彻底搞懂作用域、执行上下文、词法环境

Rockky 2021-12-19 9,961 阅读12分钟

关注

相信很多小伙伴在初学 `JavaScript` 的时候会经常对 `作用域` ， `执行上下文` ， `词法环境` 等概念混淆，其实主要还是对这些知识点没有一个清晰的认识与了解。

在我之前的一篇文章中已经对JavaScript作用域以及作用域链有一个比较详细和清晰的描述：详解JavaScript作用域与作用域链

所以这篇文章就不对作用域作过多的解释啦，接下来就主要来介绍一下本文的主角：执行上下文和词法环境。

执行上下文

首先，什么是执行上下文?

官方一点地说，执行上下文（ `Execution context stack` 简称 `ECS` ）就是一个评估和执行 `JavaScript` 代码的环境的抽象概念。通俗地说，就是每当 `Javascript` 代码在运行的时候，它都是在执行上下文中运行。

JavaScript 中有三种执行上下文

- **全局执行上下文** — 这是默认或者说基础的上下文，任何不在函数内部的代码都在全局上下文中。它会执行两件事：创建一个全局的 `window` 对象（浏览器的情况下），并且设置 `this` 的值等于这个全局对象。一个程序中只会有一个全局执行上下文。
- **函数执行上下文** — 每当一个函数被调用时, 都会为该函数创建一个新的执行上下文。每个函数都有它自己的执行上下文，不过是在函数被调用时创建的。函数上下文可以有任意多个。每当一个新的执行上下文被创建，它会按定义的顺序（将在后文讨论）执行一系列步骤。
- **Eval 函数执行上下文** — 执行在 `eval` 函数内部的代码也会有它属于自己的执行上下文，但由于 并不经常使用 `eval` ，所以在这里不作讨论。

执行上下文的生命周期包括三个阶段：**创建阶段→执行阶段→回收阶段**，本文重点介绍创建阶段。

（1）创建阶段

在 `JavaScript` 代码执行前，执行上下文将经历创建阶段。在创建阶段会发生三件事：

1. **this** 值的决定，即我们所熟知的 **This 绑定**。
2. 创建**词法环境**组件。（`LexicalEnvironment component`）
3. 创建**变量环境**组件。（`VariableEnvironment component`） （下文会解释词法环境和变量环境）

所以执行上下文用伪代码可以这样表示：

```ini
ExecutionContext = {                  // 执行上下文
    ThisBinding = <this value>,       // this绑定
    LexicalEnvironment = { ... },     // 词法环境
    VariableEnvironment = { ... },    // 变量环境
}
```

This 绑定：

在全局执行上下文中， `this` 的值指向全局对象。(在浏览器中， `this` 引用 `Window` 对象)。在函数执行上下文中， `this` 的值取决于该函数是如何被调用的。如果它被一个引用对象调用，那么 `this` 会被设置成那个对象，否则 `this` 的值被设置为全局对象或者 `undefined`（在严格模式下）。

（2）执行阶段

执行变量赋值、代码执行。

（3）回收阶段

执行上下文出栈等待虚拟机回收执行上下文

注意： 在执行阶段，如果 JavaScript 引擎不能在源码中声明的实际位置找到 `let` 变量的值，它会被赋值为 `undefined` 。

执行上下文栈

执行上下文栈（`Execution Context Stack`）（也称**调用栈**、**执行栈**），个人比较习惯叫调用栈，所以下文用调用栈来描述。它是一种拥有 `LIFO`（后进先出）数据结构的栈，被用来存储代码运行时创建的所有执行上下文。

当 JavaScript 引擎第一次遇到我们写的脚本时，它会创建一个**全局的执行上下文**并且压入当前调用栈。每当引擎遇到一个函数调用，它会为该函数创建一个**新的函数执行上下**文并压入栈的顶部。

引擎会执行那些执行上下文位于栈顶的函数。当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

让我们通过下面的代码示例来理解：

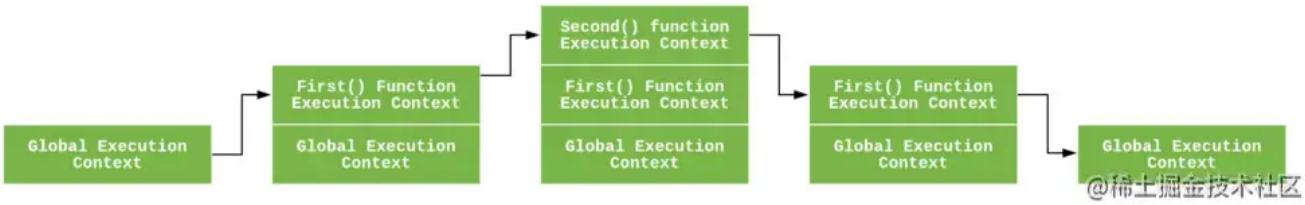
sql 复制代码

```sql
let a = 'Hello World!';

function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}

function second() {
  console.log('Inside second function');
}

first();
console.log('Inside Global Execution Context');
```



当上述代码在浏览器加载时，JavaScript 引擎创建了一个全局执行上下文并把它压入当前执行栈。当遇到 `first()` 函数调用时，JavaScript 引擎为该函数创建一个新的执行上下文并把它压

当从 `first()` 函数内部调用 `second()` 函数时，JavaScript 引擎为 `second()` 函数创建了一个新的执行上下文并把它压入当前执行栈的顶部。当 `second()` 函数执行完毕，它的执行上下文会从当前栈弹出，并且控制流程到达下一个执行上下文，即 `first()` 函数的执行上下文。

当 `first()` 执行完毕，它的执行上下文从栈弹出，控制流程到达全局执行上下文。一旦所有代码执行完毕，JavaScript 引擎从当前栈中移除全局执行上下文。

词法环境

官方定义：**词法环境**是一种规范类型，基于 `ECMAScript` 代码的词法嵌套结构来定义**标识符**和具体变量和函数的关联。一个词法环境由**环境记录器**和一个可能的**引用外部词法环境的空值**组成。

简单来说，词法环境是一种持有**标识符—变量的映射**的结构。（这里的**标识符**指的是变量/函数的名字，而**变量**是对实际对象[包含函数类型对象]或原始数据的引用）。

或者也可以这样说，词法环境就是指相应代码块内标识符与变量值、函数值之间的关联关系的一种体现。

词法环境有两种类型：

- **全局环境**（在全局执行上下文中）是没有外部环境引用的词法环境。全局环境的外部环境引用是 **null**。它拥有内建的 `Object/Array` 等、在环境记录器内的原型函数（关联全局对象，比如 `window` 对象）还有任何用户定义的全局变量，并且 `this` 的值指向全局对象。
- 在**函数环境**中，函数内部用户定义的变量存储在**环境记录器**中。并且引用的外部环境可能是全局环境，或者任何包含此内部函数的外部函数。

在词法环境的**内部**有两个组件：

- **环境记录器**：是存储变量和函数声明的实际位置。
- **外部环境的引用**：意味着它可以访问其父级词法环境。

根据词法环境的两种类型，其内部的**环境记录器**也有两种类型：

1. **声明式环境记录器**（在**函数环境**中）：存储变量、函数和参数。
2. **对象环境记录器**（在**全局环境**中）：用来定义出现在**全局上下文**中的变量和函数的关系。

注意：对于**函数环境**，**声明式环境记录器**还包含了一个传递给函数的 `arguments` 对象（此对象存储索引和参数的映射）和传递给函数的参数的 **length**。

抽象地讲，词法环境在伪代码中看起来像这样：

```dart
GlobalExectionContext = {        // 全局执行上下文
    LexicalEnvironment: {        // 词法环境
        EnvironmentRecord: {     // 环境记录器：存储变量和函数声明的实际位置
            Type: "Object",
            // 在这里绑定标识符
        }
        outer: <null>            // 对外部环境的引用：可以访问其父级词法环境
    }
}

FunctionExectionContext = {      // 函数执行上下文
    LexicalEnvironment: {
        EnvironmentRecord: {
            Type: "Declarative",
            // 在这里绑定标识符
        }
        outer: <Global or outer function environment reference>
    }
}
```

变量环境

它同样是一个词法环境，其环境记录器持有**变量声明语句**在执行上下文中创建的绑定关系。

如上所述，变量环境也是一个词法环境，所以它有着上面定义的词法环境的所有属性。

之所以在 `ES5` 的规范里要单独分出一个变量环境的概念是为 `ES6` 服务的： 在 ES6 中，**词法环境**组件和**变量环境**的一个不同就是前者被用来存储**函数声明**和变量（ `let` 和 `const` ）绑定，而后者只用来存储 `var` 变量绑定。

我们看点样例代码来理解上面的概念：

```ini
```

```
var c;
function multiply(e, f) {
    var g = 20;
    return e * f * g;
}
c = multiply(20, 30);
```

执行上下文看起来像这样：

yaml 复制代码

```yaml
GlobalExectionContext = {
  ThisBinding: <Global Object>,
  LexicalEnvironment: {       // 词法环境
    EnvironmentRecord: {
      Type: "Object",
      // 在这里绑定标识符
      a: < uninitialized >,   // let、const声明的变量
      b: < uninitialized >,   // let、const声明的变量
      multiply: < func >      // 函数声明
    }
    outer: <null>
  },
  VariableEnvironment: {     // 变量环境
    EnvironmentRecord: {
      Type: "Object",
      // 在这里绑定标识符
      c: undefined,         // var声明的变量
    }
    outer: <null>
  }
}

FunctionExectionContext = {
  ThisBinding: <Global Object>,
  LexicalEnvironment: {         // 词法环境
    EnvironmentRecord: {
      Type: "Declarative",
      // 在这里绑定标识符
      Arguments: {0: 20, 1: 30, length: 2},   // arguments对象
    },
    outer: <GlobalLexicalEnvironment>
  },
  VariableEnvironment: {        // 变量环境
     EnvironmentRecord: {
       Type: "Declarative",
```

```
    },
    outer: <GlobalLexicalEnvironment>
  }
}
```

注意 — 只有遇到调用函数 `multiply` 时，函数执行上下文才会被创建。

可能你已经注意到 `let` 和 `const` 定义的变量并没有关联任何值，但 `var` 定义的变量被设成了 `undefined` 。

这是因为在创建阶段时，引擎检查代码找出变量和函数声明，虽然函数声明完全存储在环境中，但是变量最初设置为 `undefined` （ `var` 情况下），或者未初始化（ `let` 和 `const` 情况下）。

这就是为什么你可以在声明之前访问 `var` 定义的变量（虽然是 `undefined` ），但是在声明之前访问 `let` 和 `const` 的变量会得到一个引用错误。

这就是我们说的变量声明提升。

分析程序执行全过程

- 程序启动，全局执行上下文被创建，压入调用栈
    1. 创建全局上下文的 **词法环境**
        1. 创建 **对象环境记录器** ，它用来定义出现在 **全局上下文** 中的变量和函数的关系（负责处理 `let` 和 `const` 定义的变量）
        2. 创建 **外部环境引用**，值为 **`null`**
    2. 创建全局上下文的 **变量环境**
        1. 创建 **对象环境记录器**，它持有 **变量声明语句** 在执行上下文中创建的绑定关系（负责处理 `var` 定义的变量，初始值为 `undefined` 造成声明提升）
        2. 创建 **外部环境引用**，值为 **`null`**
    3. 确定 `this` 值为全局对象（以浏览器为例，就是 `window` ）

1. 创建函数上下文的 **词法环境**

    1. 创建 **声明式环境记录器** ，存储变量、函数和参数，它包含了一个传递给函数的 `arguments` 对象（此对象存储索引和参数的映射）和传递给函数的参数的 **length**。（负责处理 `let` 和 `const` 定义的变量）
    2. 创建 **外部环境引用**，值为全局对象，或者为父级词法环境（作用域）

2. 创建函数上下文的 **变量环境**

    1. 创建 **声明式环境记录器** ，存储变量、函数和参数，它包含了一个传递给函数的 `arguments` 对象（此对象存储索引和参数的映射）和传递给函数的参数的 **length**。（负责处理 `var` 定义的变量，初始值为 `undefined` 造成声明提升）
    2. 创建 **外部环境引用**，值为全局对象，或者为父级词法环境（作用域）

3. 确定 `this` 值

- 进入函数执行上下文的执行阶段：

    1. 在上下文中运行/解释函数代码，并在代码逐行执行时分配变量值。

总结

现在我们来总结一下吧：

首先，JavaScript属于**解释型语言**，JavaScript的执行分为解释和执行两个阶段，这两个阶段所做的事并不一样：

解释阶段：

- 词法分析
- 语法分析
- 作用域规则确定

执行阶段：

- 创建执行上下文
- 执行函数代码

JavaScript解释阶段便会确定作用域规则，因此**作用域在函数定义时就已经确定了**，而不是在函数调用时确定，但是**执行上下文是函数执行之前创建的**。执行上下文最明显的就是this的指向是执行时确定的。而作用域访问的变量是编写代码的结构确定的。

作用域和执行上下文之间最大的区别是： **执行上下文在运行时确定，随时可能改变；作用域在定义时就确定，并且不会改变。**

一个作用域下可能包含若干个上下文环境。有可能从来没有过上下文环境（函数从来就没有被调用过）；有可能有过，现在函数被调用完毕后，上下文环境被销毁了；有可能同时存在一个或多个（闭包）。**同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。**

最后的最后，简要概况一下 作用域 ， 词法环境 ， 执行上下文 这三者的概念：

- **作用域**：作用域就是一个独立的区域，它可以让变量不会向外暴露出去。作用域最大的用处就是隔离变量。内层作用域可以访问外层作用域。一个作用域下可能包含若干个执行上下文。
- **词法环境**：指相应代码块内标识符与变量值、函数值之间的关联关系的一种体现。词环境内部包含环境记录器和对外部环境的引用。环境记录器是存储变量和函数声明的实际位置，对外部环境的引用意味着可以访问父级词法环境。
- **执行上下文**：JavaScript代码运行的环境。分为全局执行上下文，函数执行上下文和eval函数执行上下文（前两个较常见）。创建执行上下文时会进行this绑定、创建词法环境和变量环境。

参考

[译] 理解 JavaScript 中的执行上下文和执行栈
面试官：说说执行上下文吧

往期文章

你真的了解 script 标签吗?
详解JavaScript作用域与作用域链
用Mock.js模拟后端接口数据，看这一篇就够了！