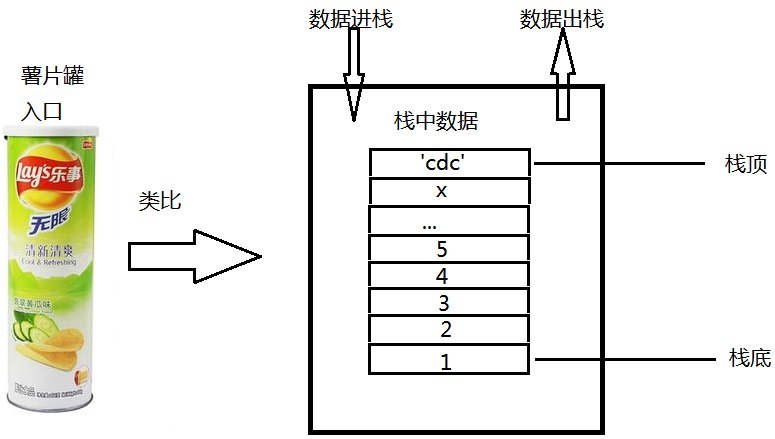
对于前端开发来说，内存空间的概念并没有像其他语言那样被重视，平时涉及到内存相关的情况比较少，通常就是内存溢出之类的。但是并不代表内存空间这块知识并不重要，相反的是，想要对于javascript有一个深刻的认知的话，必须对内存空间有一个清晰的认知！

内存空间中有三种数据结构，分别是堆(heap)，栈(stack)与队列(queue)。

**栈数据结构**

栈结构是怎样的呢？栈结构具有【先进后出，后进先出】的特点。比如在javascript的执行上下文中就是使用这种栈数据接口，来协调执行顺序(关于执行上下文会在后面文章讲述)。下面我们类比薯片罐加下图描述栈的存取方式：



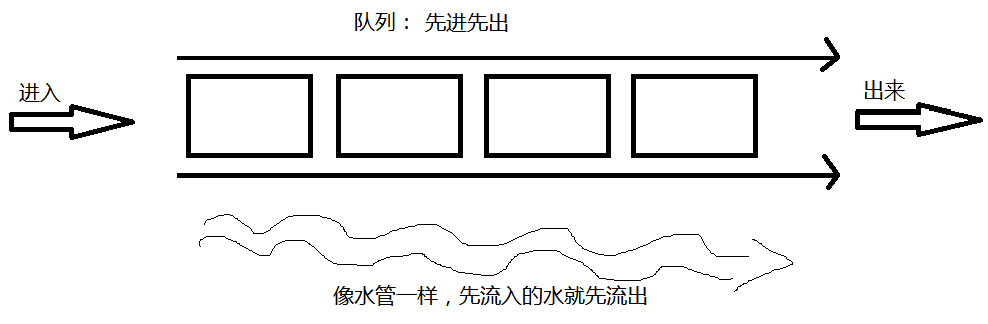
栈数据的存取跟薯片罐类似。放薯片的时候（存数据），最先放进去的薯片位于罐子底部，最后放的薯片位于罐子的顶部。吃薯片的时候（取数据），最先被拿出来的是最后放进去的那一片。如果我们想要拿最后的那片薯片，必须把前面的薯片都先取出来。这就是【先进后出，后进先出】的特点。

**堆数据结构**

堆数据结构比较类似我们去超市商场用的储物柜。什么意思呢？柜子虽然整整齐齐的摆放在一起，但是我们并不用去按顺序的打开，我们只需要记住自己储物柜的编号，便可以用钥匙打开。类似JSON格式的数据，采用key-val键值对无序的方式。

**队列数据结构**

队列是一种先进先出的数据结构。类似水管流水一样，先进去的水就先流出来，如下图：



经过上面的三个介绍，我们大致了解内存空间的数据结构，接下来，本文将介绍它们在实际中的运用。

**栈的运用--调用堆栈**

由于javascript是一种单线程编程语言，这意味着它只有一个 call stack。Call stack中文意思就是【调用堆栈】。它记录了我们程序中执行的位置。如果我们执行一个函数，则是把它入栈，如果我们从一个函数返回，则是把它弹出堆栈顶部。看下列例子：

function b (x,y) {

Return x+y

}

function a(x) {

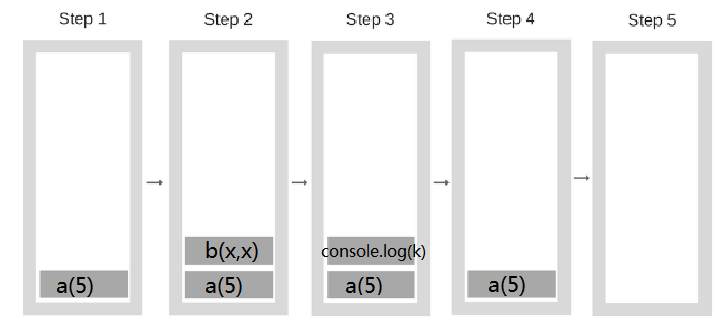
var k = b(x,x)

console.log（b）

}

a(5)

那么当引擎执行该段代码过程如下：



另外，当遇到抛错时，我们也是很容易看到栈结构的顺序的，比如以下代码：

function foo() {

throw new Error('SessionStack will help you resolve crashes :)');

}

function bar() {

foo();

}

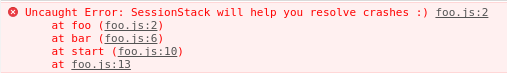
function start() {

bar();

}

start();

假设这段代码在foo.js文件中，并在chrome执行，则会出现以下错误：



**堆栈大小**

还有一种情况是调用堆栈情况达到最大值，比如以下代码：

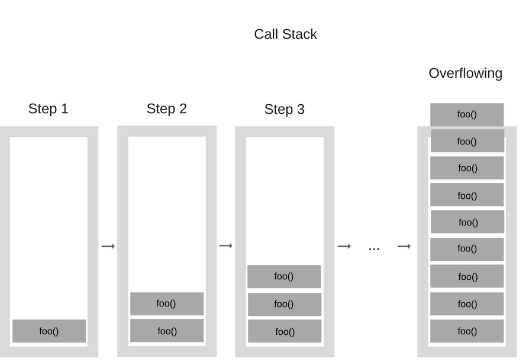
function foo() {

foo();

}

foo();

由于在引擎中执行这个代码时，会先调用foo函数，并且不断的递归调用，没有任何终止调整，所以执行时，同一个函数会不断重复的添加到调用栈中，如下图：



直到最后超出调用堆栈的实际大小，浏览器就会抛出如下错误：

IMG_256

**变量的存放**

首先我们应该知道内存中有栈和堆，那么变量应该存放在哪里呢，堆？栈？

1. 基本类型 --> 保存在栈内存中，因为这些类型在内存中分别占有固定大小的空间，通过按值来访问。基本类型一共有6种：Undefined、Null、Boolean、Number 、String和Symbol

2、引用类型 --> 保存在堆内存中，因为这种值的大小不固定，因此不能把它们保存到栈内存中，但内存地址大小的固定的，因此保存在堆内存中，在栈内存中存放的只是该对象的访问地址。当查询引用类型的变量时， 先从栈中读取内存地址， 然后再通过地址找到堆中的值。对于这种，我们把它叫做按引用访问。

**堆内存与引用数据类型**

在执行上下文中，会生成一个变量对象，拆分理解一下：变量-对象，其实就是一个存储变量的对象。对于javascript中的基础数据类型，如Undefined、Null、Boolean、Number、String，一般是保存在变量对象中，需要访问他们时，都是按值访问，我们可以直接操作保存保存在变量对象中的实际值。

然而对于引用数据类型，如数组Array、对象Object，正如上述的变量存放所说，它们值的大小不是固定的，它们不能按值访问。对于引用数据类型的访问，实际上我们是访问的是变量对象中的一个值，这个值记录着引用数据的在堆内存中的地址。

文字描述总是素然无味，图片插入才比较有画面感，看图与代码：

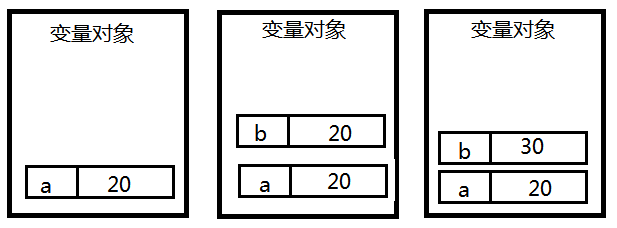
**实例一：**

var a = 20

var b = a

b = 30

在内存中，变化如下图：



在上述案例中，答案是20。因为a与b的值都属于基本类型，他们之间是按值访问的，因此var b = a时，只是把a的值20直接赋予b，并不存在应用关系。

**实例二：**

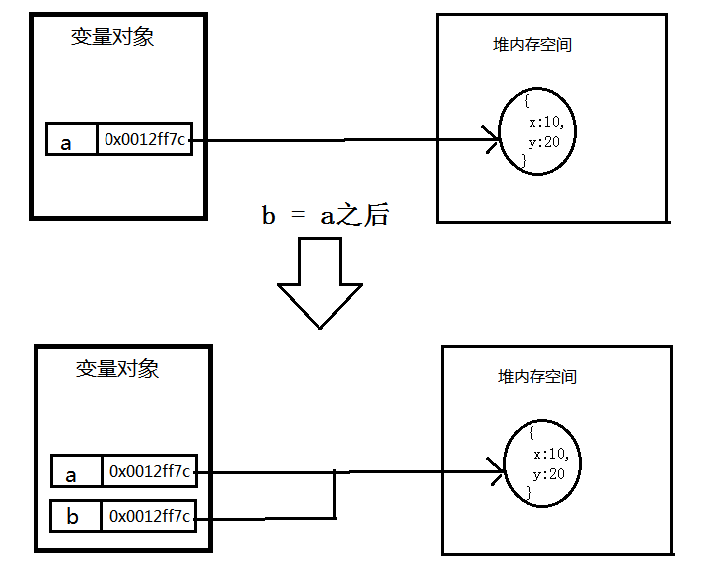
var a = { x:10, y:20}

var b = a

b.x = 15

请问此时a.x的值是多少？

在内存中，变化如下图：



上述示例，答案是15。由于a是josn数据，属于引用数据类型数据，所以a保存的值，实际上是json数据在堆内存中的地址，通过内存地址找到对应的json数据，类似我们通过门牌号找到对应的房子。在var b = a中，实际上是把内存地址值赋值给了b，所以当b.x = 15时，是通过b的内存地址，找到对应的数据，并修改。此时，再访问a.x时，也是通过内存地址去访问到修改过后的值了。

**内存空间管理**

通过上文，我们知道了变量是按照一定的数据结构方式存储在内存中，那么，内存是怎么管理这些变量占用的内存，才能保证不使用的变量不浪费占用内存呢？这其实得益于内存自动的分配与回收机制。

在javascript中内存的生命周期如下：

1. 分配变量所需的内存
2. 读写分配到的内存
3. 释放不使用的内存

示例：

var a = 20 // 内存中给变量a分配了空间

console.log(a) // 读内存中的值

a = null // 标记内存使用完毕，可释放变量占用的内存空间

Javascript自动垃圾收集机制很简单，就是找出不继续使用的值，然后释放该值占用的内存。并且垃圾收集器会每隔固定的时间段就执行一次垃圾收集。

那么怎么找出不继续使用的值呢？在javascript中使用的【标记清除】的算法来找到哪些变量不再使用，所以上述示例中a=null就是一种标记，让a对应的值失去应用，使得这个值在下一次垃圾收集器执行时能被释放。

值的注意的是，在局部作用域中，比如函数执行完毕后，我们没必要给每个不使用的变量都null，因为垃圾收集器会自己做出判断和回收，但是对于全局比那里，什么时候需要释放内存空间就比较难判断了，因为在原则上开发中我们应该减少使用过多的全局变量。

对于垃圾收集机制的详细了解，可以阅读《JavaScript高级编程》中的4.3节。