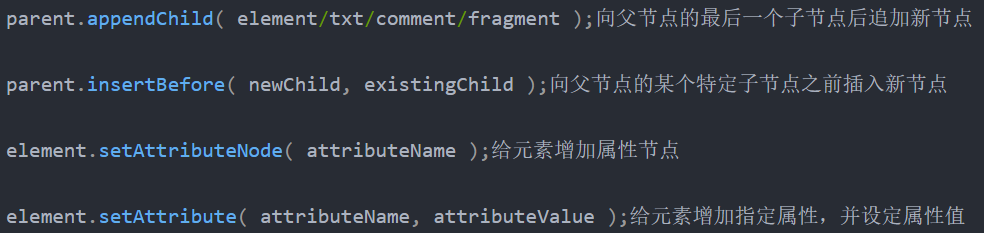
### 3.删除节点



### 4.修改节点



### 5.插入节点



### 6.设置样式



**Array.sort()方法与实现机制**

arrayObject.sort(sortby) sortby必须是函数，定义比较方式：

function sortNumber(a,b)

{

return a - b

}

### 7.节点类型



## 各JS引擎

### 1.排序的实现

**V8引擎（chrome） :**

sort 函数只给出了两种排序分别是： InsertionSort 和 QuickSort，

数组长度小于等于 10 的用插入排序 InsertionSort，

比10大的数组则使用快速排序 QuickSort

**SpiderMonkey引擎（Firefox）**

归并排序（jsarray.c 源码）

**Nitro（JavaScirptCore）：**

桶排序、归并排序

**Chakra（IE浏览器）**

快排

|  |  |  |  |
| --- | --- | --- | --- |
| 浏览器 | JS引擎 | 排序 | 底层实现 |
| Chrome | V8 | 插入、快排 | c++ |
| IE8及以下 | JScript |  |  |
| IE9及以上，IE Edge | Chakra | 快排 |  |
| Firefox | SpiderMonkey | 归并 | c |
| Safari | JavascriptCore | 桶、归并 |  |

## Ajax的请求过程

### 1.Ajax 定义：

Ajax 即“Asynchronous Javascript And XML”（异步 JavaScript 和 XML）

（1）异步通信

（2）js 脚本向服务器发起 http 通信

（3）更新部分网页

### 2.Ajax步骤:

#### （1）原生js



注：readyState的变化一次，onreadystatechange就执行一次

### 3.Ajax 的 readyState

| **状态值** | **含义** |
| --- | --- |
| 0 | 请求未初始化 |
| 1 | 服务器连接已建立 |
| 2 | 请求已接收 |
| 3 | 请求处理中 |
| 4 | 请求已完成，且响应已就绪 |

**Ajax 禁用浏览器的缓存功能**

ajax请求将数据缓存到内存中，不刷新页面就会一直被缓存在内存中，同一个URL不重复请求服务器，因此需要禁止缓存

解决的方法有：

1. 在 ajax 发送请求前加上 xhr.setRequestHeader("If-Modified-Since","0")。
2. 在 ajax 发送请求前加上 xhr.setRequestHeader("Cache-Control","no-cache")。
3. 在 URL 后面加上一个随机数： "fresh=" + Math.random();。
4. 在 URL 后面加上时间搓："nowtime=" + new Date().getTime();。
5. 如果是使用 jQuery，直接这样就可以了$.ajaxSetup({cache:false})。这样页面的所有 ajax 都会执行这条语句就是不需要保存缓存记录。

## JS的垃圾回收机制

### 1.定义

对内存的分配以及回收实行自动管理。在周期性的时间内执行。

### 2.方式

#### （1）标记清除

1）给内存中所有变量加上标记。

2）对于环境中的变量、环境中变量引用的变量，将它们的标记清除。

3）销毁带标记的值，并回收其内存空间。

（IE、Firefox、Opera、Chrome、Safari使用此方法 2008年）

#### （2）引用计数

1）某值被引用时+1，被解除引用时-1

2）删除引用次数为0的值

3）循环引用导致无法回收

### 3.自己管理

为了防止页面占用过多内存导致系统崩溃，分配给Web浏览器的内存比桌面应用要小。

因此为了优化内存占用，一旦数据不再使用，设为null释放其引用（全局变量）

释放引用不意味着释放内存，释放内存时垃圾回收运行时才进行的

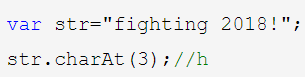
## JS中的String、Array和Math方法

### 1.String

#### （1）charAt()方法

**用于返回指定索引处的字符。返回的字符是长度为 1 的字符串。**

如果参数 index 不在 0 与 string.length-1 之间，该方法将返回一个空字符串。



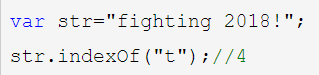
#### （2）indexOf()方法

**可返回某个指定的字符串值在字符串中首次出现的位置。**

语法：stringObject.indexOf(str,startpos)

str 必需，给定的需要检索的字符串。

startpos 可选的整数参数，规定在字符串中开始检索的位置。如省略该参数，则将从字符串的首字符开始检索。



#### （3）split()方法

**将字符串分割成字符串数组，并返回此数组**

语法：stringObject.split(separator,limit)

separator 必需，从该参数指定的地方分割。

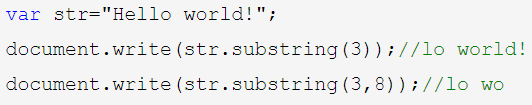
limit 可选，分割的次数



#### （4）substring()方法

**用于提取字符串中介于两个指定下标之间的字符，其内容是从 start 处到 stop-1 处的所有字符，其长度为 *s*top 减 start。**

语法：stringObject.substring(start,stop)



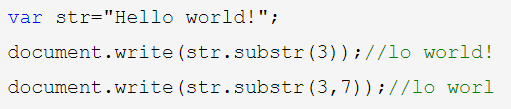
#### （5）substr()方法

**在字符串中抽取从 start 下标开始的指定数目的字符。**

语法：stringObject.substr(start,length)

start 必需，要抽取的子串的起始下标。必须是数值。如果是负数，那么该参数声明从字符串的尾部开始算起的位置。也就是说，-1 指字符串中最后一个字符，-2 指倒数第二个字符，以此类推。

length 可选，必须是数值。如果省略了该参数，那么返回从 *stringObject* 的开始位置到结尾的字串。



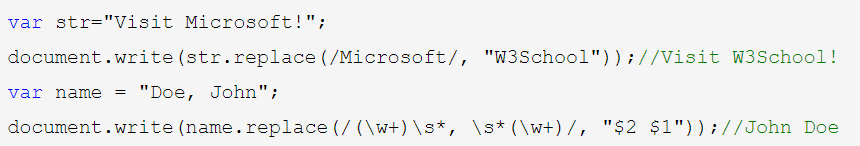
#### （6）replace()方法

**用于在字符串中用一些字符替换掉另一些字符，或替换掉一个与正则表达式匹配的子串。返回经过替换操作后形成的新的字符串，不改变原字符串**

语法：stringObject.replace(regexp/substr,newsubstr/function)

regexp/substr 必需，规定要替换掉的子字符串或要替换掉的模式的 RegExp 对象。

newsubstr/function 必需，规定了替换文本或生成替换文本的函数。

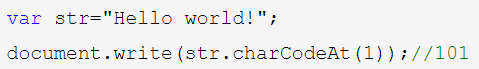


#### （7）charCodeAt()方法

**可返回指定位置的字符的 Unicode 编码。**

语法：stringObject.charCodeAt(index)

index 必需，表示字符串中某个位置的数字，即字符在字符串中的下标。

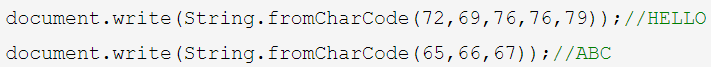


#### （8）formCharCode()方法

**接受一个指定的 Unicode 值，然后返回一个字符串。**

语法：String.fromCharCode(numX,numX,...,numX)

numX 必需，一个或多个 Unicode 值，即要创建的字符串中的字符的 Unicode 编码。



### 2.Math

#### （1）ceil()方法

**对一个数进行向上取整**

语法：Math.ceil(x)  返回大于等于x 并且与x最接近的整数

#### （2）floor()方法

**对一个数进行向下取整**

语法：Math.floor(x) 返回小于等于x 并且与x最接近的整数

#### （3）round()方法

**对一个数进行四舍五入取整**

语法：Math.round(x) 返回最接近x的整数

#### （4）random()方法

**返回介于 0 ~ 1 之间的一个随机数**

语法：Math.random() 返回0.0 ~ 1.0 之间的一个随机数。

#### （5）max()方法

**返回指定的数中带有较大的值的那个数**

语法：Math.max(x1,x2,.....) 返回x1,x2,.....中带有最高值的数字  如果有某个参数为 NaN，或是不能转换成数字的非数字值，则返回 NaN。

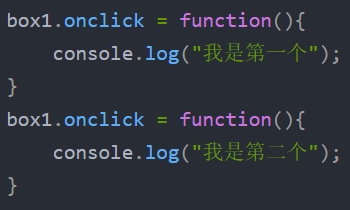
#### （6）min()方法

**返回指定的数中带有较小的值的那个数**

语法：Math.min(x1,x2,......) 返回x1,x2,.....中带有最小值的数字  如果有某个参数为 NaN，或是不能转换成数字的非数字值，则返回 NaN。

## addEventListener和onClick()的区别

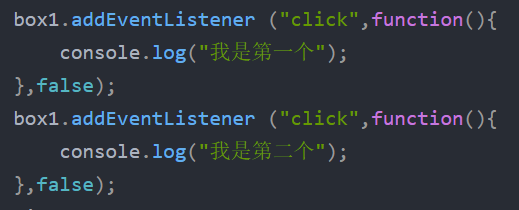
### 1.onclick()



一个click处理器在同一时间只能指向唯一的对象。因此就算对于一个对象绑定了多次，但是仍然只会出现最后的一次绑定。

移除：element.onclick = null;

### 2.addEventlistener



对于一个可以绑定的事件对象，想多次绑定事件都能运行，选用addEventListener.

false：事件冒泡

true：事件捕获

移除：removeEventListener(event, function)

1.onclick 只能给元素注册一个事件，如果存在多个，则后面的事件会覆盖前面的事件；

2.addEventListener允许给一个事件注册多个listener监听器，添加的事件不会覆盖已存在的事件；

3.addEventListener对任何DOM都是有效的，而onclick仅限于HTML；

4.addEventListener可以控制listener的触发阶段，（捕获/冒泡）。对于多个相同的事件处理器，不会重复触发，不需要手动使用removeEventListener清除；

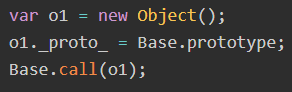
5.IE8及以下、Opera 7.0 及 Opera 更早版本可以使用attachEvent(事件名，处理函数)和detachEvent(事件名，处理函数)，注意：事件名带on。

## new和Object.create的区别

new令实例的隐式原型指向构造函数的显示原型 obj.\_\_proto\_\_ = Base.prototype

Object.create令实例的隐式原型指向传入的对象 obj.\_\_proto\_\_ = Obj

### 1.new

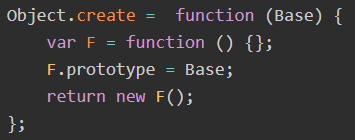


（1）创造空对象

（2）让空对象的原型（\_proto\_）指向构造函数的原型（Bae.prototype)

（3）空对象执行构造函数，获得其属性。

### 2.Object.create



（1）创造空构造函数

（2）让构造函数的原型指向对象（传入的对象不能为构造函数，因为过程中没有执行构造函数）

（3）返回空构造函数的实例，并没有调用Base的构造函数方法。

备：Object.create(Base)为什么不能传构造函数？

create过程中，传入的Base并没有被执行，被执行的是空构造函数，因此实例中只有空构造函数的属性（即没有属性），create只是将实例的原型指向传入的对象。

## DOM的location对象

### 1.定义

Location 对象实际上是 JavaScript 对象，而不是 HTML DOM 对象。

Location 对象是由 JavaScript runtime engine 自动创建的，包含有关当前 URL 的信息。

Location 对象是 Window 对象的一个部分，可通过 window.location 属性来访问。





## 浏览器从输入URL到页面渲染的整个流程（涉及到计算机网络数据传输过程、浏览器解析渲染过程）

1. **从浏览器接收url到开启网络请求线程**

（浏览器进程与线程、url解析协议/端口/域名）

1. **开启网络线程到发出一个完整的HTTP请求**

（DNS查询，HTTP，TCP/IP请求）

1. **建立连接后，前后端交互**

（这一部分包括HTTP头部、响应码、报文结构、cookie、缓存等知识）

1. **浏览器接收到HTTP数据包后的解析流程**

（解析html-词法分析，然后解析成dom树、解析css生成css规则树、合并成render树，然后layout、painting渲染、复合图层的合成、GPU绘制、外链资源的处理、loaded和DOMContentLoaded等）

1. **CSS的可视化格式模型**

（元素的渲染规则，如包含块，控制框，BFC，IFC等概念）

1. **JS引擎解析过程**

（JS的解释阶段，预处理阶段，执行阶段生成执行上下文，VO，作用域链、回收机制等等）

## [JavaScript](file:///C:\jump\super-jump\word%3fword=JavaScript)中的arguments

### 1.定义

**arguments** 是一个对应于传递给函数的参数的类数组对象。

### 2.特征

（1）是所有（非箭头）函数中都可用的**局部变量**。

（2）arguments对象不是一个 [Array](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array) 。它类似于Array，但除了length属性和索引元素之外没有任何Array属性。

### 3.属性

[**arguments.callee**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments/callee)

指向参数所属的当前执行的函数。

指向调用当前函数的函数。

[**arguments.length**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments/length)

传递给函数的参数数量。

[**arguments[@@iterator]**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/@@iterator)

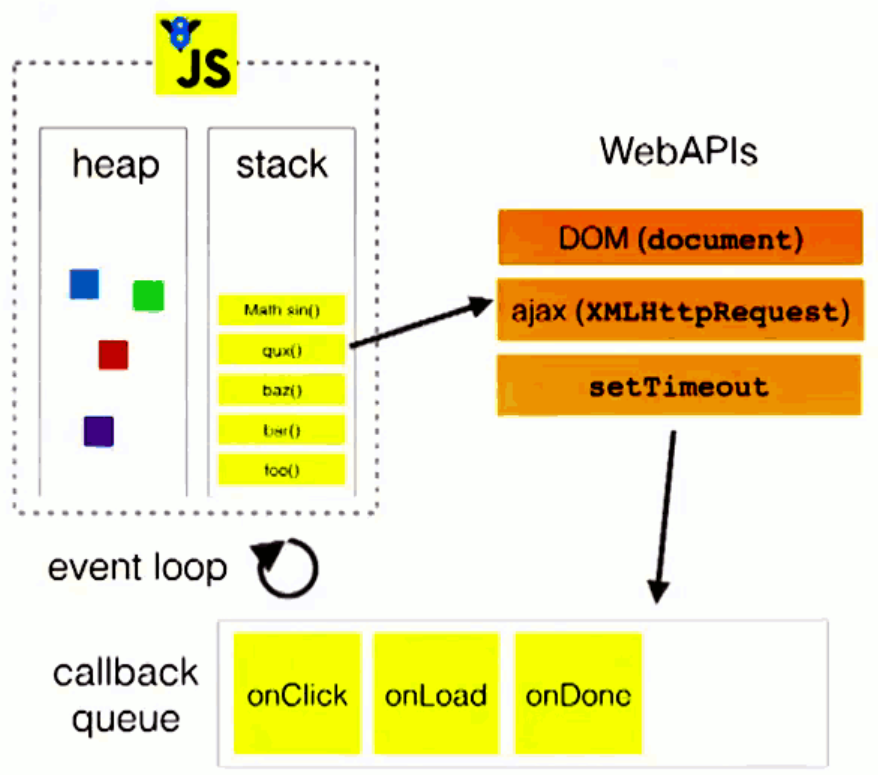
返回一个新的[Array 迭代器](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/@@iterator) 对象，该对象包含参数中每个索引的值。

## EventLoop事件循环

### 1.定义

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）。

（也就是JS主线程、浏览器子线程、回调队列之间的循环执行）



### 2.过程

1）执行栈中会存放执行上下文（执行环境），执行其中的代码。

2）遇到同步任务：

<1>不是方法：直接执行

<2>是方法：将其执行上下文放入执行栈，递归执行

<3>如果里面没有方法了，执行完后出栈。

3）遇到异步任务：

<1>将异步任务放入Event Table中注册。（setInterval是放在这里后，隔段时间入任务队列一次，如果执行栈不为空，会持续累计！）

<2>将执行完(ajax)、到时间(setTimeout)的异步任务放入任务队列callback queue

<3>当执行栈空闲时，执行任务队列中的异步任务

<4>如果其中还有异步任务，返回<1>进行递归调用

## 宏任务与微任务

### 1.宏任务

macro task，可以理解是每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）。

浏览器为了能够使得JS内部(macro)task与DOM任务能够有序的执行，会在一个(macro)task执行结束后，在下一个(macro)task 执行开始前，对页面进行重新渲染，流程如下：

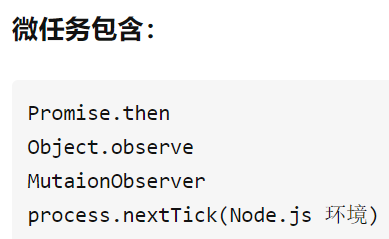


//setImmediate，类似setTimeout，在宏任务队列靠后位置执行（每次运行结果不一定相同）

### 2.微任务

micro task,可以理解是在当前 task 执行结束后立即执行的任务。也就是说，在当前task任务后，下一个task之前，在渲染之前。

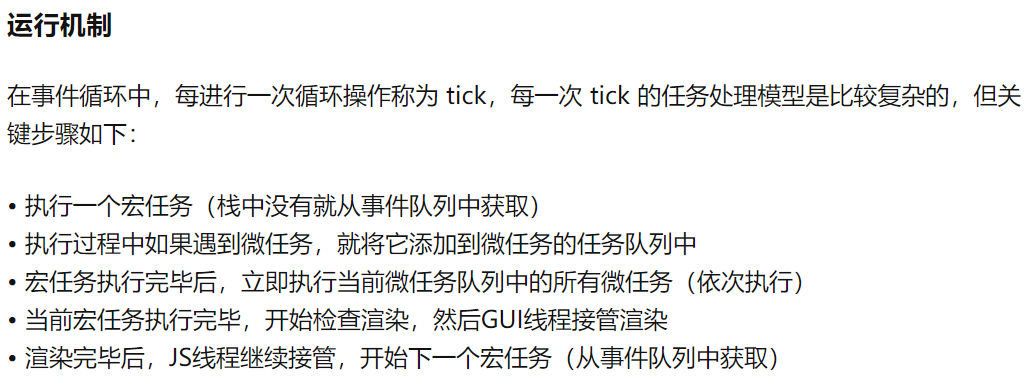
所以它的响应速度相比setTimeout（setTimeout是task）会更快，因为无需等渲染。也就是说，在某一个macrotask执行完后，就会将在它执行期间产生的所有microtask都执行完毕（在渲染前）。



//nextTick内部代码在当前宏任务<script>之后，微任务之前执行；

.then在Promise后才有返回值，Promise在前阶段微任务，then在下一阶段的微任务

### 3.执行机制



## BOM属性对象方法

### 1.定义

BOM（Browser Object Model）浏览器对象模型，提供了独立于内容而与浏览器窗口进行交互的对象。描述了与浏览器进行交互的方法和接口，可以对浏览器窗口进行访问和操作，譬如可以弹出新的窗口，改变状态栏中的文本，对Cookie的支持。

### 2.内容

#### （1）window

BOM的核心对象是window，它表示浏览器的一个实例。window对象处于JavaScript结构的最顶层，对于每个打开的窗口，系统都会自动为其定义 window 对象。

属性：



方法：



#### （2）location对象

location是BOM对象之一，它提供了与当前窗口中加载的文档有关的信息，还提供了一些导航功能。事实上，location对象是window对象的属性，也是document对象的属性；所以window.location和document.location等效。

属性：



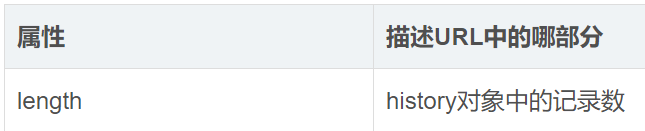
方法：



#### （3）history对象

history对象是window对象的属性，它保存着用户上网的记录，从窗口被打开的那一刻算起。

属性：



方法：



## 函数柯里化及其通用封装

### 1.定义

柯里化函数的运行过程其实是一个参数的收集过程，我们将每一次传入的参数收集起来，并在最里层里面处理。

### 2.实现



## 数组扁平化

### 1.reduce

利用reduce将上次结果与本次结果计算的特征，如果是基本数据类型就直接concat，否则递归调用。

### 2.toString/join & split

用toString/join将数组转化为字符串时，会默认在子数组中递归调用。

之后将其转化为数字（字母就无能为力了）

### 3.递归

初始化一个空数组，用map判断每个元素，如果是数组就递归，否则直接接入；

### 4.拓展运算符

拓展运算符会消除当前的外围括号，返回一系列参数，然后concat接入。多次遍历，直到some中没有数组。

### 5.es6中flat()

flat(dep)，将数据展开，深度为dep，深度代表将多深的数组扁平。

## “==”和“===”的区别

== 表示相等 （值相等）

===表示恒等（类型和值都要相等）

js在比较的时候如果是 == 会先做类型转换，再判断值得大小，如果是===类型和值必须都相等。

NaN==NaN  //返回false

NaN===NaN   //返回false

undefined==null //返回true ECMAScript 规范认为，既然 null 和  undefined 的行为很相似，并且都表示 一个无效的值，那么它们所表示的内容也具有相似性

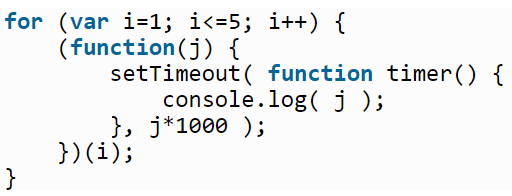
undefined===null //返回false 类型不同，直接false

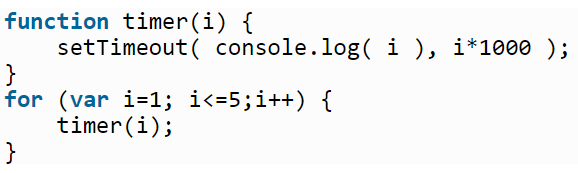
## for循环中的setTimeout

在for循环过程中，setTimeout被放入异步任务队列中，等执行栈为空后才执行，传入的参数为同一个。

解决：

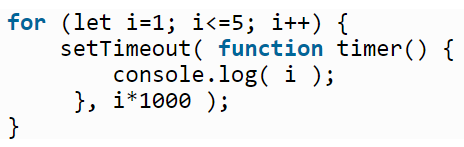
### （1）闭包





利用闭包延长变量生命周期的特点

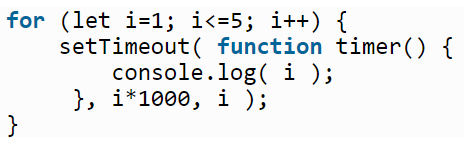
### （2）let



for中声明let后，会在每次循环中重新初始化一个let i，这样每个settimeout中参数i都是独立的；

（注意：for循环外声明let不起作用，必须在for中定义let)

### （3）setTimeout第三个参数



## Object.defineProperty实现代理

Object.defineProperty(obj, prop, descrpt{....})

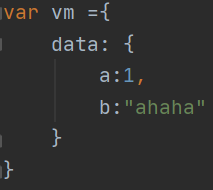
obj：为被设置的对象

prop：被设置对象的属性

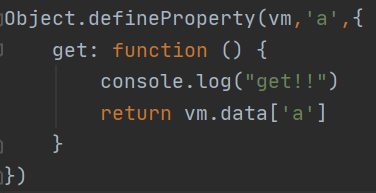
descript：当obj[prop]被调用时，做出响应

可用于做内层对象属性的代理

比如对于下面对象，我们想直接通过vm[a]或vm.a取值，但是结果会是undefined



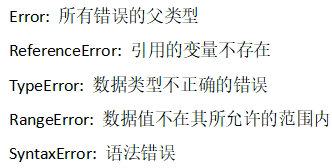
因此，可以通过defineProperty，令vm[a]/vm.a的值为vm.data[a]



这样就实现了代理

## 常见内置错误

### 1.错误类型



### 2.自定义错误提示



## parseInt与Number

### parseInt：

将含有数字的字符串部分取出，没数字返回NaN

得到的结果是整数型

为null/" "时，返回NaN

### Number：

含有字母时，返回NaN

得到的结果是Number型（可小数）

为null/" "时，返回0

## 构造函数、原型中添加方法

Promise.method()

给构造函数添加方法，但是由于没有进行this指向，因此实例访问不到。直接通过Promise.method访问。

Promise.prototype.method()

给构造函数的原型添加方法，实例可以通过原型链访问。