

一、前端为什么要关注内存？

任何一个程序的运行，都需要分配内存空间，对于一个页面来说，如果一些不再需要使用的内存没有得到及时的释放，我们称这种现象为内存泄露，内存泄露堆积会造成内存溢出，内存溢出就是我们所需要使用的内存空间大于可用内存，此时我们的程序就会出现内存溢出错误。

- 防止页面内存占用过大，引起客户端卡顿，甚至无响应
- Node.js使用V8引擎，内存对于后端服务性能至关重要，因为后端服务的持久性，后端更容易造成内存溢出（全栈工程师必备）

二、JS数据类型与JS内存机制

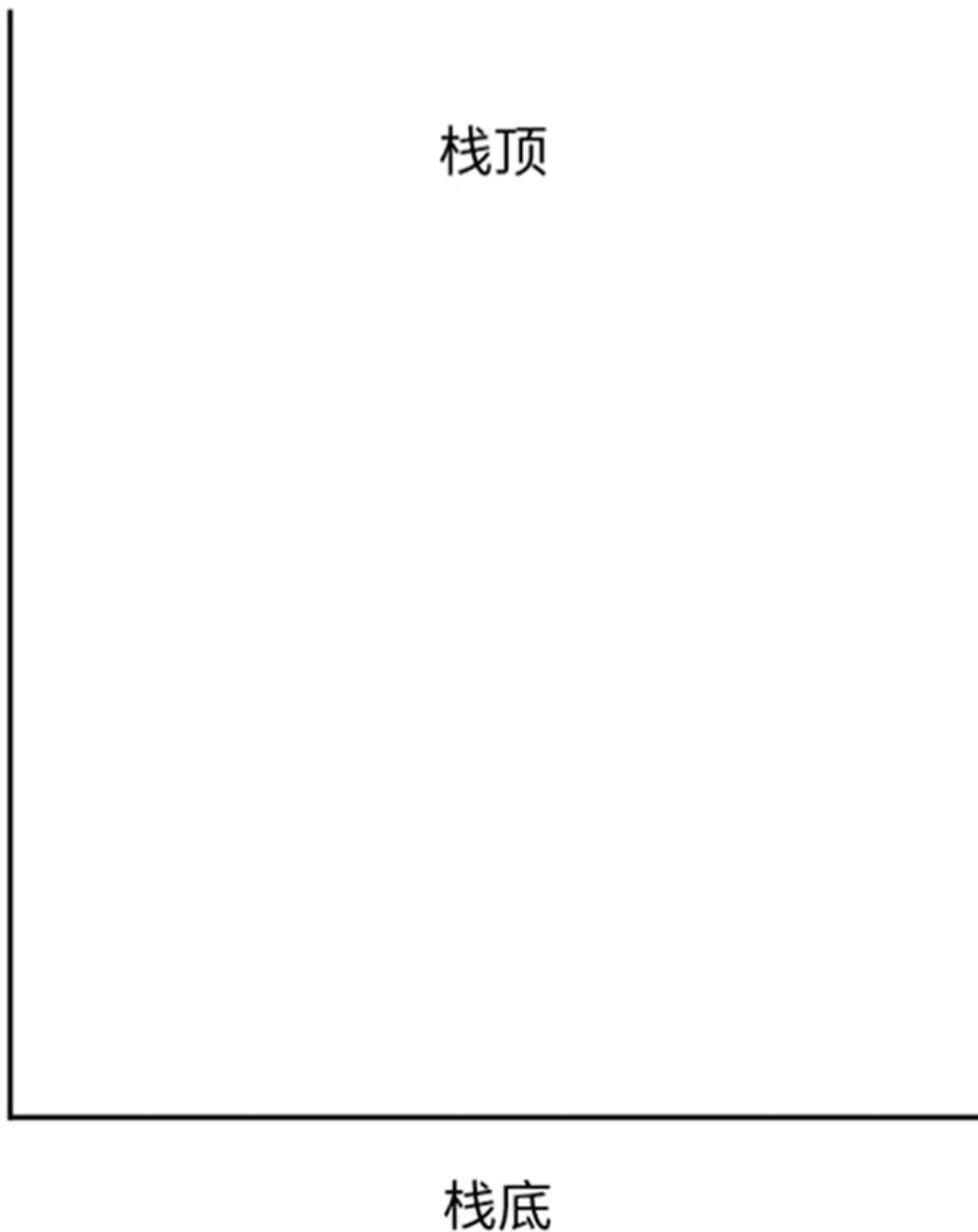
数据类型

- 原始数据类型：数字型（Number）、字符串型（String）、布尔型（Boolean）、空对象（Null）、未定义（undefined）、ES6新增Symbol类型
- 引用数据类型：Object（Array、Function都属于Object）
- 内存空间：栈内存（stack）、堆内存（heap）

栈内存

在JavaScript中，每一个数据都需要一个内存空间，JavaScript中的**原始数据类型都有固定的大小**，保存在栈内存中，**由系统自动分配存储空间**，我们可以直接进行操作，因此，**原始数据类型都是按值访问**。栈是一种运算受限的线性表，**仅允许在表的一端插入和删除运算**。例如：

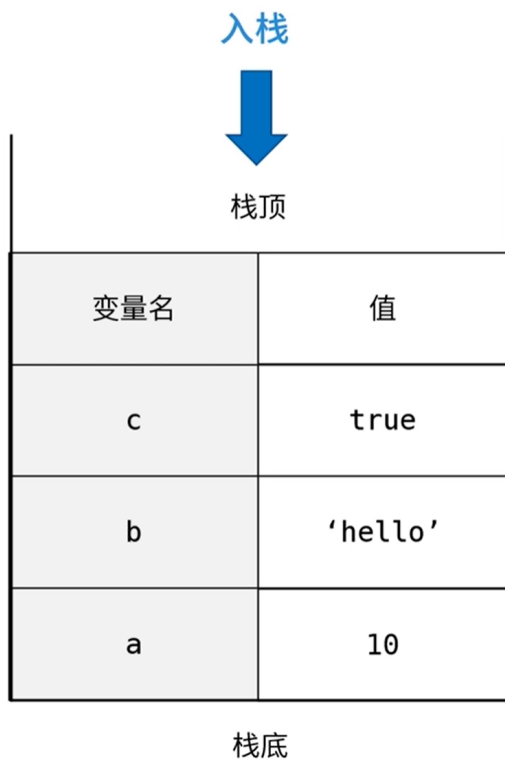
把栈想象成一个水桶，水桶只有顶端可以倒水进去，栈的顶端被称为栈顶，相对的底端被称为栈底



我们的栈只能从栈顶插入和删除数据，所有的原始数据类型都是存放在栈内存中。

向栈插入新元素，又称为进栈/入栈/压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素

例：先后定义三个变量a、b、c，这三个变量都只能从箭头所指的方向入栈，我们定义的三个变量在栈内存中，最先放入的是变量a，其次是变量b，而变量c就是我们新的栈顶元素。

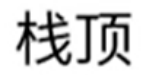


```
1 // 定义变量
2 var a = 10;
3 var b = 'hello';
4 var c = true;
```

那么，定义变量到底发生了什么？以定义变量a为例：

首先，我们会将10存入内存空间

入栈



栈底

之后，在我们当前的作用域中声明一个变量a（此时只是声明了变量，并没有赋值，此时a的默认值是undefined）

入栈



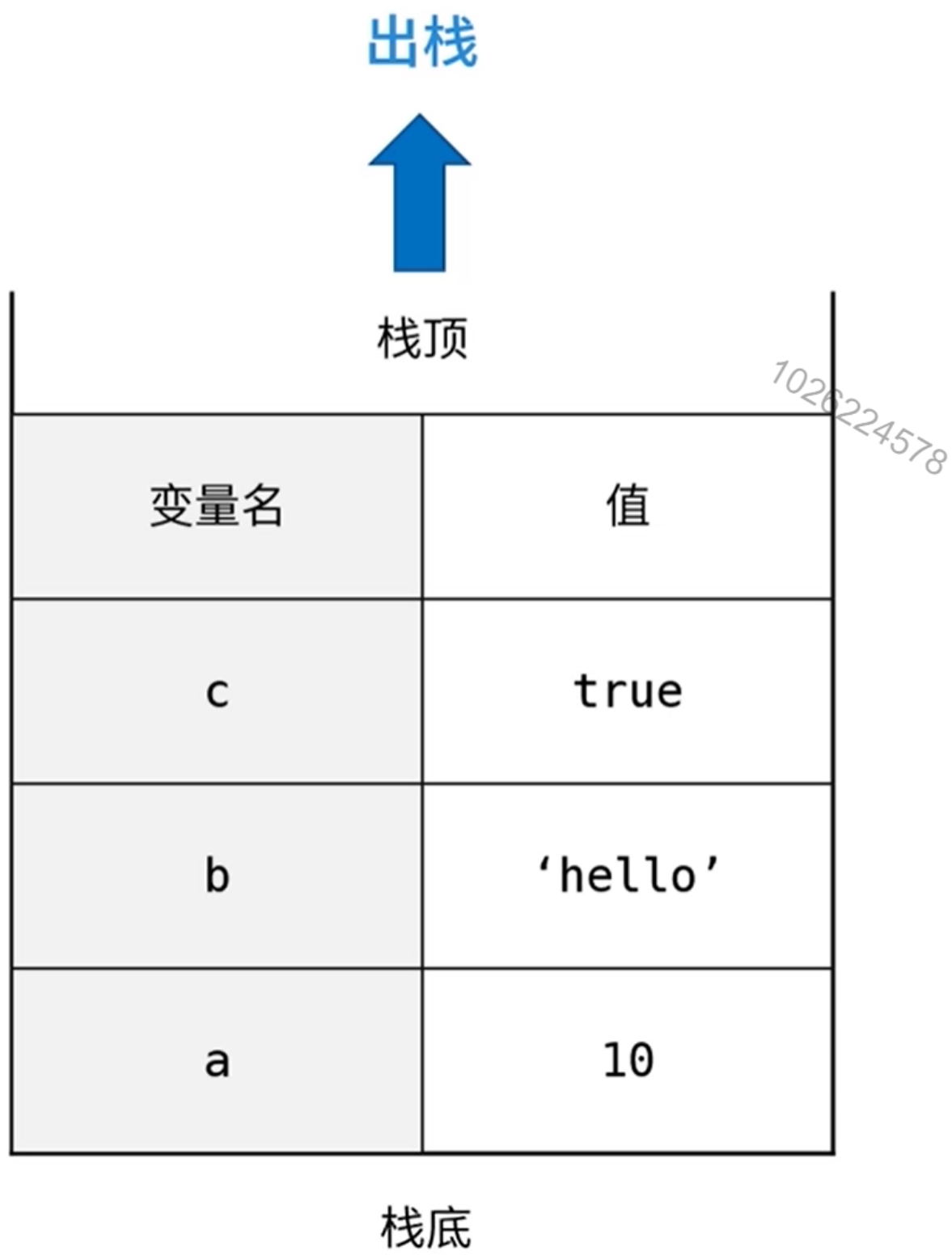
栈顶

变量名	值
a	10

栈底

当声明完了变量a以后，我们才会对变量进行赋值，赋值的过程只是将变量a与10进行一个关联，当变量a与10关联完成以后，定义变量的过程才算结束。

从一个栈删除元素又称出栈/退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素，出栈的顺序是先进后出



也就是说，在栈内存中，最先删除的是最后入栈的变量c

出栈



栈顶

变量名	值
b	'hello'
a	10

栈底

接着是删除变量b

出栈



栈顶

变量名	值
a	10

栈底

最后才会删除最先入栈的变量a

出栈



栈顶

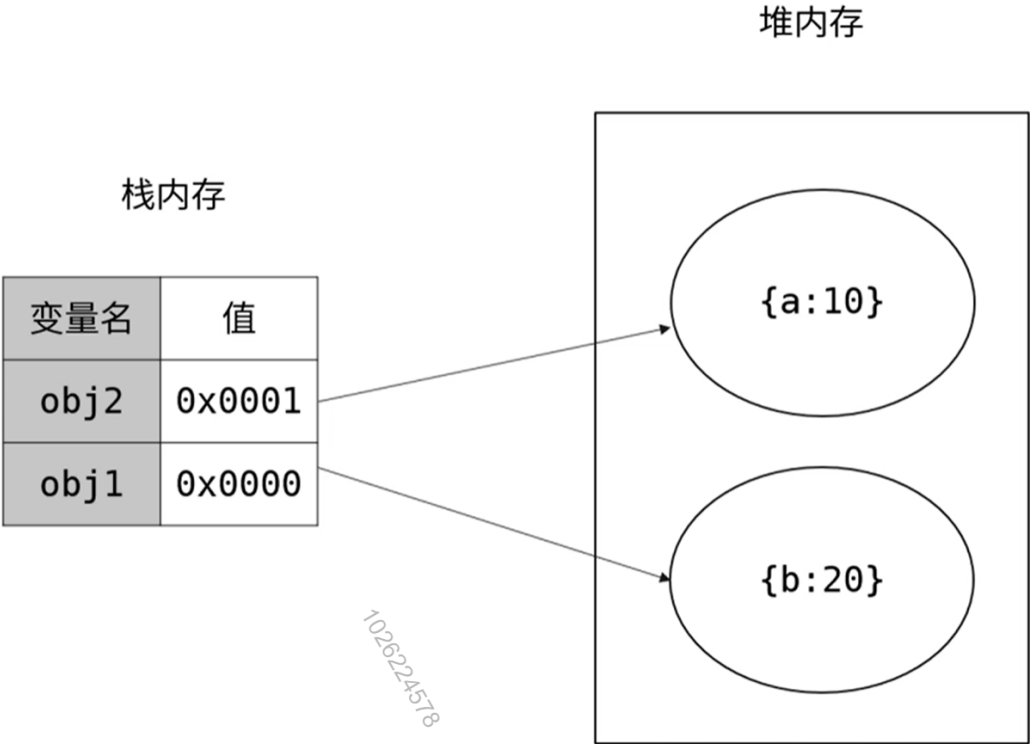


栈底

总结：栈空间先进后出，后进先出

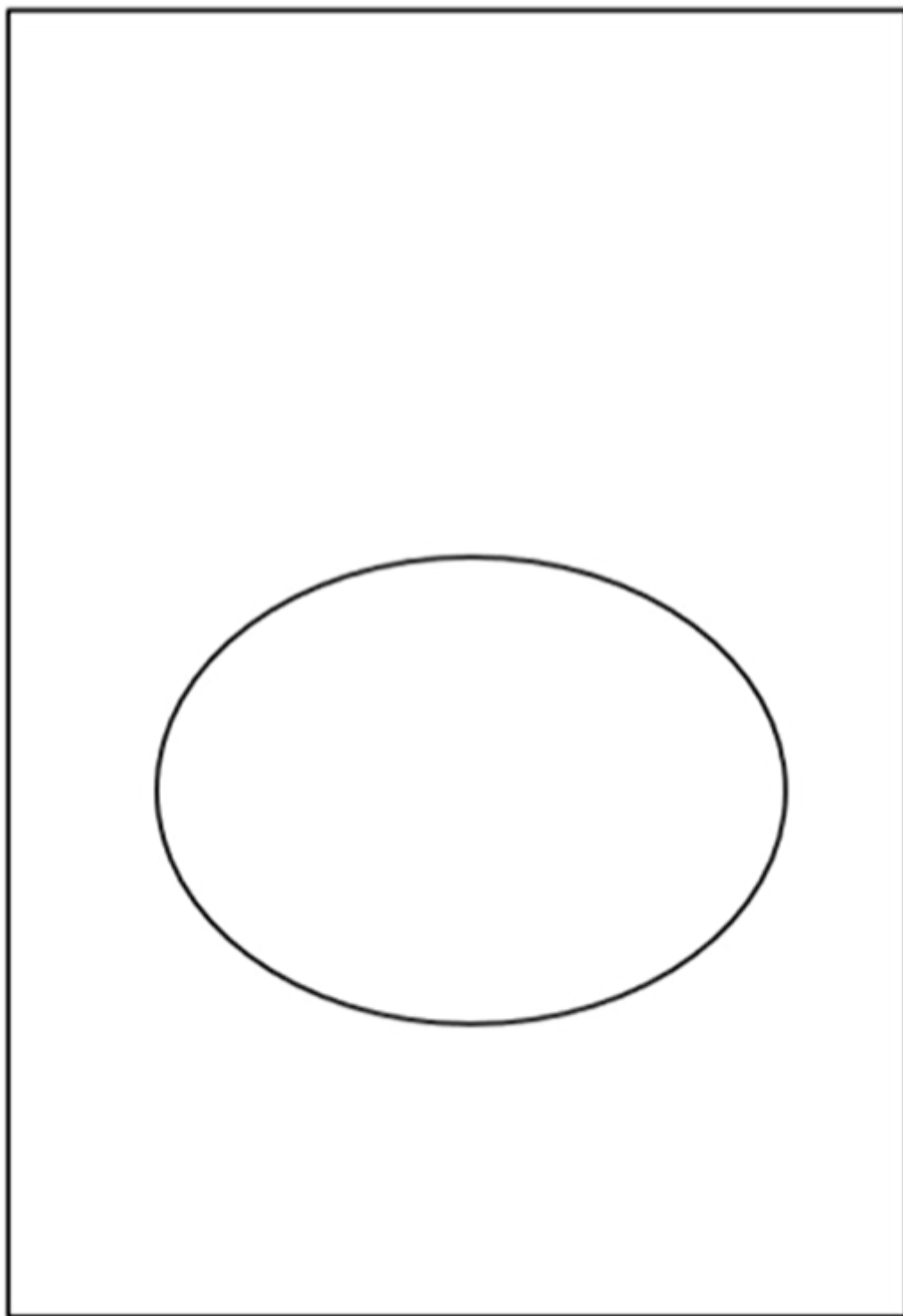
堆内存

JavaScript中的引用数据类型比如数组、对象，它们的值的大小是不固定的，相对于原始数据类型在栈内存中存储，引用类型是在堆内存中存储。JavaScript是不允许直接访问堆内存的，因此，我们也无法直接操作对象的堆内存空间，我们在操作对象时，实际上是在操作对象的引用而不是实际的对象，因此，引用类型的值都是按引用访问的，这里的引用我们可以粗浅的认为保存在栈内存中的一个内存地址，该地址与堆内存的实际值相关联，所以引用数据类型的值是保存在堆内存中的对象。



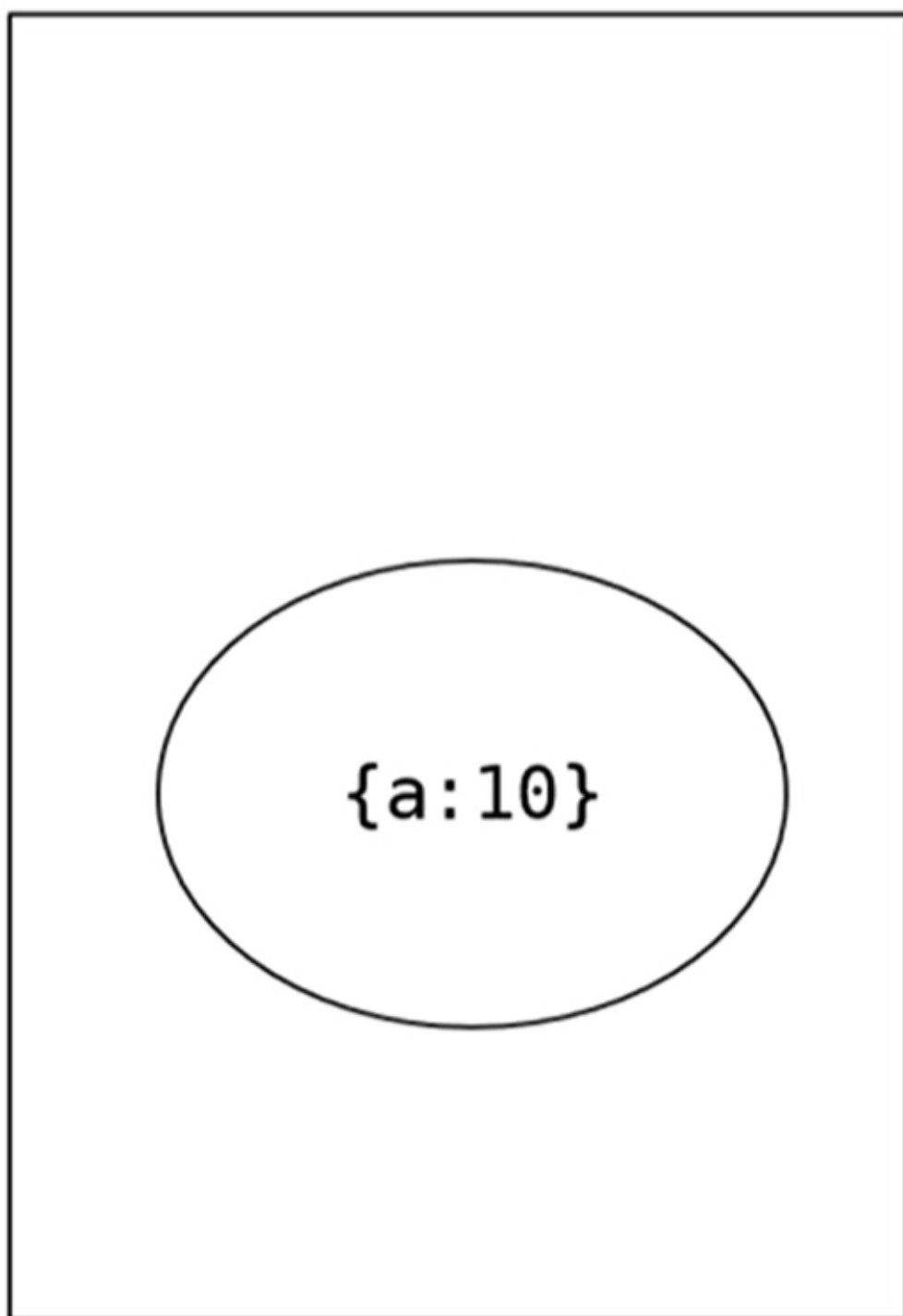
当JavaScript遇到引用类型时，比如对象，因为对象的不确定性，当我们需要存储对象的时候，首先会在堆内存中开辟一块空间

堆内存



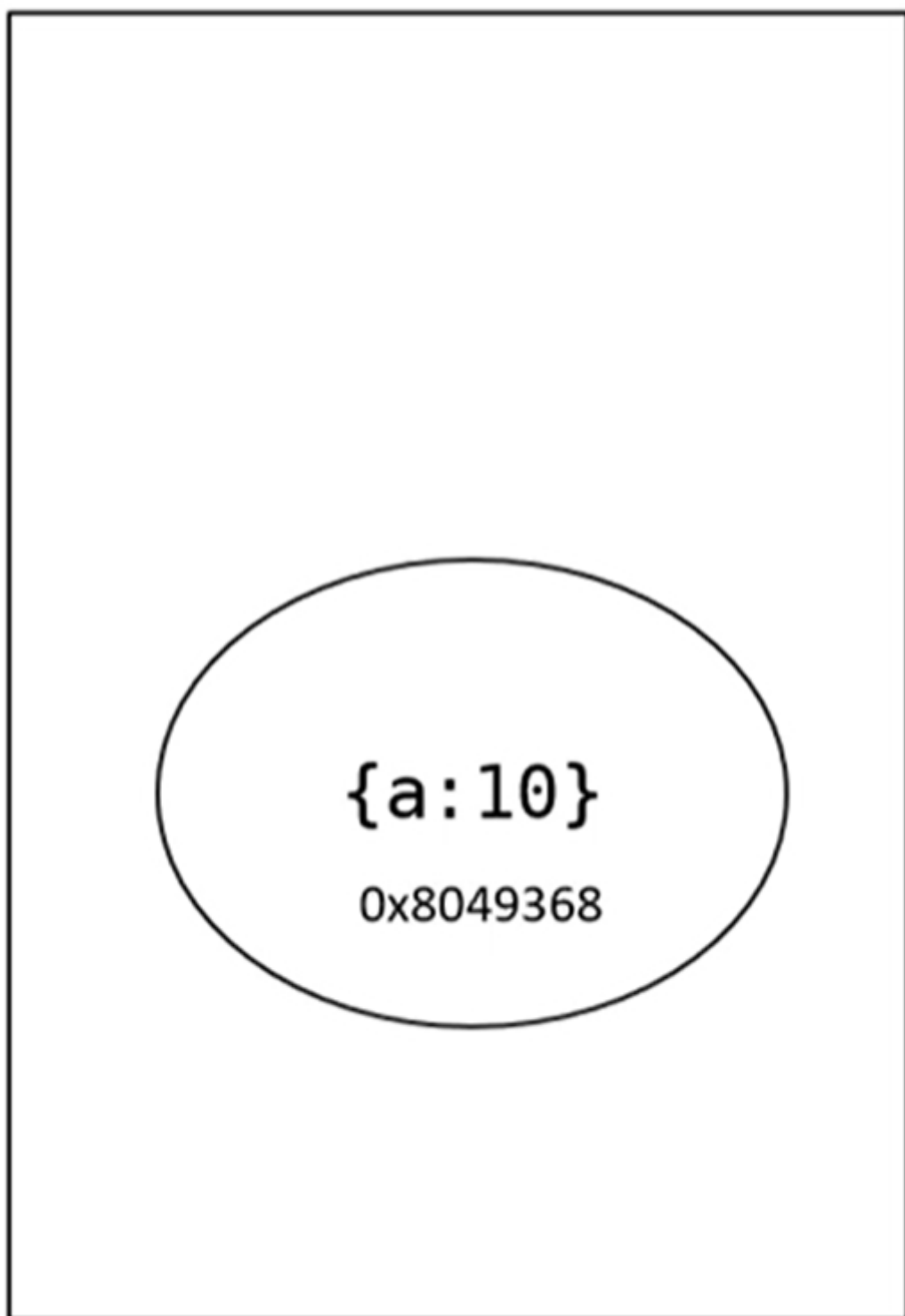
它会以键值对的形式将我们的对象存入到堆内存里

堆内存



这个堆内存会有一个16进制的内存地址，假设它的内存地址是0x8049368

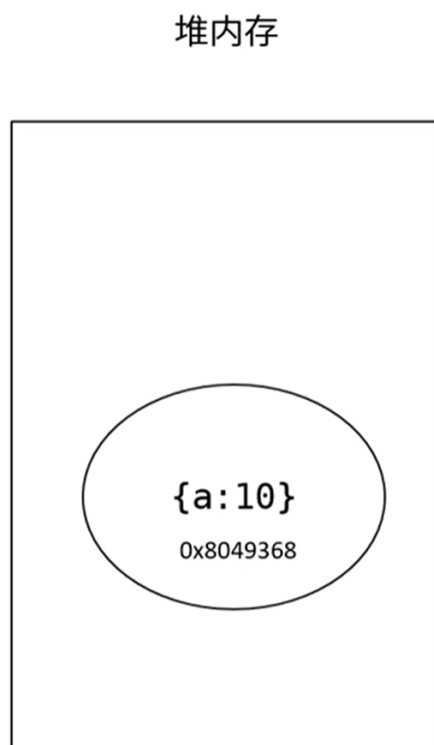
堆内存



之后我们在栈内存中声明一个变量obj1

栈内存

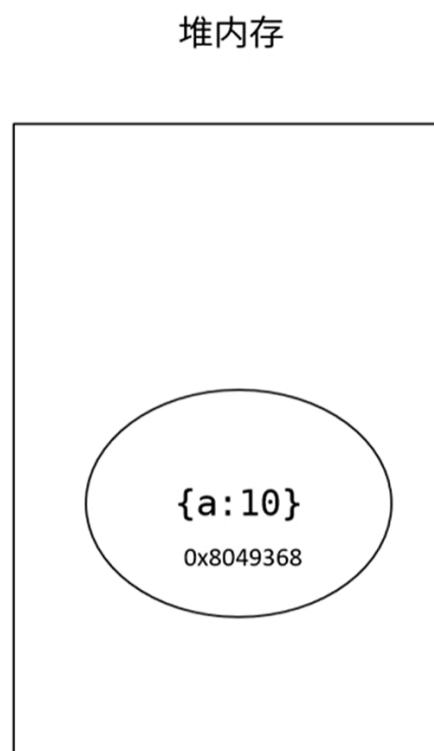
变量名	值
obj1	



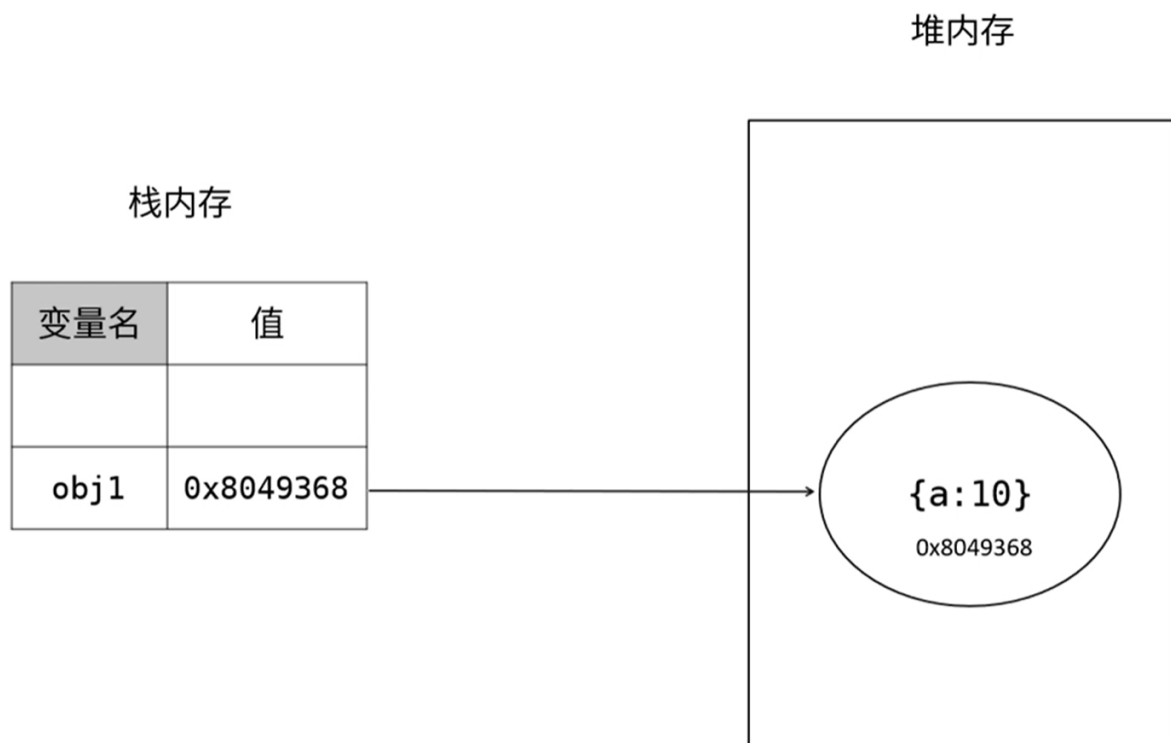
它的值就是我们刚才的堆内存地址

栈内存

变量名	值
obj1	0x8049368



此时我们通过变量obj1就可以访问我们的对象了

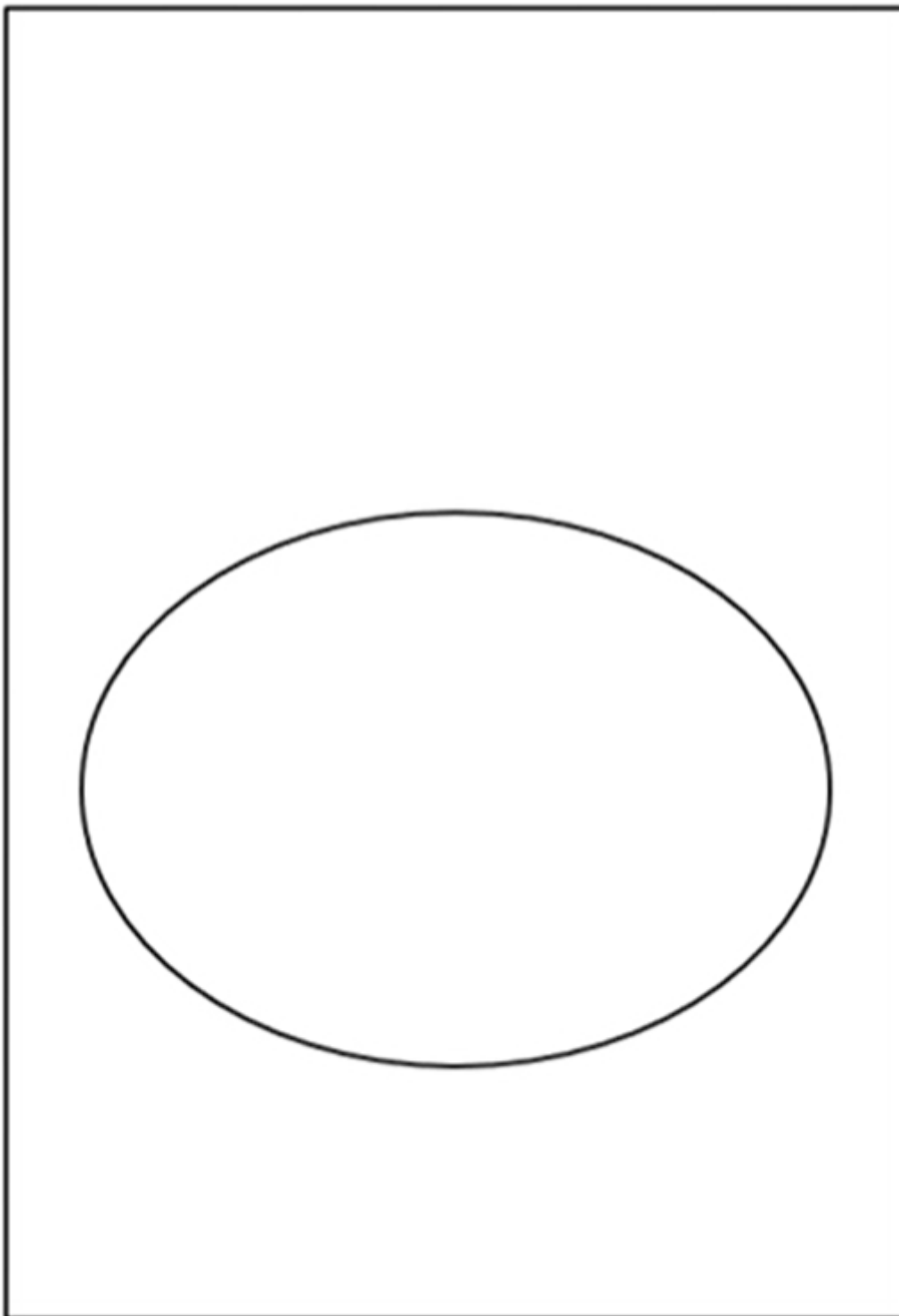


我们知道，函数也是引用类型，当我们定义一个普通函数的时候，我们的内存到底是如何工作的呢？

```
1 // 定义函数
2 function fn() {
3   var i = 10;
4   var j = 10;
5   console.log(i + j);
6 }
```

1、当我们定义函数时，首先会在堆内存中开辟一块内存空间

堆内存



2、将函数体中的代码以字符串的形式存储到内存空间中

堆内存

“代码字符串”

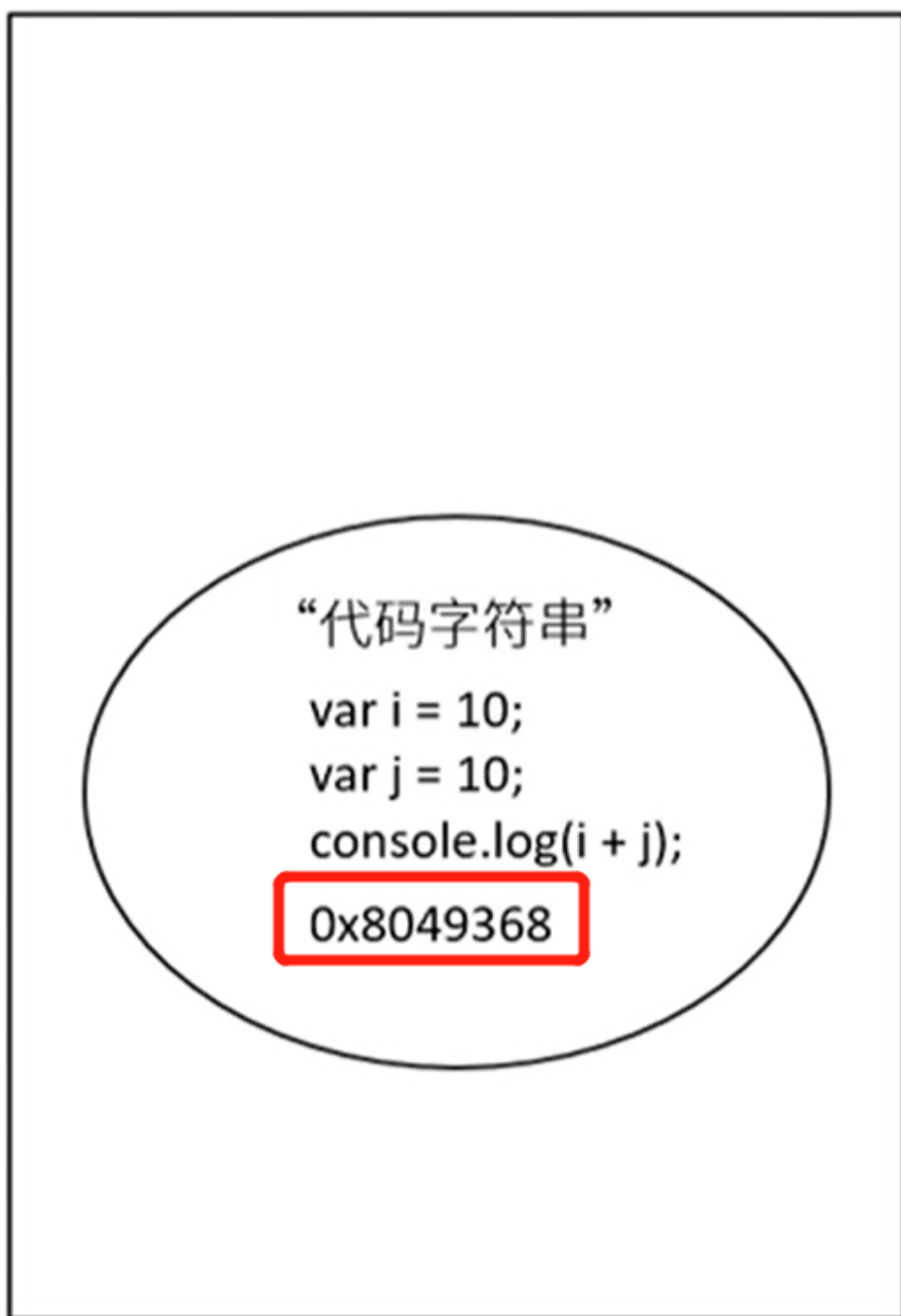
```
var i = 10;
```

```
var j = 10;
```

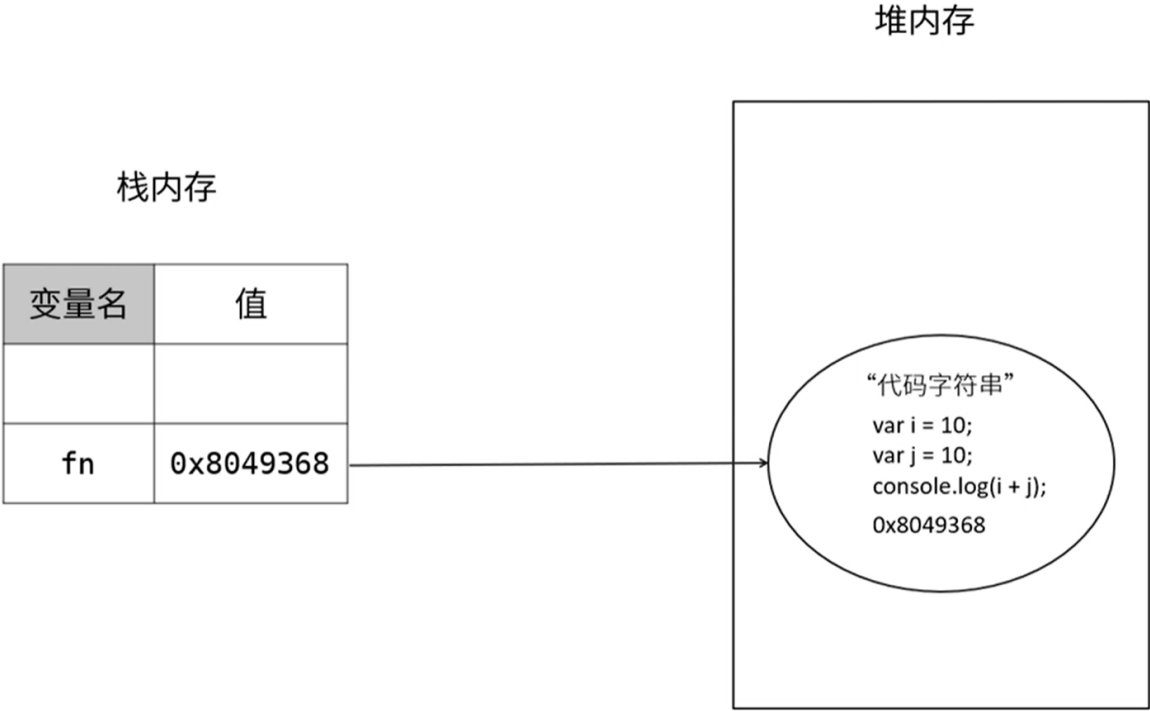
```
console.log(i + j);
```

这个空间也会有一个16进制的内存地址，假设这个内存空间的地址为0x8049368

堆内存

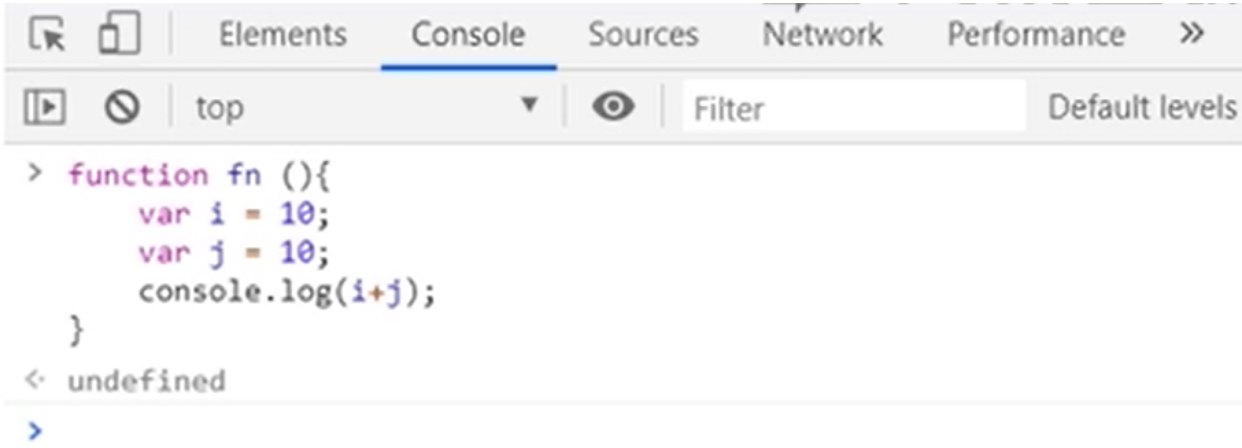


3、之后把0x8049368这个内存地址赋值给函数名，所以我们可以把函数理解为变量，此时的函数仍然以字符串的形式存在于我们的堆内存中

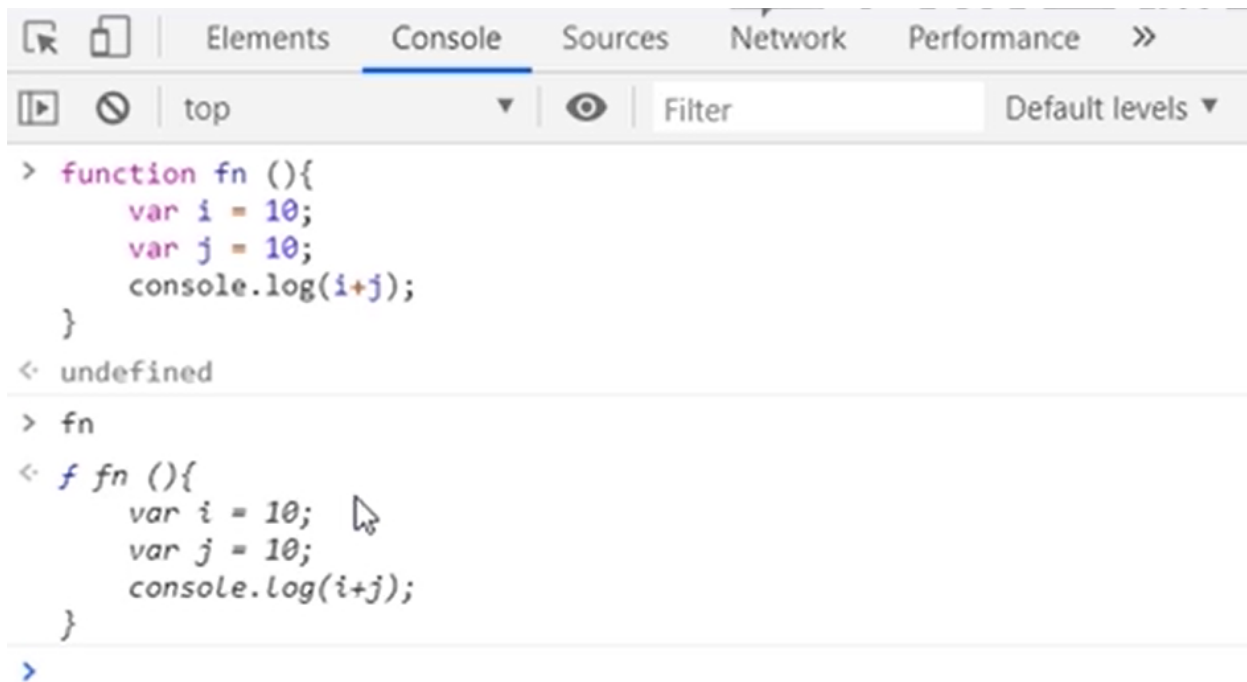


简单验证一下：

1、打开chrome控制台，将代码粘贴至控制台



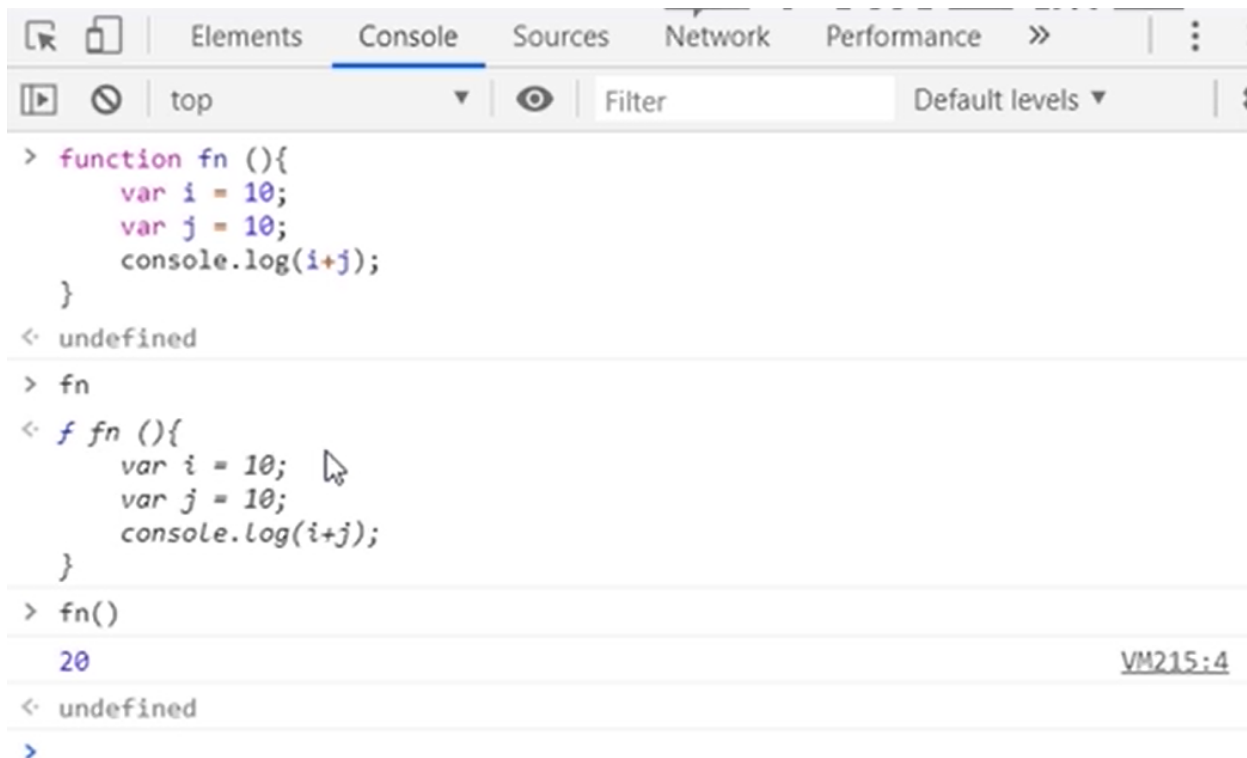
2、当我们调用这个函数的时候，如果我们不加括号，此时它只是找到函数体，也就是找到堆内存中的代码字符串



```
> function fn (){
  var i = 10;
  var j = 10;
  console.log(i+j);
}
< undefined

> fn
< f fn (){
  var i = 10;
  var j = 10;
  console.log(i+j);
}
```

3、只有当我们加上括号调用的时候，这个函数才会真正被执行



```
> function fn (){
  var i = 10;
  var j = 10;
  console.log(i+j);
}
< undefined

> fn
< f fn (){
  var i = 10;
  var j = 10;
  console.log(i+j);
}

> fn()
20
VM215:4
< undefined

>
```

这个执行的过程是，首先找到堆内存中的函数体部分，之后再将这个函数体字符串转变成我们的js代码执行，如果定义的函数不执行，这个函数其实就是以字符串的形式躺在我们的堆内存中。

总结：堆的存取是随意的,闭包中的变量并不保存中栈内存中，而是保存在堆内存中。(闭包就是内层函数可以访问外层函数的作用域) 这也就解释了函数调用之后之后为什么

么闭包还能引用到函数内的变量。

三、垃圾回收

当计算机上的动态内存不再需要时，就应该予以释放，以让出内存。在JavaScript中，垃圾回收是JavaScript引擎找出那些不再继续使用的变量，然后释放其所占用的内存。

我们知道，程序是运行在内存里的，当声明一个变量，定义一个函数时都会占用内存，而内存的容量是有限的，如果变量、函数只有产生没有消亡的过程，那内存就会被完全占用。当内存被完全占用时，不仅我们的程序无法正常运行，连其他程序也会受到影响，所以，在计算机中，我们需要垃圾回收。

在JavaScript中，垃圾回收是一种自动的内存管理机制，需要注意的是，自动的意思是JavaScript引擎可以帮助我们回收内存垃圾，但并不代表我们可以不关心内存管理，如果我们操作不当，JavaScript中依旧会出现内存溢出的情况，在JavaScript中，内存管理是自动执行的，而且是不可见的，垃圾回收器会按照固定的时间间隔周期性的执行一次释放操作。

虽然说JavaScript使用的是自动垃圾回收机制来管理内存，但自动垃圾回收机制是一把双刃剑：

- 优势：可以大幅简化程序的内存管理代码，降低程序员的负担，减少因长时间运转而带来的内存泄露问题
- 不足：意味着程序员将无法掌控内存，JavaScript中没有暴露任何关于内存的api，我们无法强迫其进行垃圾回收，更无法干预内存管理

垃圾收集策略

1、引用计数

- 引用计数 (reference counting)

跟踪记录每个值被引用的次数，如果一个值的引用次数是0，就表示这个值不再用到了，因此可以将这块内存释放

- 原理

每次引用加一，被释放减一，当这个值的引用次数变成0时，就可以将其内存空间回收

代码示例：

```
1  var obj1 = {a: 10}; // {a: 10}的引用次数加1          0+1 = 1
2  var obj2 = obj1; // {a: 10}的引用次数加1             1+1 = 2
3  var obj3 = obj1; // {a: 10}的引用次数加1             2+1 = 3
4  obj1 = {}; // obj1赋值了新的内存地址，{a: 10}的引用次数减1      3-1 = 2
5  console.log(obj1); // obj1对{a: 10}没有引用了，输出 {}
6  console.log(obj2); // obj2有对{a: 10}的引用 输出 {a: 10}
7  obj2 = null; // {a: 10}的引用次数减1                  2-1 = 1
8  console.log(obj2); // obj2对{a: 10}没有引用了，输出null
9  console.log(obj3); // obj3有对{a: 10}的引用 输出 {a: 10}
10 obj3 = null; // {a: 10}的引用次数减1                  1-1 = 0 等待下一次垃圾收集器运行回收内存
11 console.log(obj1); // obj1对{a: 10}没有引用了，输出 {}
12 console.log(obj2); // obj2对{a: 10}没有引用了，输出 null
13 console.log(obj3); // obj3对{a: 10}没有引用了，输出 null
```

- 上面的第一行代码中，我们在堆内存中开辟一块新的内存空间存入了一个普通对象，变量obj1是对这个对象的引用，当声明一个变量并将一个引用类型的值赋值给这个变量的时候，这个引用类型的值的引用次数是1
- 第二行代码中，同一个引用值又被赋值给另一个变量，这个引用类型值的引用次数加1，所以是2
- 第三行代码中同第二行，引用次数再加1，等于3
- 第四行代码中，当包含这个引用值的变量又被赋值成另一个引用值了，那么这个引用类型值的引用次数减1，所以是2
- 第五行代码中输出{}
- 第六行代码中输出{a: 10}
- 第七行代码中，当包含这个引用值的变量赋值为null，表示清除这个变量对值的引用，所以第七行的引用次数减1，等于1
- 第八行代码中输出null
- 第九行代码中输出{a: 10}
- 第十行代码中，将obj3对引用值的引用次数减1，此时所有对{a: 10}的引用都没了，所以是0
- 第十一行代码中输出obj1的新值{}

- 第十二行代码中输出null
- 第十三行代码中输出null
- 当引用次数变为0时，说明没办法访问这个值了，当垃圾收集器下一次运行时，就会释放引用次数为0的值（{a: 10}）所占的内存空间

虽然说引用计数看起来可以解决我们的内存回收问题，但是这种机制有个非常严重的Bug，那就是循环引用。

所谓循环引用，就是变量之间相互引用，**循环引用会造成JS的垃圾回收机制无法回收内存，浪费大量性能。**

例：

```
1 // 循环引用
2 function fn() {
3     var obj1 = { a: 10 };
4     var obj2 = { b: 10 };
5     obj1.a = obj2; // obj1的a属性指向obj2
6     obj2.b = obj1; // obj2的b属性指向obj1
7 }
```

在本例中，obj1的a属性引用了obj2，obj2的b属性引用了obj1，也就是说obj1和obj2的引用次数都是2，当代码执行完毕以后，会将变量obj1和obj2都置为null，但是此时obj1和obj2这两个对象的引用次数并不是0而是1，所以并不会进行垃圾回收，但是这两个对象已经没有作用了，在函数外部也不可能使用到它们，所以就会造成内存泄露。假如这个函数被重复多次的调用，就会导致大量的内存得不到回收。

在现代浏览器中重复引用已经不是问题了，这个问题出现在IE8以及以前的时代，**现代浏览器更多是采用标记清除来实现垃圾回收机制。**

标记清除指的是当变量进入环境时，这个变量标记为“进入环境”，当变量离开环境时，则将其标记为“离开环境”，最后，垃圾回收器完成内存清除工作，销毁并回收那些被标记

为“离开环境”的值所占用的内存空间。目前主流浏览器都是使用标记清除式的垃圾回收策略，只不过收集的时间间隔有所不同。

这个环境就是我们所说的执行环境。

执行环境定义了变量或函数有权访问的其它数据，决定了它们各自的行为。我们可以把执行环境理解为我们人类的生存环境，如果人类脱离了生存环境就无法生存，在JavaScript中，我们的变量或者函数只有在当前的执行环境中才可以被访问，每个执行环境都有一个与之关联的变量对象（variable object），环境中定义的所有变量和函数都保存在这个对象中，虽然我们编写的代码无法访问到这个对象，但JS解析器在处理数据时会在后台使用它。

```
1 // 创建执行环境
2 function fn() {
3     // 执行环境内的变量和函数
4     var a = 1;
5     function b() {}
6 }
7
8 /*-----相当于-----*/
9
10 // 这里相当于执行环境关联的变量对象 (variable object)
11 // 只不过这个变量对象我们无法通过代码访问
12 // 但JS解析数据时会使用它
13 var fn = {
14     a: 1,
15     b: function () {}
16 }
```

我们的执行环境分为全局执行环境和局部执行环境

- 全局执行环境
 1. 它是最外围的一个执行环境
 2. 根据ECMAScript实现所在的宿主环境不同，表示执行环境的变量对象也不一样，例如在web浏览器中，全局执行环境的关联变量对象是window，在node中，全局执行环境的关联对象是global

3. 全局变量和函数都是作为window对象的属性和方法创建的

4. 某个执行环境中的所有代码执行完毕后，该环境被销毁，保存在其中的所有变量和函数也随之销毁，但是全局环境直到引用程序退出，例如关闭网页或是浏览器时才会被销毁。

- 局部执行环境

1. 每个函数都有自己的执行环境，我们称函数的执行环境为局部执行环境
2. 每当函数被调用时就会产生一个新的执行环境
3. 函数的执行环境会被推入一个环境栈中进行执行，而在函数执行之后，栈就会将这个环境弹出，把控制权返回给之前的执行环境。
4. 当浏览器第一次加载script的时候，它默认进入全局执行环境，如果你在全局代码中调用了一个函数，那么执行流就会进入到你调用的函数当中，创建一个新的执行环境，并且把这个执行环境添加到执行栈的顶部，当函数执行完毕，执行栈将这个执行环境弹出，并将执行权返回给之前的执行环境（全局环境）

例：

```
1 function foo() {  
2   var a = 10; // 被标记进入环境  
3   var b = 'hello'; // 被标记进入环境  
4 }  
5 foo(); // 执行完毕，a 和 b 被标记离开环境，内存被回收
```

在本例中，当变量a、b进入执行环境的时候，垃圾回收器将其标记为进入环境，当变量离开环境的时候也就是函数foo执行完成的时候，垃圾回收器将其标记为离开环境，而在函数执行结束后，执行栈将其执行环境弹出，把控制权返回给之前的执行环境。

四、V8引擎内存管理机制

限制内存的原因

- 浅层原因是由于V8最初为浏览器而设计，不太可能遇到大量内存的使用场景
- 深层原因是防止因为垃圾回收所导致的线程暂停执行的时间过长

V8的垃圾回收机制的限制，按照官方说法，以1.5G的垃圾回收为例，V8做一次小的垃圾回收需要50ms以上，做一次非增量式的垃圾回收甚至要1s以上，这里的时间指的是在垃圾回收中引起JavaScript线程暂停执行的时间，很明显这是无法接受的，因此V8直接限制了内存的大小。如果说我们需要在nodejs下操作大内存的对象，可以通过修改设置去完成，或者是避开这种限制，要知道1.5G是在V8引擎层面上做出的限制，我们可以使用Buffer对象，而Buffer对象的内存分配是在C++层面进行的，C++层面的内存分配不受V8引擎限制。

V8的回收策略

- V8采用了一种分代回收的策略，将内存分为两个生代：新生代和老生代
- V8分别对新生代和老生代使用不同的垃圾回收算法来提升垃圾回收的效率

在分代的基础上，新生代的对象主要通过.....（懵逼中，后面补上。。。）