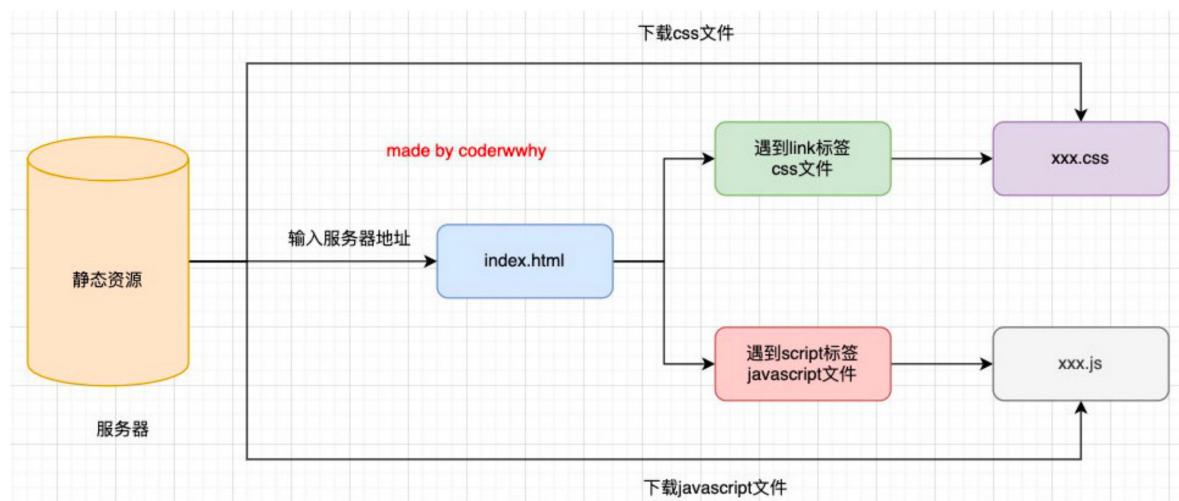


1 浏览器

浏览器工作原理



用户输入网址（域名）

dns解析域名-》 ip地址

服务器返回index.html

浏览器解析html

浏览器内核（渲染引擎）

■ 我们经常会说：不同的浏览器有不同的内核组成

- **Gecko**：早期被Netscape和Mozilla Firefox浏览器使用；
- **Trident**：微软开发，被IE4~IE11浏览器使用，但是Edge浏览器已经转向Blink；
- **Webkit**：苹果基于KHTML开发、开源的，用于Safari，Google Chrome之前也在使用；
- **Blink**：是Webkit的一个分支，Google开发，目前应用于Google Chrome、Edge、Opera等；
- 等等...

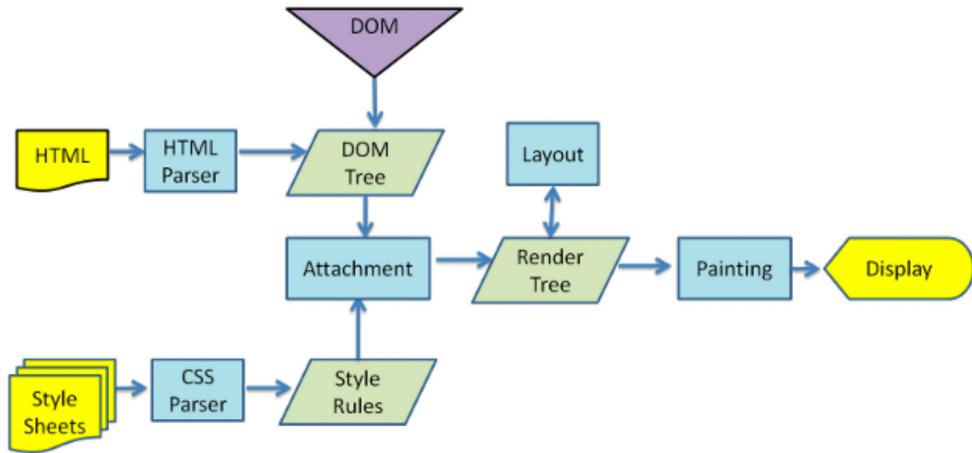
■ 事实上，我们经常说的浏览器内核指的是浏览器的排版引擎：

□ **排版引擎**（layout engine），也称为**浏览器引擎**（browser engine）、**页面渲染引擎**（rendering engine）或**样版引擎**。

浏览器渲染过程

■ 但是在这个执行过程中，HTML解析的时候遇到了JavaScript标签，应该怎么办呢？

- 会停止解析HTML，而去加载和执行JavaScript代码；



2 js引擎 (v8)

js需要转换成cpu指令，最终被cpu执行

基础知识

■ 为什么需要JavaScript引擎呢？

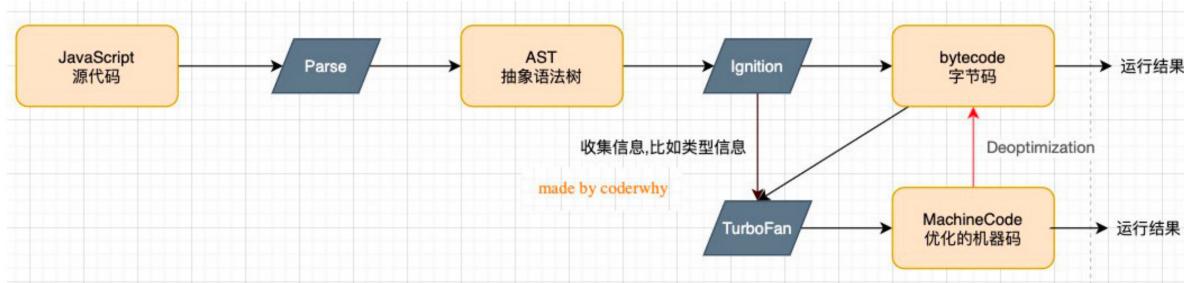
- 我们前面说过，**高级的编程语言**都是需要转成**最终的机器指令来执行的**；
- 事实上我们编写的JavaScript无论你交给**浏览器或者Node执行**，最后都是需要被**CPU执行的**；
- 但是CPU只认识自己的指令集，实际上是机器语言，才能被CPU所执行；
- 所以我们需要**JavaScript引擎**帮助我们将**JavaScript代码**翻译成**CPU指令**来执行；

■ 比较常见的JavaScript引擎有哪些呢？

- **SpiderMonkey**：第一款JavaScript引擎，由Brendan Eich开发（也就是JavaScript作者）；
- **Chakra**：微软开发，用于IE浏览器；
- **JavaScriptCore**：WebKit中的JavaScript引擎，Apple公司开发；
- **V8**：Google开发的强大JavaScript引擎，也帮助Chrome从众多浏览器中脱颖而出；

v8引擎原理

- V8是用C ++编写的Google开源高性能JavaScript和WebAssembly引擎，它用于Chrome和Node.js等。
- 它实现ECMAScript和WebAssembly，并在Windows 7或更高版本，macOS 10.12+和使用x64，IA-32，ARM或MIPS处理器的Linux系统上运行。
- V8可以独立运行，也可以嵌入到任何C ++应用程序中。



=》 parse

词法分析 (token: [{type:'keyword', value:'const'}]) =》

语法分析

=>抽象语法树

=>ignition

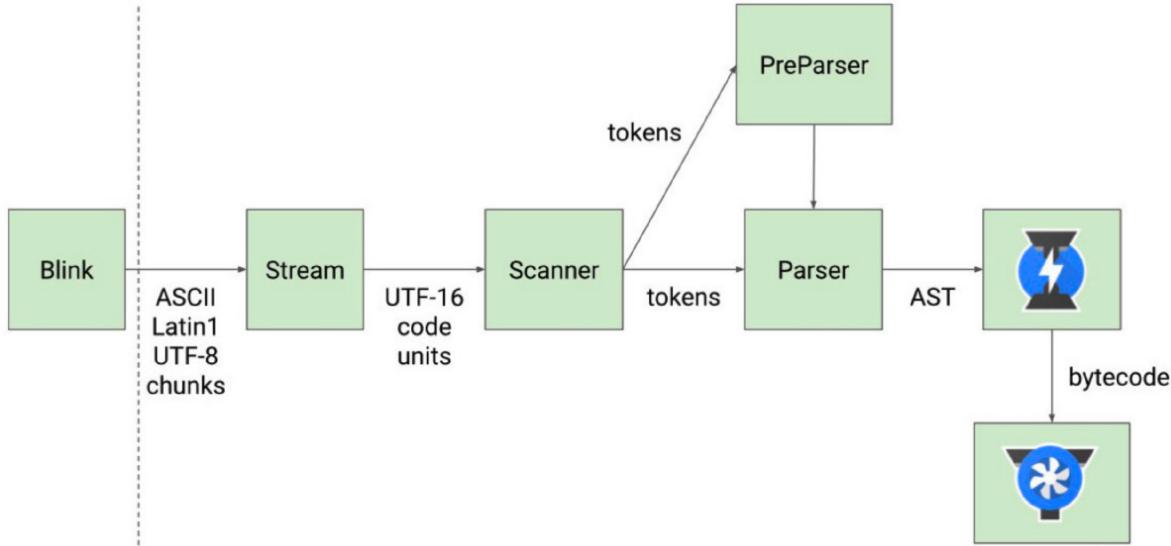
=>字节码（跨平台）

=>对应平台的机器码

deoptimization反向优化

- **Parse**模块会将JavaScript代码转换成AST（抽象语法树），这是因为解释器并不直接认识JavaScript代码；
 - 如果函数没有被调用，那么是不会被转换成AST的；
 - Parse的V8官方文档：<https://v8.dev/blog/scanner>
- **Ignition**是一个解释器，会将AST转换成ByteCode（字节码）
 - 同时会收集TurboFan优化所需要的信息（比如函数参数的类型信息，有了类型才能进行真实的运算）；
 - 如果函数只调用一次，Ignition会执行解释执行ByteCode；
 - Ignition的V8官方文档：<https://v8.dev/blog/ignition-interpreter>
- **TurboFan**是一个编译器，可以将字节码编译为CPU可以直接执行的机器码；
 - 如果一个函数被多次调用，那么就会被标记为**热点函数**，那么就会经过**TurboFan**转换成**优化的机器码**，提高代码的执行性能；
 - 但是，**机器码实际上也会被还原为ByteCode**，这是因为如果后续执行函数的过程中，**类型发生了变化**（比如**sum**函数原来执行的是**number**类型，后来执行变成了**string**类型），之前优化的机器码并不能正确的处理运算，就会逆向的转换成字节码；
 - TurboFan的V8官方文档：<https://v8.dev/blog/turbofan-jit>

v8原理图



■ 那么我们的JavaScript源码是如何被解析（Parse过程）的呢？

- Blink将源码交给V8引擎，Stream获取到源码并且进行编码转换；
- Scanner会进行词法分析（lexical analysis），词法分析会将代码转换成tokens；
- 接下来tokens会被转换成AST树，经过Parser和PreParser：
 - Parser就是直接将tokens转成AST树架构；
 - PreParser称之为预解析，为什么需要预解析呢？
 - ✓ 这是因为并不是所有的JavaScript代码，在一开始时就会被执行。那么对所有的JavaScript代码进行解析，必然会影响网页的运行效率；
 - ✓ 所以V8引擎就实现了Lazy Parsing（延迟解析）的方案，它的作用是将不必要的函数进行预解析，也就是只解析暂时需要的内容，而对函数的全量解析是在函数被调用时才会进行；
 - ✓ 比如我们在一个函数outer内部定义了另外一个函数inner，那么inner函数就会进行预解析；
- 生成AST树后，会被Ignition转成字节码（bytecode），之后的过程就是代码的执行过程（后续会详细分析）。

代码的执行过程

1. 初始化全局对象：代码被解析，v8引擎内部会帮助我们创建一个对象(GlobalObject -> go)

初始化全局对象

■ js引擎会在执行代码之前，会在堆内存中创建一个全局对象：Global Object (GO)

- 该对象 **所有的作用域（scope）** 都可以访问；
- 里面会包含**Date、Array、String、Number、setTimeout、setInterval**等等；
- 其中还有一个**window属性**指向自己；

2. 执行上下文栈：运行代码

2.1. v8为了执行代码，v8引擎内部会有一个执行上下文栈(Execution Context Stack, ECStack)(函数调用栈)

2.2. 因为我们执行的是全局代码，为了全局代码能够正常的执行，需要创建 全局执行上下文(Global Execution Context)(全局代码需要被执行时才会创建)

执行上下文栈（调用栈）

■ js引擎内部有一个**执行上下文栈（Execution Context Stack，简称ECS）**，它是用于执行代码的调用栈。

■ 那么现在它要执行谁呢？执行的是**全局的代码块**：

- 全局的代码块为了执行会构建一个 **Global Execution Context (GEC)**；
- GEC会 **被放入到ECS中** 执行；

■ GEC被放入到ECS中里面包含两部分内容：

- 第一部分：在代码执行前，在parser转成AST的过程中，会将全局定义的变量、函数等加入到**GlobalObject**中，但是**并不会赋值**；
 - ✓ 这个过程也称之为**变量的作用域提升 (hoisting)**
- 第二部分：在代码执行中，对变量赋值，或者执行其他的函数；

函数执行过程

■ 在执行的过程中执行到一个函数时，就会根据**函数体**创建一个**函数执行上下文（Functional Execution Context，简称FEC）**，并且压入到**EC Stack**中。

■ FEC中包含三部分内容：

- 第一部分：在解析函数成为AST树结构时，会创建一个Activation Object (AO)：
 - ✓ AO中包含形参、arguments、函数定义和指向函数对象、定义的变量；
- 第二部分：作用域链：由VO（在函数中就是AO对象）和父级VO组成，查找时会一层层查找；
- 第三部分：this绑定的值：这个我们后续会详细解析；

<input checked="" type="checkbox"/>	Functional Execution Context
+ VO:	形参/arguments/function/变量
+ Scope Chain:	VO/Parent VO
+ thisValue:	根据不同情况绑定this

变量查找规则

作用域链

3 js的内存管理

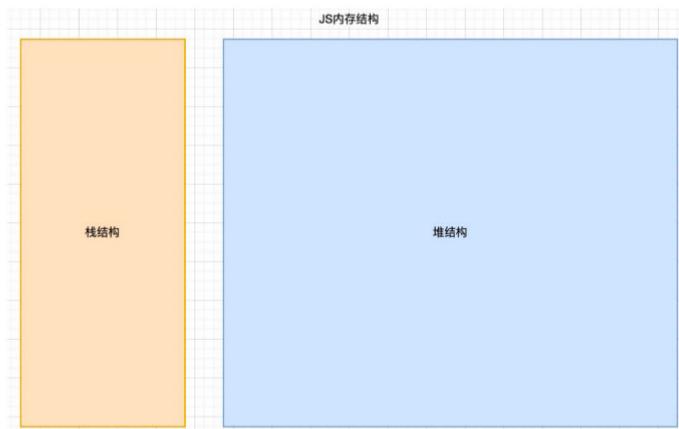
内存分配

- JavaScript会在**定义变量时**为我们分配内存。

- 但是内存分配方式是一样的吗？

- JS对于**基本数据类型内存的分配**会在执行时，直接在栈空间进行分配；

- JS对于**复杂数据类型内存的分配**会在堆内存中开辟一块空间，并且将这块空间的指针返回值变量引用；



垃圾回收

- 因为**内存的大小是有限的**，所以当**内存不再需要的时候**，我们需要对其进行释放，以便腾出**更多的内存空间**。

- 在**手动管理内存**的语言中，我们需要通过一些方式自己来释放不再需要的内存，比如**free**函数：

- 但是这种管理的方式其实**非常的低效**，影响我们**编写逻辑的代码的效率**；

- 并且这种方式对**开发者的要求也很高**，并且**一不小心就会产生内存泄露**；

- 所以大部分**现代的编程语言**都是**有自己的垃圾回收机制**：

- 垃圾回收的英文是**Garbage Collection**，简称**GC**；

- 对于**那些不再使用的对象**，我们都称之为是**垃圾**，它需要被**回收**，以释放更多的内存空间；

- 而我们的语言运行环境，比如Java的运行环境JVM，JavaScript的运行环境js引擎都会内存 **垃圾回收器**；

- **垃圾回收器**我们也会简称为**GC**，所以在很多地方你看到GC其实指的是垃圾回收器；

- 但是这里又出现了另外一个很关键的问题：**GC怎么知道哪些对象是不再使用的呢？**

- 这里就要用到**GC的算法**了

4 闭包

定义

- 这里先来看一下闭包的定义，分成两个：在计算机科学中和在JavaScript中。
- 在计算机科学中对闭包的定义（维基百科）：
 - 闭包（英语：Closure），又称词法闭包（Lexical Closure）或函数闭包（function closures）；
 - 是在支持头等函数的编程语言中，实现词法绑定的一种技术；
 - 闭包在实现上是一个结构体，它存储了一个函数和一个关联的环境（相当于一个符号查找表）；
 - 闭包跟函数最大的区别在于，当捕捉闭包的时候，它的自由变量会在补充时被确定，这样即使脱离了捕捉时的上下文，它也能照常运行；
捕捉
- 闭包的概念出现于60年代，最早实现闭包的程序是Scheme，那么我们就可以理解为什么JavaScript中有闭包：
 - 因为JavaScript中有大量的设计是来源于Scheme的；
- 我们再来看一下MDN对JavaScript闭包的解释：
 - 一个函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）；
 - 也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域；
 - 在JavaScript中，每当创建一个函数，闭包就会在函数创建的同时被创建出来；
- 那么我的理解和总结：
 - 一个普通的函数function，如果它可以访问外层作用于的自由变量，那么这个函数就是一个闭包；
 - 从广义的角度来说：JavaScript中的函数都是闭包；
 - 从狭义的角度来说：JavaScript中一个函数，如果访问了外层作用于的变量，那么它是一个闭包；

闭包的内存泄漏

- 那么我们为什么经常会说闭包是有内存泄露的呢？
 - 在上面的案例中，如果后续我们不再使用add10函数了，那么该函数对象应该要被销毁掉，并且其引用着的父作用域AO也应该被销毁掉；
 - 但是目前因为在全局作用域下add10变量对0xb00的函数对象有引用，而0xb00的作用域中AO（0x200）有引用，所以最终会造成这些内存都是无法被释放的；
 - 所以我们经常说的闭包会造成内存泄露，其实就是刚才的引用链中的所有对象都是无法释放的；
- 那么，怎么解决这个问题呢？
 - 因为当将add10设置为null时，就不再对函数对象0xb00有引用，那么对应的AO对象0x200也就不可达了；
 - 在GC的下一次检测中，它们就会被销毁掉；

```
add10 = null
```

AO不使用的属性

- 我们来研究一个问题：AO对象不会被销毁时，是否里面的所有属性都不会被释放？
 - 下面这段代码中name属于闭包的父作用域里面的变量；
 - 我们知道形成闭包之后count一定不会被销毁掉，那么name是否会被销毁掉呢？
 - 这里我打上了断点，我们可以在浏览器上看看结果；

```
function makeAdder(count) {
  let name = "why"
  return function (num) {
    debugger
    return count + num
  }
}

const add10 = makeAdder(10)
console.log(add10(5))
console.log(add10(8))
```

The screenshot shows the Chrome DevTools debugger with a stack trace for an uncaught reference error. The error message is "Uncaught ReferenceError: name is not defined" at eval (eval at <anonymous> (06_闭包的访问.js:1), <anonymous>:1:1). The stack trace shows the error occurred at 06_闭包的访问.js:4 and 06_闭包的访问.js:10.

5 this

this的指向

- 1. 函数在调用时，JavaScript会默认给this绑定一个值；
- 2. this的绑定和定义的位置（编写的位置）没有关系；
- 3. this的绑定和调用方式以及调用的位置有关系；
- 4. this是在运行时被绑定的；

绑定规则

- 绑定一：默认绑定；
- 绑定二：隐式绑定；
- 绑定三：显示绑定；
- 绑定四：new绑定；

对箭头函数无效

默认绑定：独立函数调用 fn()

隐式绑定：通过某个对象进行调用 obj.fn()

显式绑定：call() apply() bind()

new绑定：使用new关键字调用函数 new Fn()

使用new调用函数内部会执行以下操作：

1. 创建一个新的空对象
2. 新对象会被执行prototype连接
3. 新对象绑定到函数调用的this上
4. 如果函数没有返回其他对象，则返回这个对象

绑定规则优先级

其他规则

this规则之外 – 忽略显示绑定

- 我们讲到的规则已经足以应付平时的开发，但是总有一些语法，超出了我们的规则之外。（神话故事和动漫中总是有类似这样的人物）
- 如果在显示绑定中，我们传入一个null或者undefined，那么这个显示绑定会被忽略，使用默认规则：

```
function foo() {
  console.log(this);
}

var obj = {
  name: "why"
}

foo.call(obj); // obj对象
foo.call(null); // window
foo.call(undefined); // window

var bar = foo.bind(null);
bar(); // window
```

this规则之外 - 间接函数引用

- 另外一种情况，创建一个函数的 间接引用，这种情况使用默认绑定规则。

- 赋值(obj2.foo = obj1.foo)的结果是foo函数；
- foo函数被直接调用，那么是默认绑定；

```
function foo() {
  console.log(this);
}

var obj1 = {
  name: "obj1",
  foo: foo
};

var obj2 = {
  name: "obj2"
}

obj1.foo(); // obj1对象
(obj2.foo = obj1.foo)(); // window
```

箭头函数中的this

箭头函数中不绑定this，而是根据外层作用域来决定this

6

实现apply call bind

arguments

- arguments 是一个 对应于 传递给函数的参数 的 类数组(array-like)对象。

```
function foo(x, y, z) {  
    // [Arguments] { '0': 10, '1': 20, '2': 30 }  
    console.log(arguments)  
}  
  
foo(10, 20, 30)
```

- array-like意味着它不是一个数组类型，而是一个对象类型：

- 但是它却拥有数组的一些特性，比如说length，比如可以通过index索引来访问；
 - 但是它却没有数组的一些方法，比如forEach、map等；

```
console.log(arguments.length)  
console.log(arguments[0])  
console.log(arguments[1])  
console.log(arguments[2])
```

arguments转成array

```
// 1. 转化方式一:  
var length = arguments.length  
var arr = []  
for (var i = 0; i < length; i++) {  
    arr.push(arguments[i])  
}  
console.log(arr)  
  
// 2. 转化方式二  
var arr1 = Array.prototype.slice.call(arguments);  
var arr2 = [].slice.call(arguments)  
console.log(arr1)  
console.log(arr2)  
  
// 3. 转化方式三: ES6之后  
const arr3 = Array.from(arguments)  
const arr4 = [...arguments]  
console.log(arr3)  
console.log(arr4)
```

■ 箭头函数是不绑定arguments的，所以我们在箭头函数中使用arguments会去上层作用域查找

7 函数式编程(编程范式)

纯函数

- 确定的输入，一定会产生确定的输出；
- 函数在执行过程中，不能产生副作用；

■ 纯函数的维基百科定义：

- 在程序设计中，若一个函数符合以下条件，那么这个函数被称为纯函数：
 - 此函数在相同的输入值时，需产生相同的输出。
 - 函数的输出和输入值以外的其他隐藏信息或状态无关，也和由I/O设备产生的外部输出无关。
 - 该函数不能有语义上可观察的函数副作用，诸如“触发事件”，使输出设备输出，或更改输出值以外物件的内容等。

副作用

■ 那么这里又有一个概念，叫做副作用，什么又是副作用呢？

- 副作用 (side effect) 其实本身是医学的一个概念，比如我们经常说吃什么药本来是为了治病，可能会产生一些其他的副作用；
- 在计算机科学中，也引用了副作用的概念，表示在执行一个函数时，除了返回函数值之外，还对调用函数产生了附加的影响，比如修改了全局变量，修改参数或者改变外部的存储；

■ 纯函数在执行的过程中就是不能产生这样的副作用：

- 副作用往往是产生bug的“温床”。

柯里化