

06 | x: break x; 搞懂如何在循环外使用break，方知语句执行真解

2019-11-22 周爱民

《JavaScript核心原理解析》

[课程介绍 >](#)



讲述：周爱民

时长 21:08 大小 19.36M



你好，我是周爱民。

上一讲的for语句为你揭开了 JavaScript 执行环境的一角。在执行系统的厚重面纱之下，到底还隐藏了哪些秘密呢？那些所谓的执行环境、上下文、闭包或块与块级作用域，到底有什么用，或者它们之间又是如何相互作用的呢？

接下来的几讲，我就将重点为你讲述这些方面的内容。

用中断（Break）代替跳转

在 Basic 语言还很流行的时代，许多语言的设计中都会让程序代码支持带地址的“语句”。例如，Basic 就为每行代码提供一个标号，你可以把它叫做“行号”，但它又不是绝对物理的行号，通常为了增减程序的方便，会使用“1, 10, 20……”等等这样的间隔。如果想在第 10 行后追加 1 行，就可以将它的行号命名为“11”。



行号是一种很有历史的程序逻辑控制技术，更早一些可以追溯到汇编语言，或可以手写机器代码的时代（确实存在这样的时代）。那时由于程序装入位置被标定成内存的指定位置，所以这个位置也通常就是个地址偏移量，可以用数字化或符号化的形式来表达。

所有这些“为代码语句标示一个位置”的做法，其根本目的都是为了实现“GOTO 跳转”，任何时候都可以通过“GOTO 标号”的语法来转移执行流程。

然而，这种黑科技在 20 世纪的 60~70 年代就已经普遍地被先辈们批判过了。这样的编程方式只会大大地降低程序的可维护性，其正确性或正确性验证都难以保障。所以，后面的故事想必你都知道了，半个多世纪之前开始的“结构化”运动一直影响至今，包括现在我与你讨论的这个 JavaScript，都是“结构化程序设计”思想的产物。

所以，简单地说：JavaScript 中没有 GOTO 语句了。取而代之的，是分块代码，以及基于代码分块的流程控制技术。这些控制逻辑基于一个简单而明了的原则：如果代码分块中需要 GOTO 的逻辑，那么就为它设计一个“自己的 GOTO”。

这样一来，所有的 GOTO 都是“块（或块所在语句）自己知道的”。这使得程序可以在“自己知情的前提下自由地 GOTO”。整体看起来还不错，很酷。然而，问题是那些“标号”啊，或者“程序地址”之类的东西已经被先辈们干掉了，因此就算设计了 GOTO 也找不到去处，那该怎么办呢？

第一种中断

第一种处理方法最为简洁，就是约定“可以通过 GOTO 到达的位置”。

在这种情况下，JavaScript 将 GOTO 的“离开某个语句”这一行为理解为“中断（Break）该语句的执行”。由于这个中断行为是明确针对于该语句的，所以“GOTO 到达的位置”也就可以毫无分歧地约定为该语句（作为代码块）的结束位置。这是“break”作为子句的由来。它用在某些“可中断语句（*BreakableStatement*）”的内部，用于中断并将程序流程“跳转（GOTO）到语句的结束位置”。

在语法上，这表示为（该语法只作用于对“可中断语句”的中断）：


break;



所谓“可中断语句”其实只有两种，包括全部的**循环语句**，以及 **switch 语句**。在这两种语句内部使用的“break;”，采用的就是这种处理机制——中断当前语句，将执行逻辑交给下一语句。

第二种中断

与第一种处理方法的限制不同，第二种中断语句可以中断“任意的标签化语句”。所谓标签化语句，就是在一般语句之前加上“xxx:”这样的标签，用以指示该语句。就如我在文章中写的这两段示例：

 复制代码

```
1 // 标签aaa
2 aaa: {
3     ...
4 }
5
6 // 标符bbb
7 bbb: if (true) {
8     ...
9 }
```

对比这两段示例代码，你难道不会有这么一个疑惑吗？在标签 aaa 中，显然 aaa 指示的是后续的“块语句”的块级作用域；而在标签 bbb 中，if 语句是没有块级作用域的，那么 bbb 到底指示的是“if 语句”呢，还是其后的then分支中的“块语句”呢？


这个问题本质上是在“块级作用域”与“标签作用的（语句）范围”之间撕裂了一条鸿沟。由于标签 bbb 在语义上只是要“标识其后的一行语句”，因此这种指示是与“块级作用域（或词法环境）”没有关系的。简单地说，**标签化语句理解的是“位置”，而不是“（语句在执行环境中的）范围”**。

因此，中断这种标签化语句的“break”的语法，也是显式地用“标签”来标示位置的。例如：

```
break labelName;
```

所以你才会看到，我在文章中写的这两种语句都是可行的：



 复制代码


```
1 // 在if语句的两个分支中都可以使用break;
2 // （在分支中深层嵌套的语句中也是可以使用break的）
```

```

3  aaa: if (true) {
4      ...
5  }
6  else {
7      ...
8      break aaa;
9  }
10
11 // 在try...catch...finally中也可以使用break;
12 bbb: try {
13     ...
14 }
15 finally {
16     break bbb;
17 }

```

对于标签 bbb 的 finally 块中使用的这个特例，我需要再特别说明：如果在 try 或 try..finally 块中使用了 return，那么这个 break 将发生于最后一行语句之后，但是却是在 return 语句之前。例如我在文章中写的这段代码：

 复制代码

```

1  var i = 100;
2  function foo() {
3      bbb: try {
4          console.log("Hi");
5          return i++; // <-位置1: i++表达式将被执行
6      }
7      finally {
8          break bbb;
9      }
10     console.log("Here");
11     return i; // <-位置2
12 }

```

测试如下：

 复制代码

```

1  > foo()
2  Hi
3  Here
4  101

```



在这个例子中，你的预期可能会是“位置 1”返回的 100，而事实上将执行到输出“Here”并通过位置 2 返回 101。这也很好地说明了 **break** 语句本质上就是作用于其后的“一个语句”，而与它“有多少个块级作用域”无关。

执行现场的回收

break 将“语句的‘代码块’”理解为**位置**，而不是理解为作用域 / 环境，这是非常重要的前设！

然而，我在上面已经讲过了，程序代码中的“位置”已经被先辈们干掉了。他们用了半个世纪来证明了一件事情：**想要更好、更稳定和更可读的代码，那么就忘掉“（程序的）位置”这个东西吧！**

通过“作用域”来管理代码的确很好，但是作用域与“语句的位置”以及“GOTO 到新的程序执行”这样的理念是矛盾的。它们并不在同一个语义系统内，这也是**标签与变量**可以重名而不相互影响的根本原因。由于这个原因，在使用标签的代码上下文中，**执行现场的回收**就与传统的“块”以及“块级作用域”根本上不同。

JavaScript 的执行机制包括“执行权”和“数据资源”两个部分，分别映射可计算系统中的“逻辑”与“数据”。而块级作用域（也称为词法作用域）以及其他的作用域本质上就是一帧数据，以保存执行现场的一个瞬时状态（也就是每一个执行步骤后的现场快照）。而 JavaScript 的运行环境被描述为一个后入先出的栈，这个栈顶永远就是当前“执行权”的所有者持用的那一帧数据，也就是代码活动的现场。

JavaScript 的运行环境通过函数的 CALL/RETURN 来模拟上述“数据帧”在栈上的入栈与出栈过程。任何一次函数的调用，即是向栈顶压入该函数的上下文环境（也就是作用域、数据帧等等，它们在不同场合下的相同概念）。所以，包括那些在全局或模块全局中执行的代码，以及 Promise 中执行调度的那些内部处理，所有的这些 JavaScript 内部过程或外部程序都统一地被封装成函数，通过 CALL/RETURN 来激活、挂起。

所以，“作用域”就是在上述过程中被操作的一个对象。

- 作用域退出，就是函数 RETURN。
- 作用域挂起，就是执行权的转移。
- 作用域的创建，就是一个闭包的初始化。



然而如之前所说的，“**break** `labelName`;”这一语法独立于“执行过程”的体系，它表达一个位置的跳转，而不是一个数据帧在栈上的进出栈。这是 `labelName` 独立于标识符体系（也就是词法环境）所带来的附加收益！

基于对“语句”的不同理解，JavaScript 设计了一种全新方法，用来清除这个跳转所带来的影响（也就是回收跳转之前的资源分配）。而这多余出来的设计，其实也是上述收益所需要付出的代价。

语句执行的意义

对于语句的跳转来说，“离开语句”意味着清除语句所持有的一切资源，如同函数退出时回收闭包。但是，这也同样意味着“语句”中发生的一切都消失了，对于函数来说，`return` 和 `yield` 是唯一从这个现场发出信息的方式。那么语句呢？语句的执行现场从这个“程序逻辑的世界”中湮灭之后，又留下了什么呢？

NOTE: 确实存在从函数中传出信息的其他结构，但这些也将援引别的解释方式，这些就留待今后再讲了。

语句执行与函数执行并不一样。函数是求值，所以返回的是对该函数求值的结果（`Result`），该结果或是值（`Value`），或是结果的引用（`Reference`）。而语句是命令，语句执行的返回结果是该命令得以完成的状态（`Completion`, `Completion Record Specification Type`）。

注意，JavaScript 是一门混合了函数式与命令式范型的语言，而这里对函数和语句的不同处理，正是两种语言范型根本上的不同抽象模型带来的差异。

在 ECMAScript 规范层面，本质上所有 JavaScript 的执行都是语句执行（这很大程度上解释了为什么 `eval` 是执行语句）。因此，ECMAScript 规范中对执行的描述都称为“运行期语义（`Runtime Semantics`）”，它描述一个 JavaScript 内部的行为或者用户逻辑的行为的过程与结果。也就是说这些运行期语义都最终会以一个完成状态（`Completion`）来返回。例如：

- 一个函数的调用：调用函数——执行函数体（`EvaluateBody`）并得到它的“完成”结果（`result`）。
- 一个块语句的执行：执行块中的每行语句，得到它们的“完成”结果（`result`）。



这些结果（result）包括的状态有五种，称为完成的类型：normal、break、continue、return、throw。也就是说，任何语句的行为，要么是包含了有效的、可用于计算的数据值（Value）：

- 正常完成（normal）
- 一个函数调用的返回（return）

要么是一个不可（像数据那样）用于计算或传递的纯粹状态：

- 循环过程中的继续下次迭代（continue）
- 中断（break）
- 异常（throw）

NOTE: throw 是一个很特殊的流程控制语句，它与这里的讨论的流程控制有相似性，不同的地方在于：它并不需要标签。关于 throw 更多的特性，我还会在稍后的课程中给你具体地分析。

所以当运行期出现了一个称为“中断（break）”的状态时，JavaScript 引擎需要找到这个“break”标示的目标位置（**result.Target**），然后与当前语句的标签（如果有的话）对比：

- 如果一样，则取 break 源位置的语句执行结果为值（Value）并以正常完成状态返回；
- 如果不一样，则继续返回 break 状态。

这与函数调用的过程有一点类似之处：由于对“break 状态”的拦截交给语句退出（完成）之后的下一个语句，因此如果语句是嵌套的，那么其后续（也就是外层的）语句就可以得到处理这个“break 状态”的机会。举例来说：

```
1 console.log(eval(`
2   aaa: {
3     1+2;
4     bbb: {
5       3+4;
6       break aaa;
7     }
8   }
`))
```

 复制代码



在这个示例中，“break aaa”语句是发生于 bbb 标签所示块中的。但当这个中断发生时，

- 标签化语句 bbb 将首先捕获到这个语句完成状态，并携带有标签 aaa；
- 由于 bbb 语句完成时检查到的状态中的中断目标（Target）与自己的标签不同，所以它将这个状态继续作为自己的完成状态，返回给外层的 aaa 标签化语句 aaa；
- 语句 aaa 得到上述状态，并对比标签成功，返回结果为语句 3+4 的值（作为完成状态传出）。

所以，语句执行总是返回它的完成状态，且如果这个完成状态是包含值（Value）的话，那么它是可以作为 JavaScript 代码可访问的数据来使用的。例如，如果该语句被作为 eval() 来执行，那么它就是 eval() 函数返回的值。

中断语句的特殊性

最后的一个问题是：标题中的这行代码有什么特殊性呢？

相信你知道我总是会设计一些难解的，以及表面上矛盾和歧义的代码，并围绕这样的代码来组织我的专题的每一讲的内容。而今天这行代码在“貌似难解”的背后，其实并不包含任何特殊的执行效果，它的执行过程并不会对其他任何代码构成任何影响。

我列出这行代码的原因有两点。

1. 它是最小化的 break 语句的用法，你不可能写出更短的代码来做 break 的示例了；
2. 这种所谓“不会对其他任何代码构成任何影响”的语句，也是 JavaScript 中的特有设计。

首先，由于“标签化语句”必须作用于“一个”语句，而**语句**理论上的最小化形式是“空语句”。但是将空语句作为 break 的目标标签语句是不可能的，因为你还必须在标签语句所示的语句范围内使用 break 来中断。空语句以及其他一些单语句是没有这样的语句范围的，因此最小化的示例就只能是对 break 语句自身的中断。

其次，语句的返回与函数的返回有相似性。例如，函数可以不返回任何东西给外部，这种情况下外部代码得到的函数出口信息会是 undefined 值。



由于典型的函数式语言的“函数”应该是没有副作用的，所以这意味着该函数的执行过程不影响任何其他逻辑——也不在这个“程序逻辑的世界”中留下任何的状态。事实上，你还可以用“void”运算符来阻止一个函数返回的值影响它的外部世界。函数是“表达式运算”这个体系中的，因此用一个运算符来限制它的逻辑，这很合理。

虽然“**break** labelName”的中止过程是可以传出“最后执行语句”的状态的，但是你只要回忆一下这个过程就会发现一个悖论：**任何被 break 的代码上下文中，最后执行语句必然是“break 语句”本身！**所以，如果要在这个逻辑中实现“语句执行状态”的传递，那么就必须确保：

1. “break 语句”不返回任何值（ECMAScript 内部约定用“Empty”值来表示）；
2. 上述“不返回任何值”的语句，也不会影响任何语句的既有返回值。

所以，事实上我们已经探究了“break 语句”返回值的两个关键特性的由来：

- 它的类型必然是“break”；
- 它的返回值必然是“空（Empty）”。

对于 Empty 值，在 ECMAScript 中约定：在多行语句执行时它可以被其他非 Empty 值更新（UpdateEmpty），而 Empty 不可以覆盖其他任何值。

这就是空语句等也同样“不会对其他任何代码构成任何影响”的原因了。

知识回顾

今天的内容有一些非常重要的、关键的点，主要包括：

1. “GOTO 语句是有害的。”——1972 年图灵奖得主艾兹格·迪科斯彻（Edsger Wybe Dijkstra, 1968）。
2. 很多新的语句或语法被设计出来用来替代 GOTO 的效果的，但考虑到 GOTO 的失败以及无与伦比的破坏性，这些新语法都被设计为功能受限的了。
3. 任何一种 GOTO 带来的都是对“顺序执行”过程的中断以及现场的破坏，所以也都存在相应的执行现场回收的机制。



4. 有两种中断语句，它们的语义和应用场景都不相同。
5. 语句有返回值。
6. 在顺序执行时，当语句返回 Empty 的时候，不会改写既有的其他语句的返回值。
7. 标题中的代码，是一个“最小化的 break 语句示例”。

思考题

- 找到其他返回 Empty 的语句。
- 尝试完整地对比函数执行与语句执行的过程。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 4  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 加餐 | 捡豆吃豆的学问（下）：这门课该怎么学？

[下一篇](#) 07 | ``${1}``：详解JavaScript中特殊的可执行结构



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费 



精选留言 (25)

 写留言



不将就

2019-11-22

老师，问个问题，

```
{
```

```
let a=10
```

```
}
```

这是块级作用域

```
{
```

```
var a=10
```

```
}
```

在外层可以访问到a为未定义，这是不是可以说明{}这对括号里只有出现let/const才算有块级作用域？但是如下

```
if(1){
```

```
let b=10
```

```
}
```

这个if语句有括号而且用了let，老师为什么又说if语句没有块级作用域？

作者回复：不是。块级作用域与它内部声明了什么没关系。例如，一个“块语句{ }”就有一个块级作用域，哪怕它内部一行代码也没有。



对于你说的if(1)这个例子来说，这里有两个语句，一个是if语句本身，它是个“单语句”（ECMAScript就是这么定义的，NodeJS的错误提示里也有），它没有块级作用域；而后面的一对大括号“{}”，是一个“块语句”，有一个块。

传统习惯上过来的开发人员会把“if () { ... }”理解成一个语句，而在JavaScript中，这是两个语句。



👍 25



海绵薇薇

2019-11-28

Hello, 老师好：） 阅读完文章还存在如下问题，期待有解答或方向，感谢：）

```
try {  
  
    1  
  
} finally {  
  
    console.log('finally')  
  
    2  
  
}
```

输出：

> finally

> 1

1. try finally 语句输出的Result 是{type: normal, value: 1}。但是最后一个语句是finally中的2，value不应该是2吗？

```
try {  
  
    throw 1  
  
} catch(ex) {  
  
    2
```



```
}
```

这里确实输出了2。

```
function foo() {
```

```
  aaa: try {
```

```
    return 1;
```

```
  } finally {
```

```
    break aaa;
```

```
  }
```

```
}
```

return 1 Result是{type: return, value: 1}

break Result是{type: break, value: empty, target: aaa}

2. 这里finally中语句的结果却覆盖了try中语句的结果，这是一个特例吗？

作者回复: 我之前没有注意过这个例子，倒是忽略了它在语句执行上的特点。不过这并不算特例。

因为在finally{}块中的执行流程仍然会回到try{}块，例如说，你在try{}块中使用return语句，那么在return之前会执行到finally{}块，而finally{}执行完之后，还会回到try{}块里的return语句来返回。所以最终“完成并退出”整个try语句的，还是try块。

在效果上，这类似于（也就是finally{}是一个call()）：

```
try {
```

```
  return void finally(), x;
```

```
}
```

```
catch {}
```



👍 16



Elmer

2019-12-10

觉得函数执行应该是语句执行的一部分或者一个特例，返回值都已经统一为文中的result。

只不过函数执行具体实现了本身的上下文创建与回收，并用额外的栈来记录当前执行状况。两者都是流程控制的一种形式。关系应为语句执行包含函数执行。不知道理解的对不对。

作者回复: 其实真实的情况与你想的有点区别（也与我在文章中讲的有点细节上的不同）。关键在于：所有的表达式，原则上都是既可以返回完成记录，也可以返回引用，也可以返回值的。

后面两种比较容易理解，但表达式返回“完成记录”的意义在哪儿呢？多数情况下是没有意义的，但是只有允许这种情况，ECMAScript才能在表达式（的实现逻辑）中抛异常啊。所以多数情况下表达式返回值的Result都是值或引用两种，但偶尔也会返回类型为Throw的异常完成记录。也正是因为这个缘故，在ECMAScript中，所有所有的取表达式计算结果的写法，都采用类似下面这种模式：

```
...
* Let ref be the result of evaluating ...
* Let val be ? GetValue(ref)
...
```

首先，在GetValue()里面会写，如果ref不是引用，那么就直接返回，这样GetValue就会把“异常类型”的完成记录原样抛出来。然后，你注意第二行中的那个“?”号，那个表明如果GetValue()的调用结果是“异常类型”的完成记录，那么就结束当前的执行，继续把异常往外抛。

而且?号还有一个作用，就是直接从Normal类型的完成记录中把值解出来。也就是如果r是NormalCompletion，那么r = r.value。这样一来，就确保任何`?...`操作的结果，要么是异常被抛出，要么就是完成记录r中的值（r.value）。

所以，事实上整个“表达式执行”的结果Result也是支持返回值为完成记录的（而不仅仅是引用和值），只是绝大多数都过滤掉了。

接下来才是你的问题。函数执行也只是正常地返回了一个完成记录而已（如上面所说的，这是正常的行为，而不是语句执行的特例）。如果它是使用Return，那么也会在调用完成前被替换成Normal类型。然后函数调用操作会保证在完成之前得到的仅仅是一个一般的JavaScript语言类型中的数据（Result），或者非正常的完成类型。你看看这里就明白了：

```
...
// FROM: https://tc39.es/ecma262/#sec-evaluatecall

...
// 取函数调用结果
* Let result be Call(func, thisValue, argList).

// 断言：要么是非正常返回，要么就是语言类型
* Assert: If result is not an abrupt completion, then Type(result) is an ECMAScript language type.
...
```



而Call()是调用F.[[Call]]来实现的，它的主要代码就一行：

...

```
// FROM: https://tc39.es/ecma262/#sec-call
```

```
* Return ? F.[[Call]](V, argumentsList).
```

...

注意这里的？号，就是要么抛异常出去，要么就是把结果（完成记录r）中的值（r.value）取出来了。——这里再强调一个小的关键点：函数调用是不能返回规范类型中的“引用”的，也就是说结果值已经用GetValue(ref)把值取出来过了。

共 2 条评论 >

👍 8



zcdll

2019-11-22

返回 Empty 的语句，是不是还有 单独的一个 分号，和 if 不写大括号，或者大括号中为空？

作者回复: 你说的都是。不过也不止的哟。比如说break语句自己就返回empty呀，还有continue，还有for语句的某些处理，以及yield等等，都有返回Empty的情况。



👍 6



穿秋裤的男孩

2019-11-26

所谓“可中断语句”其实只有两种，包括全部的循环语句，以及 switch 语句。

老师，那forEach不属于循环语句吗？为什么break不可以在forEach中使用呢

作者回复: 如果你说的是`for each (... in ...)`，那么这个语句不在ECMAScript的规范里面，在mozilla的spidermonkey引擎里，也是被废弃的特性了。

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for_each...in

共 5 条评论 >

👍 5



林逸舟

2020-04-06

尝试完整地对比函数执行与语句执行的过程：

·操作返回值：

函数执行：在函数体的最后进行一次返回值的赋值



语句执行：在每句后更新返回值

如下所示：

```
function foo() {  
    1 + 1;  
    return 1; //函数的执行结果为1 赋值动作仅有一次  
}
```

foo();

```
{  
    1 + 1; //整个块语句的执行结果更新为2  
    2 + 2; //整个块语句的执行结果更新为4  
}
```

·堆栈顺序

函数执行与语句执行类似 是先入后出的堆栈

如下所示：

```
function bar() {  
    return;  
}
```

```
function foo() {  
    1 + 1;  
    bar()  
    return 1; //函数的执行结果为1 赋值动作仅有一次  
}
```

foo();

//开始执行foo->开始执行bar->bar执行结束->foo执行结束

```
{  
    1 + 1; //整个块语句的执行结果更新为2  
    2 + 2; //整个块语句的执行结果更新为4  
}
```



//开始执行块语句{}->执行1+1->1+1执行结束->执行2+2->2+2执行结束->块语句执行结束

·操作Result的对象

函数：getValue(ref)||ref传递给上一层表达式使用

语句：Completion传递给引擎进行使用

作者回复: 谢谢。挺好的一份答案。

其中有些东西阅读到后面的小结就可以得到印证了。^^.

共 2 条评论 >

👍 4



Real Aaron

2019-12-12

【学习方式大变化】

前5讲看下来，主要有两个感觉：

- 1、课程的内容非常深入而且重要，经常中间看到一段文字，就有一种“原来如此“的体验。
- 2、逻辑顺序看不懂，看完一讲之后，好像学到一些零碎知识，但串不一起来。

今天凑巧看到了加餐中的”学习这门课的正确姿势“，原来老师用心良苦，没有将知识点清晰的串起来是希望大家自己能主动理清思路，串出逻辑。

参考加餐中的方法，今天换了一种学习方式：一边学习内容，一边将关键词和疑惑（dots）写在本子上，反复琢磨其中的来龙去脉。最终写满了两页纸，然后将其中的各个点串起来（connecting the dots），形成了下面的笔记。

【本讲的一些记录和归纳】

1、执行结果方面：

JavaScript 是一门混合了函数式与命令式范型的语言，对函数和语句的不同处理，正是两种语言范型根本上的不同抽象模型带来的差异。

本质上所有 JavaScript 的执行都是语句执行（包括函数执行），语句执行的过程因语句类型而异，但结果都返回的是一个“完成”结果。

但【函数语句执行】和【普通语句（非函数）执行】的区别在于：函数语句执行返回的“完成”结果是值或者引用（未报异常的情况下），而普通语句执行返回的是一个完成状态（Completion）。



2、执行过程方面：

总体来讲，

JavaScript 的执行机制包含两部分：【执行权（逻辑）】和【数据资源（数据）】

JavaScript的执行（运行）环境：是一个后入先出的栈，栈顶就是当前“执行权”拥有者所持有的那一帧数据，运行环境通过函数的 CALL/RETURN 来模拟“数据帧”（也称上下文环境或作用域）在栈上的入栈和出栈过程。

但"break labelName"这一语法跟上面不同，它表达一个位置的跳转，而不是一个数据帧的进出栈。

另外，各种类型的语句执行过程（内部逻辑）也可能有差异：

2.1 函数执行过程

2.2 break 执行过程

2.3 case 执行过程

2.4 switch 执行过程

2.5 循环语句执行过程

2.6 try...catch 执行过程

【仍旧未解的疑问】

1、函数执行和语句执行返回的都是一个完成状态？还是函数执行返回的只能是值或引用？亦或是其他说法？

希望老师能解答一下，非常感谢。

作者回复：谢谢Aaron。

我想你读过前11章，看到第二篇加餐内容（“让JavaScript运行起来”）之后，你的大多数问题就都有解了，而且会对你已经领悟到的内容有许多“更新”，认识会再加深一些的。

说回你最后的两个疑问。表达式执行（包括函数执行），本质上都是求值运算，所以它们应当只返回值。但是事实上所有的执行——包括函数、表达式和语句也都“同时”是可以返回完成状态，这样才能在表达式中向外抛异常，因为异常抛出就是一个完成状态。

但是ECMAScript对所有在表达式层面上返回的“完成状态”做了处理，相当于在语言层面上“消化



了”这些状态。所以绝大多数情况下，你认为表达式执行返回的Result是值或引用就好了。稍有例外的是，函数调用返回的是一个type为Return的完成状态，只不过它在内部方法Call处理之后，也已经变成了值而已。

关于这个问题，正好是在这一课的留言中，我给Elmer的回复中解释了更多的细节。你可以看看。

共 2 条评论 >

👍 3



桔子

2019-11-23

- 1、可以理解为函数中return的设计是为了传递函数的状态，break的设计则是为了传递语句的状态么？
- 2、可以认为break;只可以中断语句，不能用在函数中，break label;可以用在函数中，它返回了上一行语句的完成状态并作为所在函数的返回值？

作者回复: 1. 可以。

2. 不太对。break labelName只与“块”相关，与函数没直接关系。语句的“块”也是有返回值的，因为JavaScript里面存在“语句执行是有值的”这个设定。

注意有许多语句是有“块（块级作用域）”的，而不仅仅是块语句（也就是一对大括号，它称为Block语句）。



👍 3



undefined

2021-04-29

我又来了...

原文 ↓

如果在 try 或 try..except 块中使用了 return

应该是

try...finally

作者回复: Oh ya.... :(~

这回是我的锅。呜呼。我请编辑们改过改过~



👍 1



孜孜

2020-06-15

表达式的result引用和语句的result有联系吗？

作者回复: 这个在11讲之后的加餐“让JavaScript运行起来”里面有讲（还有图）。在这里：

> <https://time.geekbang.org/column/article/175261>



1

Elmer

2019-12-26

求函数执行与语句执行的过程对比。

作者回复: 函数执行啊，其实是表达式执行的特例。它会通过完成记录来返回return语句返回结果。

但是，在内部过程Call()的调用中它会取出值，而不是直接返回“Return类型的完成类型”。所以在“函数调用作为表达式的操作数”时，运算处理的还是“Result/Value值”，而不是“完成记录”。

由于函数调用会“从完成记录中取出值”，所以它不能返回“引用（规范类型）”。举例来说：

...

示例

```
> obj = { foo() { return this === obj } }
```

分组表达式能返回引用

```
> (obj.foo)()
```

true

return不能返回引用

```
> (function () { return obj.foo })()
```

false

...



2

晓小东

2019-11-26

老师有一个表达式执行让我感到困惑，我在做位运算符的时候碰到这么一个现象

```
let n = 2;
```

```
let c1 = n != 0;
```




```
let c2 = (n & (n - 1)) === 0;
let c3 = n & (n - 1) === 0;
console.log(c1, c2, c3);
```

打印: true true 0

c3 结果为什么变成了0 按照表达式 左右操作数的逻辑

作者回复: 这个是优先级的问题。你在c3里面去掉了一对括号, 运算符的优先级变了。

共 2 条评论 >

👍 1



westfall

2019-11-22

第一次看到标签化语句, 请问老师, 标签化语句除了用来 break, 在实际的开发中还有哪些应用场景?



👍 1



黑山老妖

2022-01-13

1、传统习惯上过来的开发人员会把“if () { ... }”理解成一个语句, 而在JavaScript中, 这是两个语句。

2、在try{}块中使用return语句, 那么在return之前会执行到finally{}块, 而finally{}执行完之后, 还会回到try{}块里的return语句来返回。所以最终“完成并退出”整个try语句的, 还是try块。

3、·操作Result的对象

函数: getValue(ref)||ref传递给上一层表达式使用

语句: Completion传递给引擎进行使用

4、所谓“可中断语句”其实只有两种, 包括全部的循环语句, 以及 switch 语句。

5、1、执行结果方面:

JavaScript 是一门混合了函数式与命令式范型的语言, 对函数和语句的不同处理, 正是两种语言范型根本上的不同抽象模型带来的差异。

本质上所有 JavaScript 的执行都是语句执行 (包括函数执行), 语句执行的过程因语句类型而异, 但结果都返回的是一个“完成”结果。

但【函数语句执行】和【普通语句 (非函数) 执行】的区别在于: 函数语句执行返回的“完成”结果是值或者引用 (未报异常的情况下), 而普通语句执行返回的是一个完成状态 (Completion) 。



2、执行过程方面：

总体来讲，

JavaScript 的执行机制包含两部分：【执行权（逻辑）】和【数据资源（数据）】

JavaScript的执行（运行）环境：是一个后入先出的栈，栈顶就是当前“执行权”拥有者所持有的那一帧数据，运行环境通过函数的 CALL/RETURN 来模拟“数据帧”（也称上下文环境或作用域）在栈上的入栈和出栈过程。

但"break labelName"这一语法跟上面不同，它表达一个位置的跳转，而不是一个数据帧的进出栈。

另外，各种类型的语句执行过程（内部逻辑）也可能有差异：

2.1 函数执行过程

2.2 break 执行过程

2.3 case 执行过程

2.4 switch 执行过程

2.5 循环语句执行过程

2.6 try...catch 执行过程

【仍旧未解的疑问】

1、函数执行和语句执行返回的都是一个完成状态？还是函数执行返回的只能是值或引用？亦或是其他说法？表达式执行（包括函数执行），本质上都是求值运算，所以它们应当只返回值。但是事实上所有的执行——包括函数、表达式和语句也都“同时”是可以返回完成状态，这样才能在表达式中向外抛异常，因为异常抛出就是一个完成状态。

但是ECMAScript对所有在表达式层面上返回的“完成状态”做了处理，相当于在语言层面上“消化了”这些状态。所以绝大多数情况下，你认为表达式执行返回的Result是值或引用就好了。稍有例外的是，函数调用返回的是一个type为Return的完成状态，只不过它在内部方法Call处理之后，也已经变成了值而已。

1、可以理解为函数中return的设计是为了传递函数的状态，break的设计则是为了传递语句的状态么？可以

2、可以认为break;只可以中断语句，不能用在函数中，break label;可以用在函数中，它返回了上一行语句的完成状态并作为所在函数的返回值？不太对。break labelName只与“块”相关，与函数没直接关系。语句的“块”也是有返回值的，因为JavaScript里面存在“语句执行是



有值的”这个设定。

注意有许多语句是有“块（块级作用域）”的，而不仅仅是块语句（也就是一对大括号，它称为Block语句）。

函数执行啊，其实是表达式执行的特例。它会通过完成记录来返回return语句返回结果。

但是，在内部过程Call()的调用中它会取出值，而不是直接返回“Return类型的完成类型”。所以在“函数调用作为表达式的操作数”时，运算处理的还是“Result/Value值”，而不是“完成记录”。

由于函数调用会“从完成记录中取出值”，所以它不能返回“引用（规范类型）”

在js中，语句执行跟表达式执行是分开的，是两种不同概念的东西。而函数执行其实是表达式执行的一种，其中函数名（亦即是函数）是运算数，而一对括号是运算符。——这是确实的，并且这个称为“函数调用运算符”的括号也是有优先级的，你可以直接在MDN里面查到。

表面来看，函数就是一堆语句，但其实“函数执行”时的返回值是由return来决定的，对吧。而语句执行却不是，语句执行的结果值是由“最后一个有效语句”来决定的。当你使用eval()来执行一批语句时，就可以看到这个结果值了。——并且，这也是语句执行要被拿出来讨论的原因，亦即是“动态执行”执行的是语句，而不是函数，也不是表达式。

作者回复: 非常棒的总结!

关于未解疑问中的第1个，你自己的回答是对的，就是语言层面上消化了那些问题。——既然RETURN是一个完成状态，那么就一定是按语句返回来做的，但f()是一个“函数调用表达式”，那就是最终按表达式运算结果做了处理。

两个分开的小的疑问点。其一，函数的RETURN不是用来“传状态”。如果函数内有一个特别的状态，那一定是THROW，如果没这样的状态，那就一定RETURN出来个东西。函数的RETURN要与参数结合起来理解，就是一进一出的求值=>数据传递方式。——多说一句，yield带来的是多进多出的数据传递。

其二是关于break labelName的，你的理解没错。它只与块相关。

其它的都非常赞，表达的也清晰明了。再赞+n



1



ayu

2022-01-11

```
eval( 'aaa: { 1+2; bbb: { void 7; break aaa; } }') // NaN
```

老师能解释下返回 NaN 而不是 undefined 的原理吗

作者回复: 在shell中执行Node:

```
> node -p -e 'eval( "aaa: { 1+2; bbb: { void 7; break aaa; } }")'
undefined
```

结果是undefined呀?



Sam

2021-07-30

```
function testBlock () {
```

```
  let t = 1;
```

```
  try {
```

```
    t = 2;
```

```
    return t;
```

```
  }
```

```
  finally {
```

```
    t = 3
```

```
  }
```

```
}
```

```
console.log(testBlock()) //输出为2
```

想请教下老师:

1. 如果finally块, 是在try的块中return语句执行前执行话, 怎么返回的变量t是try块中赋的值
2. 在try和finally中的t变量与外部let定义的t是同一个吗?

作者回复: Q1

t作为单值表达式被先计算, 然后进入finally块, 然后再处理return 语义来返回。

Q2

在你的例子中是同一个。但try/catch/finally中如果声明了自己的let t, 那么他们与外部的t, 以及他们之间都不是同一个。



王美红

2021-03-05

老师，想问下，函数不就是语句构成的，为什么要说 函数执行和语句执行的结果不一样呢？不太懂为啥要这样分开讨论？

作者回复: 在js中，语句执行跟表达式执行是分开的，是两种不同概念的东西。而函数执行其实是表达式执行的一种，其中函数名（亦即是函数）是运算数，而一对括号是运算符。——这是确实的，并且这个称为“函数调用运算符”的括号也是有优先级的，你可以直接在MDN里面查到。

表面来看，函数就是一堆语句，但其实“函数执行”时的返回值是由return来决定的，对吧。而语句执行却不是，语句执行的结果值是由“最后一个有效语句”来决定的。当你使用eval()来执行一批语句时，就可以看到这个结果值了。——并且，这也是语句执行要被拿出来讨论的原因，亦即是“动态执行”执行的是语句，而不是函数，也不是表达式。

一点细节，可以参见这篇文章：

<https://blog.csdn.net/aimingoo/article/details/51136511>



爱呀顶呀

2020-07-13

```
// 在if语句的两个分支中都可以使用break;
// （在分支中深层嵌套的语句中也是可以使用break的）
aaa: if (true) {
    ...
}
else {
    ...
    break aaa;
}

// 在try...catch...finally中也可以使用break;
bbb: try {
    ...
}
finally {
    break bbb;
}
```

能详细解释下吗？ 看不懂呀



作者回复: 我倒没有明白你的问题是什么呢?

对于try的例子, 课程的正语言紧接着就有一个。if语句也并没有什么特殊的, 你试着写一个就明白了。

总之break就是提前中止语句执行, 在break语句之后的都不会执行到了。



卡尔

2020-06-15

函数执行, 语句执行, 表达式执行。后两者是什么关系?

作者回复: 这个在11讲之后的加餐“让JavaScript运行起来”里面有讲(还有图)。在这里:

> <https://time.geekbang.org/column/article/175261>

共 2 条评论 >



HoSalt

2020-05-11

```
aaa: if (true) {
```

```
  ...
```

```
} else {
```

```
  ...
```

```
  break bbb;
```

```
}
```

这惹人应该是break aaa 吧

作者回复: 好的。多谢。已经申请在线fix了。

