你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}
```

```
for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

## 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大

多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
```

```
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所

以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是一**种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

## 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
```

```
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了

标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、

可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

## 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于apply()的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是一**种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家

一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是一**种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的

是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

## 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

## 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
```

```
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放

起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就

像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里`argArray`表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析

阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

## 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
```

```
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n；
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了

标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、

可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

## 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

## 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是**一种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

## 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于apply()的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家一起分享，让我也有机会听听你的收获。

你好，我是周爱民。

今天这一讲的标题是一个**模板**。模板这个语法元素在JavaScript中出现得很晚，以至于总是有人感到奇怪：为什么JavaScript这么晚才弄出个模板这样的东西？

模板看起来很简单，就是把一个字符串里的东西替换一下就行了，C语言里的printf()就有类似的功能，Bash脚本里也可以直接在字符串里替换变量。这个功能非常好用，但在实现上其实很简单，无非就是字符串替换而已。

# 模板是什么？

但是，模板就是一个字符串吗？或者我们需要更准确地问一个概念上的问题：

模板是什么？

回顾之前的内容，我们说JavaScript中，有**语句**和**表达式**两种基本的可执行元素。这在语言设计的层面来讲，是很普通的，大多数语言都这么设计。少数的语言会省略掉**语句**这个语法元素，或者添加其它一些奇怪的东西，不过通常情况下它的结果就是让语言变得不那么人性。那么，是不是说，JavaScript中只有语句和表达式是可以执行的呢？

答案是“No”，譬如这里讲到的模板，其实就是一**种特殊的可执行结构**。

所有特殊可执行结构其实都是来自于某种固定的、确定的逻辑。这些逻辑语义是非常明确的，输入输出都很确定，这样才能被设计成一个标准的、易于理解的可执行结构。并且，如果在一门语言中添加太多的、有特殊含义的执行结构，那么这门语言就像上面说的，会显得“渐渐地有些奇怪了”。

语言的坏味道就是这样产生的。越来越多的抽象概念放进来，固化成一种特殊的逻辑或结构，试图通过非正常的逻辑来影响程序员的思维过程，于是就会渐渐地变得令人不愉快了。

如果我们抛开JavaScript核心库或者标准语言运行时里面的那些东西，例如Map、Set等等，专门考察一下在语言及语法层面定义的特殊可执行结构的话，都会有哪些可执行结构浮出水面呢？

# 参数表

第一个不太容易注意到的东西就是参数表。

在JavaScript语言的内核中，参数表其实是一个独立的语法组件：

- 对于函数来说，参数表就是在函数调用时传入的参数0到n;
- 对于构造器以及构造器的new运算来说，参数表就是new运算的一个运算数。

这二者略微有一点区别，在远古时期的JavaScript中，它们是很难区分的。然而在ECMAScript的规范中，这个参数表被统一成了标准的List。这个List也是一种ECMAScript中的规范类型，与引用、属性描述符等等规范类型类似，它在相关的操作中是作为一个独立的部分参与运算的。

要证实这一点是很容易的。例如在JavaScript的反射机制中，使用代理对象就能拿到一个函数调用的入参，或者new运算过程中传入的参数，它们都表示成一个标准的数组：

```
handler.apply = function(target, thisArgument, argArray) {
  ...
}
```

这里argArray表示为一个数组，但这只是参数表在传入后通过“特殊可执行结构”执行的结果。如果追究这个行为背后的逻辑，那么这个列表实际上是根据形式参数的样式（Formal of Parameters），按照传入参数逐一匹配出来的。这个所谓“**逐一匹配**”，就是我们说的“**特殊的可执行的逻辑**”。

任何实际参数在传入一个函数的形式参数时，都会经历这样的一个执行过程，它是“将函数实例化”这个内部行为的一个处理阶段。

我们之前也说过了，所谓“**将函数实例化**”就是将函数从源代码文本变成一个可以执行的、运行期的闭包的过程。

在这个过程中，参数表作为可执行结构，它的执行结果就是将传入的参数值变成与形式参数规格一致的实际参数，最终将这些参数中所有的值与它们“在形式参数表中的名字”绑定起来，作为函数闭包中可以访问的名字。

说完这段，我估计你听得都累了。听起来很啰嗦很复杂，但是简单化地讲呢，就是把参数放在arguments列表中，然后让arguments中的值与参数表中的名字对应起来。而这就是对“参数表（argArray）”这个可执行结构的全部操作。

了解这个有什么用呢？很有用。

其一，我们要记得，JavaScript中有个东西没有参数表，那就是箭头函数，那么上面的逻辑是如何实现的呢？

其二，我们还要知道JavaScript中有种形式参数的风格，称为“简单参数（*Simple Parameter List*）”，这与argArray的使用存在莫大的关系。

关于这两点，我们往简化里说，就是箭头函数也是采用与上述过程完全一致的处理逻辑，只是在最后没有向闭包绑定arguments这个名字而已。而所谓简单参数，就是可以在形式参数表中可以明确数出参数个数的、没有使用扩展风格声明参数的参数表。

# 扩展风格的参数表

什么是扩展风格的参数表呢？它也称为“非简单的参数列表（*Non-Simple Parameter List*）”，这就与其它几种可执行结构有关了，例如说**缺省参数**。

事实上，缺省参数是非常有意思的可执行结构，它长得就是下面这个样子：

```
function foo(x = 100) {
  ...
}
```

这意味着在语法分析期，JavaScript就得帮助该参数登记下“100”这个值。然后在实际处理这个参数时，至少需要一个赋值表达式的操作，用来将这个值与它的名字绑定起来。所以，foo()函数调用时，总有一段执行逻辑来访问形式参数表以及执行这个赋值表达式。

让问题变得更复杂的地方在于：这个值“100”可以是一个表达式的运算结果，由于表达式可以引用上下文中的其它变量，因此上面的所谓“登记”，就不能只是记下一个字面量值那么简单，必须登记一个表达式，并且在运行期执行它。例如：

```
var x = 0;
function foo(i = x++) {
  console.log(i);
}
foo(); // 1st
foo(); // 2nd
```

这样每次调用foo()的时候，“x++”就都会得到执行了。所以，缺省参数就是一种可执行结构，是参数表作为可执行结构的逻辑中的一部分。同样的，**剩余参数**和**参数展开**都具有类似的性质，也都是参数表作为可执行结构的逻辑中的一部分。

既然提到参数展开，这里是可以略微多讨论一下的，因为它与后面还要讲到的另外一种可执行结构有关。参数展开是唯一一个可以影响“传入参数个数”的语法。例如：

```
foo(...args)
```

这个语法的关键处不在于形式参数的声明，而在于实际参数的传入。

这里传入时实际只用到了一个参数，即“args”，但是“…”语法对这个数组进行了展开，并且根据args.length来扩展了参数表的长度/大小。由于其它参数都是按实际个数计数的，所以这里的参数展开就成了唯一能动态创建和指定参数个数的语法。

这里之所以强调这一语法，是因为在传统的JavaScript中，这一语法是使用foo.apply()来替代的。历史中，“new Function()”这个语法没有类似于`apply()`的创建和处理参数表的方式，所以早期的JavaScript需要较复杂的逻辑，或者是调用eval()来处理动态的new运算。

这个过程相当麻烦，真的是“谁用谁知道”。而如今，它可以只使用一行代码替代：

```
new Func(...args)
```

这正是我们之前说“函数和（使用new运算的）构造器的参数表不一样”所带来的差异。那么这个参数展开是怎么实现的呢？答案是**迭代器**。

参数展开其实是数组展开的一种应用，而数组展开在本质上就是依赖迭代器的。

你可以在任何内置迭代器的对象（亦即是Symbol.iterator这个符号属性有值的对象）上使用展开语法，使它们按迭代顺序生成相应多个“元素（elements）”，并将这些元素用在需要的地方，而不仅仅是将它展开。例如`yield*`，又例如**模板赋值**。我们知道迭代器是有一组界面约定的，那么这个迭代器界面本质上也是一种可执行结构。

# 赋值模板

赋值模板是我们今天要讲到的第三种可执行结构。

模板赋值是ECMAScript 6之后提供一种声明标识符的语法，该语法依赖一个简单的赋值过程，可以抽象地理解为下面这样：

```
a = b
```

等号的左侧称为赋值模板（AssignmentPattern），而右侧称为值(Value）。

在JavaScript中，任何出现类似语法或语义过程的位置，本质上都可以使用模板赋值的。也就是说，即使没有这个“赋值符号（等号）”，只要语义是“向**左操作数**（lhs）上的标识符，赋以**右操作数**（rhs）的值”，那么它就适用于模板赋值。

很显然，我们前面说的“向参数表中的形式参数（的名字），赋以实际参数的值”，也是这样的一个过程。所以，JavaScript在语法上很自然地就支持了在参数表中使用模板赋值，以及在任何能够声明一个变量或标识符的地方，来使用模板赋值。例如：

```
function foo({x, y}) {
  ...
}

for (var {x, y} in obj) {
  ...
}
```

而所有这些地方的赋值模板，都是在语法解析期就被分析出来，并在JavaScript内部作为一个可执行结构存放着。然后在运行期，会用它们来完成一个“从右操作数按模板取值，并赋值给左操作数”的过程。这与将函数的参数表作为**样式**（Formal）存放起来，然后在运行期逐一匹配传入值是异曲同工的。

所有上述的执行结构，我们都可以归为一个大类，称为“**名字和值的绑定**”。

也就是说，所有这些执行的结果都是一个名字，执行的语义就是给这个名字一个值。显然这是不够的，因为除了给这个名字一个值之外，最终还得使用这个名字以便进行更多的运算。那么，这个“找到名字并使用名字”的过程，就称为“**发现**（Resolve binding）”，而其结果，就称为“**引用**（reference）”。

任何的名字，以及任何的字面量的值，本质上都可以作为一个被发现的对象，并且在实际应用中也是如此。在代码的语法分析阶段，发现一个名字与发现一个值本质上没有什么不同，所以如下的两行代码：

```
a = 1
1 = 1
```

其实在JavaScript中都可以通过语法解析，并且进入实际的代码执行阶段。所以“1=1”是一个运行期错误（ReferenceError），而不是语法错误（SyntaxError）。那么所谓的“发现的结果——引用（Reference）”，也就不是简单的一个语法标识符，而是一个可执行结构了。更进一步地说，如下面这些代码，每一个都会导致一个引用（的可执行结构）：

```
a
1
"use strict"
obj.foo
```

正是因此，所以上面的第三行代码才会成为一个“可以导致当前作用域切换为严格模式”的**指令**。因为它是引用，也是可执行结构。对待它，JavaScript只需要像调用函数一样，将它处理成一段确定逻辑就可以了。

这几个引用中有一个非常特殊的引用，就是obj.foo，它被称为属性引用（Property Reference）。属性引用不是简单的标识符引用，而是一个属性存取运算的结果。所以，表达式运算的结果可以是一个引用。那么它的特殊性在哪里呢？它是为数不多的、可以存储原表达式信息，并将该信息“传递”到后续表达式的特殊结构。严格地说，所有的引用都可以设计成这个样子，只不过属性引用是我们最常见到的罢了。

然而，为什么要用“引用（Reference）”这种结构来承担这一责任呢？

这与JavaScript中的“方法调用”这一语义的特殊实现有关。JavaScript并不是静态分析的，因此它无法在语法阶段确定“obj.foo”是不是一个函数，也不知道用户代码在得到“obj.foo”这个属性之后要拿来做什么用。

```
obj.foo()
```

直到运行期处理到下一个运算（例如上面这样的运算时），JavaScript引擎才会意识到：哦，这里要调用一个方法。

然而，方法调用的时候是需要将obj作为foo()函数的this值传入，这个信息只能在上一步的属性存取“obj.foo”中才能得到。所以obj.foo作为一个属性引用，就有责任将这个信息保留下来，传递给它的下一个运算。只有这样，才能完成一次“将函数作为对象方法调用”的过程。

引用作为函数调用（以及其它某些运算）的“左操作数（lhs）”时，是需要传递上述信息的。这也就是“引用”这种可执行结构的确定逻辑。

本质上来说，它就是要帮助JavaScript的执行系统来完成“发现/解析（Resolve）”过程，并在必要时保留这个过程中的信息。在引擎层面，如果一个过程只是将“查找的结果展示出来”，那么它最终就表现为值；如果包括这个过程信息，通常它就表现为引用。

那么作为一个执行系统来讲，JavaScript执行的最终的结果到底表达为一个引用呢，还是一个值呢？答案是“值”。

因为你没有办法将一个引用（包括它的过程信息）在屏幕上打印出来，而且即便打印出来，用户也没有兴趣。用户真正关心的是打印出来的那个结果，例如在屏幕上显示“Hello world”。所以无论如何，JavaScript创建引用也好，处理这些引用或特殊结构的执行过程也好，最终目的，还是计算求值。

# 模板字面量

回到我们今天的话题上来。我们为什么要讲这些可执行结构呢？事实上，我们在标题中的列出的这行代码是一个**模板字面量**（TemplateLiteral）：

```
`${1}`
```

而模板字面量是上述所有这些可执行结构的集大成者。它本身是一个特殊的可执行结构，但是它调动了包括引用、求值、标识符绑定、内部可执行结构存储，以及执行函数调用在内的全部能力。这是JavaScript厘清了所有基础的可执行结构之后，才在语法层面将它们融会如一的结果。

# 知识回顾

接下来我们对今天的这一行代码做个总结，并对相关的内容再做些补充。

标题中的代码称为**模板字面量**，是一种可执行结构。JavaScript中有许多类似的可执行结构，它们通常要用固定的逻辑，并在确定的场景下，交付JavaScript的一些核心语法的能力。

与参数表和赋值模板有相似的地方，模板字面量也是将它的形式规格（Formal）作为可执行结构来保存的。

只是参数表与赋值模板关注的是名字，因此存储的是“名字（lhs）”与“名字的值（rhs）的取值方法”之间的关系，执行的结果是argArray或在当前作用域中绑定的名字等。

而模板字面量关注的是值，它存储的是“结果”与“结果的计算过程”之间的关系。由于模板字面量的执行结果是一个字符串，所以当它作为值来读取时，就会激活它的运算求值过程，并返回一个字符串值。

模板字面量与所有其它字面量（能作为引用）相似，它也可以作为引用。

```
1=1
```

“1=1”包括了“1”作为引用和值（lhs和rhs）的两种形式，在语法上是成立的。

```
foo`${1}`
```

所以上面这行代码在语法上也是成立的。因为在这个表达式中，`${1}`使用的不是模板字面量的值，而是它的一个“（类似于引用的）结构”。

“模板字面量调用（TemplateLiteral Call）”是唯一一个会使用模板字面量的引用形态（并且也没有直接引用它的内部结构）的操作。这种引用形态的模板字面量也被称为“标签模板（Tagged Templates）”，主要包括模板的位置和那些可计算的标签的信息。例如：

```
> var x = 1;
> foo = (...args) => console.log(...args);
> foo`${x}`
[ '', '' ] 1
```

模板字面量的内部结构中，主要包括将模板多段截开的一个数组，原始的模板文本（raw）等等。在引擎处理模板时，只会将该模板解析一次，并将这些信息作为一个可执行结构缓存起来（以避免多次解析降低性能），此后将只使用该缓存的一个引用。当它作为字面量被取值时，JavaScript会在当前上下文中计算各个分段中的表达式，并将表达式的结果填回到模板从而拼接成一个结果，最后返回给用户。

# 思考题

关于模板的话题其实还有很多可探索的空间，所以建议你仔细阅读一下ECMAScript规范，以便对今天的内容有更深入的理解，例如ECMAScript中如何利用模板的缓存。今天的思考题是：

- 为什么ECMAScript要创建一个“模板调用”这样古怪的语法呢？

当然，JavaScript内部其实还有很多其它的可执行结构，我今后还会讲到一些。或者你现在就可以开始去发掘，希望你能与大家

一起分享，让我也有机会听听你的收获。