你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript

中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果"至少会返回一个undefined值"，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓"引用"，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个"执行表达式并'取值'"的操作，以便确保不会有"引用"类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号"? *opName*()"语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个"处理（*opName()*）"的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓"表达式语句"（这是排在"最简单语句榜"的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1.**Let** exprRef be the result of evaluating Expression.
2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的"***let** ... result of evaluating ...*"。其中的"result of evaluating…"基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解"执行一行语句"与"执行一个表达式"的不同，并且由此决定它们返回的是一个"引用记录"还是"完成记录"（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将"引用"理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或Parse Tree中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程[[Call]]，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的throw 1就排在这个“最简单榜”第三名。

NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果"至少会返回一个undefined值"，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓"引用"，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个"执行表达式并'取值'"的操作，以便确保不会有"引用"类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号"? *opName*()"语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个"处理（*opName()*）"的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓"表达式语句"（这是排在"最简单语句榜"的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1.**Let** exprRef be the result of evaluating Expression.
2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的"***let** ... result of evaluating ...*"。其中的"result of evaluating…"基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解"执行一行语句"与"执行一个表达式"的不同，并且由此决定它们返回的是一个"引用记录"还是"完成记录"（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将"引用"理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它"外部的代码块（或`Parse Tree`中的父级结点）"间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执

行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2. **Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1. **Let** exprRef be the result of evaluating Expression.
> 2. **Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let*** *... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

ThrowStatement : **throw** *Expression*;
1.**Let** exprRef be the result of evaluating Expression.
2.**Let** exprValue be ? GetValue(exprRef).
3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

Return Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

Return Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;
```

```
// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
```

```
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的throw 1就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

throw语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个Let语句，以及最后一个Return返回值。当然，这里的Let/Return是自然语言的语法描述，是

ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)
```

```
// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
```

```
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的throw 1就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

throw语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个Let语句，以及最后一个Return返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

Return Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

Return Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值2;

> NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用

作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

1. **Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

Return Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存

在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1.**Let** exprRef be the result of evaluating Expression.
2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执

行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

1. **Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

Return Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；

2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值2;

NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1. **Let** exprRef be the result of evaluating Expression.
> 2. **Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回empty的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回empty，因此不覆盖既有的值，所以整个语句块的执行结果是2。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回empty，于是第2行的结果1覆盖了它；而第3行的结果仍然是empty所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果"至少会返回一个undefined值"，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定if语句不返回empty，所以第1行返回的值2将被覆盖，最终显示为undefined。而在此之前（例如NodeJS 4），它将返回值2;

NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：empty，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓"引用"，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个"执行表达式并'取值'"的操作，以便确保不会有"引用"类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的throw 1语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号"? *opName*()"语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个"处理（*opName()*）"的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个throw语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是? GetValue(exprRef)就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓"表达式语句"（这是排在"最简单语句榜"的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1.**Let** exprRef be the result of evaluating Expression.

2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let*** *... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或Parse Tree中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程[[Call]]，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
```

```
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果"至少会返回一个undefined值"，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓"引用"，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个"执行表达式并'取值'"的操作，以便确保不会有"引用"类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号"? *opName*()"语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个"处理（*opName()*）"的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓"表达式语句"（这是排在"最简单语句榜"的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1.**Let** exprRef be the result of evaluating Expression.
2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的"***let*** *... result of evaluating ...*"。其中的"result of evaluating…"基本上算是ECMAScript中一个约

定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或Parse Tree中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程[[Call]]，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回empty，因此不覆盖既有的值，所以整个语句块的执行结果是2。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回empty，于是第2行的结果1覆盖了它；而第3行的结果仍然是empty所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果"至少会返回一个undefined值"，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定if语句不返回empty，所以第1行返回的值2将被覆盖，最终显示为undefined。而在此之前（例如NodeJS 4），它将返回值2;

> NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：empty，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓"引用"，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个"执行表达式并'取值'"的操作，以便确保不会有"引用"类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的throw 1语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号"? *opName*()"语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个"处理（*opName()*）"的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个throw语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是? GetValue(exprRef)就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓"表达式语句"（这是排在"最简单语句榜"的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的"***let*** *... result of evaluating ...*"。其中的"result of evaluating…"基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解"执行一行语句"与"执行一个表达式"的不同，并且由此决定它们返回的是一个"引用记录"还是"完成记录"（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将"引用"理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、

return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或Parse Tree中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程[[Call]]，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
```

```
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是

normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

`throw`语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

NOTE: 参见13.2.13 Block -> RS: Evaluation, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

2. **Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

ExpressionStatement: *Expression*;
1. **Let** exprRef be the result of evaluating Expression.
2. **Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替

换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中的`ReturnIfAbrupt(x)`向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级`Parse Node`，也就是`Script`或`Module`类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的`throw 1`就排在这个“最简单榜”第三名。

NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

throw语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个Let语句，以及最后一个Return返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行throw语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的[[target]]字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的NormalCompletion()也是一个简写，完整的表示法与上面的ThrowCompletion()也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有[[Target]]）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句throw 1，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以throw 1为例，谁才是throw语句的执行者呢？

在语句之外看语句

在JavaScript中，除了eval()之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是eval()，由于它的语义是“语句执行并求值”，所以事实上从eval()的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值`2`将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值`2`;

> NOTE: 参考阅读[《前端要给力之：语句在JavaScript中的值》]。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值1，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算GetValue()来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的Parse Tree/Nodes中的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。

你好，我是周爱民，欢迎回到我的专栏。

今天我将为你介绍的是在ECMAScript规范中，实现起来“最简单”的JavaScript语法榜前三名的JavaScript语句。

标题中的throw 1就排在这个“最简单榜”第三名。

> NOTE: 预定的加餐将是下一讲的内容，敬请期待。^^.

为什么讲最简单语法榜

为什么要介绍这个所谓的“最简单的JavaScript语法榜”呢？

在我看来，在ECMAScript规范中，对JavaScript语法的实现，尤其是语句、表达式，以及基础特性最核心的部分等等，都可以在对这前三名的实现过程和渐次演进关系中展示出来。甚至基本上可以说，你只要理解了最简单榜的前三名，也就理解了设计一门计算机语言的基础模型与逻辑。

throw语句在ECMAScript规范描述中，它的执行实现逻辑只有三行：

> *ThrowStatement* : **throw** *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Let** exprValue be ? GetValue(exprRef).
> 3.**Return** ThrowCompletion(***exprValue***).

这三行代码描述包括两个`Let`语句，以及最后一个`Return`返回值。当然，这里的Let/Return是自然语言的语法描述，是ECMAScript规范中的写法，而不是某种语言的代码。

将这三行代码倒过来看，最后一行的ThrowCompletion()调用其实是一个简写，完整的表示法是一行用于返回完成记录的代码。这里的“记录”，也是一种在ECMAScript规范中的“规范类型”，与之前一直在讲的“引用规范类型”类似，都是ECMAScript特有的。

> **Return** Completion { [Type]: ***exprValue***, [[Target]]: empty }

在ECMAScript规范的书写格式中，一对大括号“{ }”是记录的字面量（Record Literals）表示。也就是说，执行`throw`语句，在引擎层面的效果就是：**返回一个类型为"throw"的一般记录。**

> NOTE：在之前的课程中讲到标签化语句的时候，提及过上述记录中的`[[target]]`字段的作用，也就是仅仅用作“**break** *labelName*”和“**continue** *labelName*”中的标签名。

这行代码也反映了“**JavaScript语句执行是有值（Result）的**”这一事实。也就是说，任何JavaScript语句执行时总是会“返回”一个值，包括空语句。

空语句其实也是上述“最简单榜”的Top 1，因为它在ECMAScript的实现代码有且仅有一行：

> 1.**Return** NormalCompletion(***empty***).

其中的`NormalCompletion()`也是一个简写，完整的表示法与上面的`ThrowCompletion()`也类似，不过其中的传入参数argument在这里是empty。

> **Return** Completion { [Type]: ***argument***, [[Target]]: empty }

而传入参数argument在这里是empty，这是ECMAScript规范类型中的一个特殊值，理解为规范层面可以识别的Null值就可以了（例如它也用来指没有`[[Target]]`）。也就是说，所谓“空语句（*Empty statement*）”，就是返回结果为“空值（*Empty*）”的一般语句。类似于此的，这一讲标题中的语句`throw 1`，就是一个返回"throw"类型结果的语句。

> NOTE：这样的返回结果（Result）在ECMAScript中称为完成记录，这在之前的课程中已经讲述过了。

然而，向谁“返回”呢？以`throw 1`为例，谁才是`throw`语句的执行者呢？

在语句之外看语句

在JavaScript中，除了`eval()`之外，从无“如何执行语句”一说。

这是因为任何情况下，“装载脚本+执行脚本”都是引擎自身的行为，用户代码在引擎内运行时，如“鱼不知水”一般，是难以知道语句本身的执行情况的。并且，即使是`eval()`，由于它的语义是“语句执行并求值”，所以事实上从`eval()`的结果来看是无法了解语句执行的状态的。

因为“求值”就意味着去除了“执行结果（Result）”中的状态信息。

ECMAScript为JavaScript提供语言规范，出于ECMAScript规范书写的特殊性，它也同时是引擎实现的一个规范。在ECMAScript中，所有语句都被解析成待处理的结点，最顶层的位置总是被称为_Script_或_Module_的一个块（块语句），其他的语句将作为它的一级或更深层级的、嵌套的子级结点，这些结点称为“*Parse Node*”，它们构成的整个结构称为“*Parse Tree*”。

无论如何，语句总是一个树或子树，而表达式可以是一个子树或一个叶子结点。

> NOTE：空语句可以是叶子结点，因为没有表达式做它的子结点。

执行语句与执行表达式在这样的结构中是没有明显区别的，而所谓“执行代码”，在实现上就被映射成执行这个树上的子树（或叶子结点）。

所谓“顺序执行的语句”表现在_“Parse Tree_”这个树上，就是同一级的子树。它们之间平行（相同层级），并且相互之间没有“相互依赖的运算”，所以它们的值（也就是尝试执行它们共同的父结点所对应的语句）就将是最后一个语句的结果。所有顺序执行语句的结果向前覆盖，并返回最终语句的结果（Result）。

事实上在表达式中，也存在相同语句的执行过程。也就是如下两段代码在执行效果上其实没有什么差异：

```
// 表达式的顺序执行
1, 2, 3, 4;

// 语句的顺序执行
1; 2; 3; 4;
```

更进一步地说，如下两种语法，其抽象的语义上也是一样的：

```
// 通过分组来组合表达式
(1, 2, 3, 4)

// 通过块语句来组合语句
{1; 2; 3; 4;}
```

所以，从语法树的效果上来看，所谓“语句的执行者”，其实就是它外层的语句；而最外层的语句，总是被称为_Script_或_Module_的一个块，并且它会将结果返回给shell、主进程或`eval()`。

除了`eval()`之外，所有外层语句都并不依赖内层语句的返回值；除了shell程序或主进程程序之外，也没有应用逻辑来读取这些语句缺省状态下的值。

值的覆盖与读取

语句的五种完成状态（normal, break, continue, return, 以及 throw）中，“Nomal（缺省状态）”大多数情况下是不被读取的，break和continue用于循环和标签化语句，而return则是用于函数的返回值。于是，所有的状态中，就只剩下了本讲标题中的`throw 1`所指向的，也就是“异常抛出（throw）”这个状态。

> NOTE: 有且仅有return和throw两个状态是确保返回时携带有效值（包括undefined）的。其他的完成类型则不同，可能在返回时携带“空（empty）”值，从而需要在语句外的代码（shell、主进程或eval）进行特殊的处理。

return语句总是显式地返回值或隐式地置返回值为undefined，也就是说它总是返回值，而break和continue则是不携带返回值的。那么是不是说，当一个“**语句块**”的最终语句是break或continue以及其他一些不携带返回值的语句时，该“**语句块**”总是没有返回值的呢？

答案是否。

ECMAScript语言约定，在块中的多个语句顺序执行时，遵从两条规则：

1. 在向前覆盖既有的语句完成值时，`empty`值不覆盖任何值；
2. 部分语句在没有有效返回值，且既有语句的返回值是`empty`时，默认用`undefined`覆盖之。

规则1比较容易理解，表明一个语句块会尽量将块中最后有效的值返回出来。

例如在之前的课程中提到的空语句、break语句等，都是返回`empty`的，不覆盖既有的值，所以它们也就不影响整个语句块的执行。又例如当你将一个有效返回的语句放到空语句后面的时候，那么这个语句的返回，也就越过空语句，覆盖了其它现有的有效结果值。

```
# Run in NodeJS
> eval(`{
  1;
  2;
  ; // empty
  x:break x; // empty
}`)
2
```

在这个例子中的后面两行语句都返回`empty`，因此不覆盖既有的值，所以整个语句块的执行结果是`2`。又例如：

```
# Run in NodeJS
> eval(`{
  ; // empty
  1;
  ; // empty
}`)
1
```

在这个例子中第1行代码执行结果返回`empty`，于是第2行的结果`1`覆盖了它；而第3行的结果仍然是`empty`所以不导致覆盖，因此整个语句的返回值将是1。

> *NOTE: 参见13.2.13 Block -> RS: Evaluation*, 以及15.2.1.23 Module -> RS: Evaluation中，对**UpdateEmpty**(s, sl)的使用。

而上述的规则2，就比较复杂一些了。这出现在if、do…while、while、for/for…in/for…of、with、switch和try语句块中。在ECMAScript 6之后，这些语句约定不会返回empty，因此它的执行结果“至少会返回一个undefined值”，而在此之前，它们的执行结果是不确定的，既可能返回undefined值，也可能返回empty，并导致上一行语句值不覆盖。举例来说：

```
# Run in NodeJS 5.10+ (or NodeJS 4)
> eval(`{
  2;
  if (true);
}`)
undefined
```

由于ES6约定`if`语句不返回`empty`，所以第1行返回的值2将被覆盖，最终显示为`undefined`。而在此之前（例如NodeJS 4），它将返回值2；

> NOTE: 参考阅读《前端要给力之：语句在JavaScript中的值》。

由此一来，ECMAScript规范约定了JavaScript中所有语句的执行结果的可能范围：`empty`，或一个既有的执行结果（包括undefined）。

引用的值

现在还存在最后一个问题：所谓“引用”，算是什么值？

回顾第一讲的内容：表达式的本质是求值运算，而引用是不能直接作为最终求值的操作数的。因此引用实际上不能作为语句结果来返回，并且它在表达式计算中也仅是作为中间操作数（而非表达最终值的操作数）。所以在语句返回值的处理中，总是存在一个“执行表达式并‘取值’”的操作，以便确保不会有“引用”类型的数据作为语句的最终结果。而这，也就是在ECMAScript规中的`throw 1`语句的第二行代码的由来：

> 2.**Let** exprValue be ? GetValue(exprRef).

事实上在这里的符号“? *opName*()”语法也是一个简写，在ECMAScript中它表示一个ReturnIfAbrupt(x)的语义：如果设一个“处理（*opName()*）”的结果是x，那么，

> 如果x是特殊的（非normal类型的）完成记录，则返回x；否则返回一个以x.[[value]]为值的、normal类型的完成记录。

简而言之，就是在GetValue()这个操作外面再封装一次异常处理。这往往是很有效的，例如一个`throw`语句，它自己的throw语义还没有执行到呢，结果在处理它的操作数时就遇到一个异常，这该怎么办呢？

```
throw 1/0;
```

那么exprRef作为表达式的计算结果，其本身就将是一个异常，于是`? GetValue(exprRef)`就可以返回这个异常对象（而不是异常的值）本身了。类似地，所谓“表达式语句”（这是排在“最简单语句榜”的第二名的语句）就直接返回这个值：

> *ExpressionStatement*: *Expression*;
> 1.**Let** exprRef be the result of evaluating Expression.
> 2.**Return** ? GetValue(exprRef).

还有一行代码

现在还有一行代码，也就是第一行的“***let** ... result of evaluating ...*”。其中的“result of evaluating…”基本上算是ECMAScript中一个约定俗成的写法。不管是执行语句还是表达式，都是如此。这意味着引擎需要按之前我讲述过的那些执行逻辑来处理对应的代码块、表达式或值（操作数），然后将结果作为Result返回。

ECMAScript所描述的引擎，能够理解“执行一行语句”与“执行一个表达式”的不同，并且由此决定它们返回的是一个“引用记录”还是“完成记录”（规范类型）。当外层的处理逻辑发现是一个引用时，会再根据当前逻辑的需要将“引用”理解为左操作数（取引用）或右操作数（取值）；否则当它是一个完成记录时，就尝试检测它的类型，也就是语句的完成状态（throw、return、normal或其他）。

所以，throw语句也好，return语句也罢，所有的语句与它“外部的代码块（或`Parse Tree`中的父级结点）”间其实都是通过这个**完成状态**来通讯的。而外部代码块是否处理这个状态，则是由外部代码自己来决定的。

而几乎所有的外部代码块在执行一个语句（或表达式）时，都会采用上述的ReturnIfAbrupt(x)逻辑来封装，也就是说，如果是normal，则继续处理；否则将该完成状态原样返回，交由外部的、其他的代码来处理。所以，就有了下面这样一些语法设计：

1. 循环语句用于处理非标签化的continue与break，并处理为normal；否则，
2. 标签语句用于拦截那些“向外层返回”的continue和break；且，如果能处理（例如是目标标签），则替换成normal。
3. 函数的内部过程`[[Call]]`，将检查“函数体执行”（将作为一个块语句执行）所返回状态是否是return类型，如果是，则替换成normal。

显而易见，所有语句行执行结果状态要么是normal，要么就是还未被拦截的throw类型的语句完成状态。

try语句用于处理那些漏网之鱼（throw状态）：在catch块中替换成normal，以表示try语句正常完成；或在finally中不做任何处理，以继续维持既有的完成状态，也就是throw。

值1

最后，本小节标题中的代码中只剩下了一个值`1`，在实际使用中，它既可以是一个其他表达式的执行结果，也可以是一个用户定义或创建的对象。无论如何，只要它是一个JavaScript可以处理的结果Result（引用或值），那么它就可以通过内部运算`GetValue()`来得到一个真实数据，并放在一个throw类型的完成记录中，通过一层一层的`Parse Tree/Nodes`中

的ReturnIfAbrupt(x)向上传递，直到有一个try块捕获它。例如：

```
try {
  throw 1;
catch(e) {
  console.log(e);  // 1
}
```

或者，它也可能溢出到代码的最顶层，成为根级Parse Node，也就是Script或Module类型的全局块的返回值。

这时，引擎或Shell程序就会得到它，于是……你的程序挂了。

知识回顾

在最近几讲，我讲的内容从语句执行到函数执行，从引用类型到完成类型，从循环到迭代，基本上已经完成了关于JavaScript执行过程的全部介绍。

当然，这些都是在串行环境中发生的事情，至于并行环境下的执行过程，在专栏的后续文章中我还会再讲给你。

作为一个概述，建议你回顾一下本专栏之前所讲的内容。包括（但不限于）：

1. 引用类型与值类型在ECMAScript和JavaScript中的不同含义；
2. 基本逻辑（顺序、分支与循环）在语句执行和函数执行中的不同实现；
3. 流程控制逻辑（中断、跳转和异步等）的实现方法，以及它们的要素，例如循环控制变量；
4. JavaScript执行语句和函数的过程，引擎层面从装载到执行完整流程；
5. 理解语法解析让物理代码到标记（Token）、标识符、语句、表达式等抽象元素的过程；
6. 明确上述抽象元素的静态含义与动态含义之间的不同，明确语法元素与语义组件的实例化。

综合来看，JavaScript语言是面向程序员开发来使用的，是面子上的活儿，而ECMAScript既是规范也是实现，是藏在引擎底下的事情。ECMAScript约定了一整套的框架、类型与体系化的术语，根本上就是为了严谨地叙述JavaScript的实现。并且，它提供了大量的语法或语义组件，用以规范和实现将来的JavaScript。

直到现在，我向你的讲述的内容，在ECMAScript中大概也是十不过一。这些内容主要还是在刻画ECMAScript规范的梗概，以及它的核心逻辑。

从下一讲开始，我将向你正式地介绍JavaScript最重要的语言特性，也就是它的面向对象系统。

当然，一如本专栏之前的风格，我不会向你介绍类型x.toString()这样的、可以在手册上查阅的内容，我的本意，在于与你一起学习和分析：

JavaScript是怎样的一门语言，以及它为什么是这样的一种语言。