你好，我是winter。

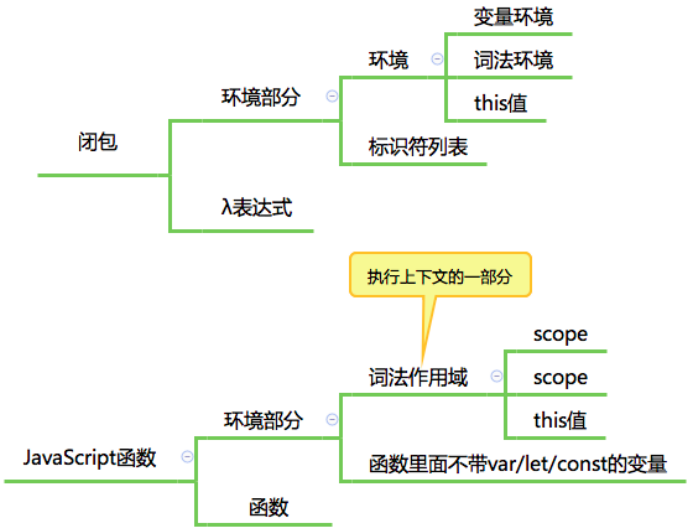
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。

More generally, this derived environment consists of E, modified by pairing the identifier(s) in $bv X$ with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a “closure,” comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE : S$$

and C is replaced by tC. We describe this step as follows: “Scanning X causes *locationEXE* to be loaded.”

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是“带有一系列信息的λ表达式”。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分
  - 环境：函数的词法环境（执行上下文的一部分）
  - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是“函数”，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的“作用域”吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的“环境部分”。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包“环境部分”相对应的术语是“词法环境”，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为：“执行上下文”。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。
- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
```

```
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从`var`、`let`、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

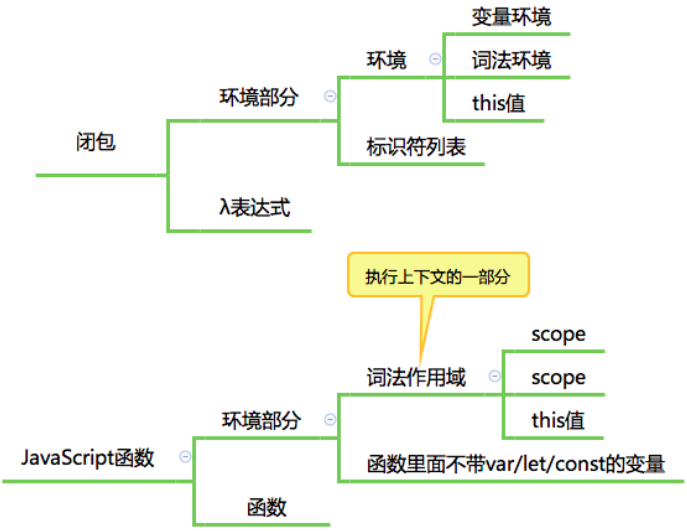
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



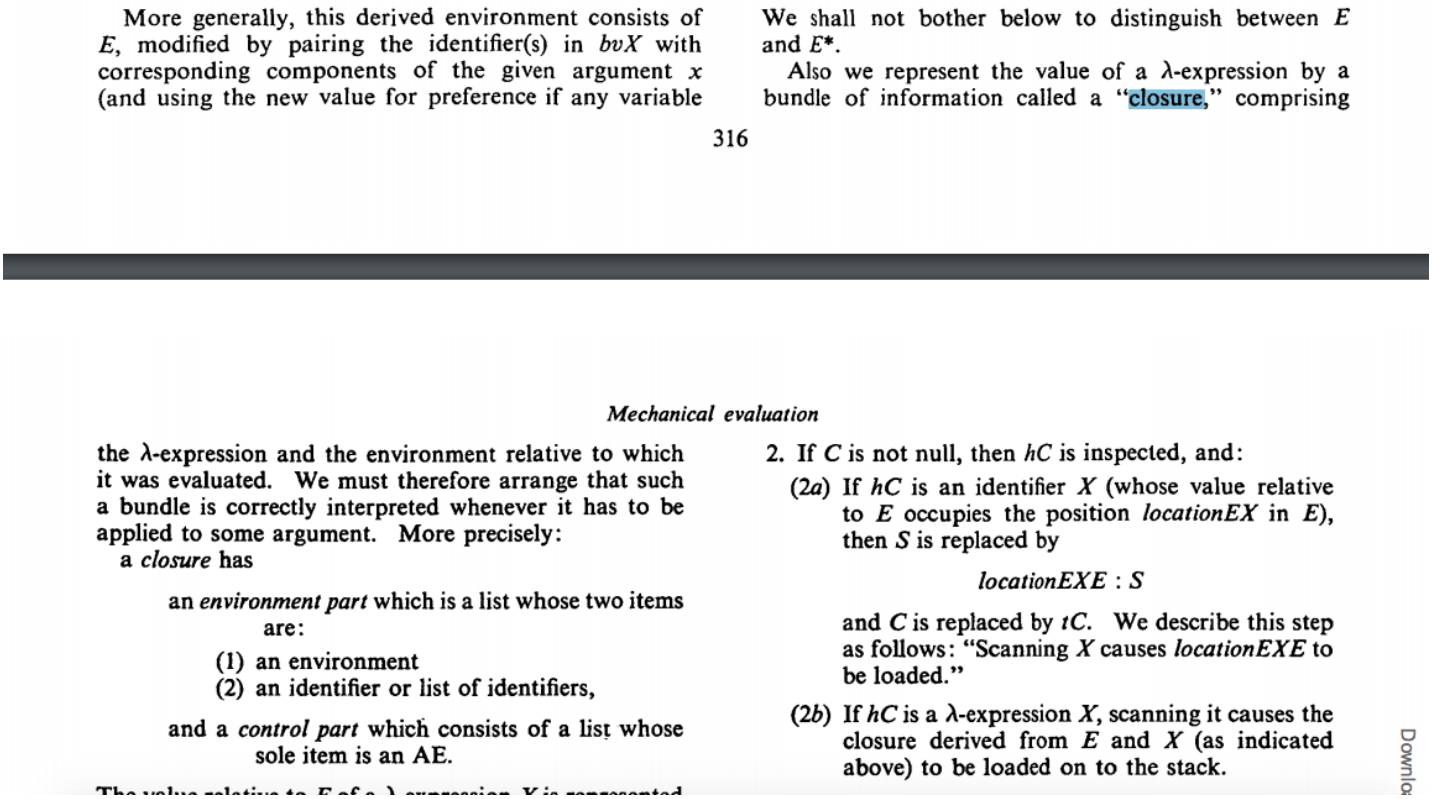
看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。



More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE : S$$

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分
  - 环境：函数的词法环境（执行上下文的一部分）
  - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是"函数"，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的"作用域"吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的"环境部分"。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包"环境部分"相对应的术语是"词法环境"，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为："执行上下文"。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。

- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

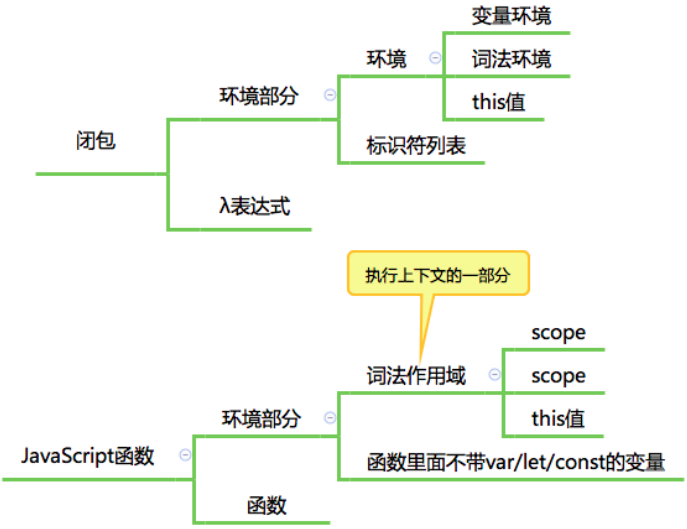
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。

More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

locationEXE : S

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分

    - 环境：函数的词法环境（执行上下文的一部分）
    - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是“函数”，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的“作用域”吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的“环境部分”。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包“环境部分”相对应的术语是“词法环境”，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为：“执行上下文”。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。
- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
```

```
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

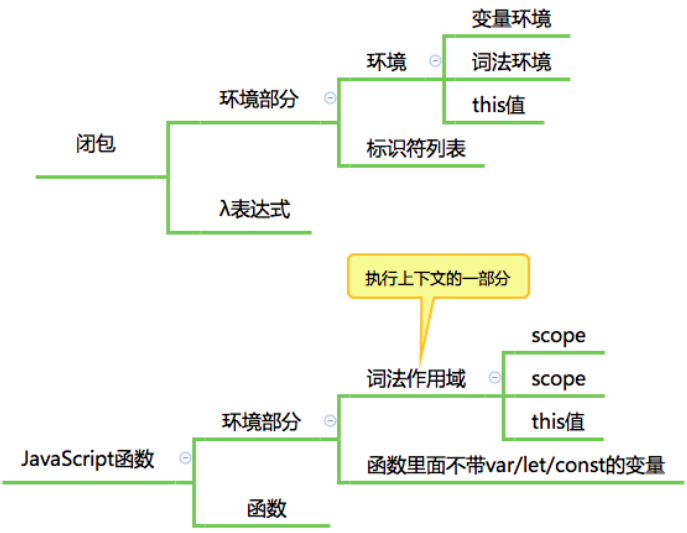
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



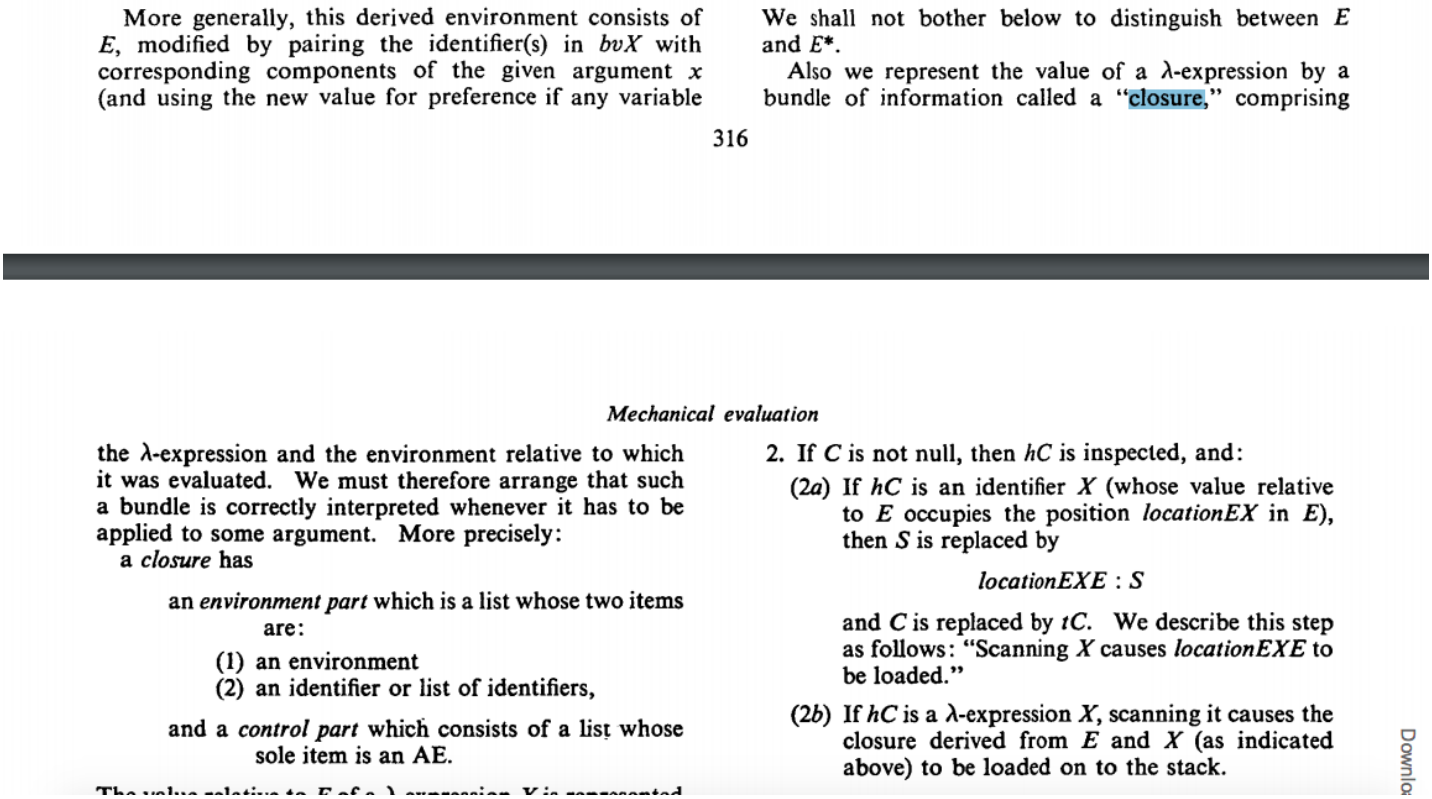
看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。



More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to E of a λ-expression X is represented

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE : S$$

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分
  - 环境：函数的词法环境（执行上下文的一部分）
  - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是"函数"，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的"作用域"吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的"环境部分"。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包"环境部分"相对应的术语是"词法环境"，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为："执行上下文"。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。

- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从`var`、`let`、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

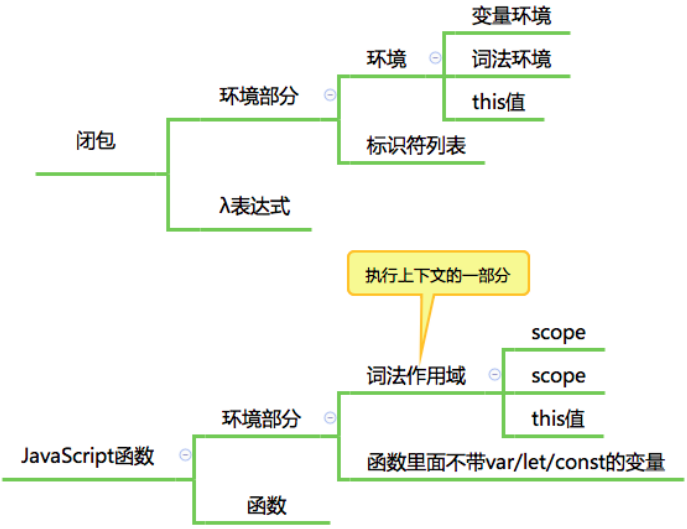
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。

More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

locationEXE : S

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分

    - 环境：函数的词法环境（执行上下文的一部分）
    - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是“函数”，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的“作用域”吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的“环境部分”。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包“环境部分”相对应的术语是“词法环境”，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为：“执行上下文”。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。
- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
```

```
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

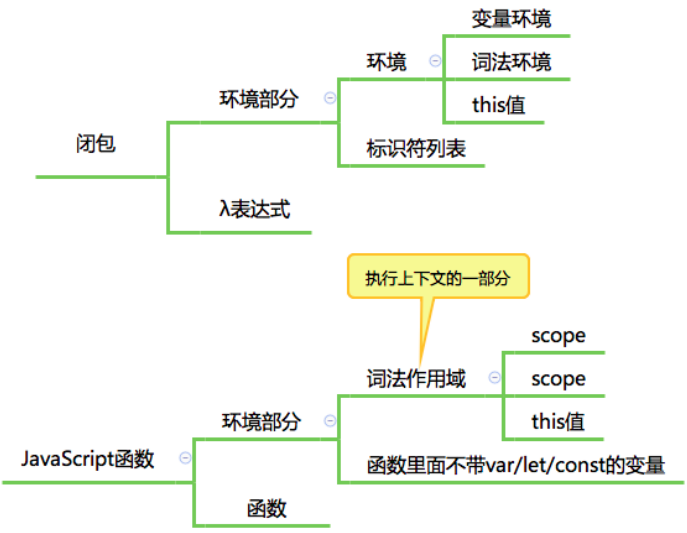
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



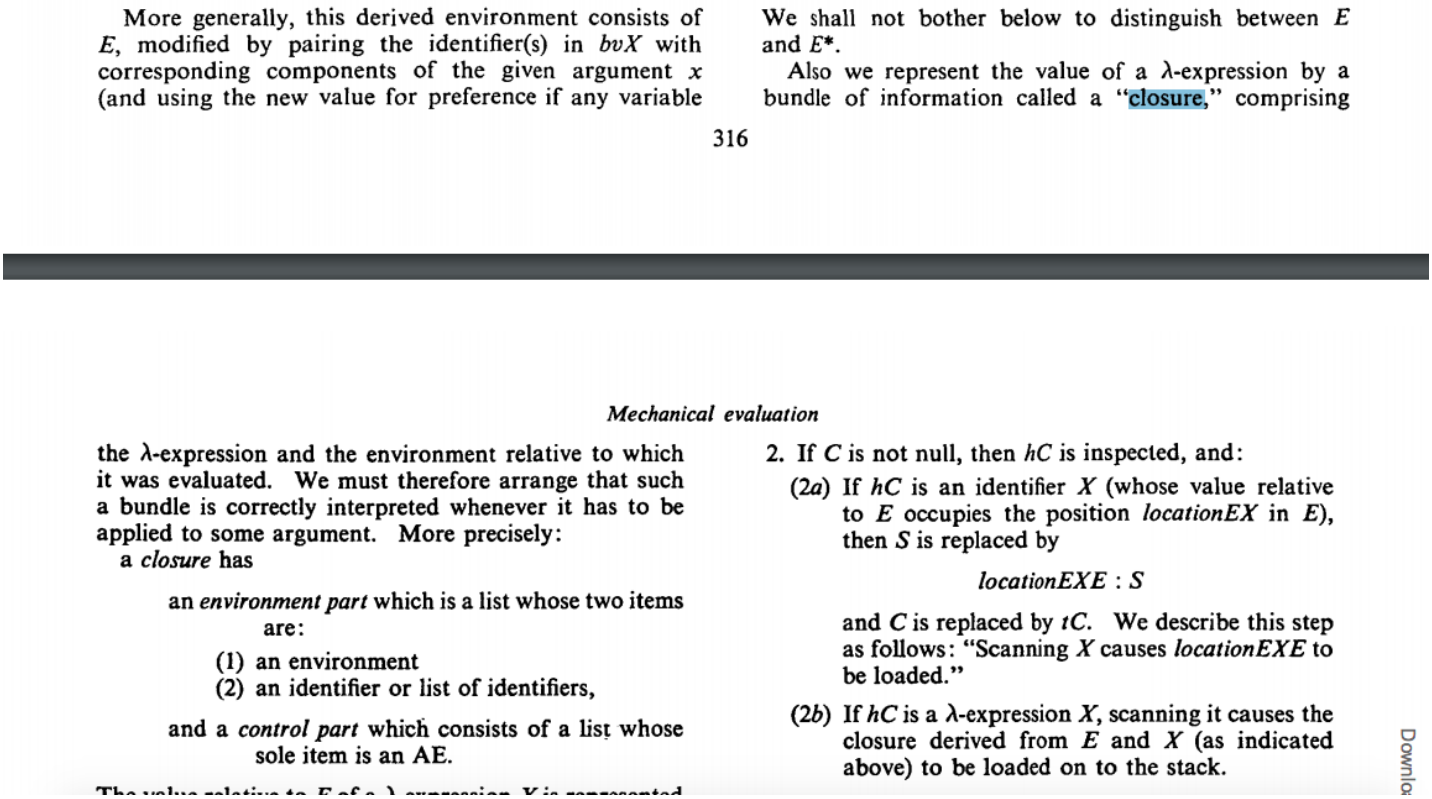
看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。



More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

The value relative to E of a λ-expression X is represented

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE : S$$

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

Downlo

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分
  - 环境：函数的词法环境（执行上下文的一部分）
  - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是"函数"，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的"作用域"吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的"环境部分"。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包"环境部分"相对应的术语是"词法环境"，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为："执行上下文"。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。

- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

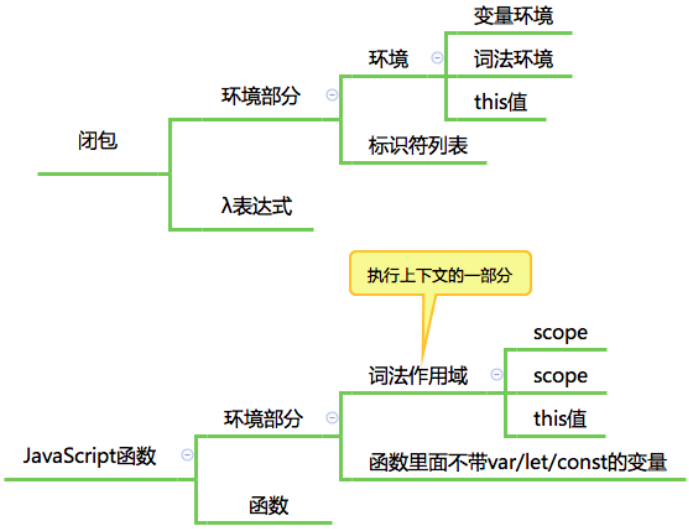
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。

More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

locationEXE : S

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分

    - 环境：函数的词法环境（执行上下文的一部分）
    - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是“函数”，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的“作用域”吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的“环境部分”。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包“环境部分”相对应的术语是“词法环境”，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为：“执行上下文”。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。
- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
```

```
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？

你好，我是winter。

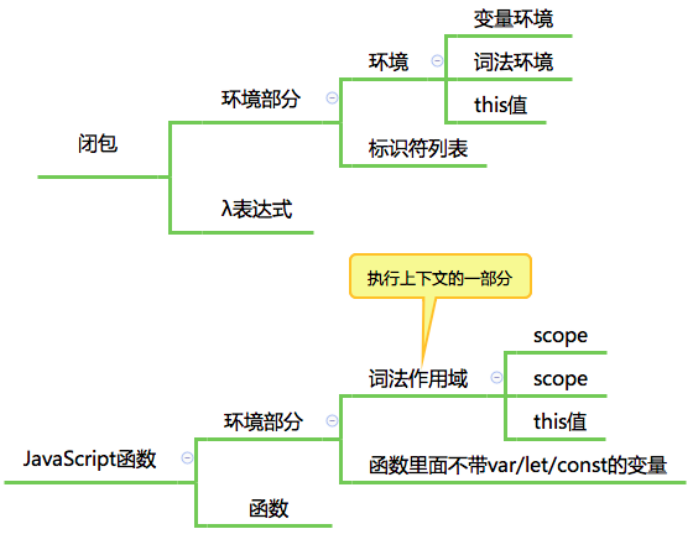
在上一课，我们了解了JavaScript执行中最粗粒度的任务：传给引擎执行的代码段。并且，我们还根据“由JavaScript引擎发起”还是“由宿主发起”，分成了宏观任务和微观任务，接下来我们继续去看一看更细的执行粒度。

一段JavaScript代码可能会包含函数调用的相关内容，从今天开始，我们就用两节课的时间来了解一下函数的执行。

我们今天要讲的知识在网上有不同的名字，比较常见的可能有：

- 闭包；
- 作用域链；
- 执行上下文；
- this值。

实际上，尽管它们是表示不同的意思的术语，所指向的几乎是同一部分知识，那就是函数执行过程相关的知识。我们可以简单看一下图。



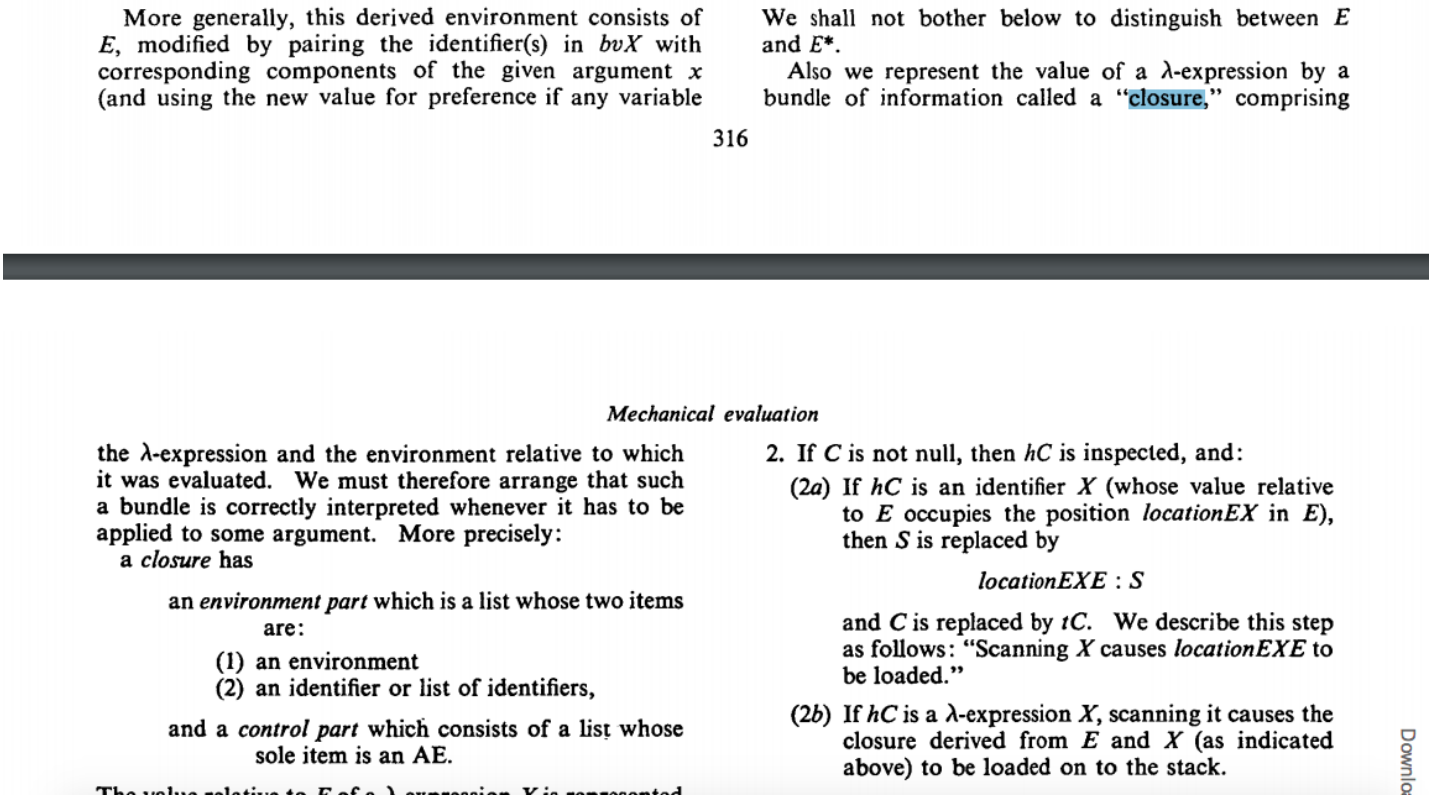
看着也许会有点晕，别着急，我会和你共同理一下它们之间的关系。

当然，除了让你理解函数执行过程的知识，理清这些概念也非常重要。所以我们先来讲讲这个有点复杂的概念：闭包。

闭包

闭包翻译自英文单词closure，这是个不太好翻译的词，在计算机领域，它就有三个完全不相同的意义：编译原理中，它是处理语法产生式的一个步骤；计算几何中，它表示包裹平面点集的凸多边形（翻译作凸包）；而在编程语言领域，它表示一种函数。

闭包这个概念第一次出现在1964年的《The Computer Journal》上，由P. J. Landin在《The mechanical evaluation of expressions》一文中提出了applicative expression和closure的概念。



More generally, this derived environment consists of E, modified by pairing the identifier(s) in bvX with corresponding components of the given argument x (and using the new value for preference if any variable

We shall not bother below to distinguish between E and E^*.

Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising

316

Mechanical evaluation

the λ-expression and the environment relative to which it was evaluated. We must therefore arrange that such a bundle is correctly interpreted whenever it has to be applied to some argument. More precisely:

a *closure* has

an *environment part* which is a list whose two items are:

(1) an environment
(2) an identifier or list of identifiers,

and a *control part* which consists of a list whose sole item is an AE.

2. If C is not null, then hC is inspected, and:

(2*a*) If hC is an identifier X (whose value relative to E occupies the position *locationEX* in E), then S is replaced by

$$locationEXE : S$$

and C is replaced by tC. We describe this step as follows: "Scanning X causes *locationEXE* to be loaded."

(2*b*) If hC is a λ-expression X, scanning it causes the closure derived from E and X (as indicated above) to be loaded on to the stack.

在上世纪60年代，主流的编程语言是基于lambda演算的函数式编程语言，所以这个最初的闭包定义，使用了大量的函数式术语。一个不太精确的描述是"带有一系列信息的λ表达式"。对函数式语言而言，λ表达式其实就是函数。

我们可以这样简单理解一下，闭包其实只是一个绑定了执行环境的函数，这个函数并不是印在书本里的一条简单的表达式，闭包与普通函数的区别是，它携带了执行的环境，就像人在外星中需要自带吸氧的装备一样，这个函数也带有在程序中生存的环境。

这个古典的闭包定义中，闭包包含两个部分。

- 环境部分
  - 环境
  - 标识符列表
- 表达式部分

当我们把视角放在JavaScript的标准中，我们发现，标准中并没有出现过closure这个术语，但是，我们却不难根据古典定义，在JavaScript中找到对应的闭包组成部分。

- 环境部分
  - 环境：函数的词法环境（执行上下文的一部分）
  - 标识符列表：函数中用到的未声明的变量
- 表达式部分：函数体

至此，我们可以认为，JavaScript中的函数完全符合闭包的定义。它的环境部分是函数词法环境部分组成，它的标识符列表是函数中用到的未声明变量，它的表达式部分就是函数体。

这里我们容易产生一个常见的概念误区，有些人会把JavaScript执行上下文，或者作用域（Scope，ES3中规定的执行上下文的一部分）这个概念当作闭包。

实际上JavaScript中跟闭包对应的概念就是"函数"，可能是这个概念太过于普通，跟闭包看起来又没什么联系，所以大家才不自觉地把这个概念对应到了看起来更特别的"作用域"吧（其实我早年也是这么理解闭包，直到后来被朋友纠正，查了资料才改正过来）。

执行上下文：执行的基础设施

相比普通函数，JavaScript函数的主要复杂性来自于它携带的"环境部分"。当然，发展到今天的JavaScript，它所定义的环境部分，已经比当初经典的定义复杂了很多。

JavaScript中与闭包"环境部分"相对应的术语是"词法环境"，但是JavaScript函数比λ函数要复杂得多，我们还要处理this、变量声明、with等等一系列的复杂语法，λ函数中可没有这些东西，所以，在JavaScript的设计中，词法环境只是JavaScript执行上下文的一部分。

JavaScript标准把一段代码（包括函数），执行所需的所有信息定义为："执行上下文"。

因为这部分术语经历了比较多的版本和社区的演绎，所以定义比较混乱，这里我们先来理一下JavaScript中的概念。

执行上下文在ES3中，包含三个部分。

- scope：作用域，也常常被叫做作用域链。
- variable object：变量对象，用于存储变量的对象。
- this value：this值。

在ES5中，我们改进了命名方式，把执行上下文最初的三个部分改为下面这个样子。

- lexical environment：词法环境，当获取变量时使用。
- variable environment：变量环境，当声明变量时使用。
- this value：this值。

在ES2018中，执行上下文又变成了这个样子，this值被归入lexical environment，但是增加了不少内容。

- lexical environment：词法环境，当获取变量或者this值时使用。

- variable environment：变量环境，当声明变量时使用。
- code evaluation state：用于恢复代码执行位置。
- Function：执行的任务是函数时使用，表示正在被执行的函数。
- ScriptOrModule：执行的任务是脚本或者模块时使用，表示正在被执行的代码。
- Realm：使用的基础库和内置对象实例。
- Generator：仅生成器上下文有这个属性，表示当前生成器。

我们在这里介绍执行上下文的各个版本定义，是考虑到你可能会从各种网上的文章中接触这些概念，如果不把它们理清楚，我们就很难分辨对错。如果是我们自己使用，我建议统一使用最新的ES2018中规定的术语定义。

尽管我们介绍了这些定义，但我并不打算按照JavaScript标准的思路，从实现的角度去介绍函数的执行过程，这是不容易被理解的。

我想试着从代码实例出发，跟你一起推导函数执行过程中需要哪些信息，它们又对应着执行上下文中的哪些部分。

比如，我们看以下的这段JavaScript代码：

```
var b = {}
let c = 1
this.a = 2;
```

要想正确执行它，我们需要知道以下信息：

1. var 把 b 声明到哪里；
2. b 表示哪个变量；
3. b 的原型是哪个对象；
4. let 把 c 声明到哪里；
5. this 指向哪个对象。

这些信息就需要执行上下文来给出了，这段代码出现在不同的位置，甚至在每次执行中，会关联到不同的执行上下文，所以，同样的代码会产生不一样的行为。

在这两篇文章中，我会基本覆盖执行上下文的组成部分，本篇我们先讲var声明与赋值，let，realm三个特性来分析上下文提供的信息，分析执行上下文中提供的信息。

var 声明与赋值

我们来分析一段代码：

```
var b = 1
```

通常我们认为它声明了b，并且为它赋值为1，var声明作用域函数执行的作用域。也就是说，var会穿透for 、if等语句。

在只有var，没有let的旧JavaScript时代，诞生了一个技巧，叫做：立即执行的函数表达式（IIFE），通过创建一个函数，并且立即执行，来构造一个新的域，从而控制var的范围。

由于语法规定了function关键字开头是函数声明，所以要想让函数变成函数表达式，我们必须得加点东西，最常见的做法是加括号。

```
(function(){
    var a;
    //code
}());


(function(){
    var a;
    //code
})();
```

但是，括号有个缺点，那就是如果上一行代码不写分号，括号会被解释为上一行代码最末的函数调用，产生完全不符合预期，并且难以调试的行为，加号等运算符也有类似的问题。所以一些推荐不加分号的代码风格规范，会要求在括号前面加上分号。

```
;(function(){
    var a;
    //code
}())


;(function(){
    var a;
    //code
})()
```

我比较推荐的写法是使用void关键字。也就是下面的这种形式。

```
void function(){
    var a;
    //code
}();
```

这有效避免了语法问题，同时，语义上void运算表示忽略后面表达式的值，变成undefined，我们确实不关心IIFE的返回值，所以语义也更为合理。

值得特别注意的是，有时候var的特性会导致声明的变量和被赋值的变量是两个b，JavaScript中有特例，那就是使用with的时候：

```
var b;
void function(){
    var env = {b:1};
    b = 2;
    console.log("In function b:", b);
    with(env) {
        var b = 3;
        console.log("In with b:", b);
    }
}();
console.log("Global b:", b);
```

在这个例子中，我们利用立即执行的函数表达式（IIFE）构造了一个函数的执行环境，并且在里面使用了我们一开头的代码。

可以看到，在Global function with三个环境中，b的值都不一样，而在function环境中，并没有出现var b，这说明with内的var b作用到了function这个环境当中。

var b = {} 这样一句对两个域产生了作用，从语言的角度是个非常糟糕的设计，这也是一些人坚定地反对在任何场景下使用with的原因之一。

let

let是 ES6开始引入的新的变量声明模式，比起var的诸多弊病，let做了非常明确的梳理和规定。

为了实现let，JavaScript在运行时引入了块级作用域。也就是说，在let出现之前，JavaScript的 if for 等语句皆不产生作用域。

我简单统计了下，以下语句会产生let使用的作用域：

- for;
- if;
- switch;
- try/catch/finally。

Realm

在最新的标准（9.0）中，JavaScript引入了一个新概念Realm，它的中文意思是“国度”“领域”“范围”。这个英文的用法就有点比喻的意思，几个翻译都不太适合JavaScript语境，所以这里就不翻译啦。

我们继续来看这段代码：

```
var b = {}
```

在 ES2016 之前的版本中，标准中甚少提及{}的原型问题。但在实际的前端开发中，通过iframe等方式创建多window环境并非罕见的操作，所以，这才促成了新概念Realm的引入。

Realm中包含一组完整的内置对象，而且是复制关系。

对不同Realm中的对象操作，会有一些需要格外注意的问题，比如 instanceOf 几乎是失效的。

以下代码展示了在浏览器环境中获取来自两个Realm的对象，它们跟本土的Object做instanceOf时会产生差异：

```
var iframe = document.createElement('iframe')
document.documentElement.appendChild(iframe)
iframe.src="javascript:var b = {};"

var b1 = iframe.contentWindow.b;
var b2 = {};

console.log(typeof b1, typeof b2); //object object

console.log(b1 instanceof Object, b2 instanceof Object); //false true
```

可以看到，由于b1、 b2由同样的代码“ {} ”在不同的Realm中执行，所以表现出了不同的行为。

结语

在今天的课程中，我帮你梳理了一些概念：有编程语言的概念闭包，也有各个版本中的JavaScript标准中的概念：执行上下文、作用域、this值等等。

之后我们又从代码的角度，分析了一些执行上下文中所需要的信息，并从var、let、对象字面量等语法中，推导出了词法作用域、变量作用域、Realm的设计。

最后留给你一个问题：你喜欢使用let还是var？听过今天的课程，你的想法是否有改变呢？为什么？