ES6~ES13新特性（一）

王红元 coderwhy

目录
content

新的ECMA代码执行描述

- **在执行学习JavaScript代码执行过程中，我们学习了很多ECMA文档的术语：**
  - 执行上下文栈：Execution Context Stack，用于执行上下文的栈结构；
  - 执行上下文：Execution Context，代码在执行之前会先创建对应的执行上下文；
  - 变量对象：Variable Object，上下文关联的VO对象，用于记录函数和变量声明；
  - 全局对象：Global Object，全局执行上下文关联的VO对象；
  - 激活对象：Activation Object，函数执行上下文关联的VO对象；
  - 作用域链：scope chain，作用域链，用于关联指向上下文的变量查找；

- **在新的ECMA代码执行描述中（ES5以及之上），对于代码的执行流程描述改成了另外的一些词汇：**
  - 基本思路是相同的，只是对于一些词汇的描述发生了改变；
  - 执行上下文栈和执行上下文也是相同的；

词法环境（Lexical Environments）

- **词法环境是一种规范类型，用于在词法嵌套结构中定义关联的变量、函数等标识符；**
  - 一个词法环境是由环境记录（Environment Record）和一个外部词法环境（o*ute;r* Lexical Environment）组成；
  - 一个词法环境经常用于关联一个函数声明、代码块语句、try-catch语句，当它们的代码被执行时，词法环境被创建出来；

> A *Lexical Environment* is a specification type used to define the association of *Identifier*s to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an *outer* Lexical Environment. Usually a Lexical Environment is associated with some specific syntactic structure of ECMAScript code such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch* clause of a *TryStatement* and a new Lexical Environment is created each time such code is evaluated.

- **也就是在ES5之后，执行一个代码，通常会关联对应的词法环境；**
  - 那么执行上下文会关联哪些词法环境呢?

Table 24: Additional State Components for ECMAScript Code Execution Contexts

Component	Purpose
LexicalEnvironment	Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.
VariableEnvironment	Identifies the Lexical Environment whose EnvironmentRecord holds bindings created by *VariableStatements* within this execution context.

LexicalEnvironment和VariableEnvironment

- LexicalEnvironment用于处理let、const声明的标识符：

> **let** and **const** declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initializer* is assigned the value of its *Initializer*'s *AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a **let** declaration does not have an *Initializer* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

- VariableEnvironment用于处理var和function声明的标识符：

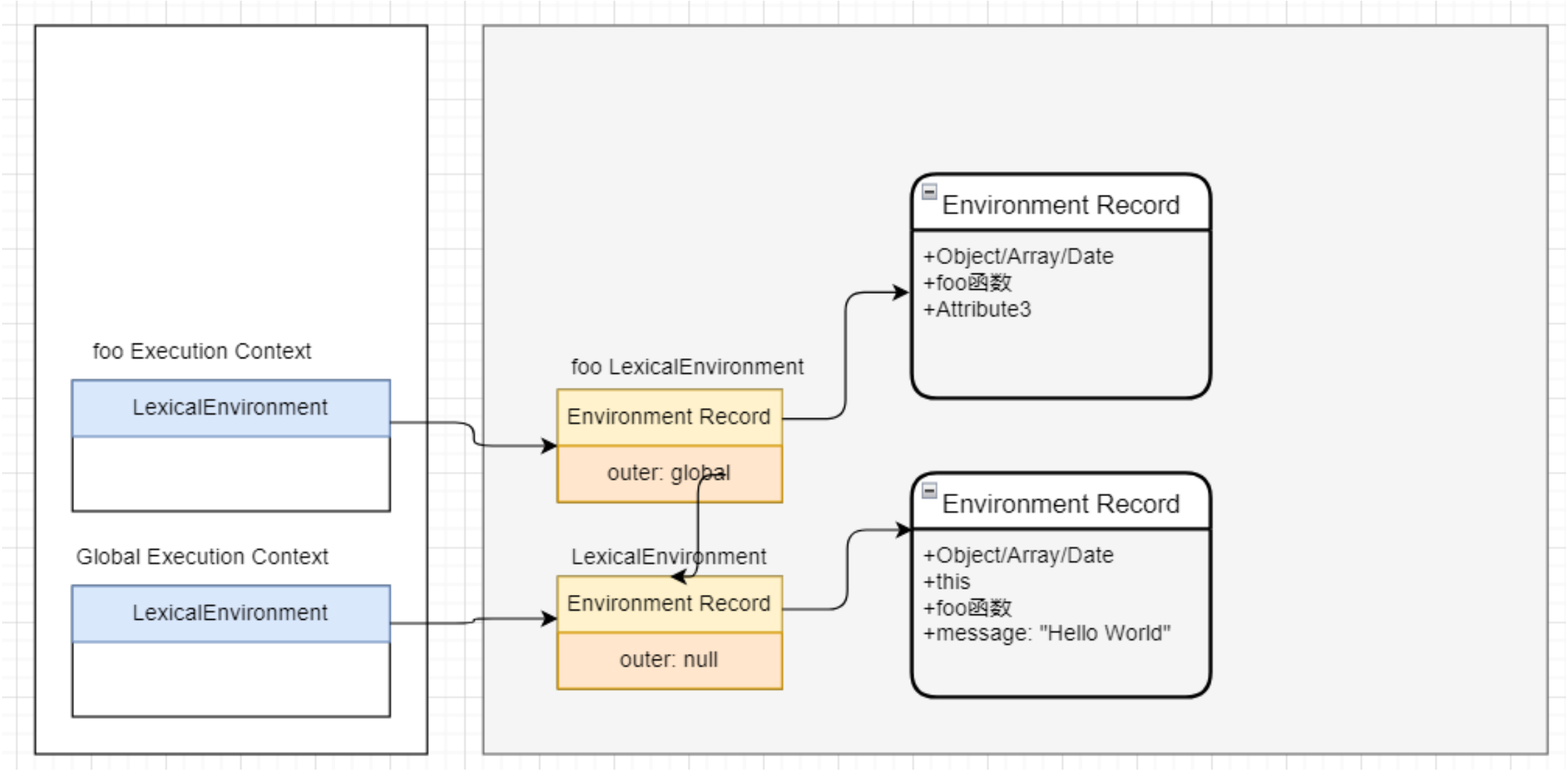
> A **var** statement declares variables that are scoped to the running execution context's VariableEnvironment. Var variables are created when their containing Lexical Environment is instantiated and are initialized to **undefined** when created. Within the scope of any VariableEnvironment a common *BindingIdentifier* may appear in more than one *VariableDeclaration* but those declarations collectively define only one variable. A variable defined by a *VariableDeclaration* with an *Initializer* is assigned the value of its *Initializer*'s *AssignmentExpression* when the *VariableDeclaration* is executed, not when the variable is created.

环境记录（Environment Record）

- **在这个规范中有两种主要的环境记录值:声明式环境记录和对象环境记录。**
  - 声明式环境记录：声明性环境记录用于定义ECMAScript语言语法元素的效果，如函数声明、变量声明和直接将标识符绑定与ECMAScript语言值关联起来的Catch子句。
  - 对象式环境记录：对象环境记录用于定义ECMAScript元素的效果，例如WithStatement，它将标识符绑定与某些对象的属性关联起来。

There are two primary kinds of *Environment Record* values used in this specification: *declarative Environment Records* and *object Environment Records*. Declarative Environment Records are used to define the effect of ECMAScript language syntactic elements such as *FunctionDeclarations*, *VariableDeclarations*, and *Catch* clauses that directly associate identifier bindings with ECMAScript language values. Object Environment Records are used to define the effect of ECMAScript elements such as *WithStatement* that associate identifier bindings with the properties of some object. Global Environment Records and function Environment Records are specializations that are used for specifically for *Script* global declarations and for top-level declarations within functions.

新ECMA描述内存图


foo Execution Context
LexicalEnvironment
Global Execution Context
LexicalEnvironment
foo LexicalEnvironment
Environment Record
outer: global
LexicalEnvironment
Environment Record
outer: null
Environment Record
+Object/Array/Date
+foo函数
+Attribute3
Environment Record
+Object/Array/Date
+this
+foo函数
+message: "Hello World"

let/const基本使用

- **在ES5中我们声明变量都是使用的var关键字，从ES6开始新增了两个关键字可以声明变量：let、const**
  - let、const在其他编程语言中都是有的，所以也并不是新鲜的关键字；
  - 但是let、const确确实实给JavaScript带来一些不一样的东西；
- **let关键字：**
  - 从直观的角度来说，let和var是没有太大的区别的，都是用于声明一个变量；
- **const关键字：**
  - const关键字是constant的单词的缩写，表示常量、衡量的意思；
  - 它表示保存的数据一旦被赋值，就不能被修改；
  - 但是如果赋值的是引用类型，那么可以通过引用找到对应的对象，修改对象的内容；
- 注意：
  - 另外let、const不允许重复声明变量；

let/const作用域提升

- let、const和var的另一个重要区别是作用域提升：
  - 我们知道var声明的变量是会进行作用域提升的；
  - 但是如果我们使用let声明的变量，在声明之前访问会报错；

```
console.log(foo) // ReferenceError: Cannot access 'foo' before initialization

let foo = "foo"
```

- 那么是不是意味着foo变量只有在代码执行阶段才会创建的呢?
  - 事实上并不是这样的，我们可以看一下ECMA262对let和const的描述；
  - **这些变量会被创建在包含他们的词法环境被实例化时，但是是不可以访问它们的，直到词法绑定被求值；**

> **let** and **const** declarations define variables that are scoped to the running execution context's LexicalEnvironment. The variables are created when their containing Lexical Environment is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initializer* is assigned the value of its *Initializer*'s *AssignmentExpression* when the *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a **let** declaration does not have an *Initializer* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

暂时性死区 (TDZ)

- 我们知道，在let、const定义的标识符真正执行到声明的代码之前，是不能被访问的
  - 从块作用域的顶部一直到变量声明完成之前，这个变量处在暂时性死区（TDZ，temporal dead zone）

```
{
  console.log(name)

  let name = "why"
}
```

```
Uncaught ReferenceError: Cannot access 'name' before initialization
    at 05_暂时性死区.html:14:19
```

- 使用术语 “temporal” 是因为区域取决于执行顺序（时间），而不是编写代码的位置；

```
function foo() {
  console.log(message)
}

let message = "Hello World"
foo()
```

let/const有没有作用域提升呢?

- **从上面我们可以看出，在执行上下文的词法环境创建出来的时候，变量事实上已经被创建了，只是这个变量是不能被访问的。**
  - 那么变量已经有了，但是不能被访问，是不是一种作用域的提升呢?
- **事实上维基百科并没有对作用域提升有严格的概念解释，那么我们自己从字面量上理解;**
  - **作用域提升**：在声明变量的作用域中，如果这个变量可以在声明之前被访问，那么我们可以称之为作用域提升;
  - 在这里，它虽然被创建出来了，但是不能被访问，我认为不能称之为作用域提升;
- 所以我的观点是let、const没有进行作用域提升，但是会在解析阶段被创建出来。

Window对象添加属性

- 我们知道，在全局通过var来声明一个变量，事实上会在window上添加一个属性：
  - 但是let、const是不会给window上添加任何属性的。
- 那么我们可能会想这个变量是保存在哪里呢?

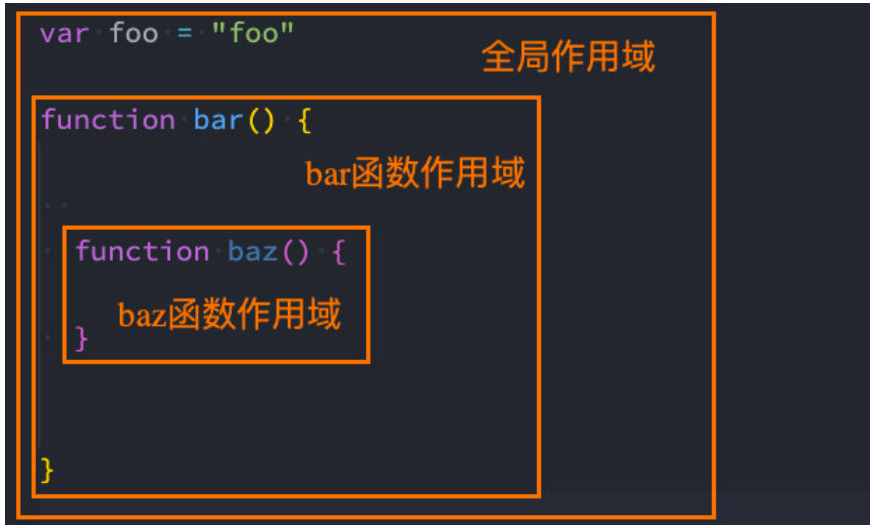
A global Environment Record is logically a single record but it is specified as a composite encapsulating an object Environment Record and a declarative Environment Record. The object Environment Record has as

Table 19: Additional Fields of Global Environment Records

Field Name	Value	Meaning
[[ObjectRecord]]	Object Environment Record	Binding object is the global object. It contains global built-in bindings as well as *FunctionDeclaration*, *GeneratorDeclaration*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, and *VariableDeclaration* bindings in global code for the associated realm.
[[GlobalThisValue]]	Object	The value returned by **this** in global scope. Hosts may provide any ECMAScript Object value.
[[DeclarativeRecord]]	Declarative Environment Record	Contains bindings for all declarations in global code for the associated realm code except for *FunctionDeclaration*, *GeneratorDeclaration*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, and *VariableDeclaration bindings*.
[[VarNames]]	List of String	The string names bound by *FunctionDeclaration*, *GeneratorDeclaration*, *AsyncFunctionDeclaration*, *AsyncGeneratorDeclaration*, and *VariableDeclaration* declarations in global code for the associated realm.

var的块级作用域

- 在我们前面的学习中，JavaScript只会形成两个作用域：**全局作用域和函数作用域**。



- ES5中放到一个代码中定义的变量，外面是可以访问的：

```
// var没有块级作用域
{
  // 编写语句
  var foo = "foo"
}

console.log(foo) // foo 可以访问到
```

通过`var`声明的变量或者非严格模式下 (non-strict mode) 创建的函数声明**没有**块级作用域。在语句块里声明的变量的作用域不仅是其所在的函数或者 script 标签内，所设置变量的影响会在超出语句块本身之外持续存在。 换句话说，这种语句块不会引入一个作用域。尽管单独的语句块是合法的语句，但在 JavaScript 中你不会想使用单独的语句

let/const的块级作用域

- 在ES6中新增了块级作用域，并且通过let、const、function、class声明的标识符是具备块级作用域的限制的：

```
{
  let foo = "foo"
  function bar() {
    console.log("bar")
  }
  class Person {}
}

console.log(foo) // ReferenceError: foo is not defined
bar() // 可以访问
var p = new Person() // ReferenceError: foo is not defined
```

使用 `let` 和 `const` 声明的变量是有块级作用域的。

- 但是我们会发现函数拥有块级作用域，但是外面依然是可以访问的：
  - 这是因为引擎会对函数的声明进行特殊的处理，允许像var那样进行提升；

块级作用域的应用

- 我来看一个实际的案例：获取多个按钮监听点击

```
<button>按钮1</button>
<button>按钮2</button>
<button>按钮3</button>
<button>按钮4</button>
```

- 使用let或者const来实现：

```
var btns = document.getElementsByTagName("button")
for (let i = 0; i < btns.length; i++) {
  btns[i].onclick = function() {
    console.log("第" + i + "个按钮被点击")
  }
}
```

var、let、const的选择

- **那么在开发中，我们到底应该选择使用哪一种方式来定义我们的变量呢?**
- **对于var的使用:**
  - 我们需要明白一个事实，var所表现出来的特殊性：比如作用域提升、window全局对象、没有块级作用域等都是一些历史遗留问题;
  - 其实是JavaScript在设计之初的一种语言缺陷;
  - 当然目前市场上也在利用这种缺陷出一系列的面试题，来考察大家对JavaScript语言本身以及底层的理解;
  - 但是在实际工作中，我们可以使用最新的规范来编写，也就是不再使用var来定义变量了;
- **对于let、const:**
  - 对于let和const来说，是目前开发中推荐使用的;
  - 我们会优先推荐使用const，这样可以保证数据的安全性不会被随意的篡改;
  - 只有当我们明确知道一个变量后续会需要被重新赋值时，这个时候再使用let;
  - 这种在很多其他语言里面也都是一种约定俗成的规范，尽量我们也遵守这种规范;

字符串模板基本使用

- 在ES6之前，如果我们想要将字符串和一些动态的变量（标识符）拼接到一起，是非常麻烦和丑陋的（ugly）。
- ES6允许我们使用字符串模板来嵌入JS的变量或者表达式来进行拼接：
  - 首先，我们会使用 `` 符号来编写字符串，称之为模板字符串；
  - 其次，在模板字符串中，我们可以通过 **${expression}** 来嵌入动态的内容；

```
const name = "why"
const age = 18
const height = 1.88

console.log(`my name is ${name}, age is ${age}, height is ${height}`)
console.log(`我是成年人吗? ${age >= 18 ? '是': '否'}`)

function foo() {
  return "function is foo"
}

console.log(`my function is ${foo()}`)
```

标签模板字符串使用

- 模板字符串还有另外一种用法：标签模板字符串（Tagged Template Literals）。
- 我们一起来看一个普通的JavaScript的函数：

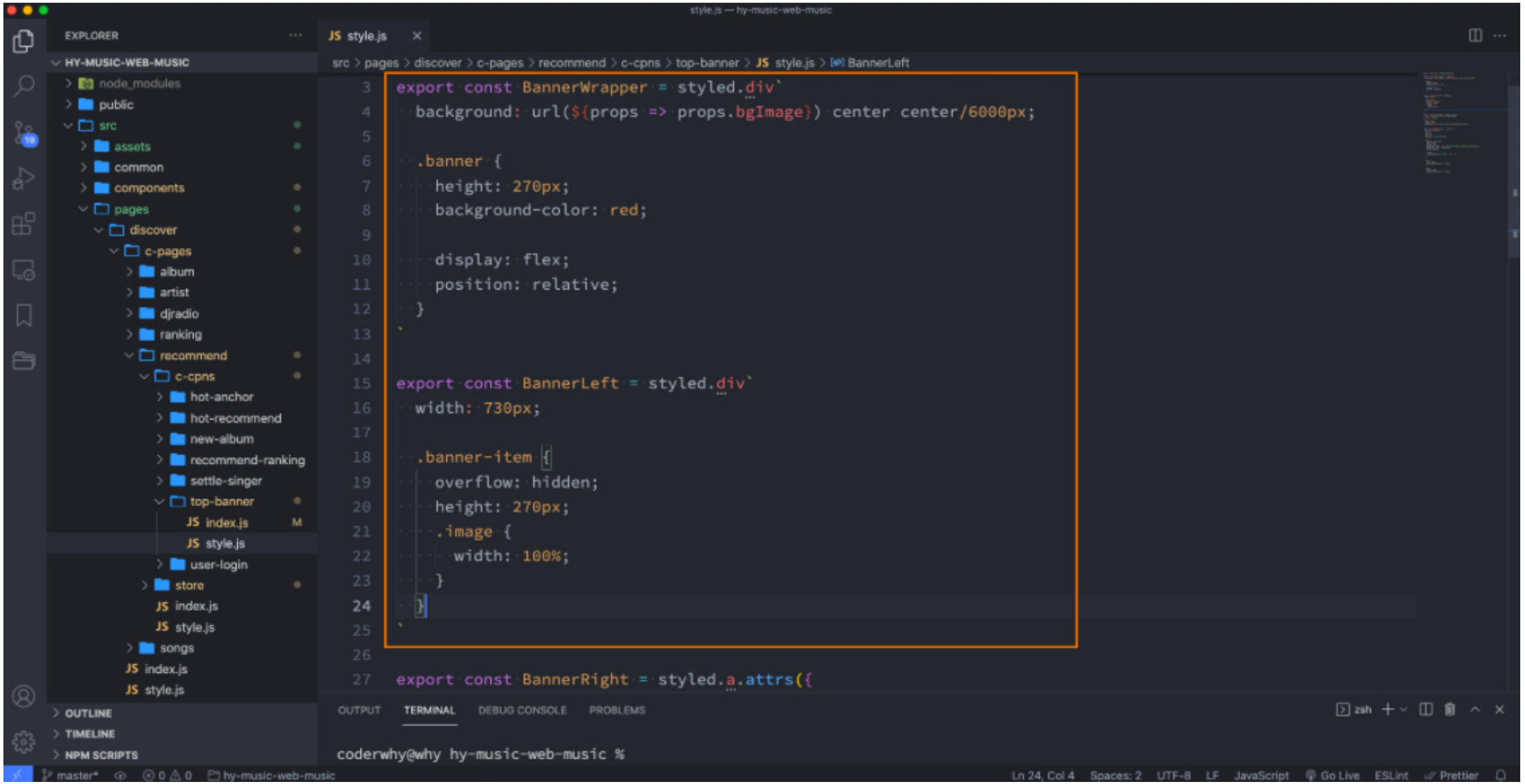
```
function foo(...args) {
  console.log(args)
}

// [ 'Hello World' ]
foo("Hello World")
```

- 如果我们使用标签模板字符串，并且在调用的时候插入其他的变量：
  - 模板字符串被拆分了；
  - 第一个元素是数组，是被模块字符串拆分的字符串组合；
  - 后面的元素是一个个模块字符串传入的内容；

```
const name = "why"
const age = 18
// [ [ 'Hello ', ' World ', '' ], 'why', 18 ]
foo`Hello ${name} World ${age}`
```

React的styled-components库



函数的默认参数

- 在ES6之前，我们编写的函数参数是没有默认值的，所以我们在编写函数时，如果有下面的需求：
  - 传入了参数，那么使用传入的参数；
  - 没有传入参数，那么使用一个默认值；
- 而在ES6中，我们允许给函数一个默认值：

```
function foo(x = 20, y = 30) {
  console.log(x, y)
}

foo(50, 100) // 50 100
foo() // 20 30
```

```
function foo() {
  var x =
    arguments.length > 0 && arguments[0] !== undefined ? arguments[0] : 20;
  var y =
    arguments.length > 1 && arguments[1] !== undefined ? arguments[1] : 30;
  console.log(x, y);
}
```

函数默认值的补充

- 默认值也可以和解构一起来使用：

```
// 写法一:
function foo({name, age} = {name: "why", age: 18}) {
  console.log(name, age)
}

// 写法二:
function foo({name = "why", age = 18} = {}) {
  console.log(name, age)
}
```

- 另外参数的默认值我们通常会将其放到最后（在很多语言中，如果不放到最后其实会报错的）：
  - 但是JavaScript允许不将其放到最后，但是意味着还是会按照顺序来匹配；
- 另外默认值会改变函数的length的个数，默认值以及后面的参数都不计算在length之内了。

函数的剩余参数（已经学习）

- **ES6中引用了rest parameter，可以将不定数量的参数放入到一个数组中：**
  - 如果最后一个参数是 ... 为前缀的，那么它会将剩余的参数放到该参数中，并且作为一个数组；

```
function foo(m, n, ...args) {
  console.log(m, n)
  console.log(args)
}
```

- **那么剩余参数和arguments有什么区别呢?**
  - 剩余参数只包含那些没有对应形参的实参，而 arguments 对象包含了传给函数的所有实参；
  - arguments对象不是一个真正的数组，而rest参数是一个真正的数组，可以进行数组的所有操作；
  - arguments是早期的ECMAScript中为了方便去获取所有的参数提供的一个数据结构，而rest参数是ES6中提供并且希望以此来替代arguments的；

- **注意：剩余参数必须放到最后一个位置，否则会报错。**

函数箭头函数的补充

- **在前面我们已经学习了箭头函数的用法，这里进行一些补充：**
  - 箭头函数是没有显式原型prototype的，所以不能作为构造函数，使用new来创建对象；
  - 箭头函数也不绑定this、arguments、super参数；

```
var foo = () => {
  console.log("foo")
}

console.log(foo.prototype) // undefined

// TypeError: foo is not a constructor
var f = new foo()
```

14.2.16 Runtime Semantics: Evaluation

ArrowFunction : *ArrowParameters* => *ConciseBody*

1. If the function code for this *ArrowFunction* is strict mode code (10.2.1), let *strict* be **true**. Otherwise let *strict* be **false**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *parameters* be CoveredFormalsList of *ArrowParameters*.
4. Let *closure* be FunctionCreate(Arrow, *parameters*, *ConciseBody*, *scope*, *strict*).
5. Return *closure*.

NOTE An *ArrowFunction* does not define local bindings for **arguments**, **super**, **this**, or **new.target**. Any reference to **arguments**, **super**, **this**, or **new.target** within an *ArrowFunction* must resolve to a binding in a lexically enclosing environment. Typically this will

展开语法

- **展开语法(Spread syntax)**：
  - 可以在函数调用/数组构造时，将数组表达式或者string在语法层面展开；
  - 还可以在构造字面量对象时，将对象表达式按key-value的方式展开；
- **展开语法的场景：**
  - 在函数调用时使用；
  - 在数组构造时使用；
  - 在构建对象字面量时，也可以使用展开运算符，这个是在ES2018（ES9）中添加的新特性；
- **注意：展开运算符其实是一种浅拷贝；**

数值的表示

- 在ES6中规范了二进制和八进制的写法：

```
const num1 = 100
// b -> binary
const num2 = 0b100
// octonary
const num3 = 0o100
// hexadecimal
const num4 = 0x100
```

- 另外在ES2021新增特性：数字过长时，可以使用_作为连接符

```
// ES2021新增特性
const num5 = 100_000_000
```

Symbol的基本使用

- Symbol是什么呢？Symbol是ES6中新增的一个基本数据类型，翻译为符号。
- **那么为什么需要Symbol呢?**
  - 在ES6之前，对象的属性名都是字符串形式，那么很容易造成属性名的冲突；
  - 比如原来有一个对象，我们希望在其中添加一个新的属性和值，但是我们在不确定它原来内部有什么内容的情况下，很容易造成冲突，从而覆盖掉它内部的某个属性；
  - 比如我们前面在讲apply、call、bind实现时，我们有给其中添加一个fn属性，那么如果它内部原来已经有了fn属性了呢?
  - 比如开发中我们使用混入，那么混入中出现了同名的属性，必然有一个会被覆盖掉；
- Symbol就是为了解决上面的问题，用来**生成一个独一无二的值。**
  - Symbol值是通过Symbol函数来生成的，生成后可以作为属性名；
  - 也就是在ES6中，对象的属性名可以使用字符串，也可以使用Symbol值；
- **Symbol即使多次创建值，它们也是不同的：**Symbol函数执行后每次创建出来的值都是独一无二的；
- **我们也可以在创建Symbol值的时候传入一个描述description**：这个是ES2019（ES10）新增的特性；

Symbol作为属性名

- 我们通常会使用Symbol在对象中表示唯一的属性名：

```
const s1 = Symbol("abc")
const s2 = Symbol("cba")

const obj = {}

// 1.写法一：属性名赋值
obj[s1] = "abc"
obj[s2] = "cba"

// 2.写法二：Object.defineProperty
Object.defineProperty(obj, s1, {
  enumerable: true,
  configurable: true,
  writable: true,
  value: "abc"
})

// 3.写法三：定义字面量是直接使用
const info = {
  [s1]: "abc",
  [s2]: "cba"
}
```

```
console.log(Object.getOwnPropertySymbols(info))

const symbolKeys = Object.getOwnPropertySymbols(info)
for (const key of symbolKeys) {
  console.log(info[key])
}
```

相同值的Symbol

- **前面我们讲Symbol的目的是为了创建一个独一无二的值，那么如果我们现在就是想创建相同的Symbol应该怎么来做呢?**
  - 我们可以使用Symbol.for方法来做到这一点;
  - 并且我们可以通过Symbol.keyFor方法来获取对应的key;

```
const s1 = Symbol.for("abc")
const s2 = Symbol.for("abc")

console.log(s1 === s2) // true
const key = Symbol.keyFor(s1)
console.log(key) // abc
const s3 = Symbol.for(key)
console.log(s2 === s3) // true
```