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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
```

```
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
```

```
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
```

```
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
```

```
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。

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：

```
// 标签aaa
aaa: {
   ...
}

// 标符bbb
bbb: if (true) {
   ...
}
```

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：

```
// 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;
}
```

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：

```
var i = 100;
function foo() {
  bbb: try {
    console.log("Hi");
    return i++; //  <-位置1：i++表达式将被执行
  }
  finally {
    break bbb;
  }
  console.log("Here");
  return i; //  <-位置2
}
```

测试如下：

```
> foo()
Hi
Here
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是唯二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这个称为“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：

```
console.log(eval(`
  aaa: {
    1+2;
    bbb: {
     3+4;
     break aaa;
    }
  }
`)); // 输出值：7
```

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，最后执行语句必然会是“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. 任何的一种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。