对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

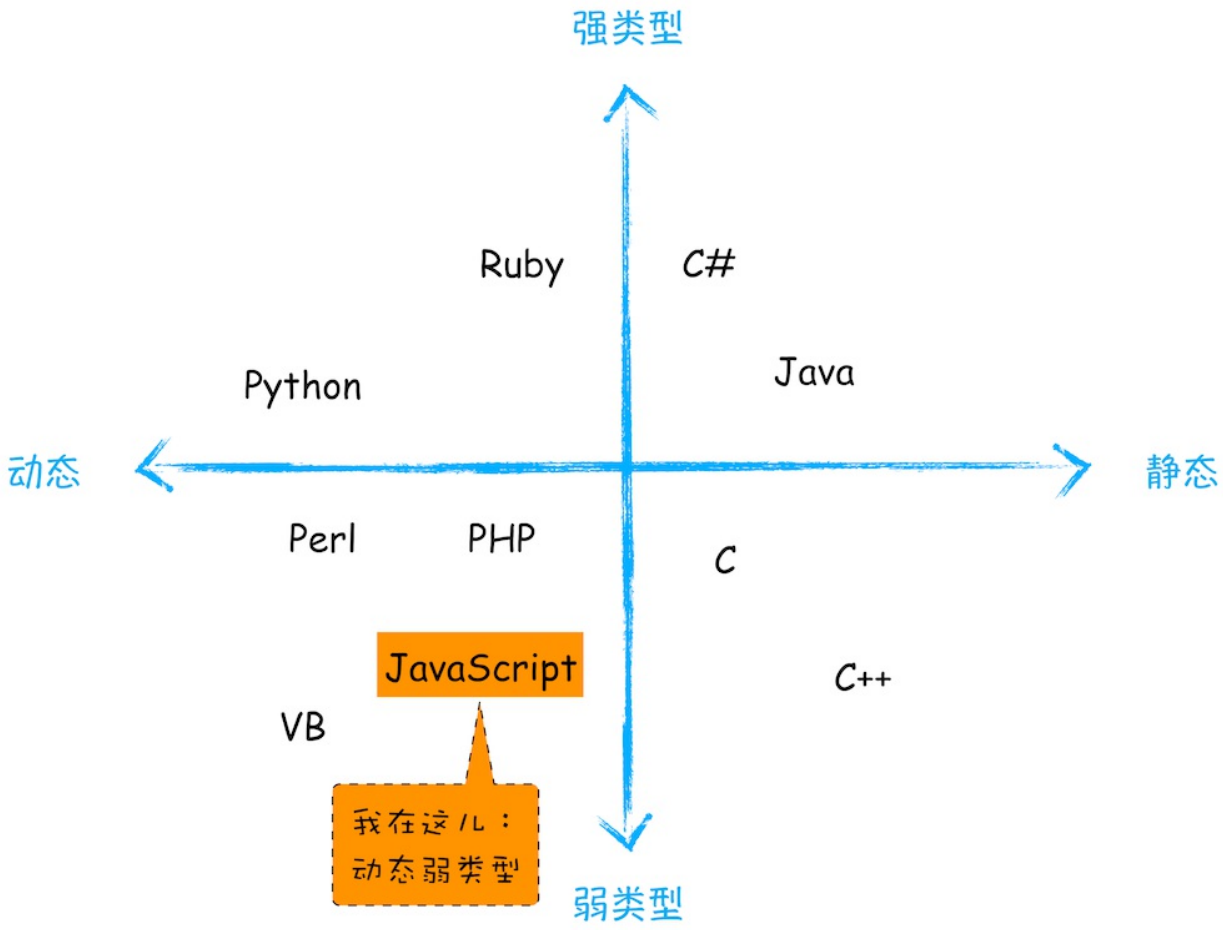
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

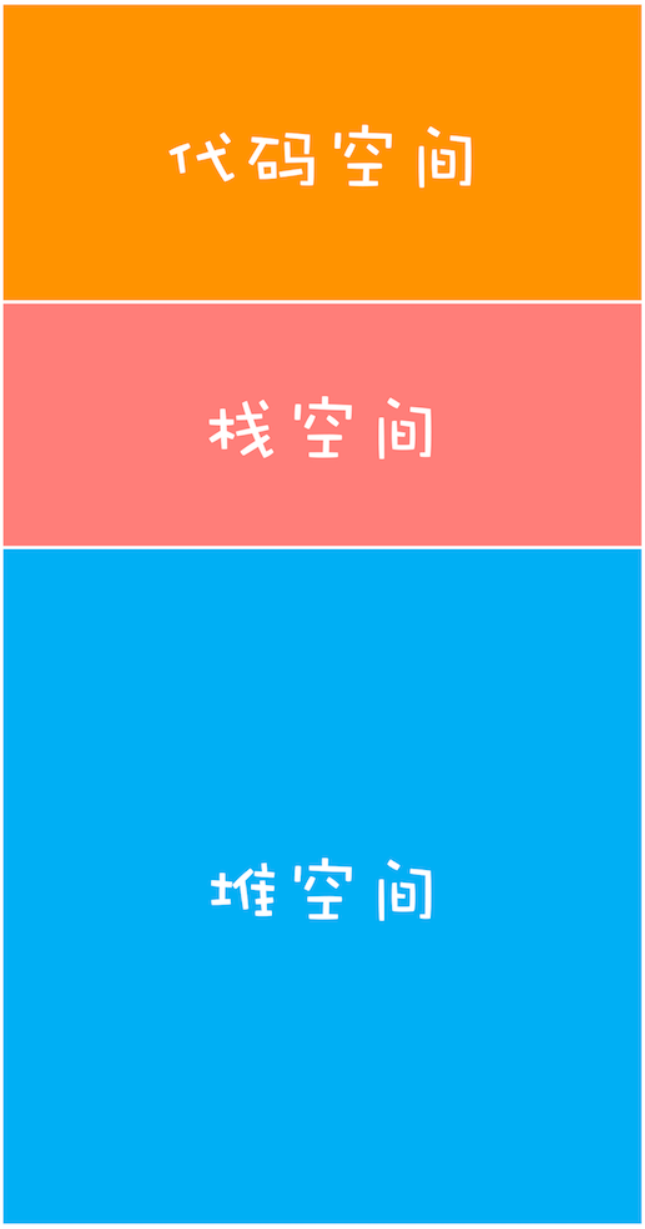
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

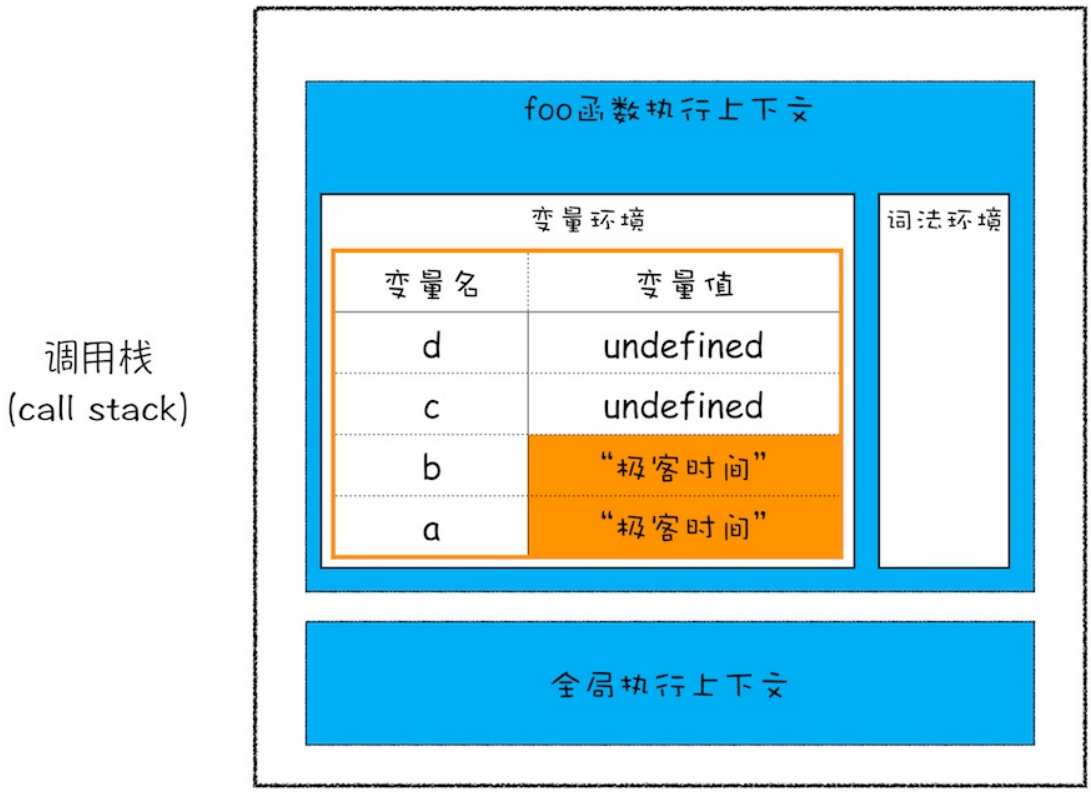
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

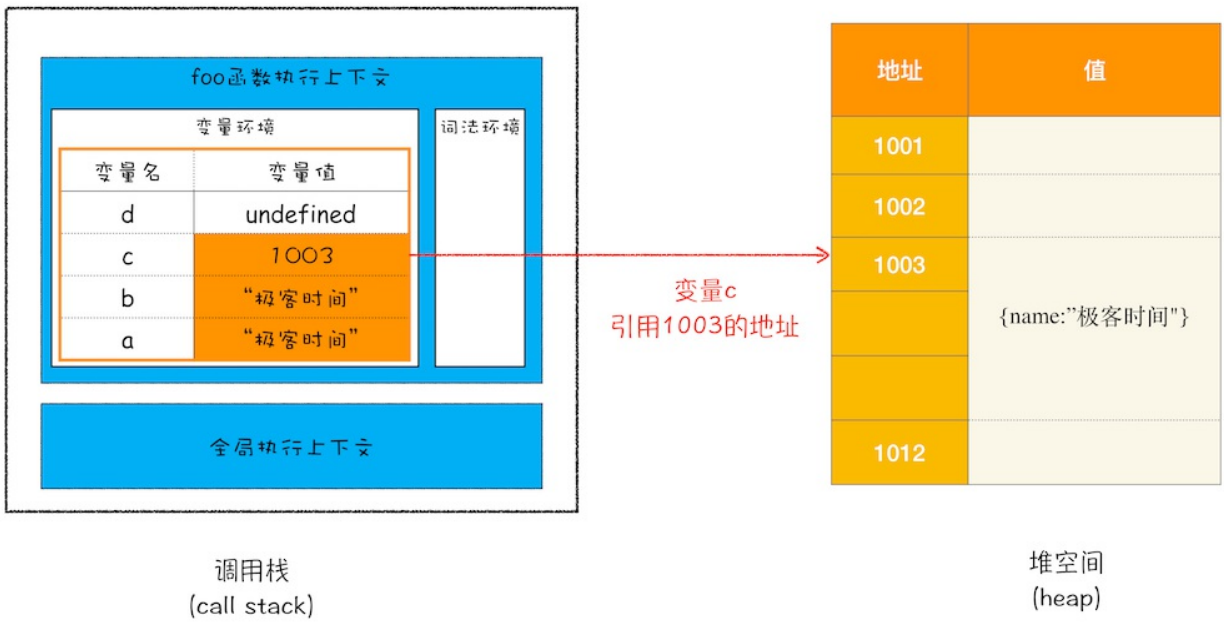
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

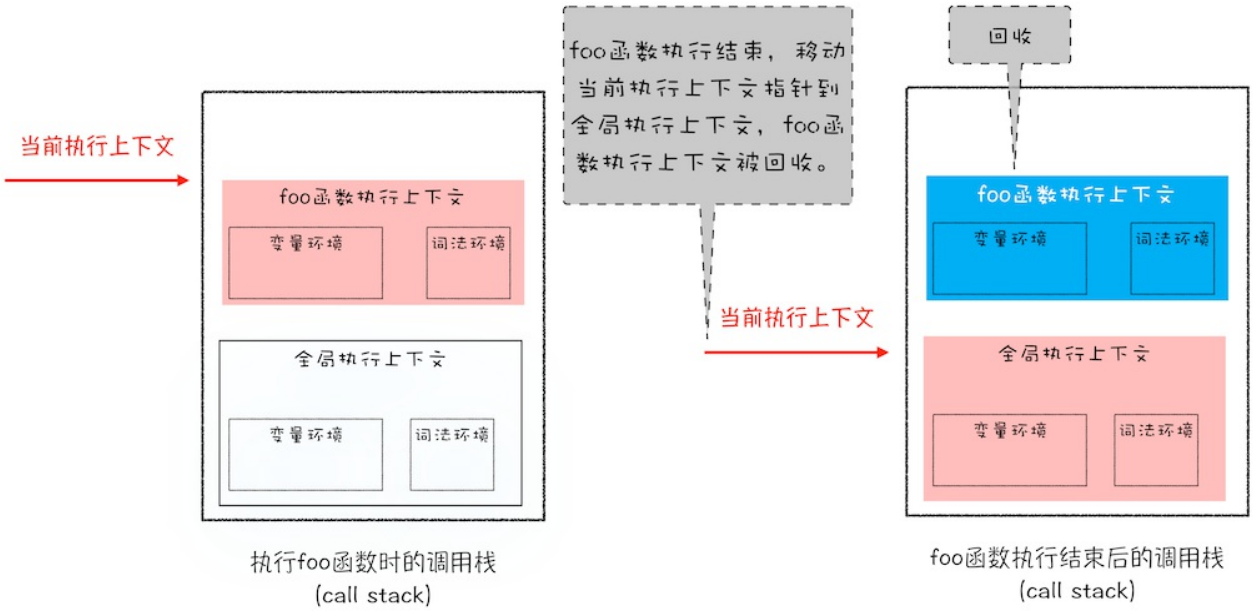


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



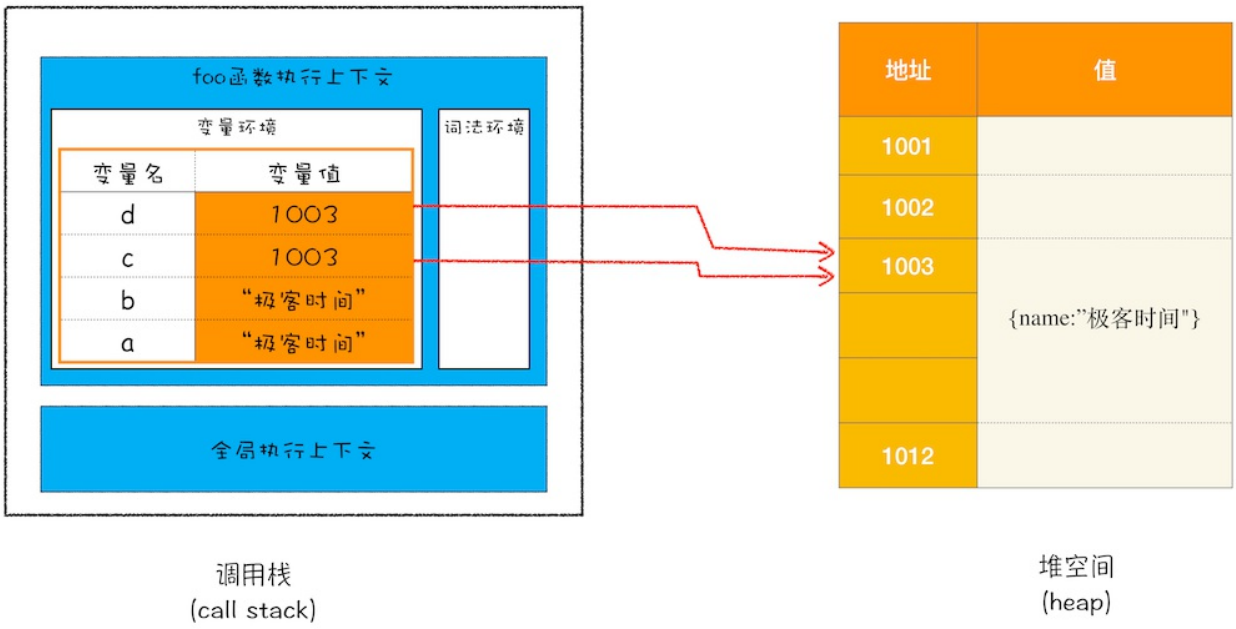
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

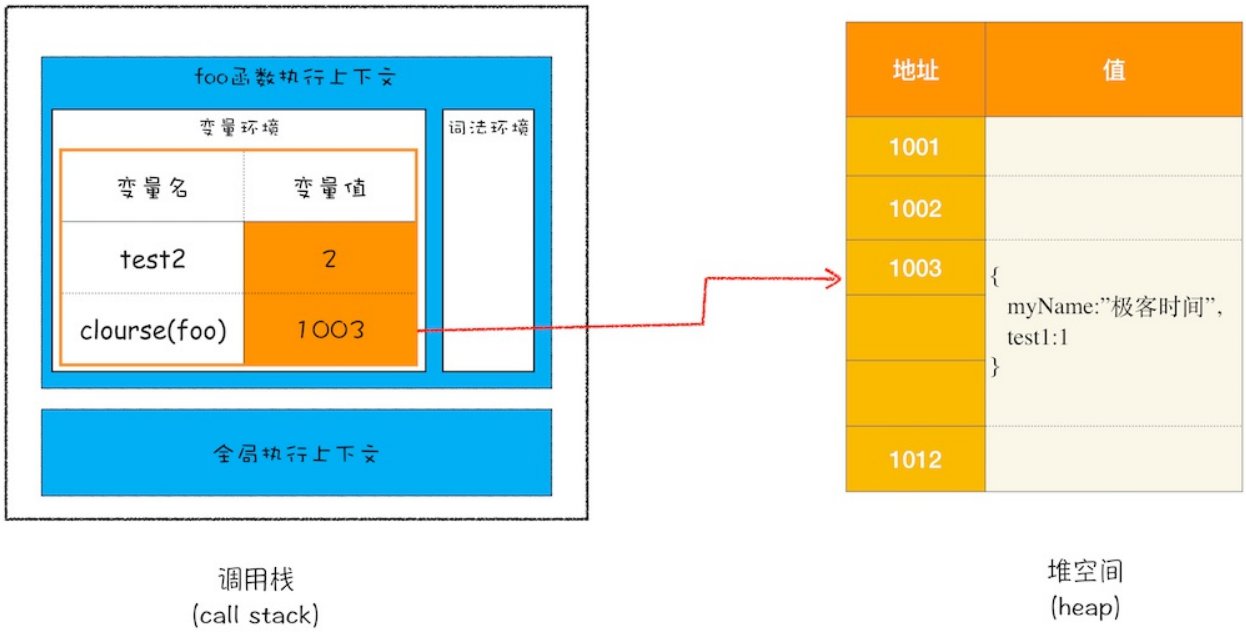
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“closure(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“closure(foo)”对象中。这时候堆中的“closure(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

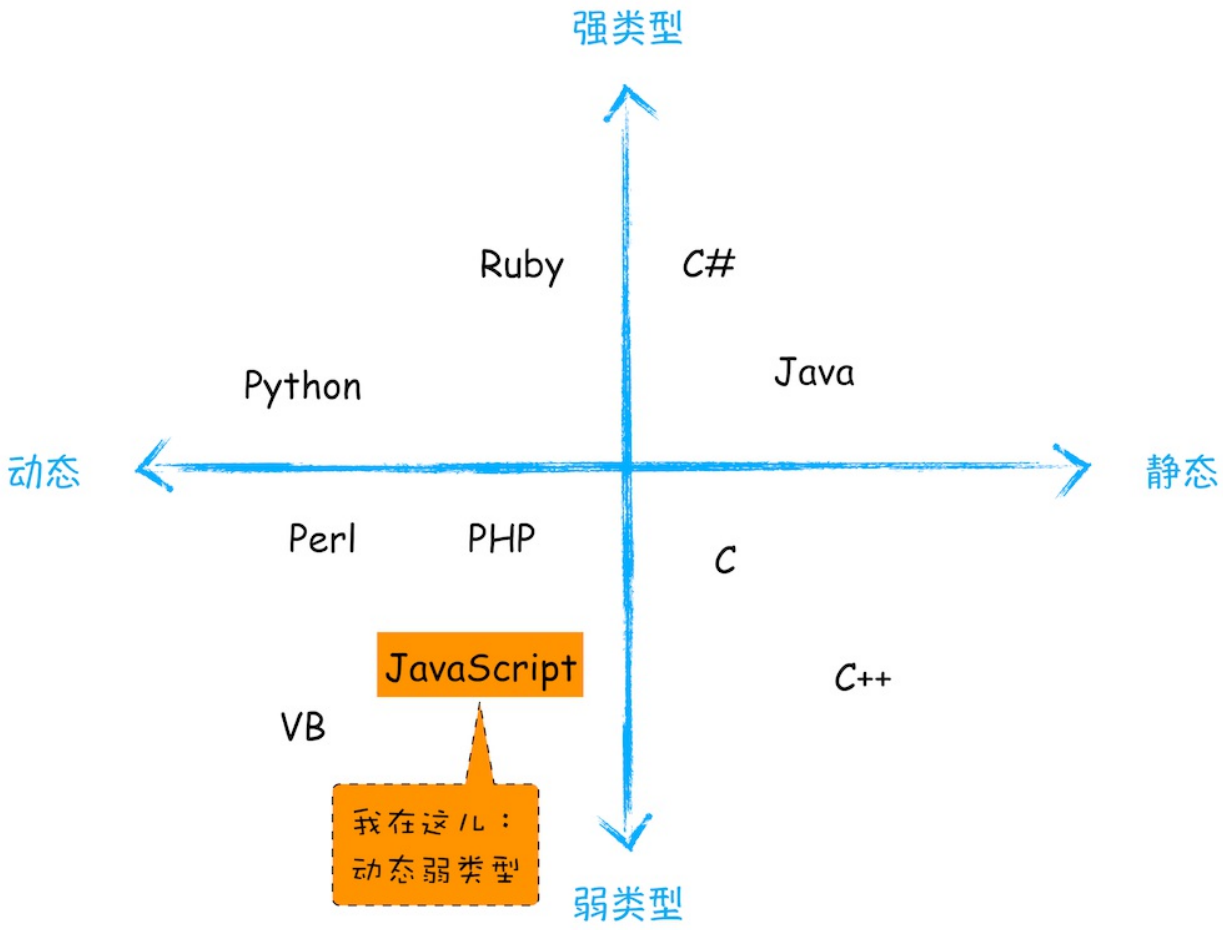
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

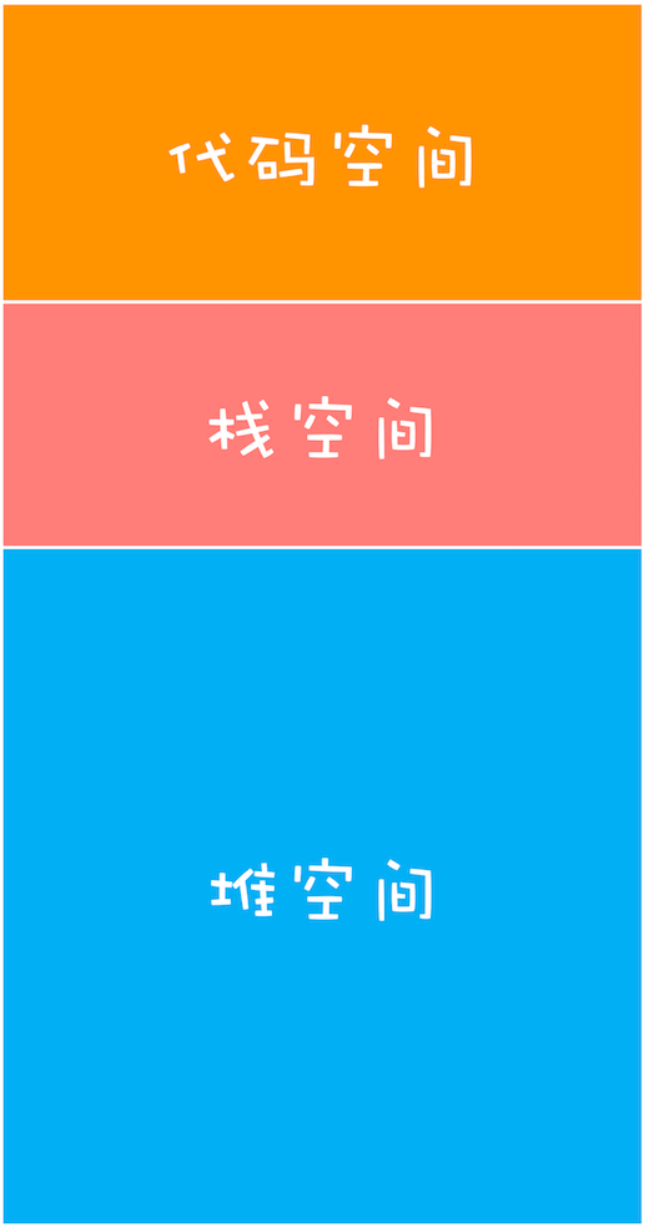
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

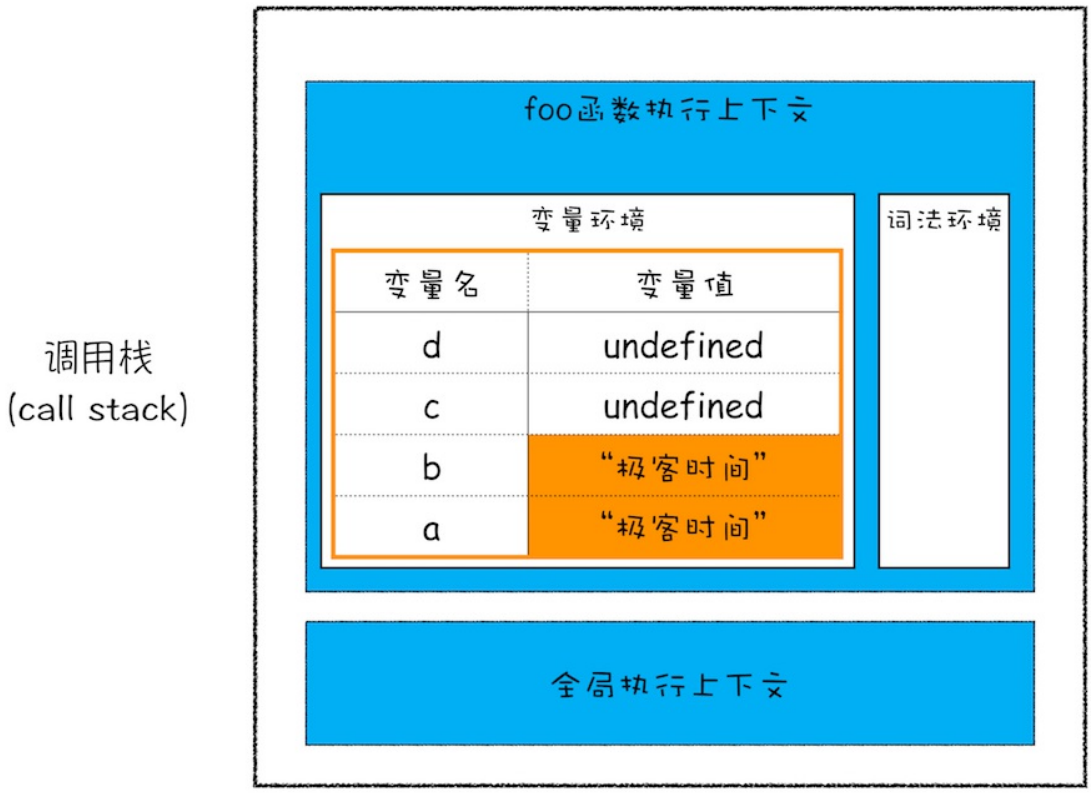
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

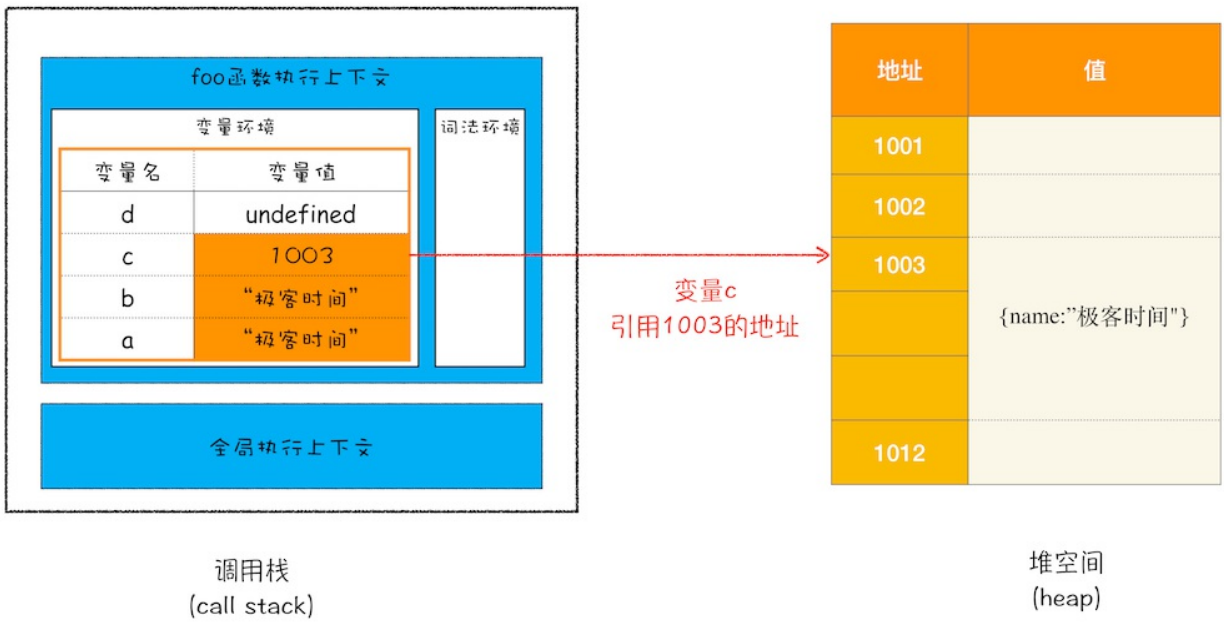
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

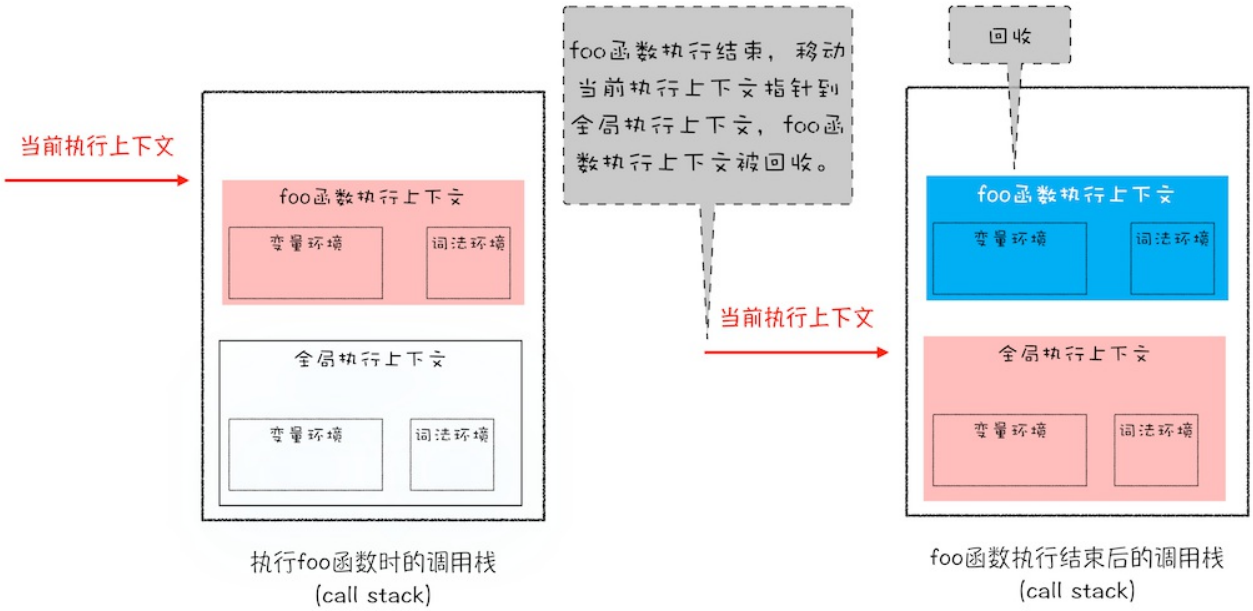


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



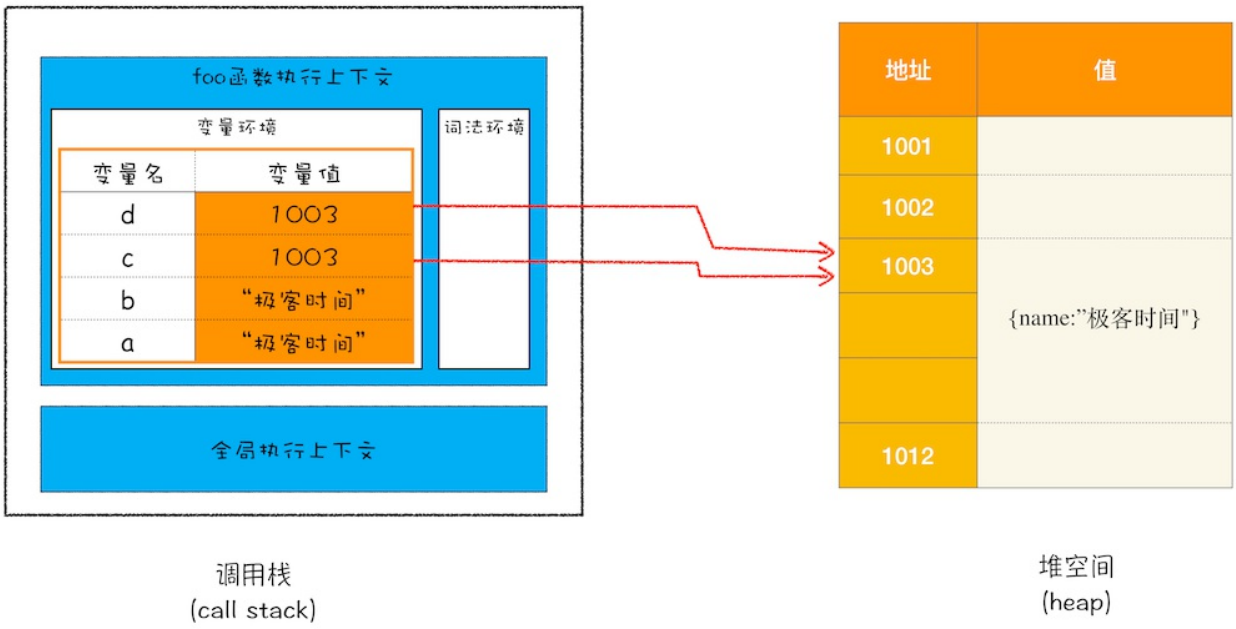
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

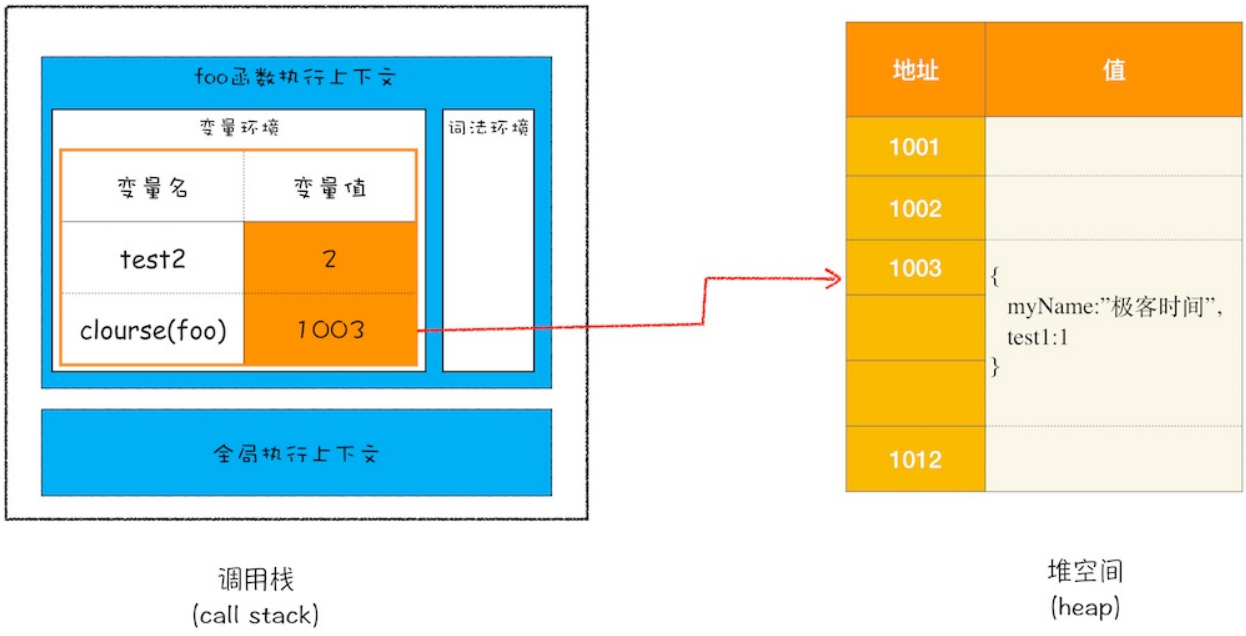
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

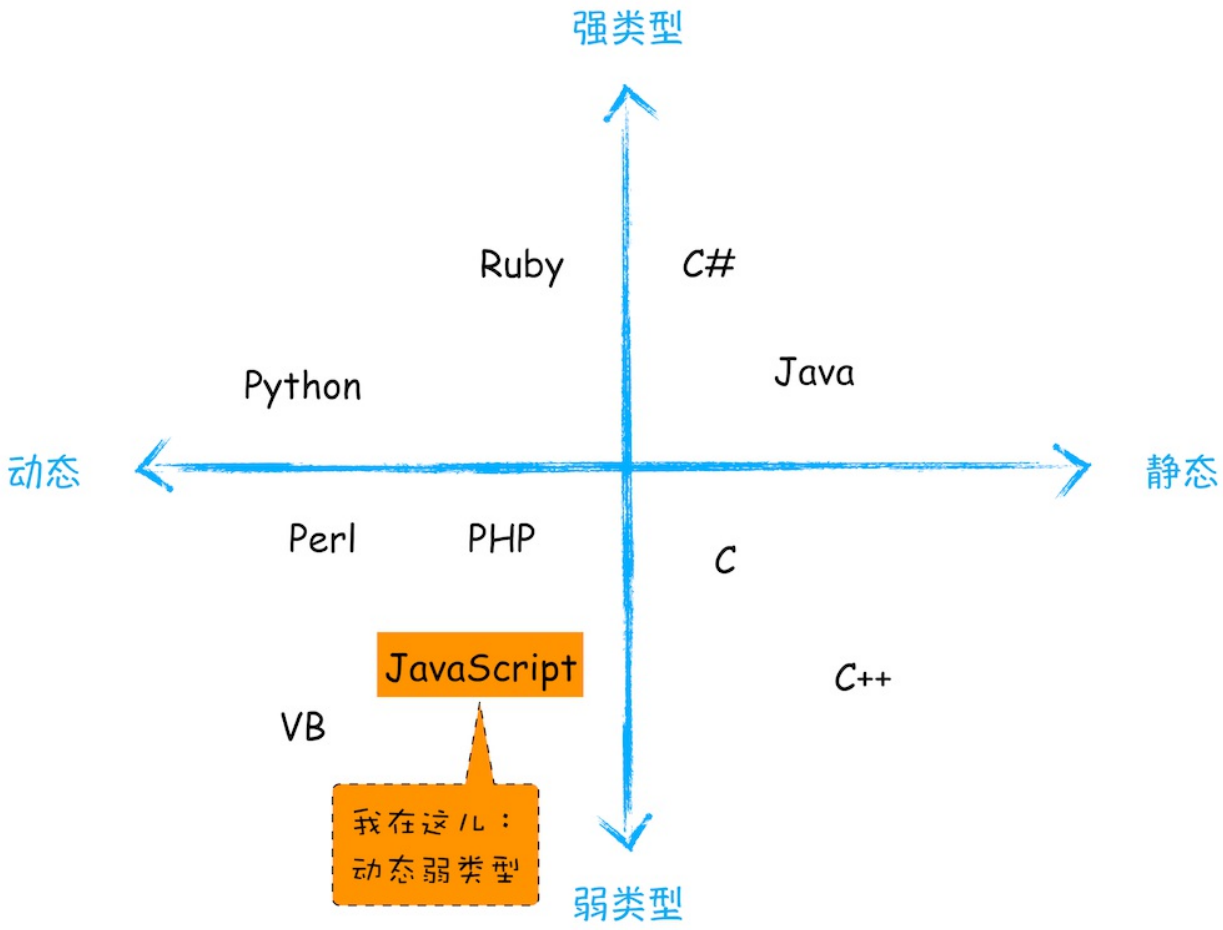
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

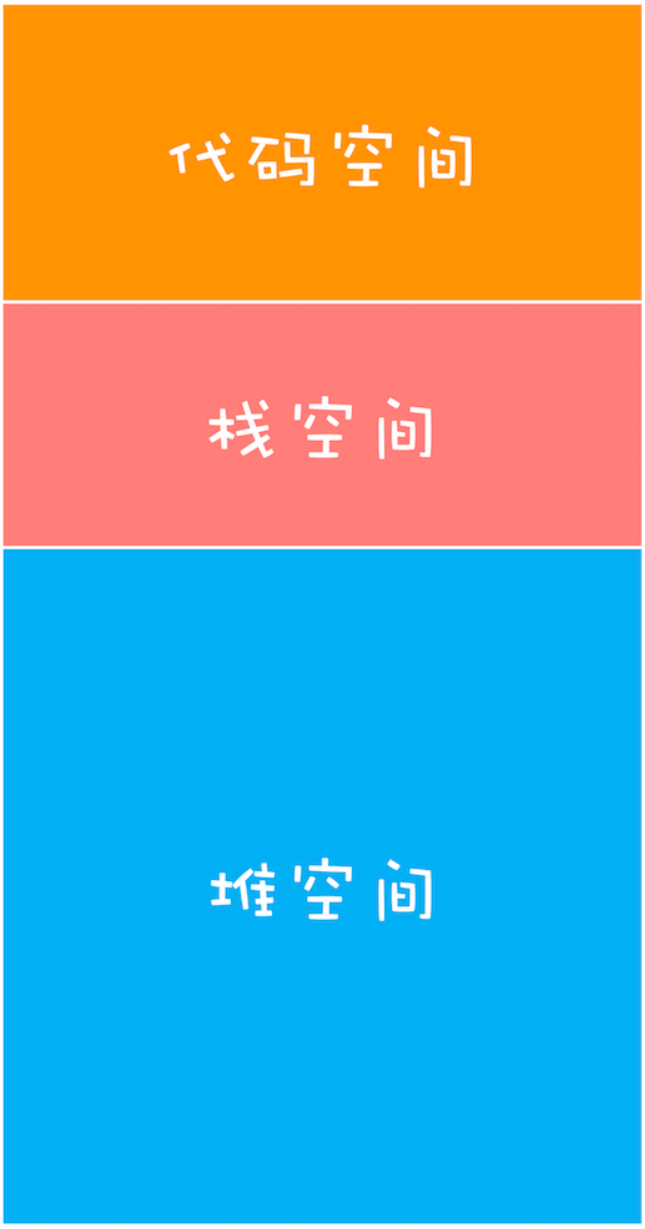
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

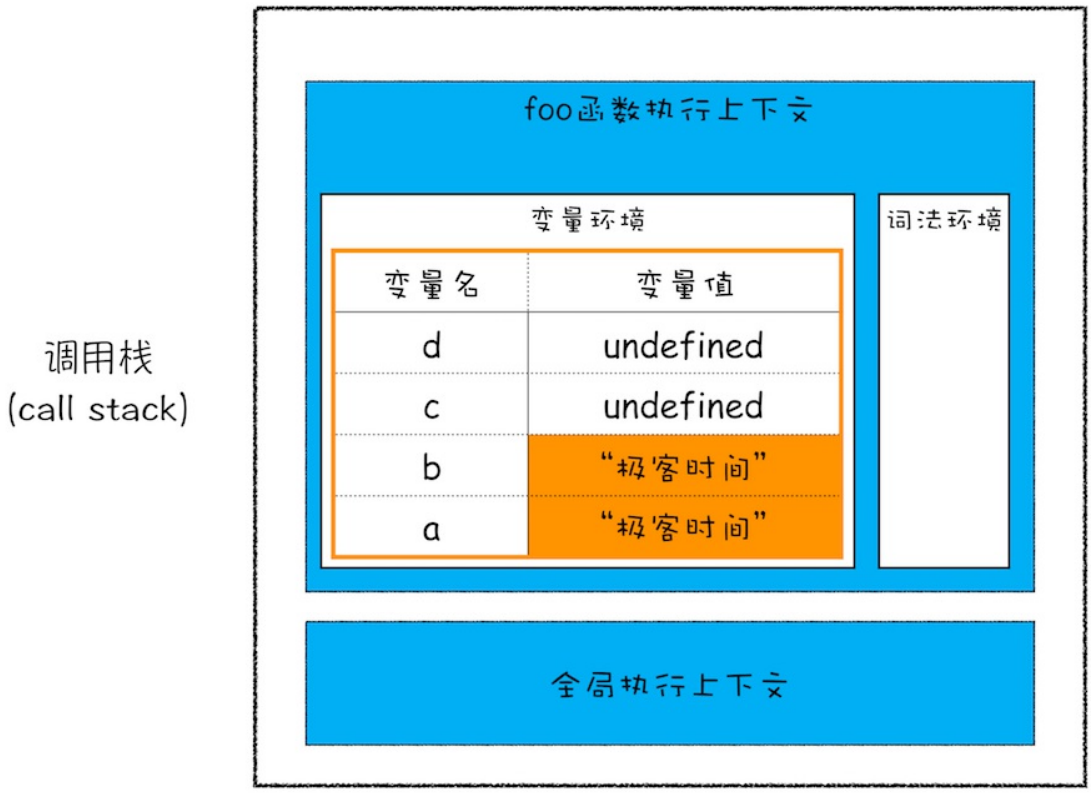
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

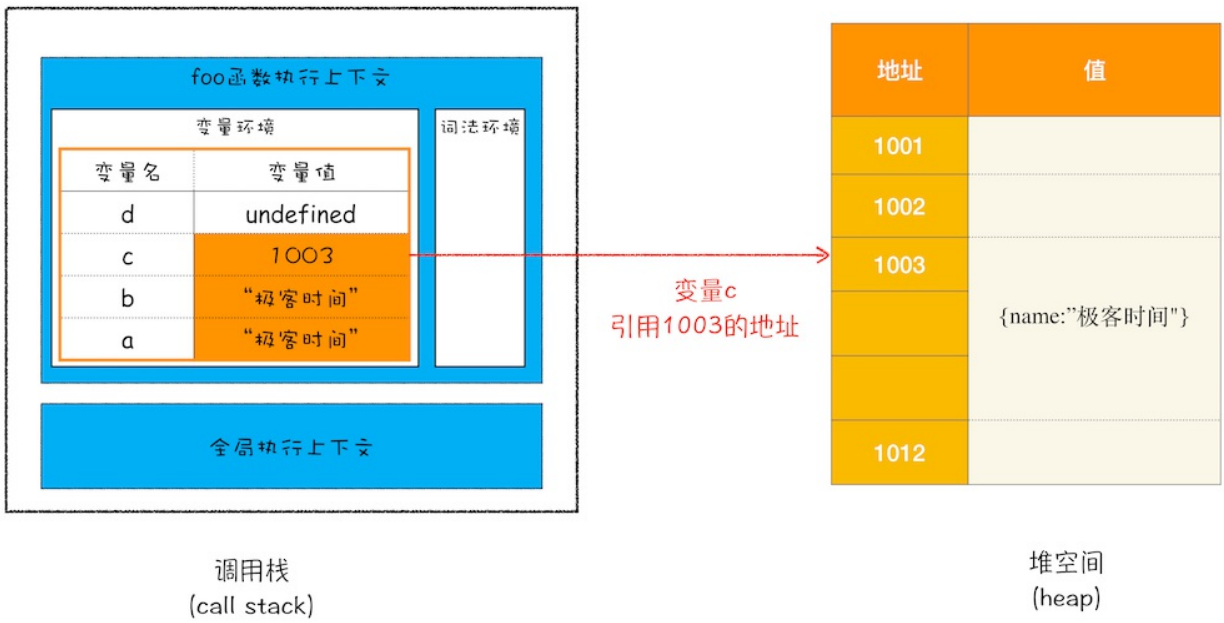
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

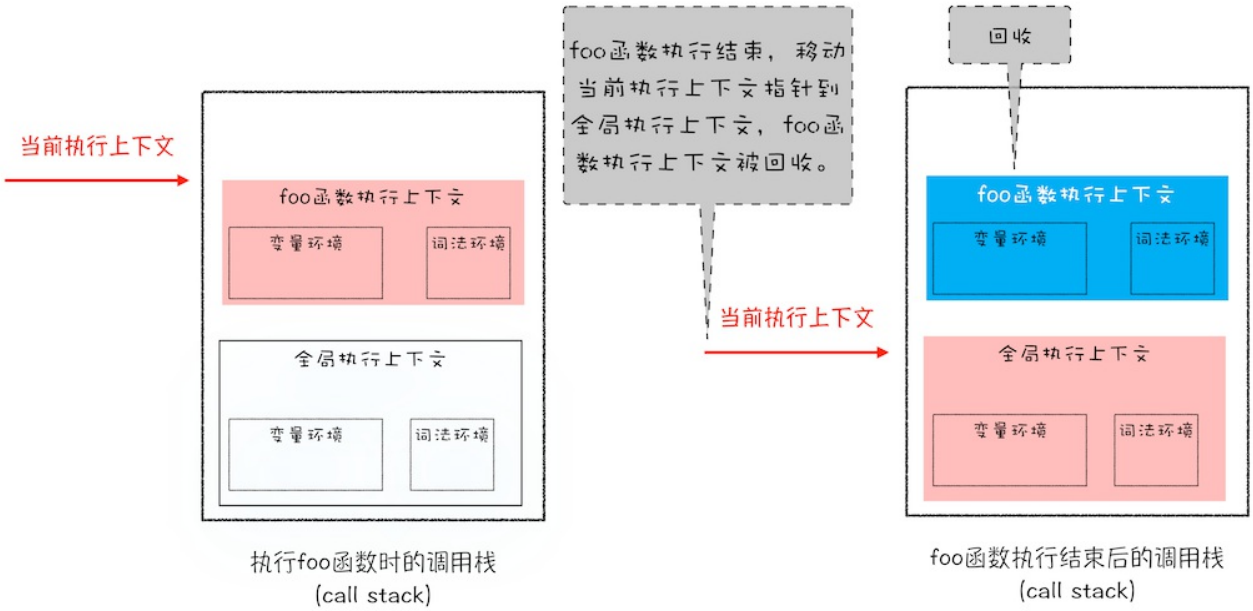


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



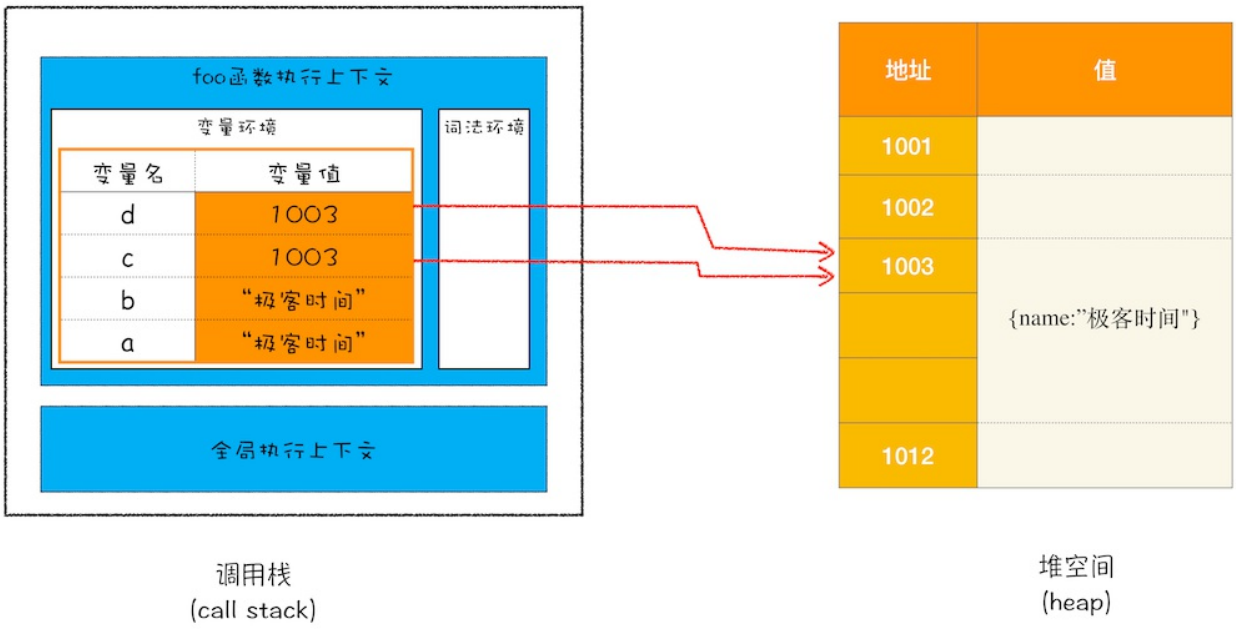
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

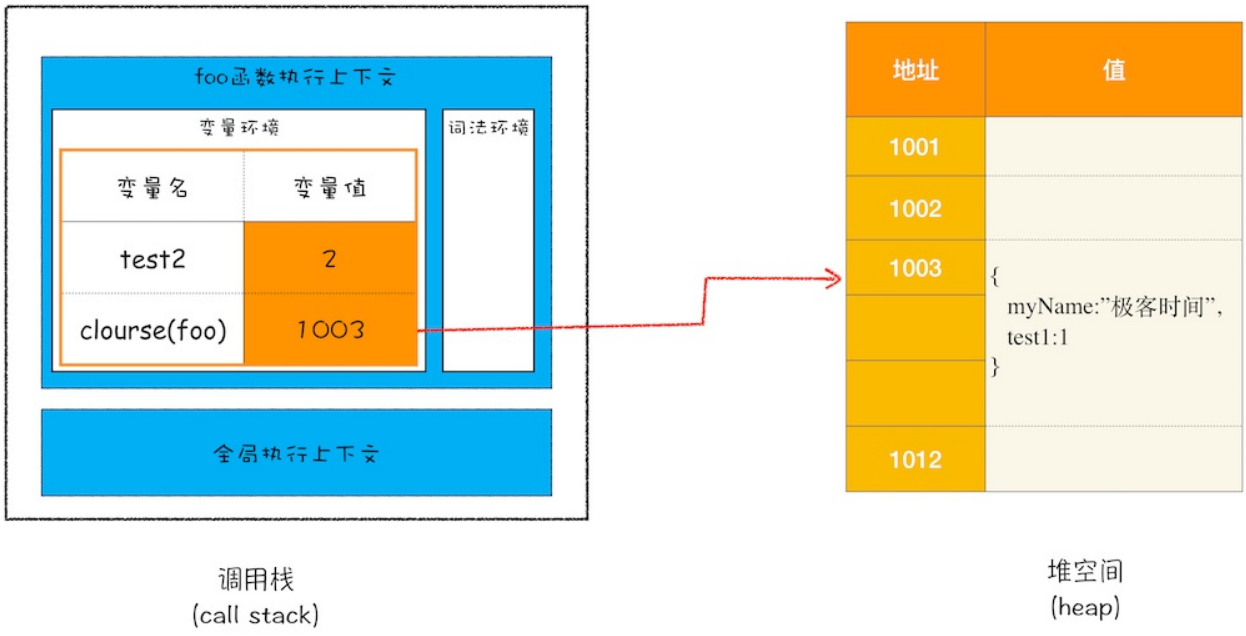
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

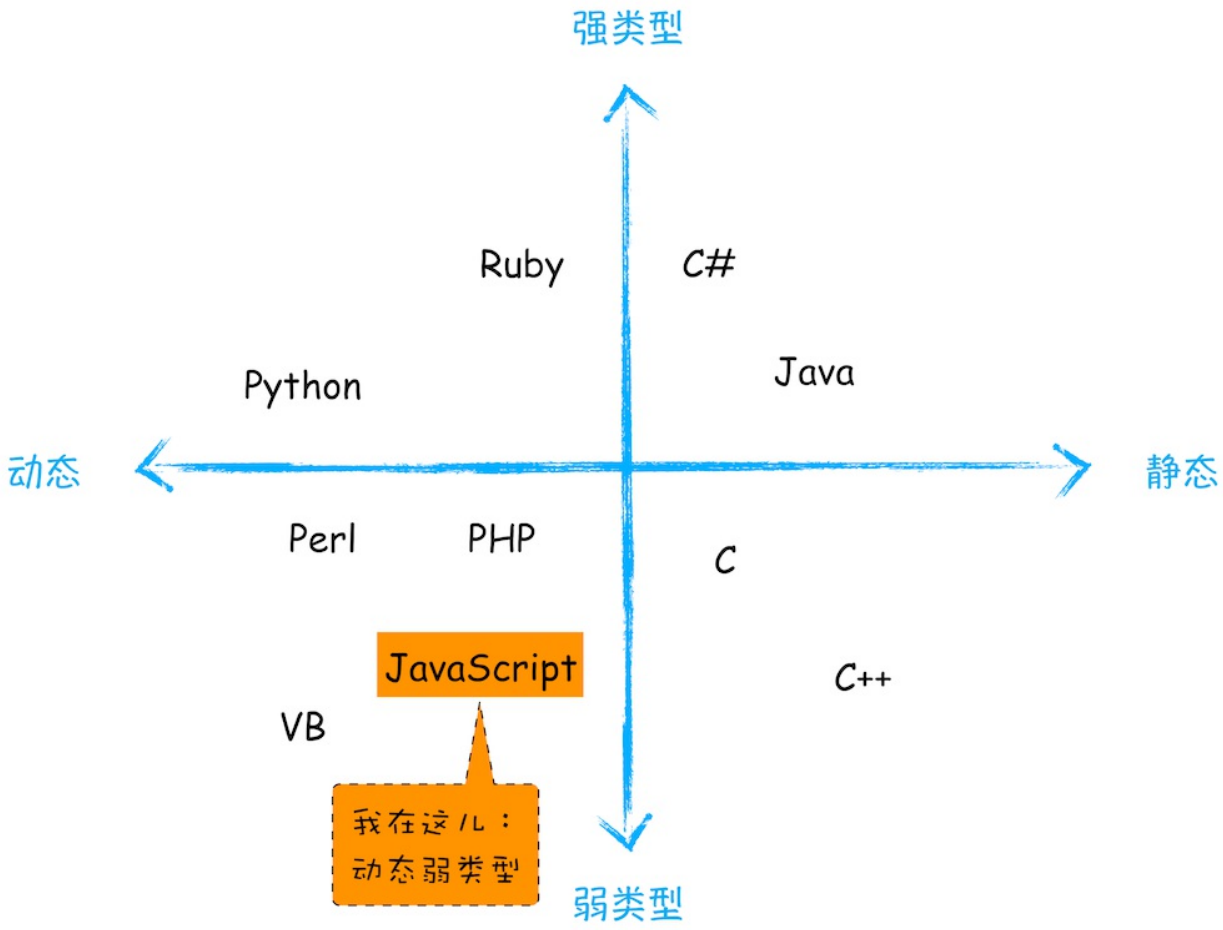
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

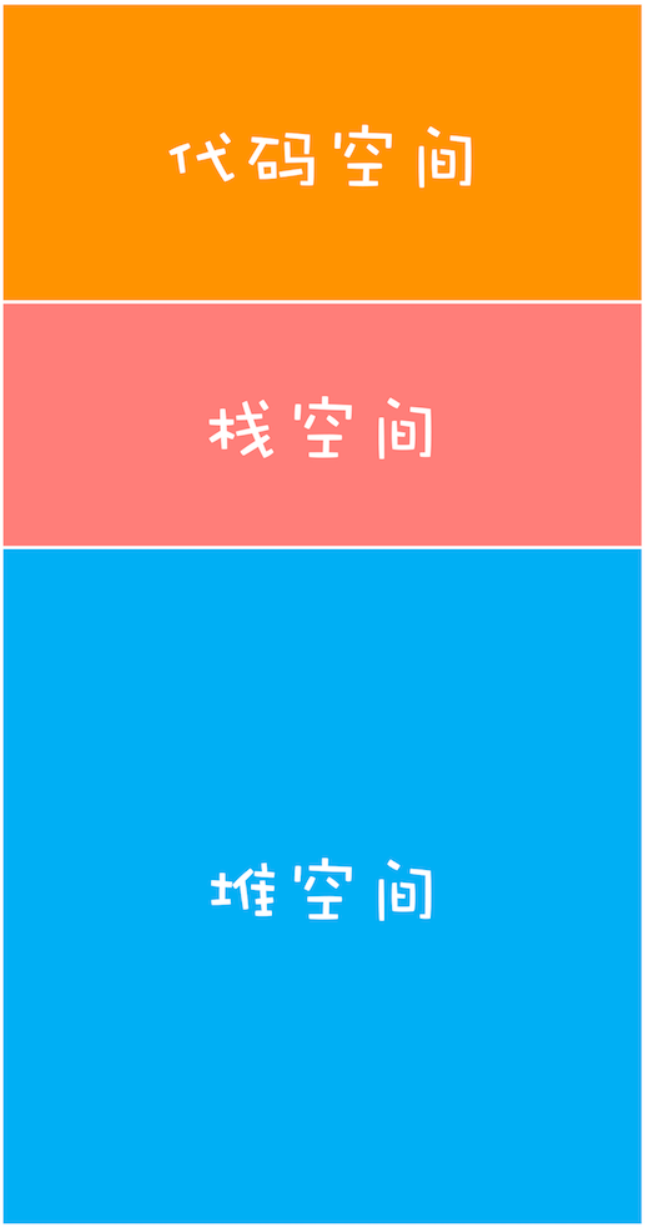
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

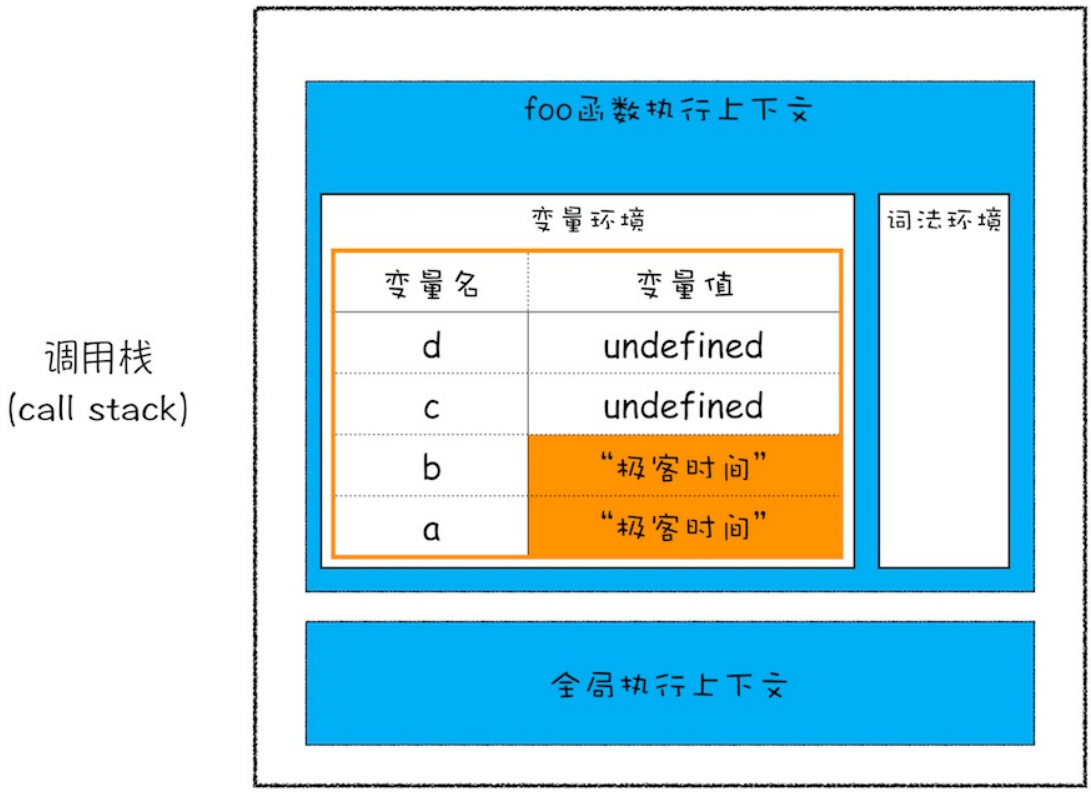
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

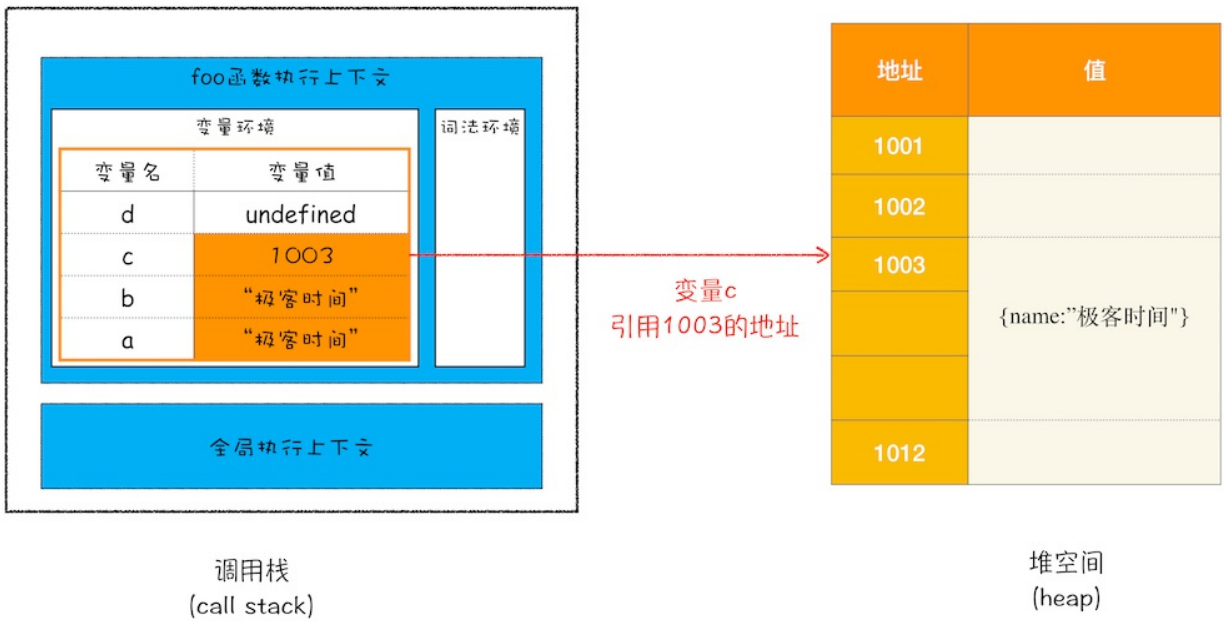
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

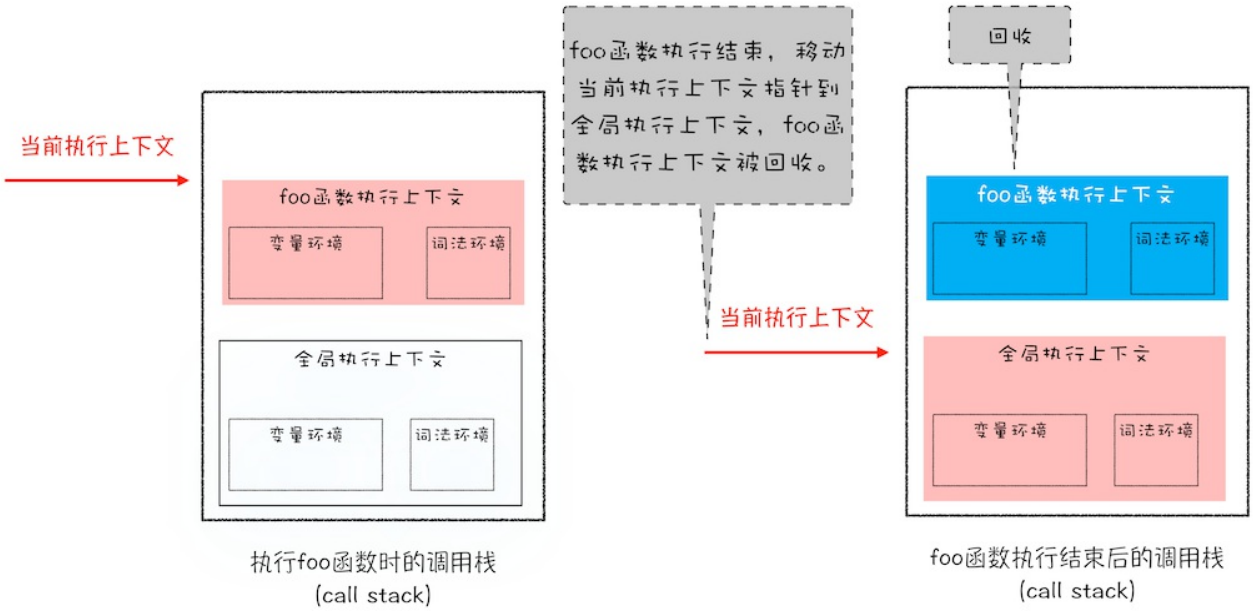


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



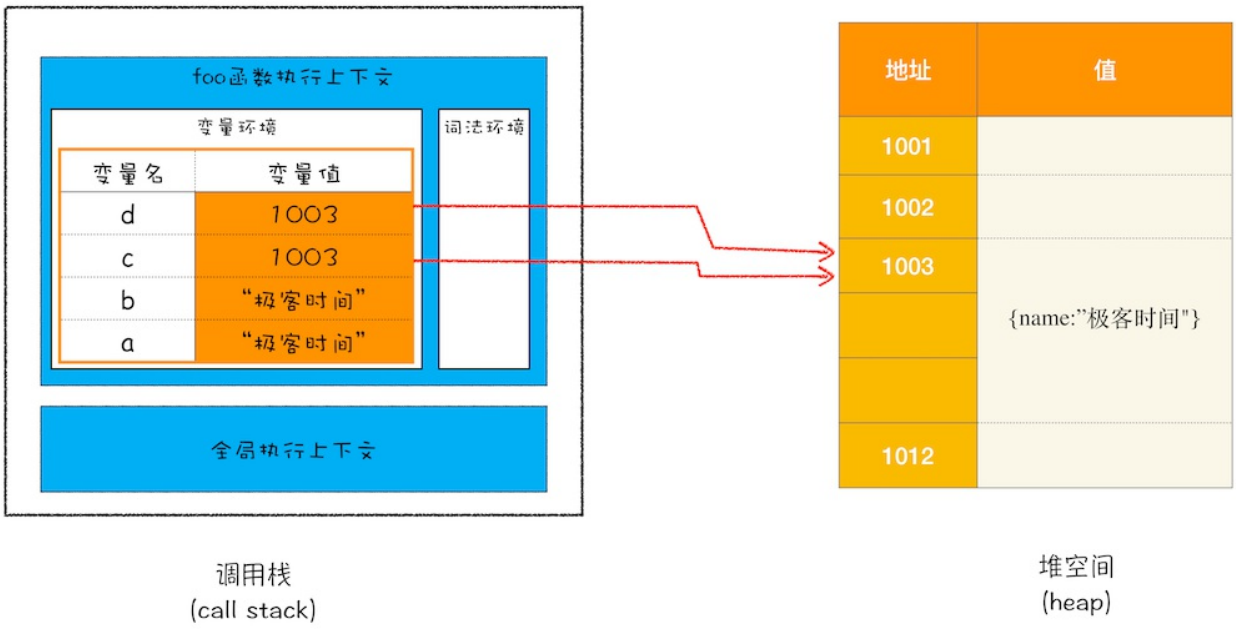
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

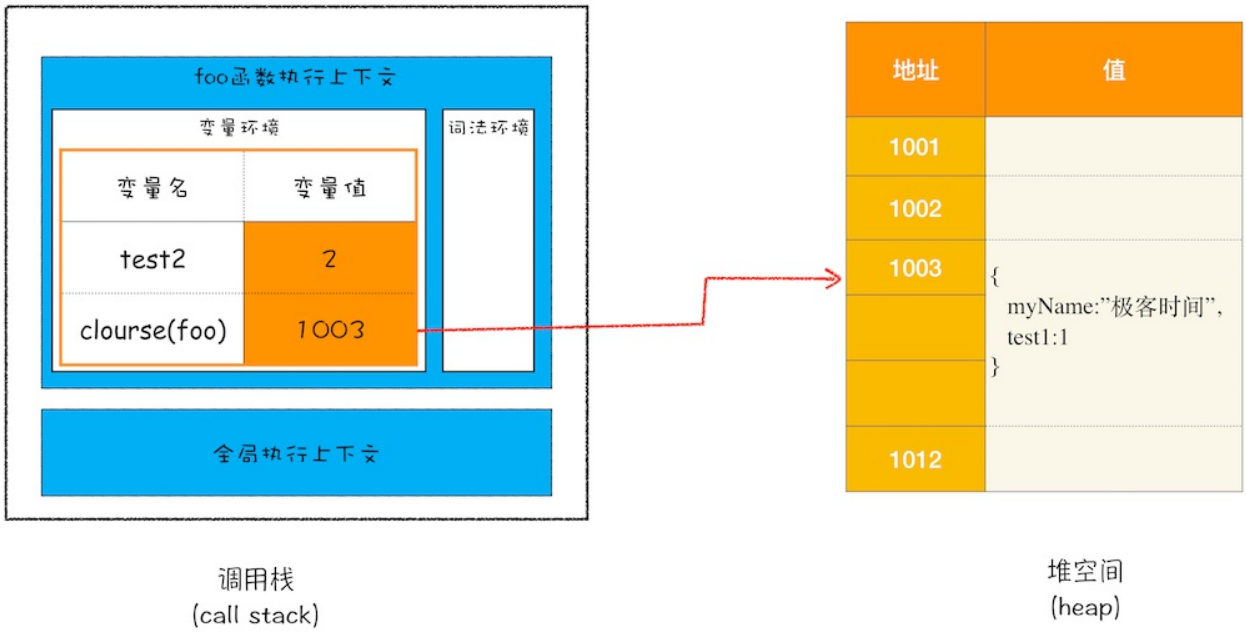
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

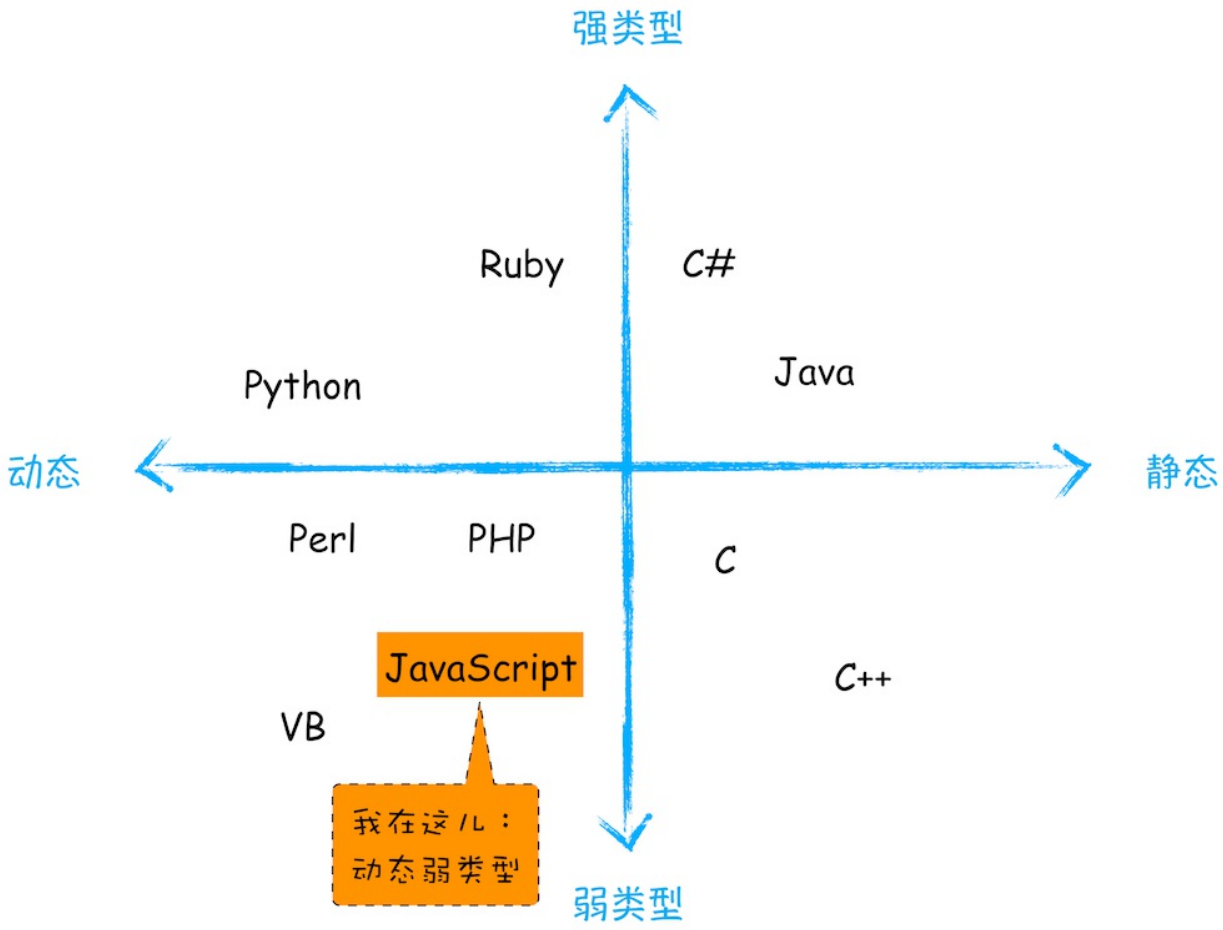
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

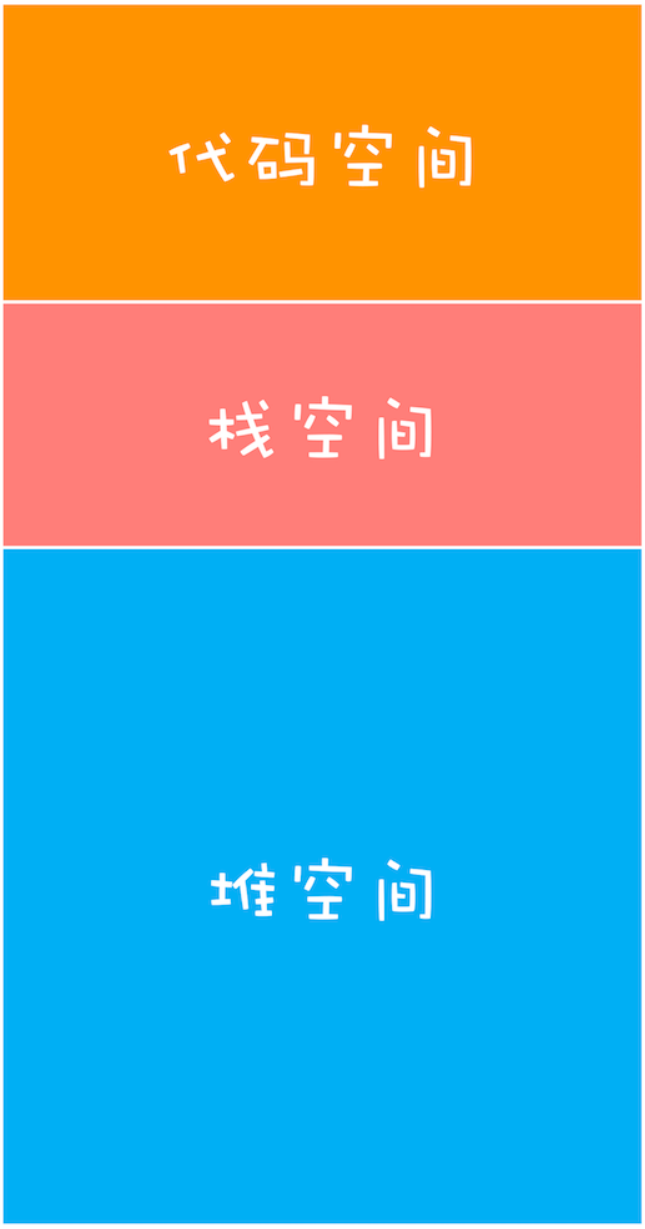
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

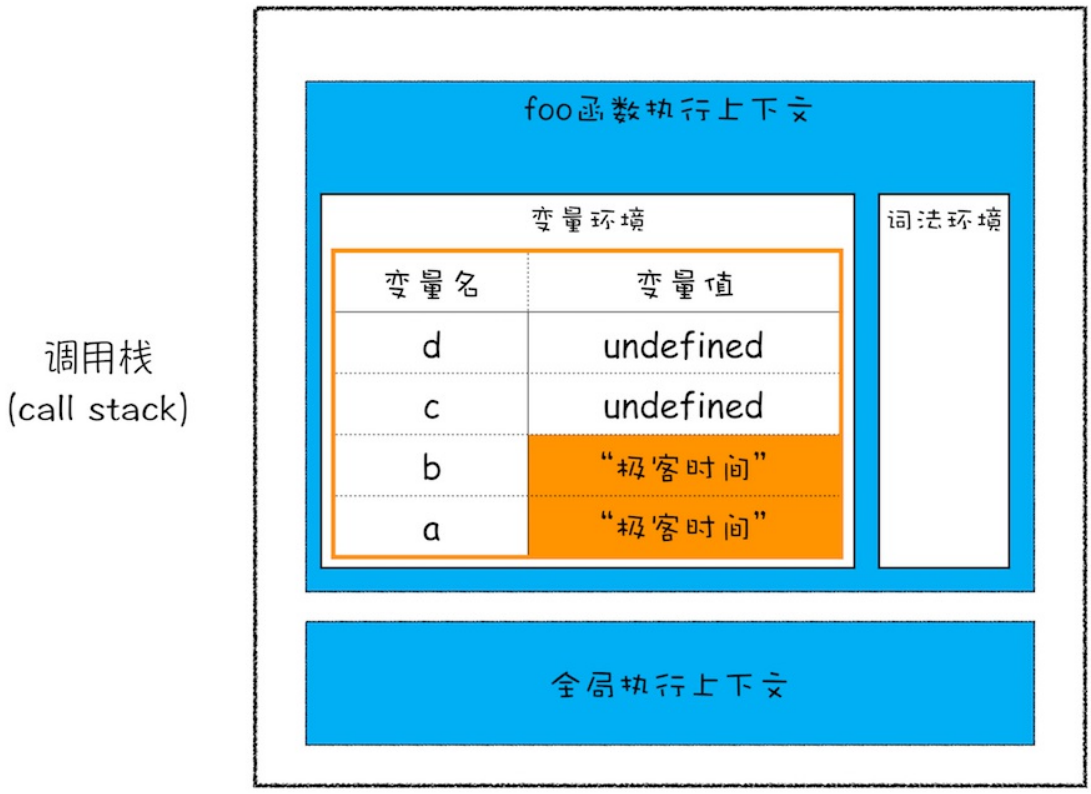
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

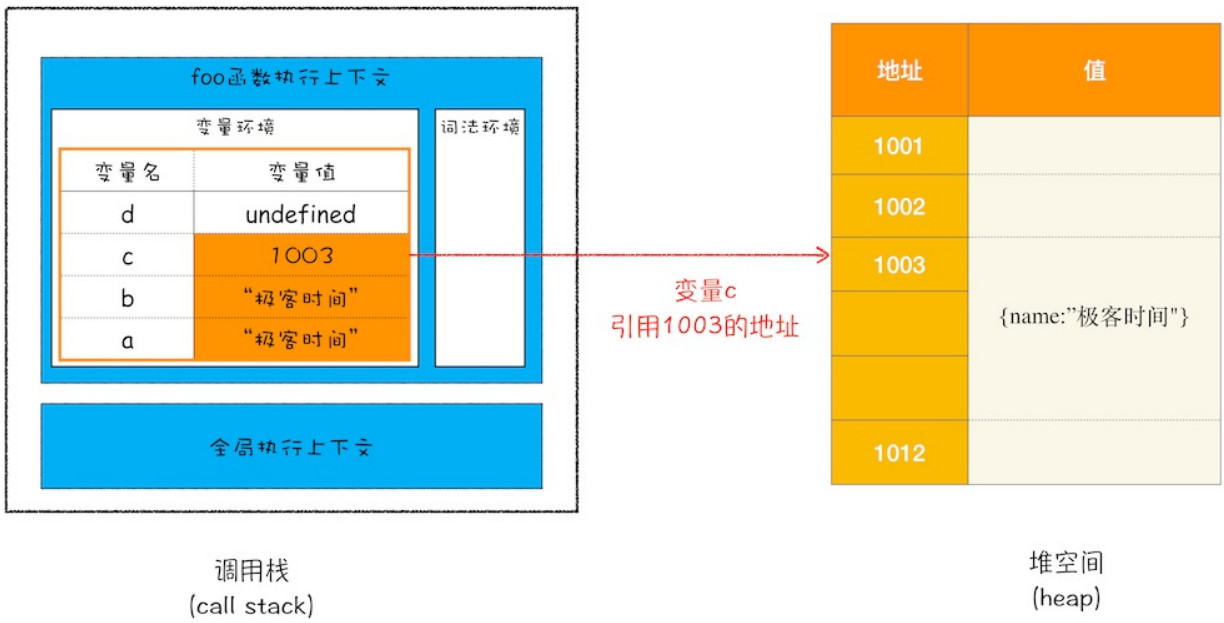
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

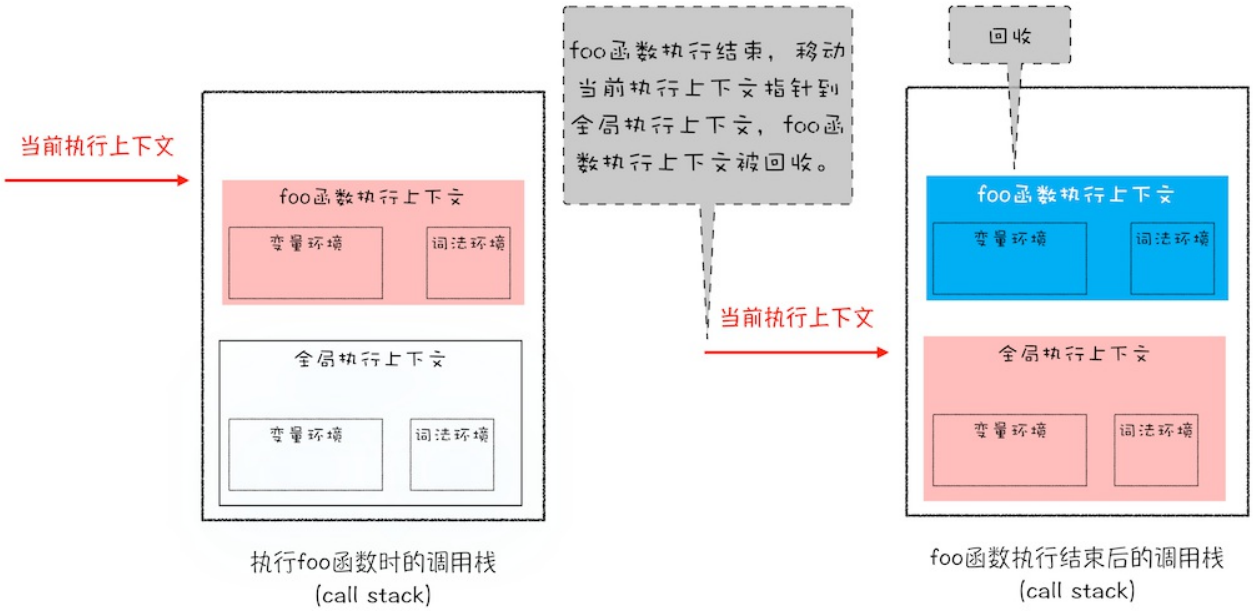


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



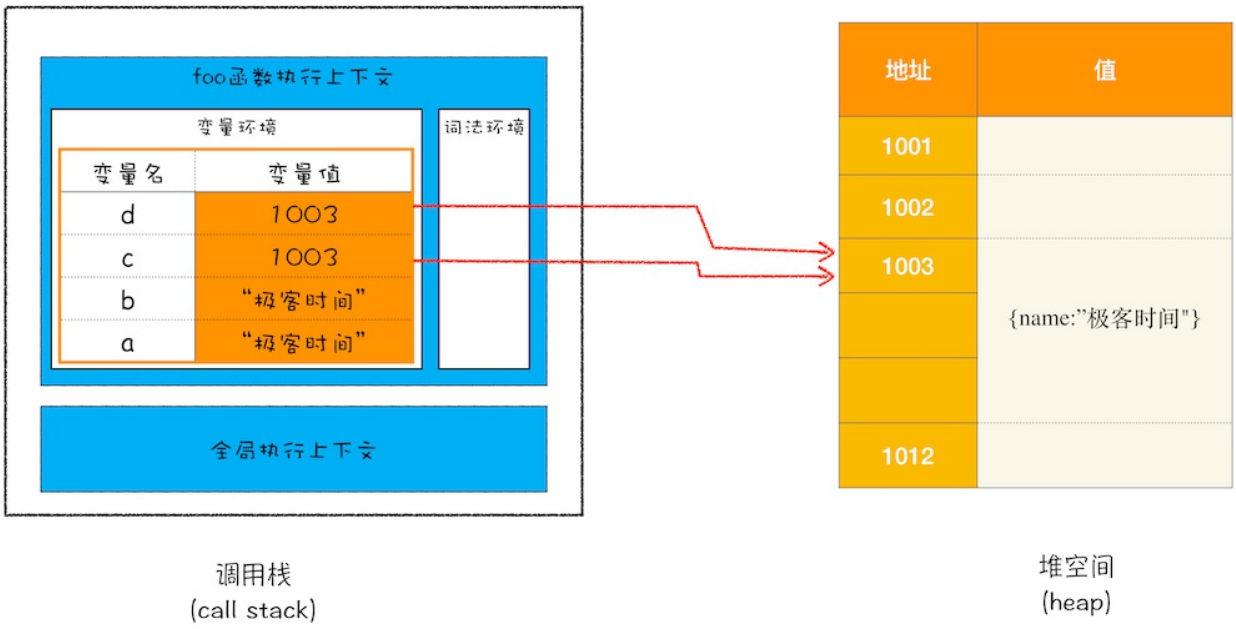
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

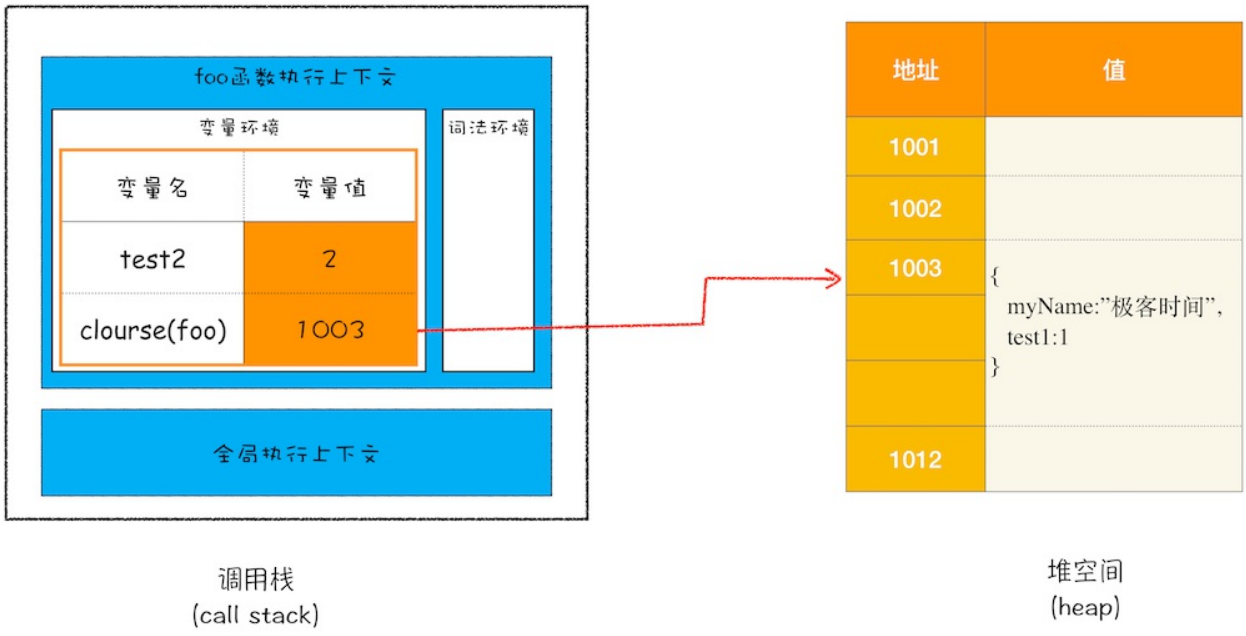
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“closure(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“closure(foo)”对象中。这时候堆中的“closure(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

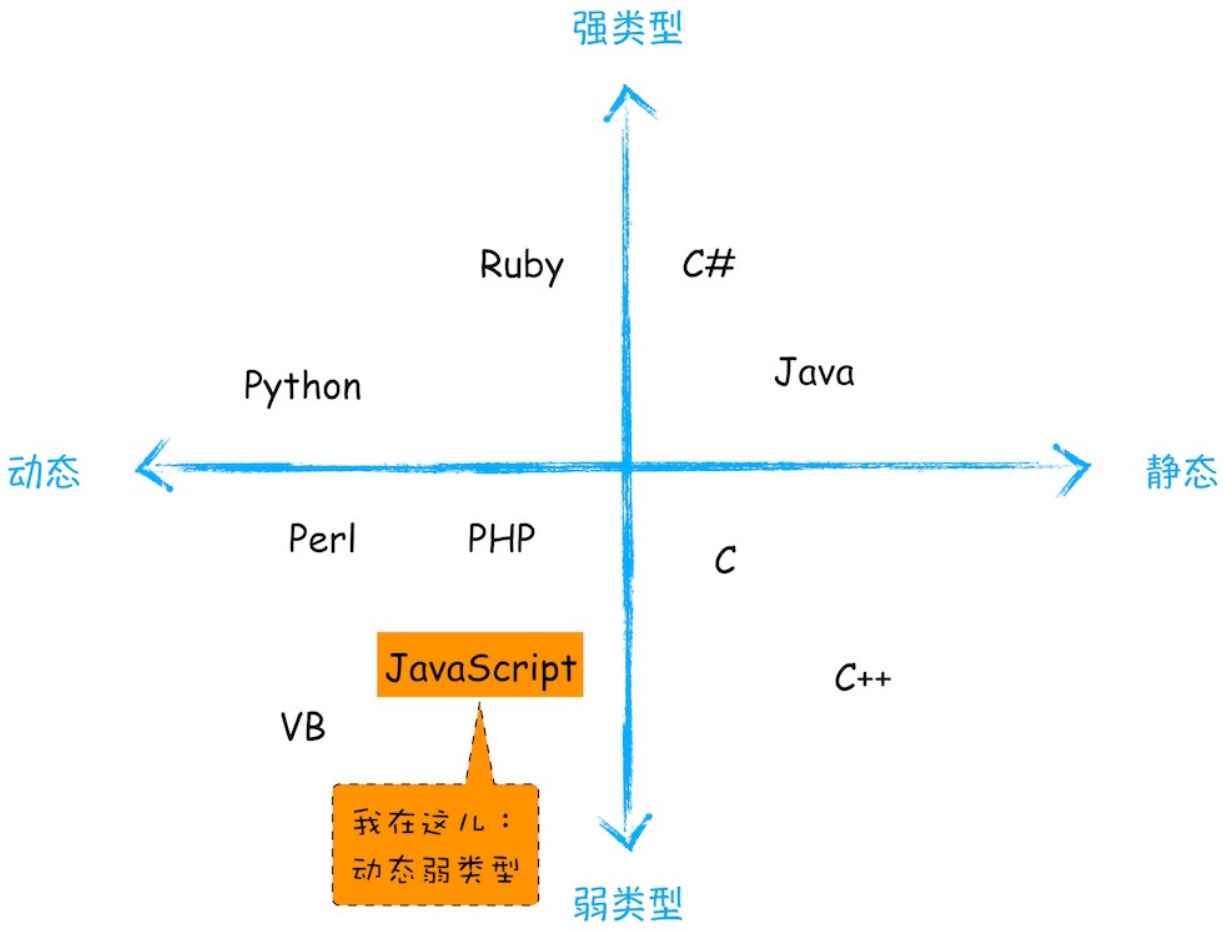
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

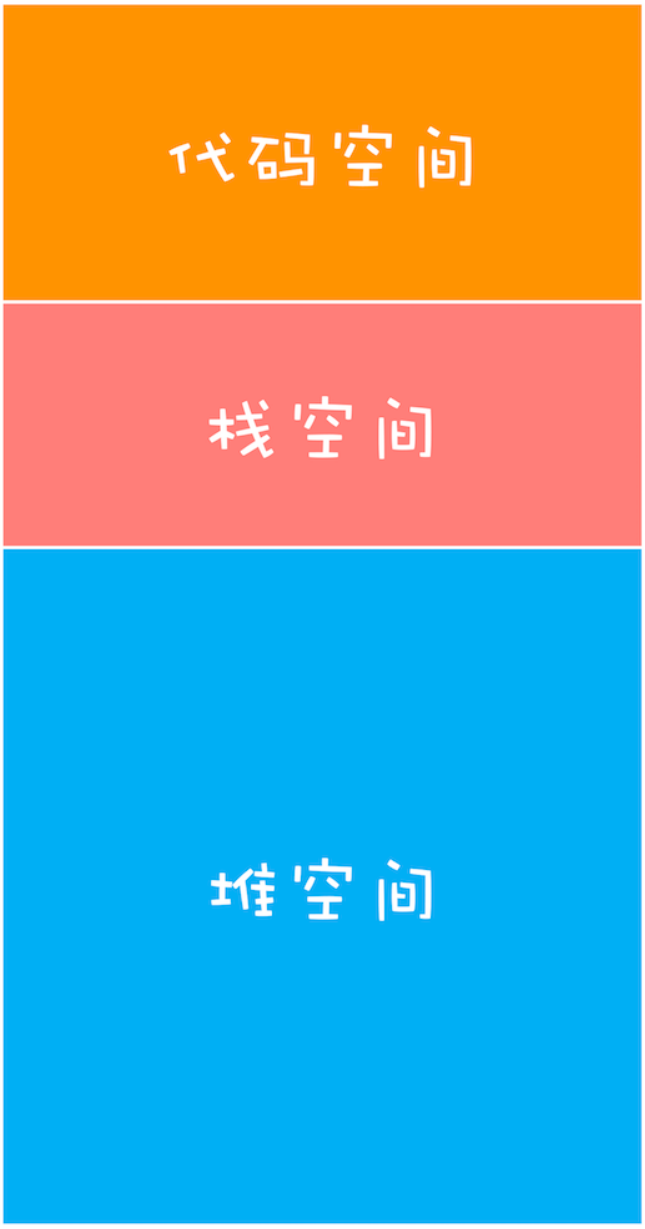
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

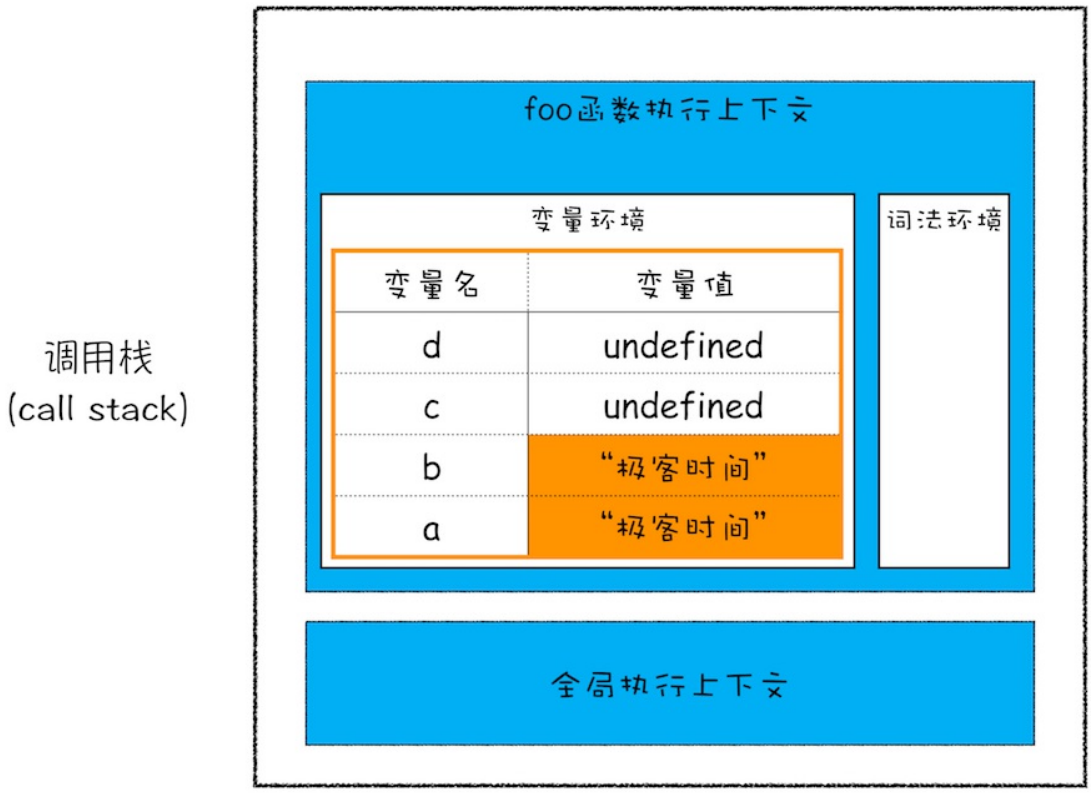
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

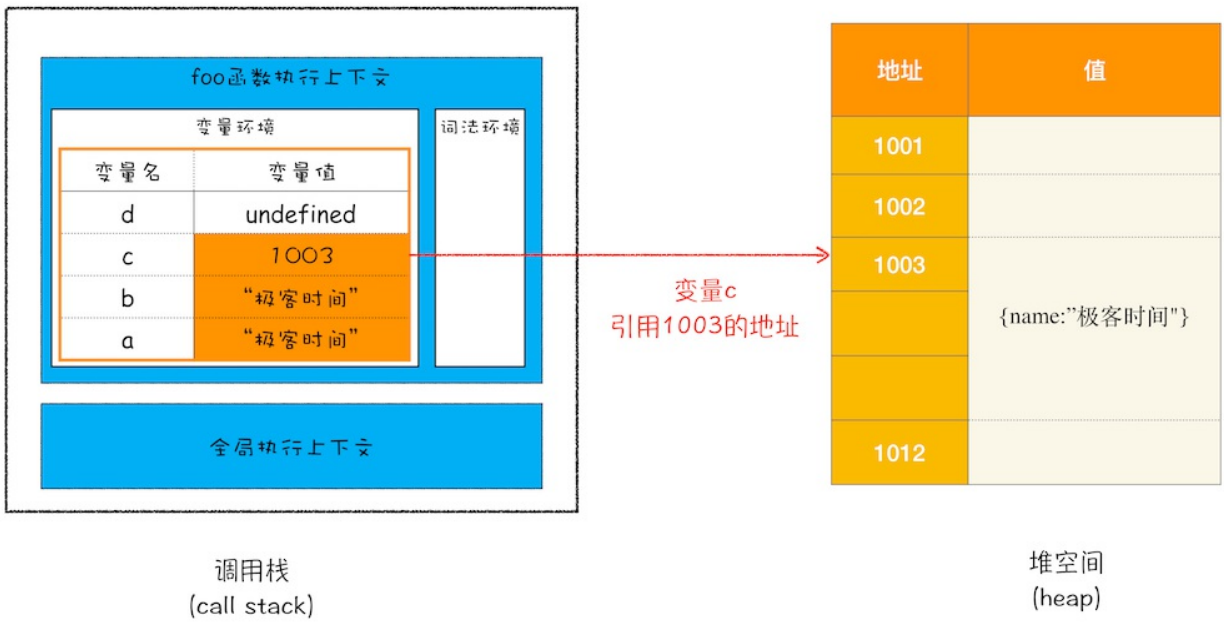
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

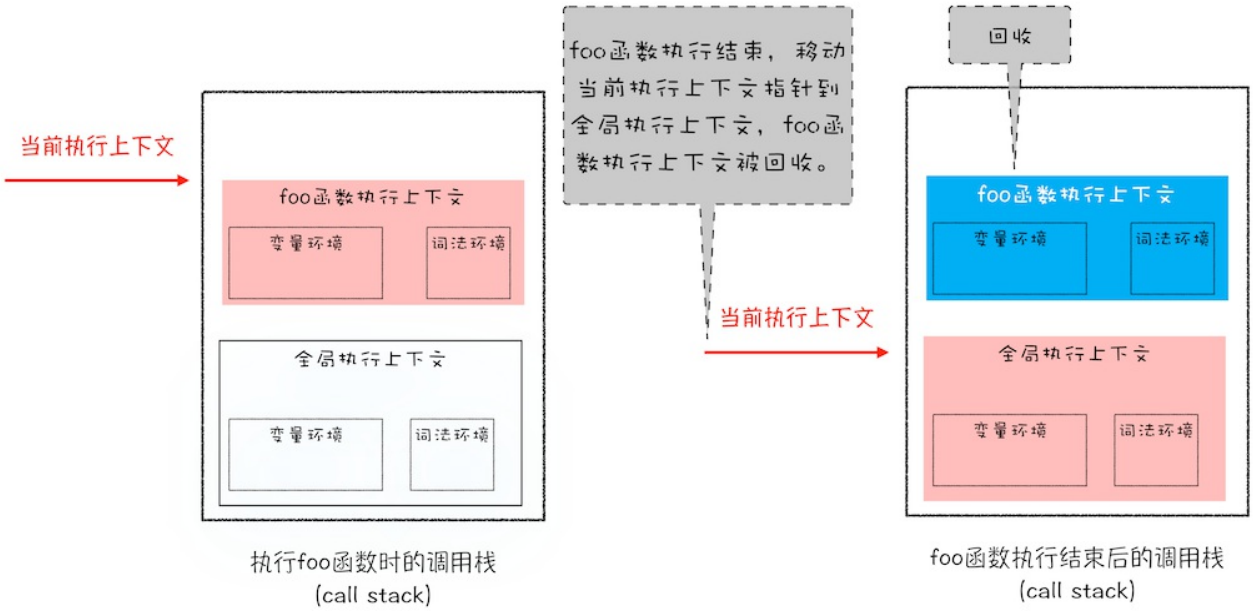


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



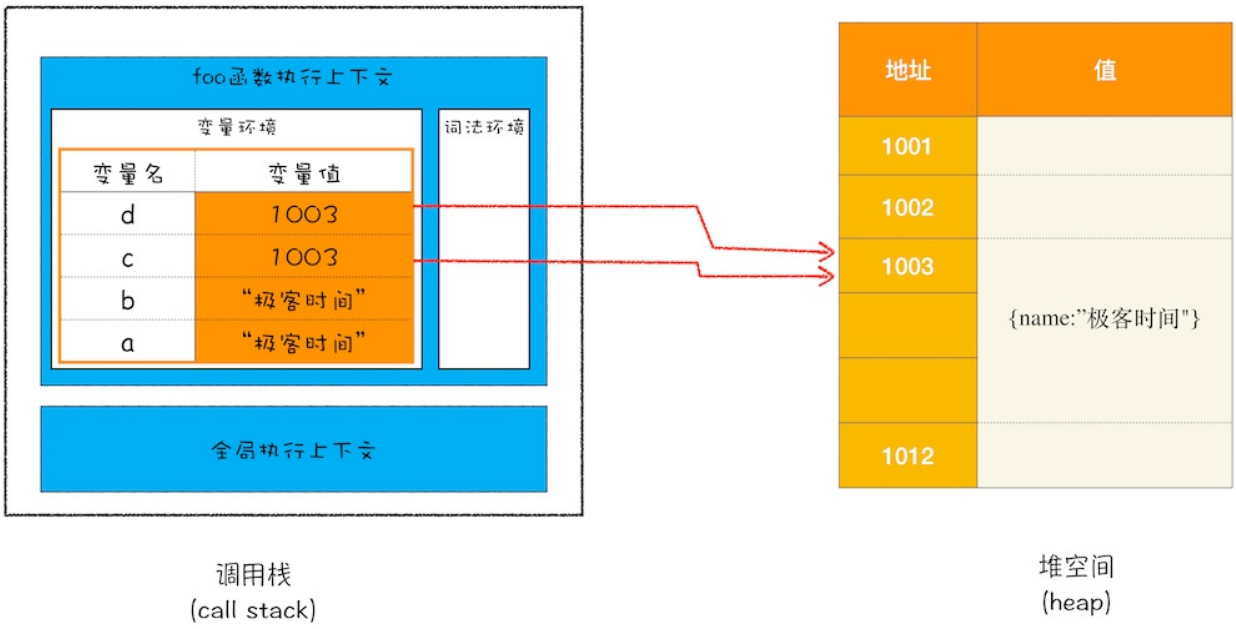
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

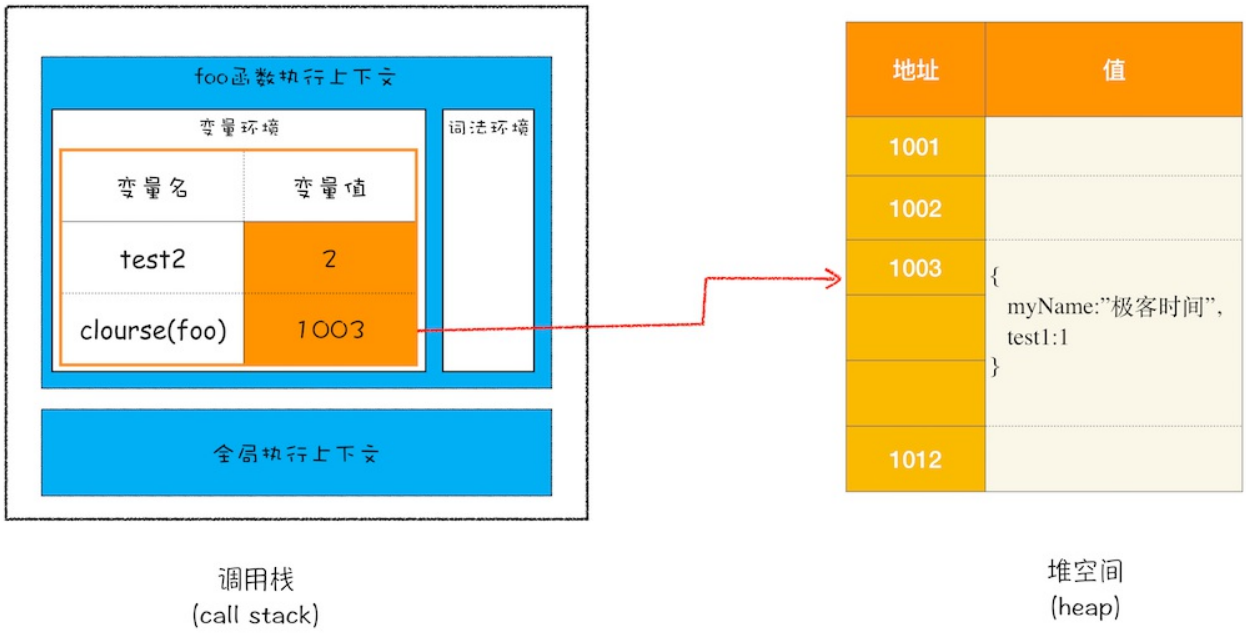
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

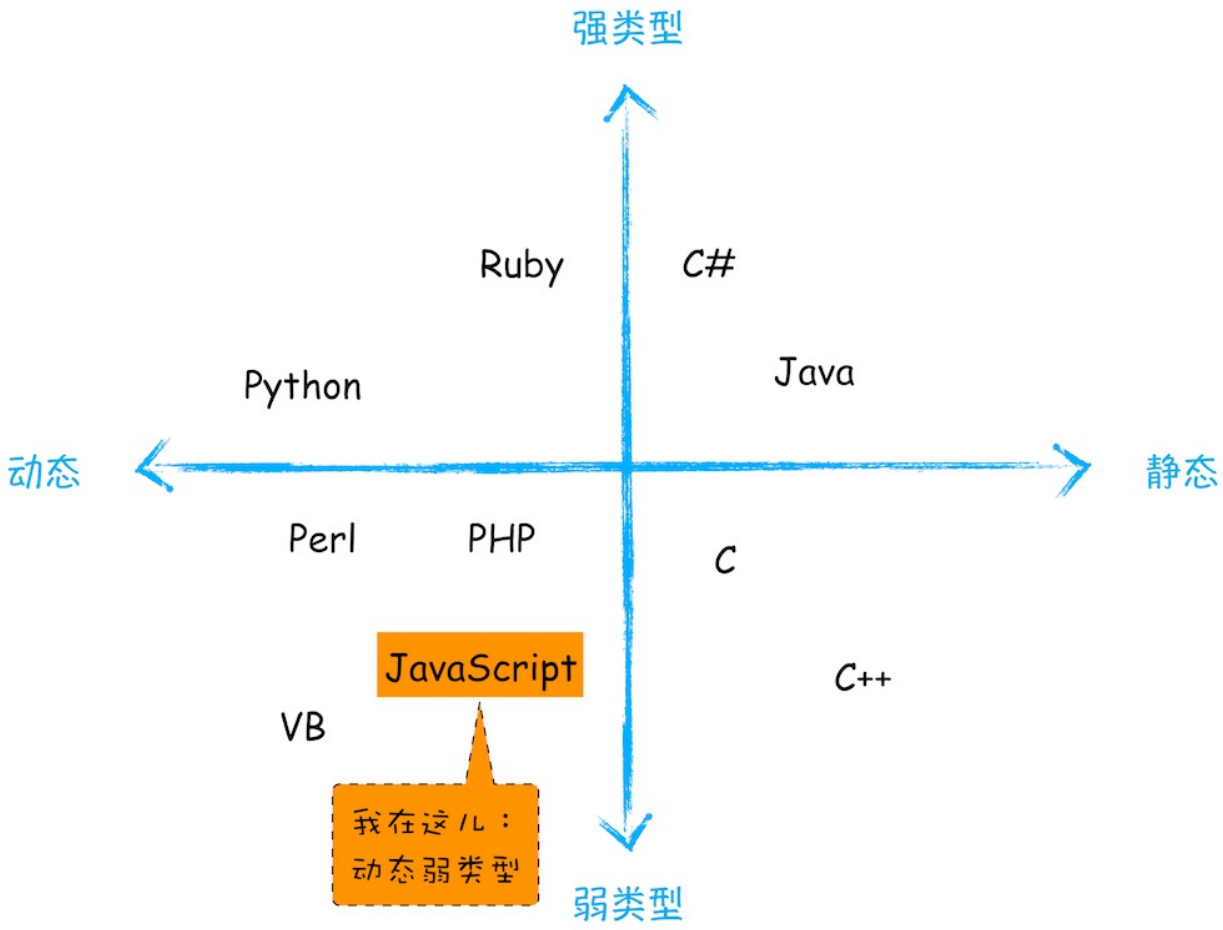
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

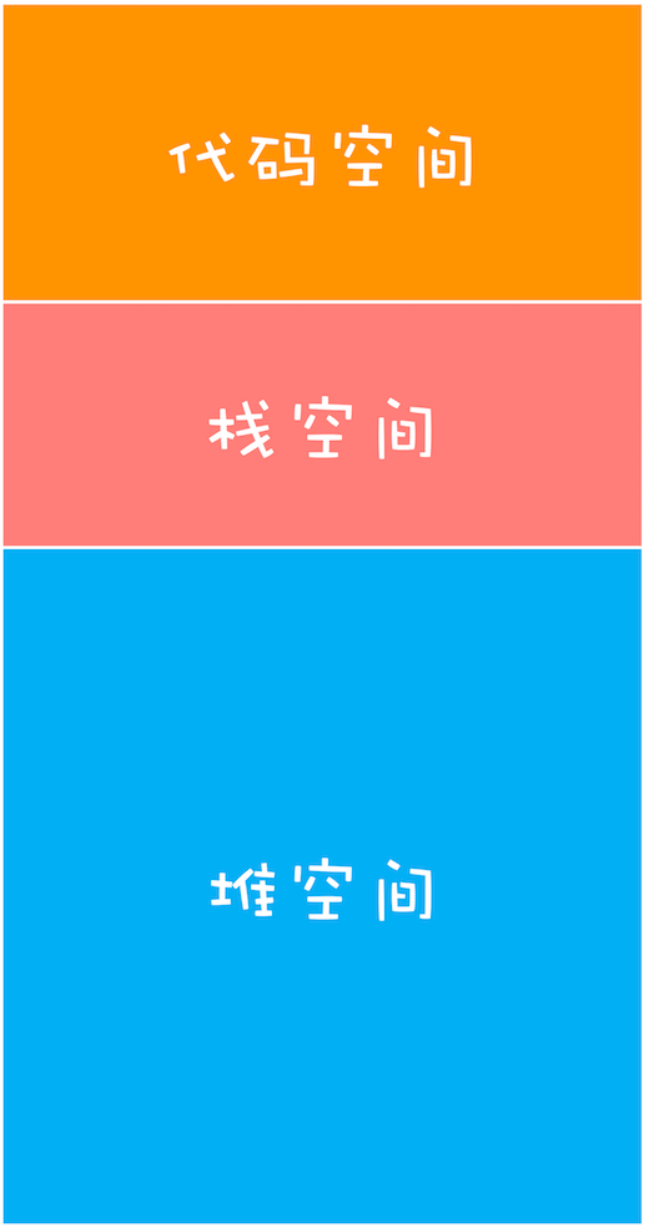
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

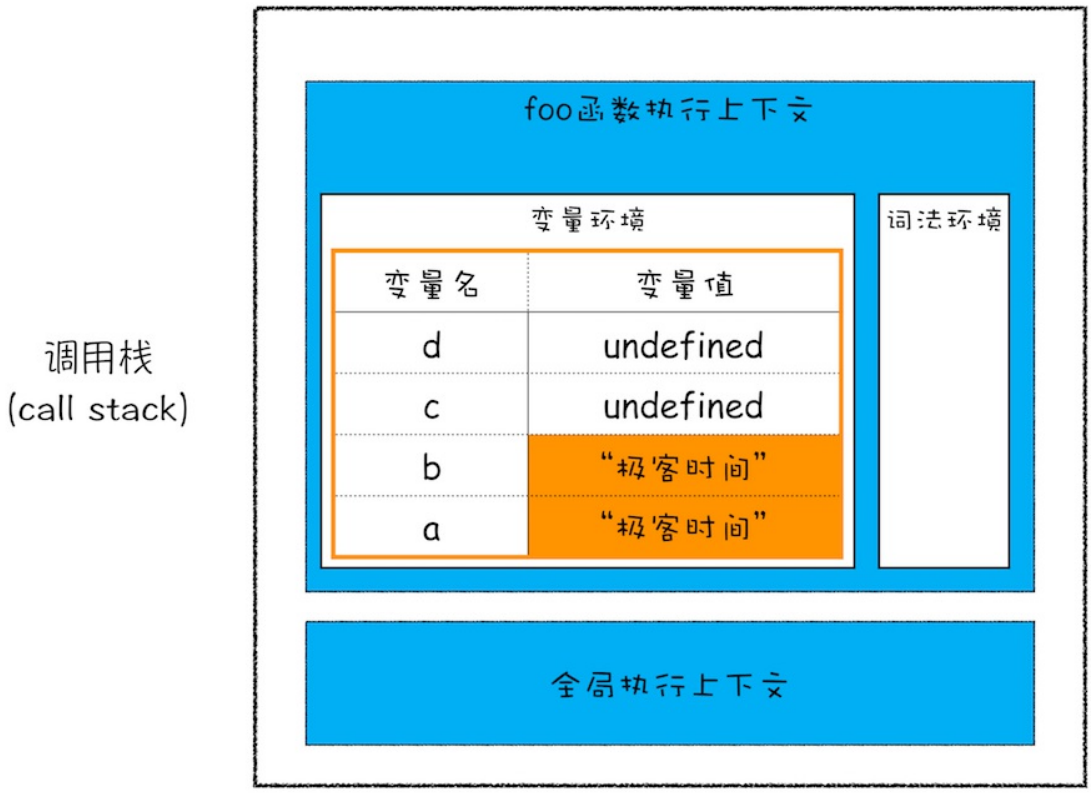
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

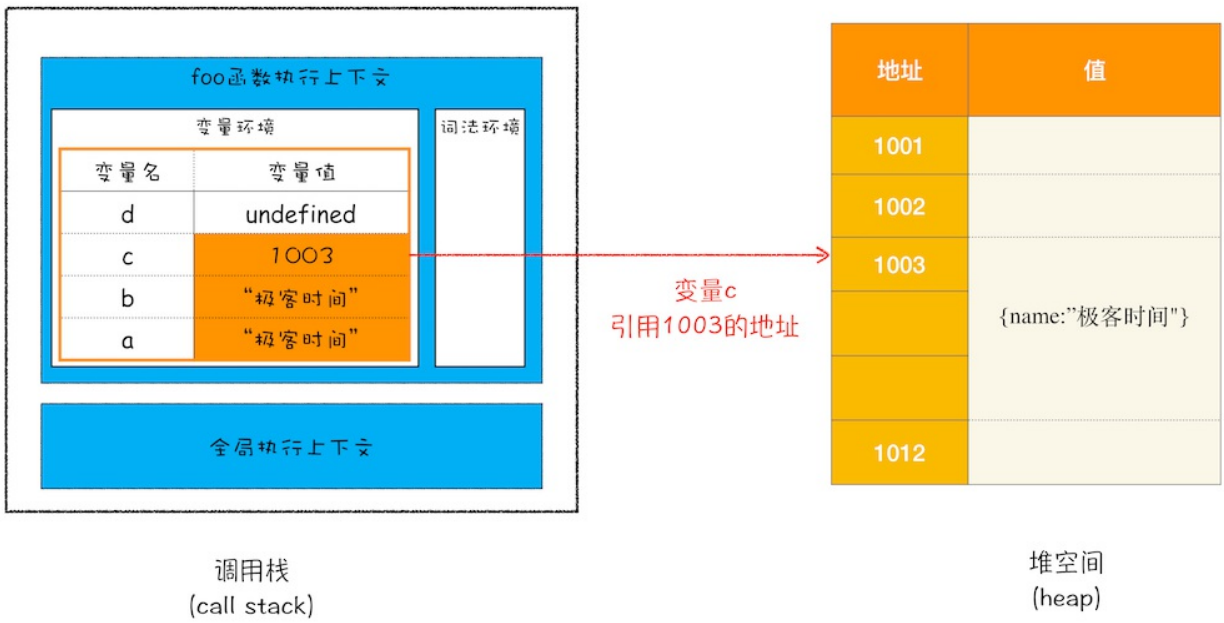
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

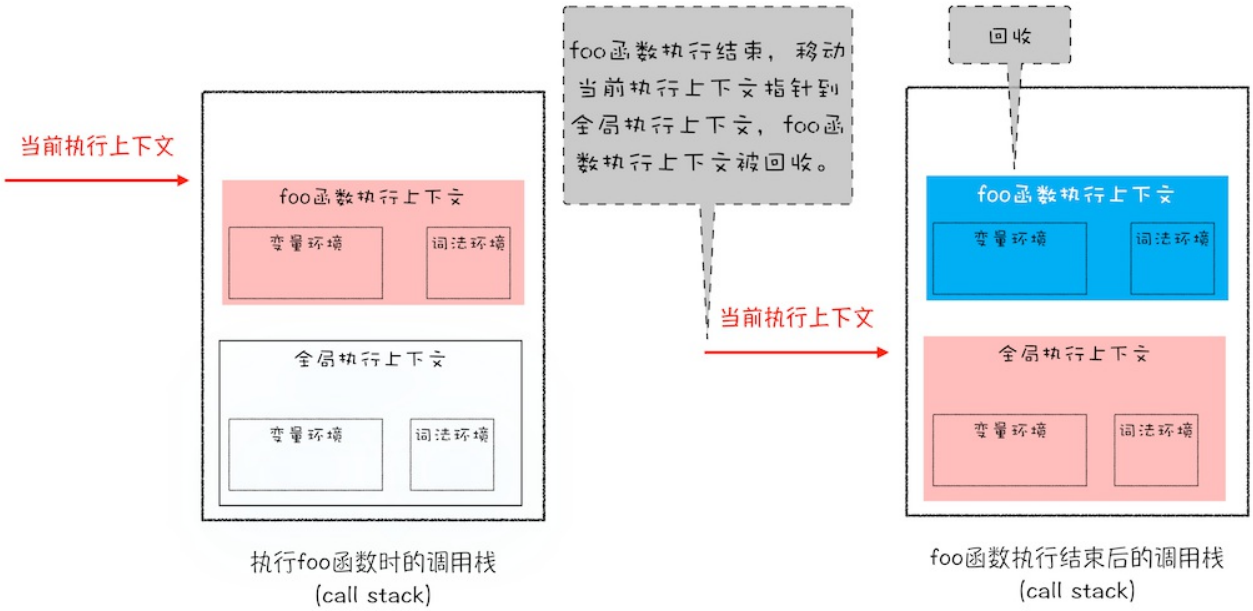


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



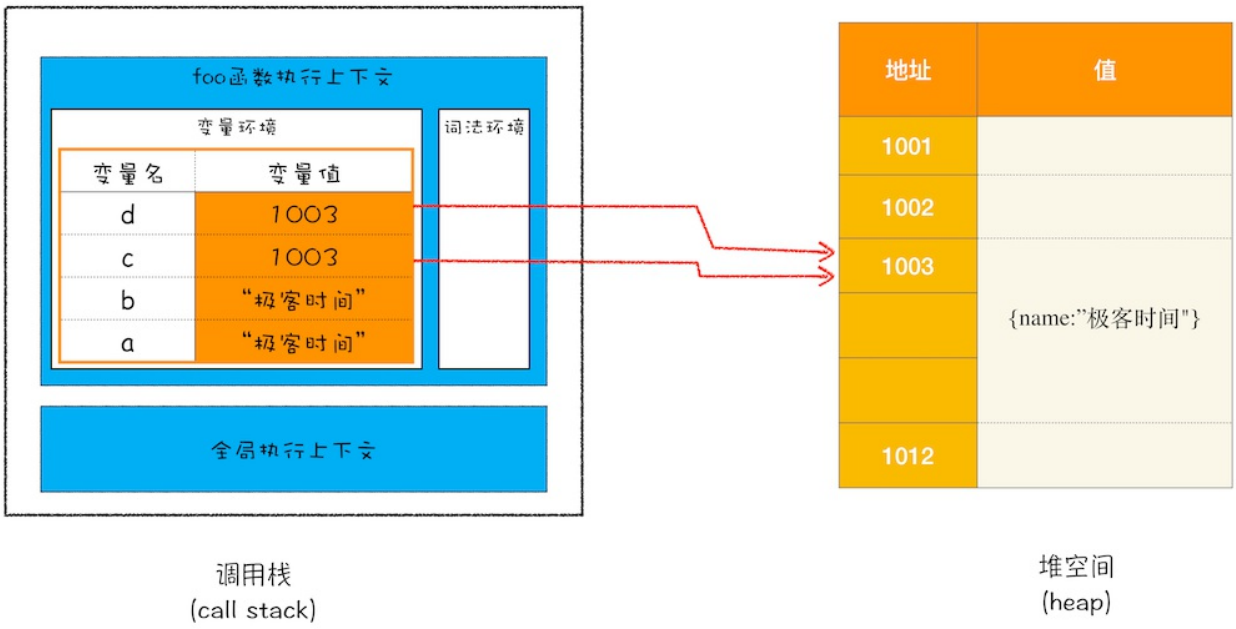
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

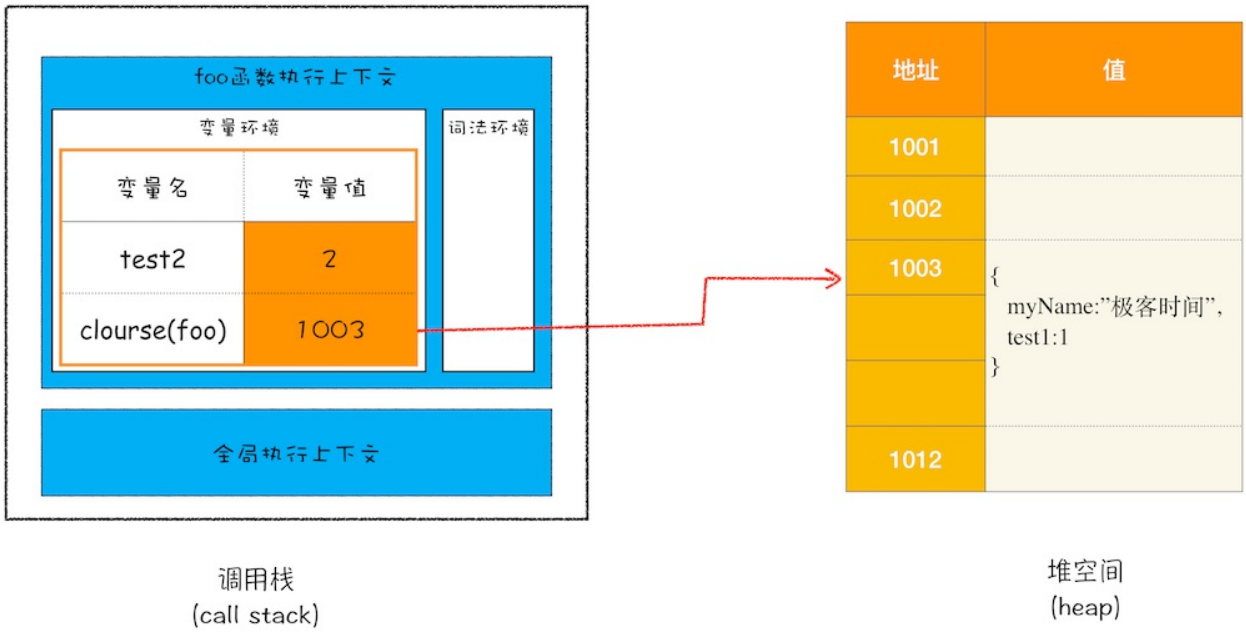
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

对于前端开发者来说，JavaScript的内存机制是一个不被经常提及的概念，因此很容易被忽视。特别是一些非计算机专业的同学，对内存机制可能没有非常清晰的认识，甚至有些同学根本就不知道JavaScript的内存机制是什么。

但是如果你想成为行业专家，并打造高性能前端应用，那么你就必须要搞清楚**JavaScript的内存机制**了。

其实，要搞清楚JavaScript的内存机制并不是一件很困难的事，在接下来的三篇文章（数据在内存中的存放、JavaScript处理垃圾回收以及V8执行代码）中，我们将通过内存机制的介绍，循序渐进带你走进JavaScript内存的世界。

今天我们讲述第一部分的内容——JavaScript中的数据是如何存储在内存中的。虽然JavaScript并不需要直接去管理内存，但是在实际项目中为了能避开一些不必要的坑，你还是需要了解数据在内存中的存储方式的。

让人疑惑的代码

首先，我们先看下面这两段代码：

```
function foo(){
    var a = 1
    var b = a
    a = 2
    console.log(a)
    console.log(b)
}
foo()
```

```
function foo(){
    var a = {name:"极客时间"}
    var b = a
    a.name = "极客邦"
    console.log(a)
    console.log(b)
}
foo()
```

若执行上述这两段代码，你知道它们输出的结果是什么吗？下面我们就来一个一个分析下。

执行第一段代码，打印出来a的值是2，b的值是1，这没什么难以理解的。

接着，再执行第二段代码，你会发现，仅仅改变了a中name的属性值，但是最终a和b打印出来的值都是`{name:"极客邦"}`。这就和我们预期的不一致了，因为我们想改变的仅仅是a的内容，但b的内容也同时被改变了。

要彻底弄清楚这个问题，我们就得先从“JavaScript是什么类型的语言”讲起。

JavaScript是什么类型的语言

每种编程语言都具有内建的数据类型，但它们的数据类型常有不同之处，使用方式也很不一样，比如C语言在定义变量之前，就需要确定变量的类型，你可以看下面这段C代码：

```
int main()
{
   int a = 1;
   char* b = "极客时间";
   bool c = true;
   return 0;
}
```

上述代码声明变量的特点是：在声明变量之前需要先定义变量类型。**我们把这种在使用之前就需要确认其变量数据类型的称为静态语言**。

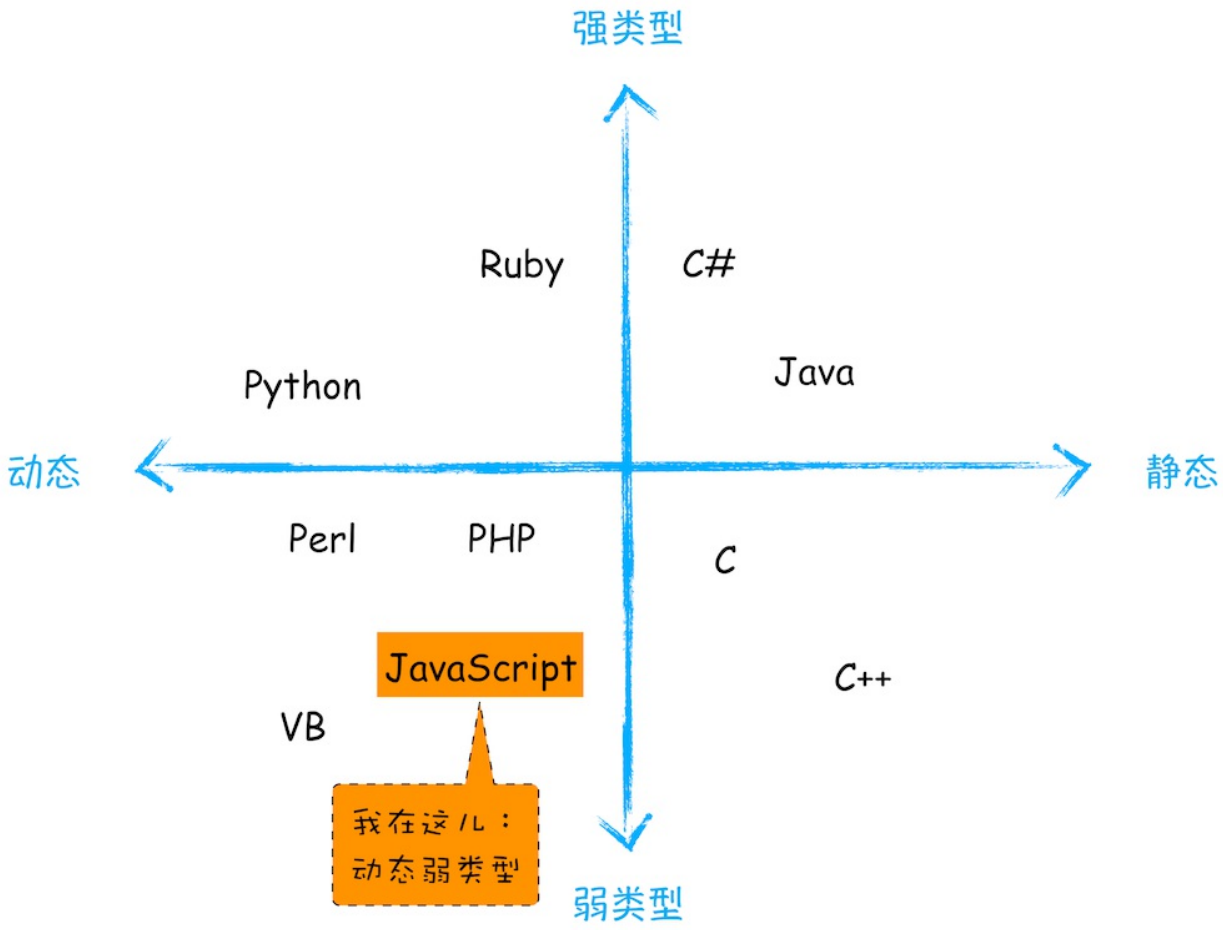
相反地，我们把在运行过程中需要检查数据类型的语言称为动态语言。比如我们所讲的JavaScript就是动态语言，因为在声明变量之前并不需要确认其数据类型。

虽然C语言是静态，但是在C语言中，我们可以把其他类型数据赋予给一个声明好的变量，如：

```
c = a
```

前面代码中，我们把int型的变量a赋值给了bool型的变量c，这段代码也是可以编译执行的，因为在赋值过程中，C编译器会把int型的变量悄悄转换为bool型的变量，我们通常把这种偷偷转换的操作称为**隐式类型转换**。而**支持隐式类型转换的语言称为弱类型语言，不支持隐式类型转换的语言称为强类型语言**。在这点上，C和JavaScript都是弱类型语言。

对于各种语言的类型，你可以参考下图：



语言类型图

JavaScript的数据类型

现在我们知道了，**JavaScript是一种弱类型的、动态的语言**。那这些特点意味着什么呢？

- **弱类型**，意味着你不需要告诉JavaScript引擎这个或那个变量是什么数据类型，JavaScript引擎在运行代码的时候自己会计算出来。
- **动态**，意味着你可以使用同一个变量保存不同类型的数据。

那么接下来，我们再来看看JavaScript的数据类型，你可以看下面这段代码：

```
var bar
bar = 12
bar = "极客时间"
bar = true
bar = null
bar = {name:"极客时间"}
```

从上述代码中你可以看出，我们声明了一个bar变量，然后可以使用各种类型的数据值赋予给该变量。

在JavaScript中，如果你想要查看一个变量到底是什么类型，可以使用“typeof”运算符。具体使用方式如下所示：

```
var bar
console.log(typeof bar)  //undefined
bar = 12
console.log(typeof bar) //number
bar = "极客时间"
console.log(typeof bar)//string
bar = true
console.log(typeof bar) //boolean
bar = null
console.log(typeof bar) //object
bar = {name:"极客时间"}
console.log(typeof bar) //object
```

执行这段代码，你可以看到打印出来了不同的数据类型，有undefined、number、boolean、object等。那么接下来我们就来谈谈JavaScript到底有多少种数据类型。

其实JavaScript中的数据类型一种有8种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

了解这些类型之后，还有三点需要你注意一下。

第一点，使用typeof检测Null类型时，返回的是Object。这是当初JavaScript语言的一个Bug，一直保留至今，之所以一直没修改过来，主要是为了兼容老的代码。

第二点，Object类型比较特殊，它是由上述7种类型组成的一个包含了key-value对的数据类型。如下所示：

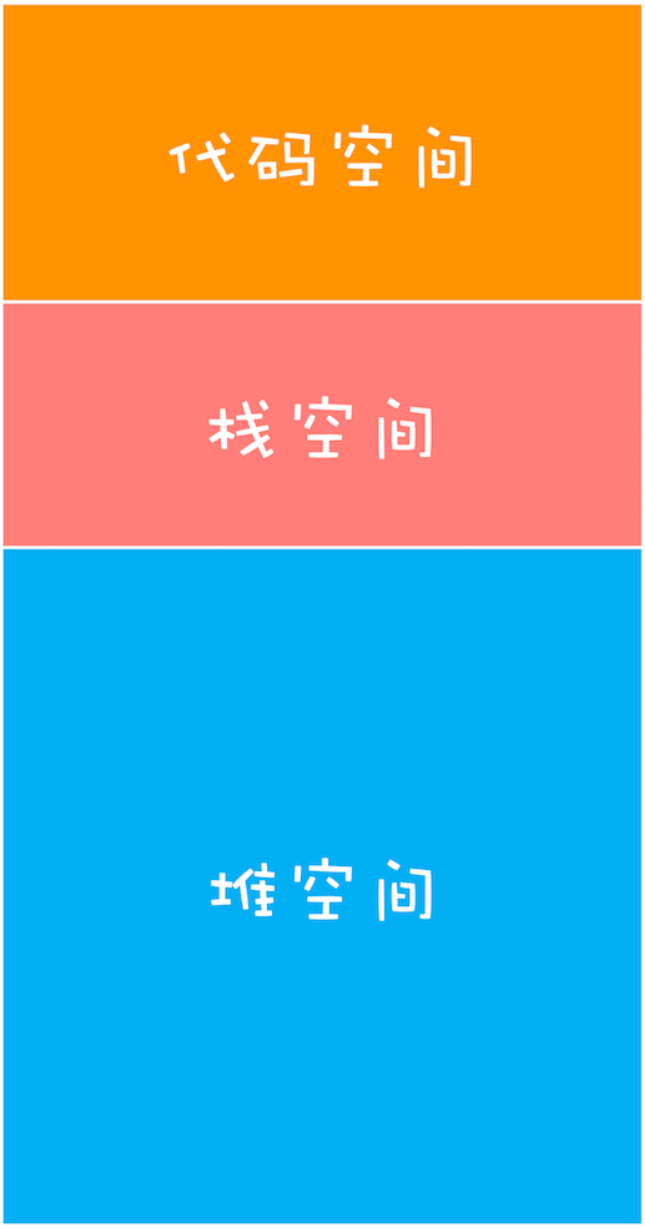
```
let myObj = {
        name:'极客时间',
        update:function(){....}
        }
```

从中你可以看出来，Object是由key-value组成的，其中的vaule可以是任何类型，包括函数，这也就意味着你可以通过Object来存储函数，Object中的函数又称为方法，比如上述代码中的update方法。

第三点，我们把前面的7种数据类型称为**原始类型**，把最后一个对象类型称为**引用类型**，之所以把它们区分为两种不同的类型，是因为它们在内存中存放的位置不一样。到底怎么个不一样法呢？接下来，我们就来讲解一下JavaScript的原始类型和引用类型到底是怎么储存的。

内存空间

要理解JavaScript在运行过程中数据是如何存储的，你就得先搞清楚其存储空间的种类。下面是我画的JavaScript的内存模型，你可以参考下：



JavaScript内存模型

从图中可以看出， 在JavaScript的执行过程中， 主要有三种类型内存空间，分别是**代码空间**、**栈空间**和**堆空间**。

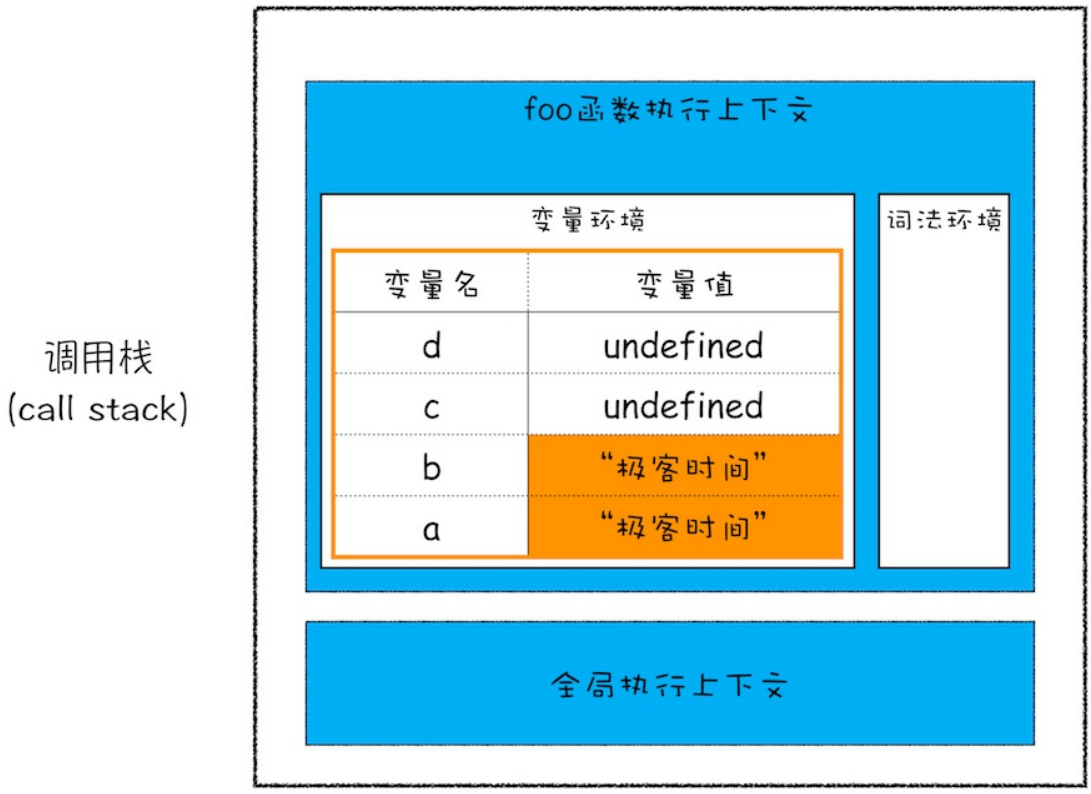
其中的代码空间主要是存储可执行代码的，这个我们后面再做介绍，今天主要来说说栈空间和堆空间。

栈空间和堆空间

这里的栈空间就是我们之前反复提及的调用栈，是用来存储执行上下文的。为了搞清楚栈空间是如何存储数据的，我们还是先看下面这段代码：

```
function foo(){
    var a = "极客时间"
    var b = a
    var c = {name:"极客时间"}
    var d = c
}
foo()
```

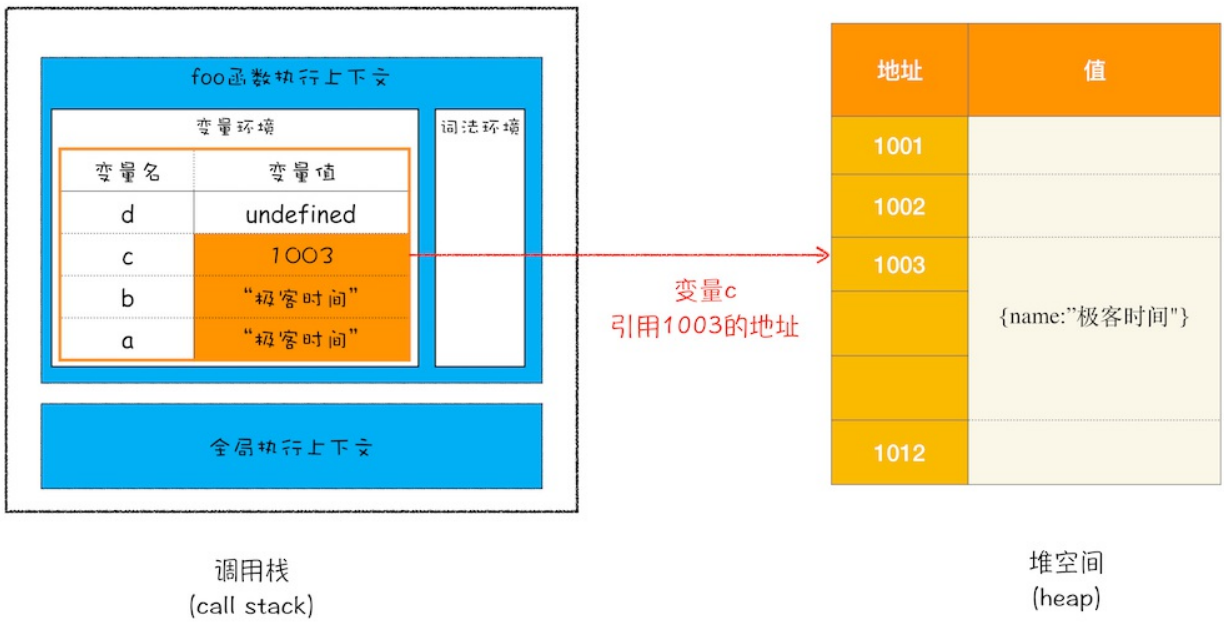
前面文章我们已经讲解过了，当执行一段代码时，需要先编译，并创建执行上下文，然后再按照顺序执行代码。那么下面我们来看看，当执行到第3行代码时，其调用栈的状态，你可以参考下面这张调用栈状态图：



执行到第3行时的调用栈状态图

从图中可以看出来，当执行到第3行时，变量a和变量b的值都被保存在执行上下文中，而执行上下文又被压入到栈中，所以你也可以认为变量a和变量b的值都是存放在栈中的。

接下来继续执行第4行代码，由于JavaScript引擎判断右边的值是一个引用类型，这时候处理的情况就不一样了，JavaScript引擎并不是直接将该对象存放到变量环境中，而是将它分配到堆空间里面，分配后该对象会有一个在“堆”中的地址，然后再将该数据的地址写进c的变量值，最终分配好内存的示意图如下所示：

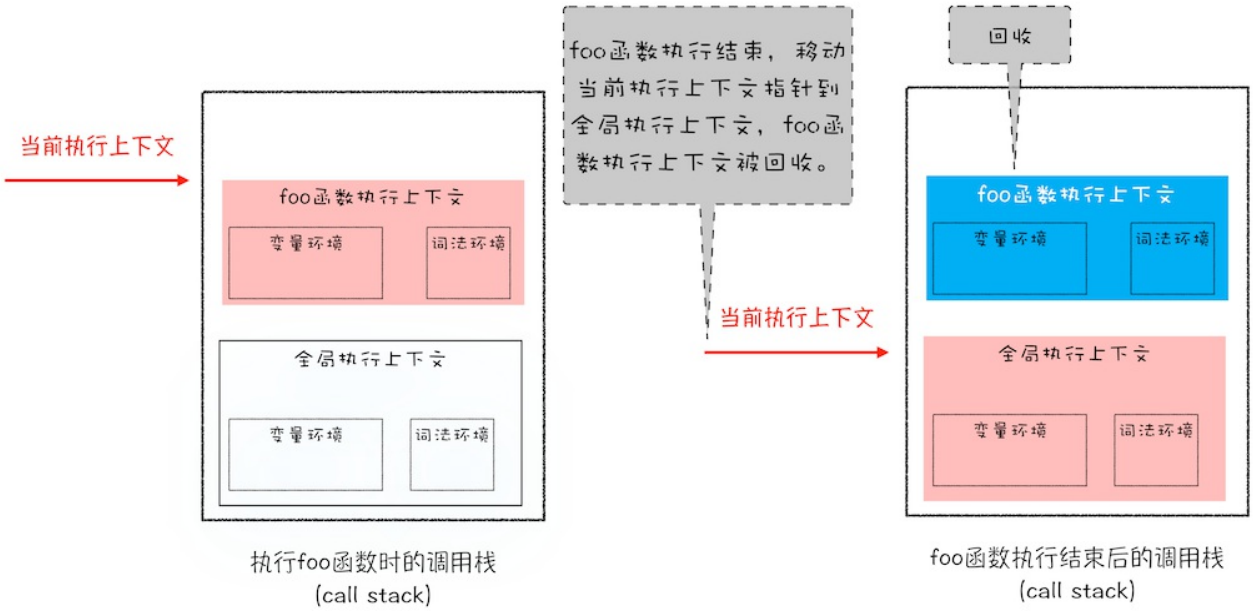


对象类型是“堆”来存储

从上图你可以清晰地观察到，对象类型是存放在堆空间的，在栈空间中只是保留了对象的引用地址，当JavaScript需要访问该数据的时候，是通过栈中的引用地址来访问的，相当于多了一道转手流程。

好了，现在你应该知道了**原始类型的数据值都是直接保存在“栈”中的，引用类型的值是存放在“堆”中的**。不过你也许会好奇，为什么一定要分“堆”和“栈”两个存储空间呢？所有数据直接存放在“栈”中不就可以了吗？

答案是不可以的。这是因为JavaScript引擎需要用栈来维护程序执行期间上下文的状态，如果栈空间大了话，所有的数据都存放在栈空间里面，那么会影响到上下文切换的效率，进而又影响到整个程序的执行效率。比如文中的foo函数执行结束了，JavaScript引擎需要离开当前的执行上下文，只需要将指针下移到上个执行上下文的地址就可以了，foo函数执行上下文栈区空间全部回收，具体过程你可以参考下图：



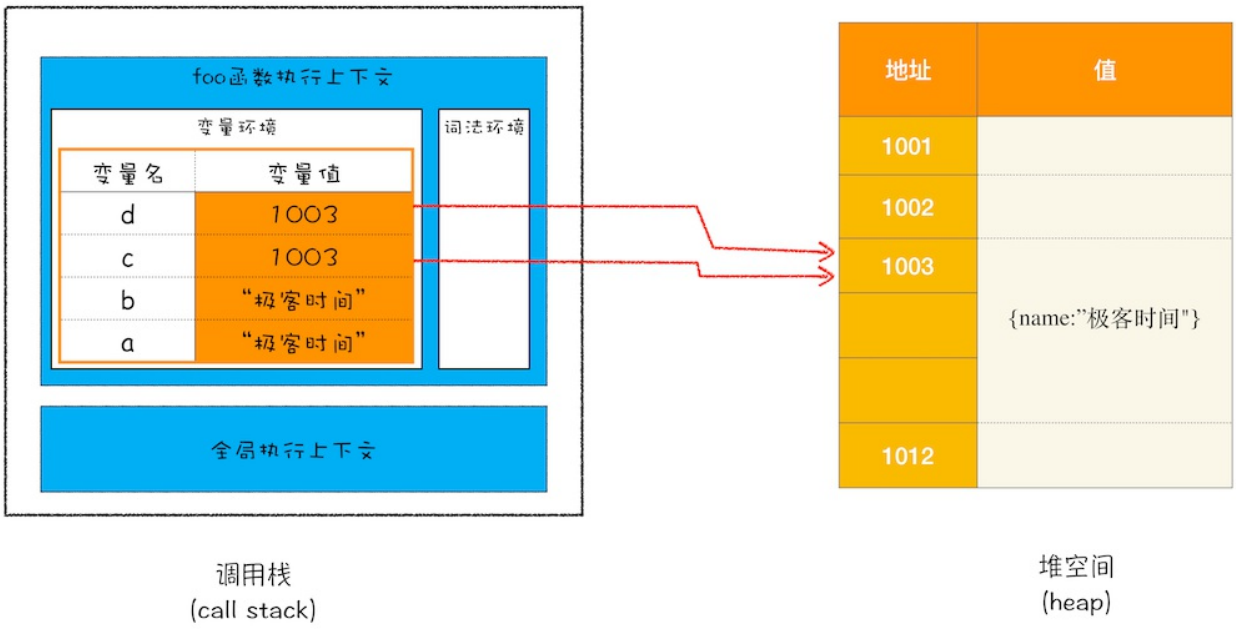
调用栈中切换执行上下文状态

所以**通常情况下，栈空间都不会设置太大，主要用来存放一些原始类型的小数据**。而引用类型的数据占用的空间都比较大，所以这一类数据会被存放到堆中，**堆空间很大，能存放很多大的数据**，不过缺点是分配内存和回收内存都会占用一定的时间。

解释了程序在执行过程中为什么需要堆和栈两种数据结构后，我们还是回到示例代码那里，看看它最后一步将变量c赋值给变量d是怎么执行的？

在JavaScript中，赋值操作和其他语言有很大的不同，**原始类型的赋值会完整复制变量值，而引用类型的赋值是复制引用地址**。

所以d=c的操作就是把c的引用地址赋值给d，你可以参考下图：



引用赋值

从图中你可以看到，变量c和变量d都指向了同一个堆中的对象，所以这就很好地解释了文章开头的那个问题，通过c修改name的值，变量d的值也跟着改变，归根结底它们是同一个对象。

再谈闭包

现在你知道了作用域内的原始类型数据会被存储到栈空间，引用类型会被存储到堆空间，基于这两点的认知，我们再深入一步，探讨下闭包的内存模型。

这里以《10 | 作用域链和闭包：代码中出现相同的变量，JavaScript引擎是如何选择的？》中关于闭包的一段代码为例：

```
function foo() {
    var myName = "极客时间"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        setName:function(newName){
            myName = newName
        },
        getName:function(){
```

```
            console.log(test1)
            return myName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("极客邦")
bar.getName()
console.log(bar.getName())
```

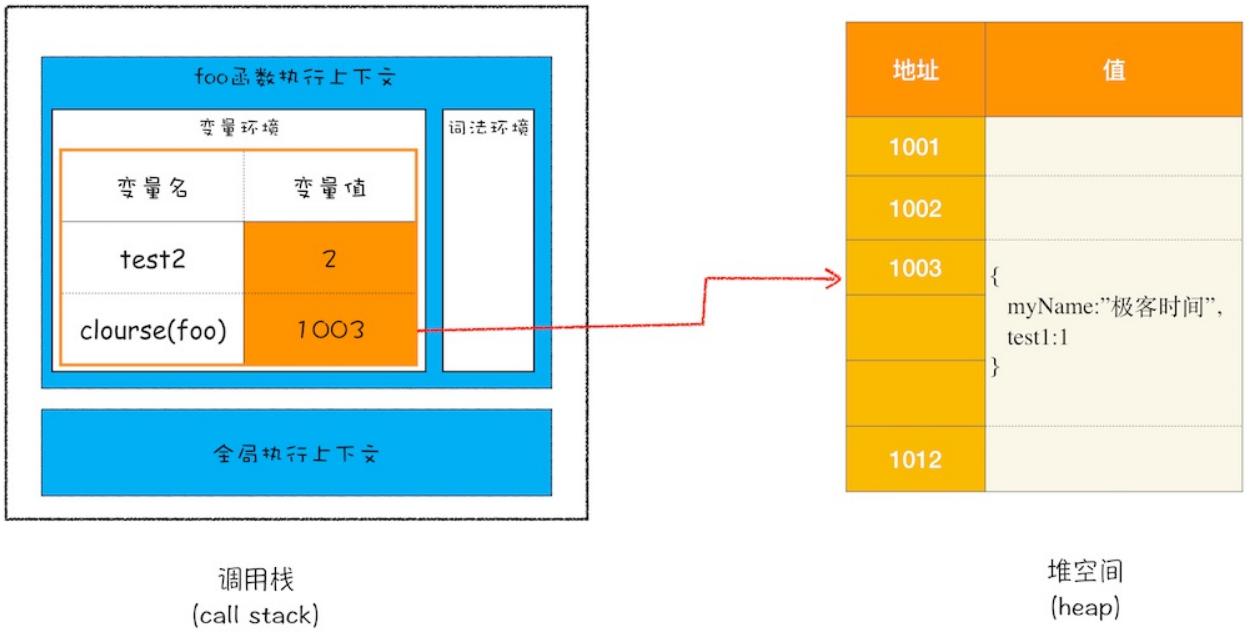
当执行这段代码的时候，你应该有过这样的分析：由于变量myName、test1、test2都是原始类型数据，所以在执行foo函数的时候，它们会被压入到调用栈中；当foo函数执行结束之后，调用栈中foo函数的执行上下文会被销毁，其内部变量myName、test1、test2也应该一同被销毁。

但是在那篇文章中，我们介绍了当foo函数的执行上下文销毁时，由于foo函数产生了闭包，所以变量myName和test1并没有被销毁，而是保存在内存中，那么应该如何解释这个现象呢？

要解释这个现象，我们就得站在内存模型的角度来分析这段代码的执行流程。

1. 当JavaScript引擎执行到foo函数时，首先会编译，并创建一个空执行上下文。
2. 在编译过程中，遇到内部函数setName，JavaScript引擎还要对内部函数做一次快速的词法扫描，发现该内部函数引用了foo函数中的myName变量，由于是内部函数引用了外部函数的变量，所以JavaScript引擎判断这是一个闭包，于是在堆空间创建换一个“clourse(foo)”的对象（这是一个内部对象，JavaScript是无法访问的），用来保存myName变量。
3. 接着继续扫描到getName方法时，发现该函数内部还引用变量test1，于是JavaScript引擎又将test1添加到“clourse(foo)”对象中。这时候堆中的“clourse(foo)”对象中就包含了myName和test1两个变量了。
4. 由于test2并没有被内部函数引用，所以test2依然保存在调用栈中。

通过上面的分析，我们可以画出执行到foo函数中“return innerBar”语句时的调用栈状态，如下图所示：



闭包的产生过程

从上图你可以清晰地看出，当执行到foo函数时，闭包就产生了；当foo函数执行结束之后，返回的getName和setName方法都引用“clourse(foo)”对象，所以即使foo函数退出了，“clourse(foo)”依然被其内部的getName和setName方法引用。所以在下次调用`bar.setName`或者`bar.getName`时，创建的执行上下文中就包含了“clourse(foo)”。

总的来说，产生闭包的核心有两步：第一步是需要预扫描内部函数；第二步是把内部函数引用的外部变量保存到堆中。

总结

好了，今天就讲到这里，下面我来简单总结下今天的要点。

我们介绍了JavaScript中的8种数据类型，它们可以分为两大类——**原始类型和引用类型**。

其中，原始类型的数据是存放在**栈**中，引用类型的数据是存放在**堆**中的。堆中的数据是通过引用和变量关联起来的。也就是说，JavaScript的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

然后我们分析了，在JavaScript中将一个原始类型的变量a赋值给b，那么a和b会相互独立、互不影响；但是将引用类型的变量a赋值给变量b，那会导致a、b两个变量都同时指向了堆中的同一块数据。

最后，我们还站在内存模型的视角分析了闭包的产生过程。

思考时间

在实际的项目中，经常需要完整地拷贝一个对象，也就是说拷贝完成之后两个对象之间就不能互相影响。那该如何实现呢？

结合下面这段代码，你可以分析下它是如何将对象jack拷贝给jack2，然后在完成拷贝操作时两个jack还互不影响的呢。

```
let jack = {
    name : "jack.ma",
    age:40,
    like:{
```

```
        dog:{
            color:'black',
            age:3,
        },
        cat:{
            color:'white',
            age:2
        }
    }
}
function copy(src){
    let dest
    //实现拷贝代码，将src的值完整地拷贝给dest
   //在这里实现
    return dest
}
let jack2 = copy(jack)

//比如修改jack2中的内容，不会影响到jack中的值
jack2.like.dog.color = 'green'
console.log(jack.like.dog.color) //打印出来的应该是 "black"
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。