coderwhy
事件循环V8引擎面试题

讲解人：coderwhy

20XX-X

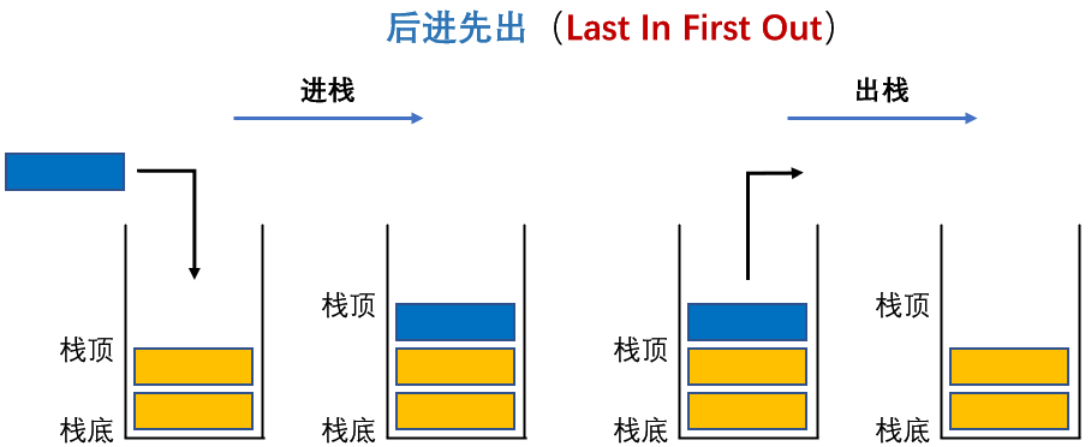
目录
content

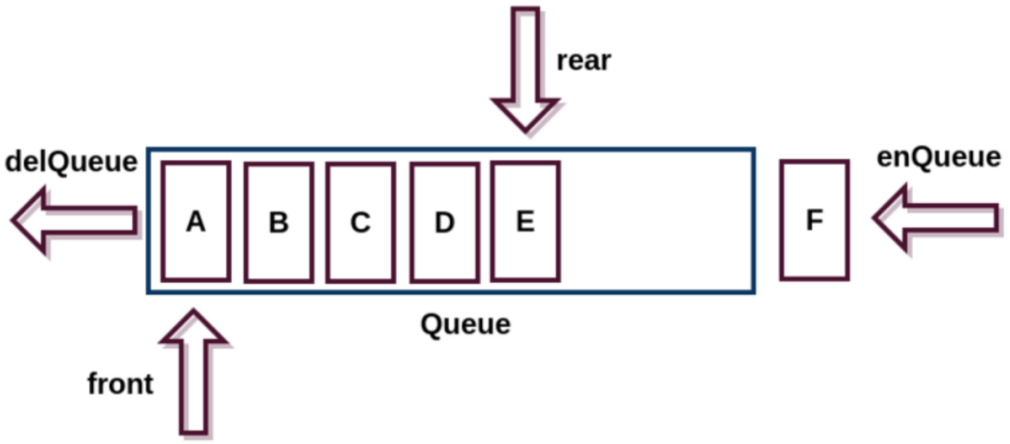
什么是浏览器的事件循环？

- **浏览器是一个进程吗，它里面只有一个线程吗？**
  - 目前多数的浏览器其实都是多进程的，当我们打开一个tab页面时就会开启一个新的进程，这是为了防止一个页面卡死而造成所有页面无法响应，整个浏览器需要强制退出；
  - 每个进程中又有很多的线程，其中包括执行JavaScript代码的线程；
- **浏览器的事件循环是一个在JavaScript引擎和渲染引擎之间协调工作的机制。**
  - 因为JavaScript是单线程的，所以所有需要被执行的操作都需要通过一定的机制来协调它们有序的进行。
  - 它的主要任务是监视调用栈（Call Stack）和任务队列（Task Queue）。
  - 当调用栈为空时，事件循环会从任务队列中取出任务执行。
- **JavaScript的代码执行是在一个单独的线程中执行的：**
  - 这就意味着JavaScript的代码，在同一个时刻只能做一件事；
  - 如果这件事是非常耗时的，就意味着当前的线程就会被阻塞；
- **所以真正耗时的操作，实际上并不是由JavaScript线程在执行的：**
  - 浏览器的每个进程是多线程的，那么其他线程可以来完成这个耗时的操作；
  - 比如网络请求、定时器，我们只需要在特定的时候，去执行应该有的回调函数即可；

栈结构和队列结构

- **调用栈（Call Stack）**
  - JavaScript是单线程的，调用栈是一个后进先出（LIFO）的数据结构，用于存储在程序执行过程中创建的所有执行上下文（Execution Contexts）。
  - 每当函数被调用时，它的执行上下文就会被推入栈中。函数执行完毕后，其上下文会从栈中弹出。
- **任务队列（Task Queue）**
  - 任务队列是一种先进先出（FIFO）的数据结构，用于存储待处理的事件。
  - 这些事件可能包括用户交互事件（如点击、滚动等）、网络请求完成、定时器到期等。

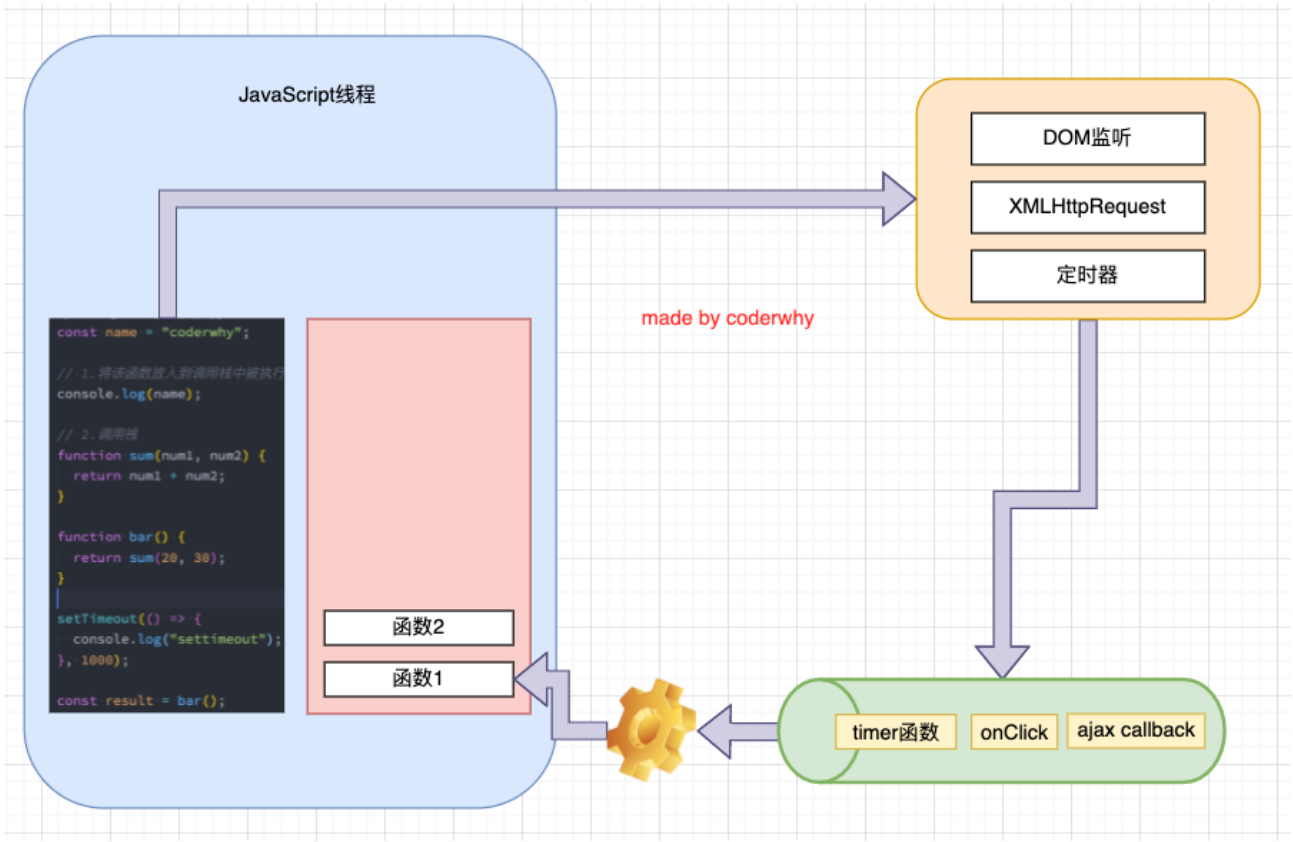




浏览器的事件循环

- **如果在执行JavaScript代码的过程中，有异步操作呢？**
  - 中间我们插入了一个setTimeout的函数调用；
  - 这个函数被放到入调用栈中，执行会立即结束，并不会阻塞后续代码的执行；

```
function sum(num1, num2) {
  return num1 + num2;
}

function bar() {
  return sum(20, 30);
}

setTimeout(() => {
  console.log("settimeout");
}, 1000);

const result = bar();

console.log(result);
```



事件循环的执行步骤

- **浏览器的事件循环通过以下步骤处理异步操作：**
- **执行全局脚本**：加载页面时，浏览器会首先执行全局脚本。
- **宏任务和微任务**：
  - **宏任务**（MacroTasks）：包括脚本（script）、setTimeout、setInterval、I/O、UI rendering 等。
  - **微任务**（MicroTasks）：包括 Promise.then、MutationObserver、process.nextTick（仅在Node.js中）等。
- **事件循环的周期**：
  - 执行当前宏任务。
  - 执行完当前宏任务后，检查并执行所有微任务。
    - ✓ 在微任务执行期间产生的新的微任务也会被连续执行，直到微任务队列清空。
  - 渲染更新界面（如果有必要）。
  - 请求下一个宏任务，重复上述过程。

02-什么是宏任务，什么是微任务？并解释一下它们在事件循环中的角色和区别？

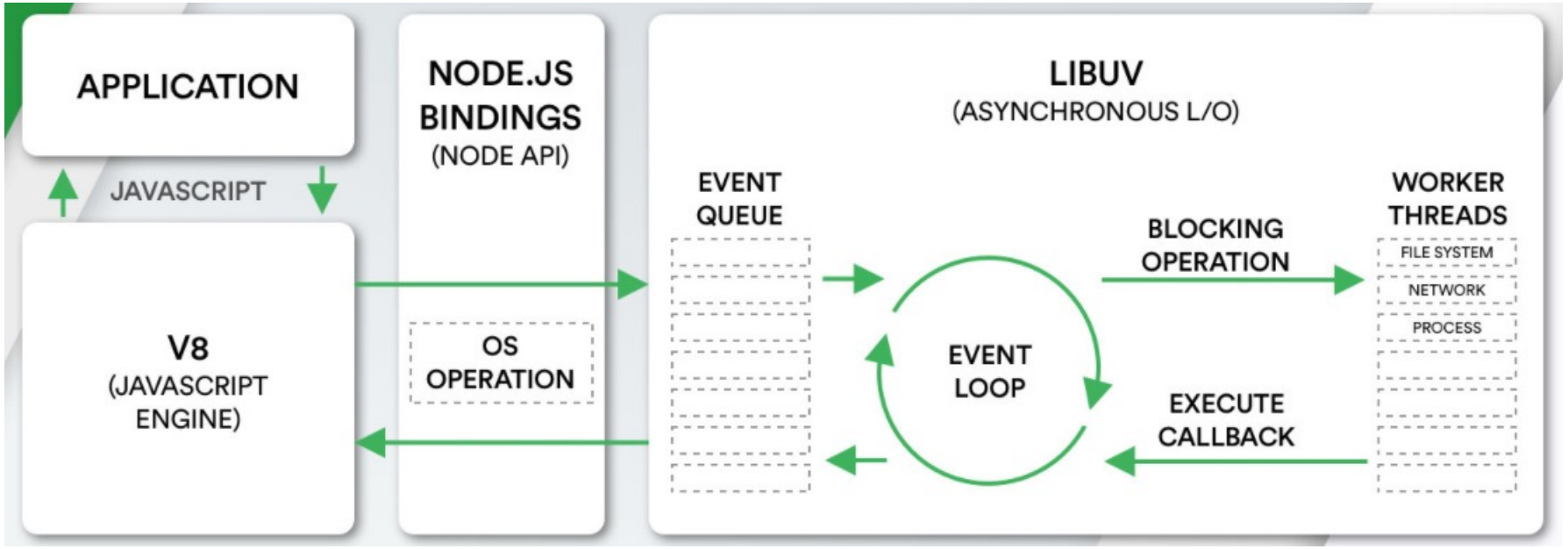
- **在浏览器的事件循环中，宏任务和微任务是两类不同的任务，它们在异步操作处理上具有不同的优先级和执行时机。**
- **宏任务（MacroTasks）**
  - 宏任务是一个比较大的任务单位，可以看作是一个独立的工作单元。
  - 当一个宏任务执行完毕后，浏览器可以在两个宏任务之间进行页面渲染或处理其他事务（比如执行微任务）。
- **常见的宏任务包括**：
  - 完整的脚本（如一个<script>标签）
  - setTimeout
  - setInterval
  - I/O操作（浏览器中的Ajax、Fetch，Node中的文件系统、网络请求、数据库交互）
  - UI交互事件
  - setImmediate（在Node.js中）
  - 等等...

02-什么是宏任务，什么是微任务？并解释一下它们在事件循环中的角色和区别？

- **微任务通常是在当前宏任务完成后立即执行的小任务，它们的执行优先级高于宏任务。**
  - 微任务的执行会在下一个宏任务开始前完成，即在当前宏任务和下一个宏任务之间。
- **常见的微任务包括**：
  - Promise.then（Promise的回调）、Promise.catch和Promise.finally
  - MutationObserver（监视DOM变更的API，在Vue2源码中也有使用它来实现微任务的调度）
  - process.nextTick（仅在Node.js中，待会儿Node事件循环详细讲解）
  - queueMicrotask（显示创建微任务的API）
- **宏任务和微任务区别**
  - **执行顺序**：
    - ✓ 事件循环在执行宏任务队列中的一个宏任务后，会查看微任务队列。如果微任务队列中有任务，事件循环会连续执行所有微任务直到微任务队列为空。
    - ✓ 宏任务的执行可能触发更多的微任务，而这些微任务会在任何新的宏任务之前执行，确保微任务可以快速执行。
  - **用途不同**：
    - ✓ 由于微任务具有较高的执行优先级，它们适合用于需要尽快执行的小任务，例如处理异步的状态更新。
    - ✓ 宏任务适合用于分割较大的、需要较长时间执行的任务，以避免阻塞UI更新或其他高优先级的操作。

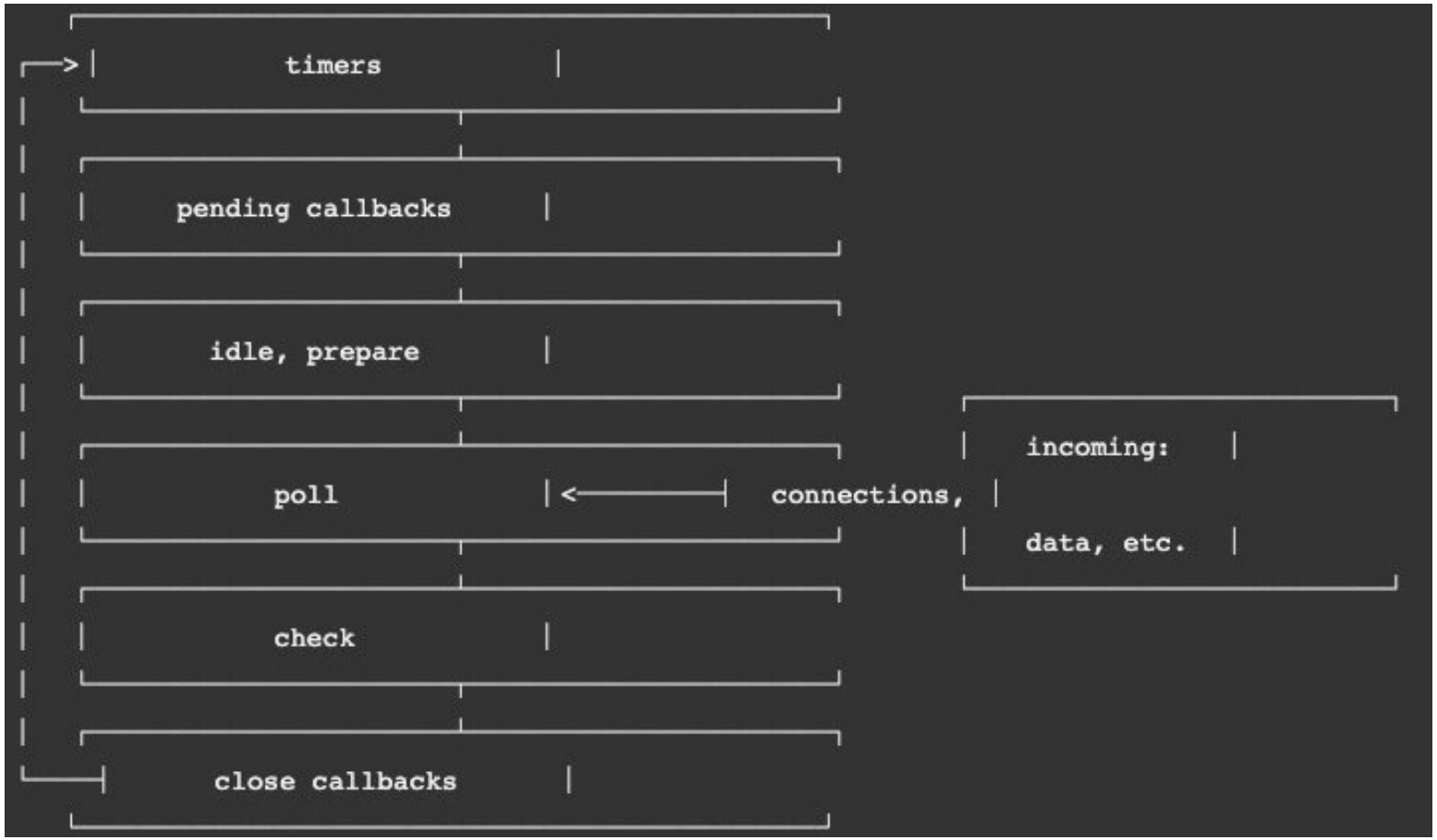
Node事件循环介绍

- **浏览器中的EventLoop是根据HTML5定义的规范来实现的，不同的浏览器可能会有不同的实现。**
- **而Node中的事件循环是由是由libuv实现的，这是一个处理异步事件的C库。**
  - libuv是一个多平台的专注于异步IO的库，它最初是为Node开发的，但是现在也被使用到Luvit、Julia、pyuv等其他地方；
- **这里我们来给出一个Node的架构图：**
  - 我们会发现libuv中主要维护了一个EventLoop和worker threads（线程池）；
  - EventLoop负责调用系统的一些其他操作：文件的IO、Network、child-processes等



Node事件循环阶段

- Node.js的事件循环包含几个主要阶段，每个阶段都有自己的特定类型的任务：
- 我们来看一下官方给出的图片：



每个阶段的详细解析

- **对每个阶段进行详细的解释：**
- **timers（定时器）**：这一阶段执行setTimeout和setInterval的回调函数。
- **Pending Callbacks（待定回调）**：executes I/O callbacks deferred to the next loop iteration（官方的解释）
  - 这意味着在这个阶段，Node处理一些上一轮循环中未完成的I/O任务。
  - 具体来说，这些是一些被推迟到下一个事件循环迭代的回调，通常是由于某些操作无法在它们被调度的那一轮事件循环中完成。
  - 比如操作系统在连接TCP时，接收到ECONNREFUSED（连接被拒绝）。
- **idle, prepare（空闲、准备）**：只用于系统内部调用。
- **poll（轮询）：**检索新的 I/O 事件；执行与 I/O 相关的回调。
  - **检索新的I/O事件**：这一部分，libuv负责检查是否有I/O操作（如文件读写、网络通信）完成，并准备好了相应的回调函数。
  - **执行I/O相关的回调**：几乎所有类型的I/O回调都会在这里执行，除了那些特别由timers和setImmediate安排的回调以及某些关闭回调（close callbacks）。
- **check（检查）**：setImmediate() 的回调在这个阶段执行。
- **close callbacks（关闭回调）**：如 socket.on('close', ...) 这样的回调在这里执行。

Node宏任务和微任务

- **我们会发现从一次事件循环的Tick来说，Node的事件循环更复杂，它也分为微任务和宏任务：**
  - 宏任务（macrotask）：setTimeout、setInterval、IO事件、setImmediate、close事件；
  - 微任务（microtask）：Promise的then回调、process.nextTick、queueMicrotask；
- **但是，Node中的事件循环中微任务队列划分的会更加精细：**
  - next tick queue：process.nextTick；
  - other queue：Promise的then回调、queueMicrotask；
- **那么它们的整体执行时机是怎么样的呢？**
- **1.调用栈执行**：Node.js 首先执行全局脚本或模块中的同步代码。
  - 这些代码在调用栈中执行，直到栈被清空。
- **2.处理 process.nextTick() 队列**：一旦调用栈为空，Node.js 会首先处理 process.nextTick() 队列中的所有回调。
  - 这确保了任何在同步执行期间通过 process.nextTick() 安排的回调都将在进入任何其他阶段之前执行。
- **3.处理其他微任务**：处理完 process.nextTick() 队列后，Node.js 会处理 Promise 等微任务队列。
  - 这些微任务包括由 Promise.then()、Promise.catch() 或 Promise.finally() 等安排的回调。

开始事件循环的各个阶段

- **4.开始事件循环的各个阶段**：
  - **timers阶段**：处理 setTimeout() 和 setInterval() 回调。
  - **I/O 回调阶段**：处理大多数类型的I/O相关回调。
  - **poll阶段**：等待新的I/O事件，处理poll队列中的事件。
  - **check阶段**：处理 setImmediate() 回调。
  - **close回调阶段**：处理如 socket.on('close', ...) 的回调。

- **这里有一个特别的Node处理：微任务在事件循环过程中的处理**
  - 在事件循环的任何阶段之间，以及在上述每个阶段内部的任何单个任务后。
  - Node.js 会再次处理 process.nextTick() 队列和 Promise 微任务队列。
  - 这确保了在事件循环的任何时刻，微任务都可以优先和迅速的被处理。

04-描述process.nextTick在Node.js中事件循环的执行顺序，以及其与微任务的关系。

- **在 Node.js 中，process.nextTick() 是一个在事件循环的各个阶段/各个宏任务之间允许开发者插入操作的功能：**
  - 其特点是具有极高的优先级，可以在当前操作完成后、任何进一步的I/O事件（包括由事件循环管理的其他微任务）处理之前执行。
- **process.nextTick() 的执行顺序：**

1. **调用栈清空**：Node.js 首先执行完当前的调用栈中的所有同步代码。
2. **执行 process.nextTick() 队列**：一旦调用栈为空，Node.js 会检查 process.nextTick() 队列。
   - 如果队列中有任务，Node.js 会执行这些任务，即使在当前事件循环的迭代中有其他微任务或宏任务排队等待。
3. **处理其他微任务**：在 process.nextTick() 队列清空之后，Node.js 会处理由 Promises 等产生的微任务队列。
4. **继续事件循环**：处理完所有微任务后，Node.js 会继续进行到事件循环的下一个阶段或者下一个任务（例如 timers、I/O callbacks、poll 等）。

- **与微任务的关系**
  - **优先级**：process.nextTick() 创建的任务和 Promise 优先级是不同的，但它们也是一种微任务，并且在所有微任务中具有最高的执行优先级。这意味着 process.nextTick() 的回调总是在其他微任务（例如 Promise 回调）之前执行。
  - **微任务队列**：在任何事件循环阶段或宏任务之间，以及在宏任务内部可能触发的任何点，Node.js 都可能执行 process.nextTick()。执行完这些任务后，才会处理 Promise 微任务队列。
- **题外话：process.nextTick() 的命名在 Node.js 社区中曾经引起过一些讨论，因为它可能会导致一些误解。**

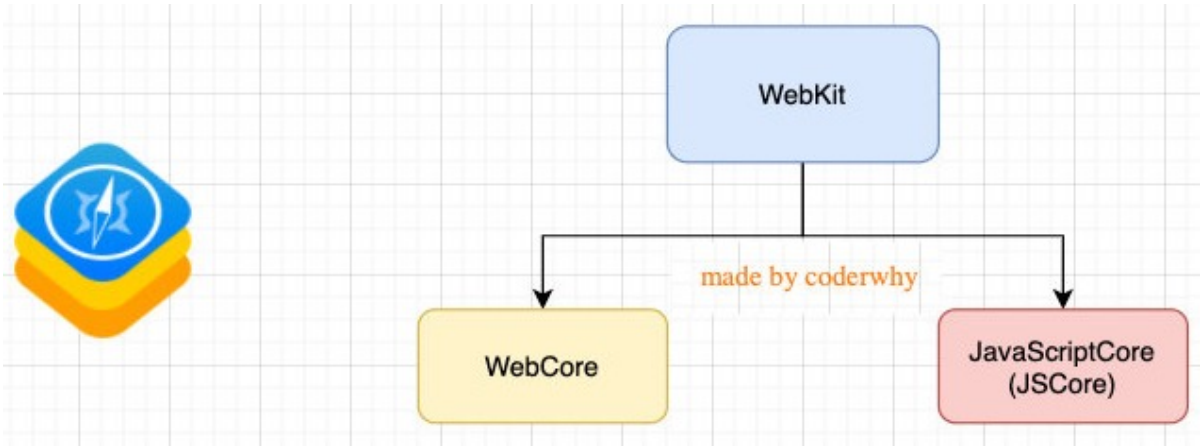
代码测试

- 代码的测试：

```
console.log('Start of script');

setTimeout(() => {
  console.log('First setTimeout');

  queueMicrotask(() => {
    console.log("queueMicrotask execution")
  })

  process.nextTick(() => {
    console.log('nextTick execution');
  });
}, 0);

setTimeout(() => {
  console.log('Second setTimeout');
}, 0);

console.log('End of script');
```

JavaScript引擎

- **JavaScript代码下载好之后，是如何一步步被执行的呢？**
- **我们知道，浏览器内核主要是由两部分组成的，以webkit为例：**
  - WebCore：负责HTML解析、布局、渲染等等相关的工作；
  - JavaScriptCore：解析、执行JavaScript代码；
- Webkit源码地址：https://github.com/WebKit/WebKit

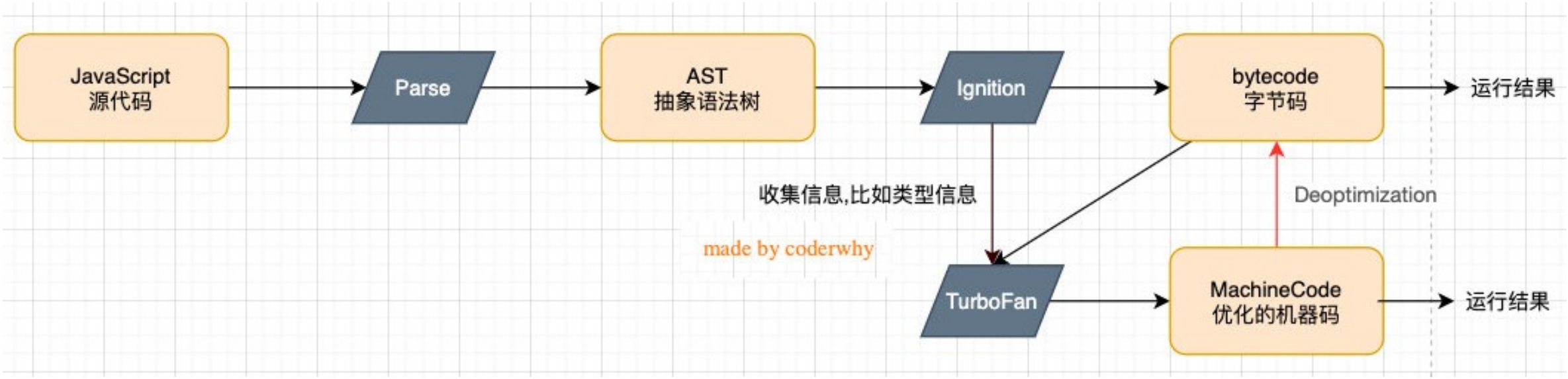


- **JavaScript引擎非常多，我们来介绍几个比较重要的：**
  - SpiderMonkey：第一款JavaScript引擎，由Brendan Eich开发（也就是JavaScript作者），在1996年发布；
  - Chakra：Chakra 最初是 Internet Explorer 9 的 JavaScript 引擎，并在后续成为了 Edge 浏览器的引擎，直到 Microsoft 转向 Chromium 架构并采用了 V8。
  - JavaScriptCore：JavaScriptCore 是 WebKit 浏览器引擎的一部分，主要用于 Apple 的 Safari 浏览器，它也被用在所有 iOS 设备的应用中。
  - V8引擎：V8 是 Chrome 浏览器和 Node.js 的 JavaScript 引擎，也是我们后续讲解的重点。

- 接下来我们以V8引擎为例，来讲解一下JavaScript代码具体的执行过程。

V8引擎的执行原理

- **我们来看一下官方对V8引擎的定义：**
  - 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。



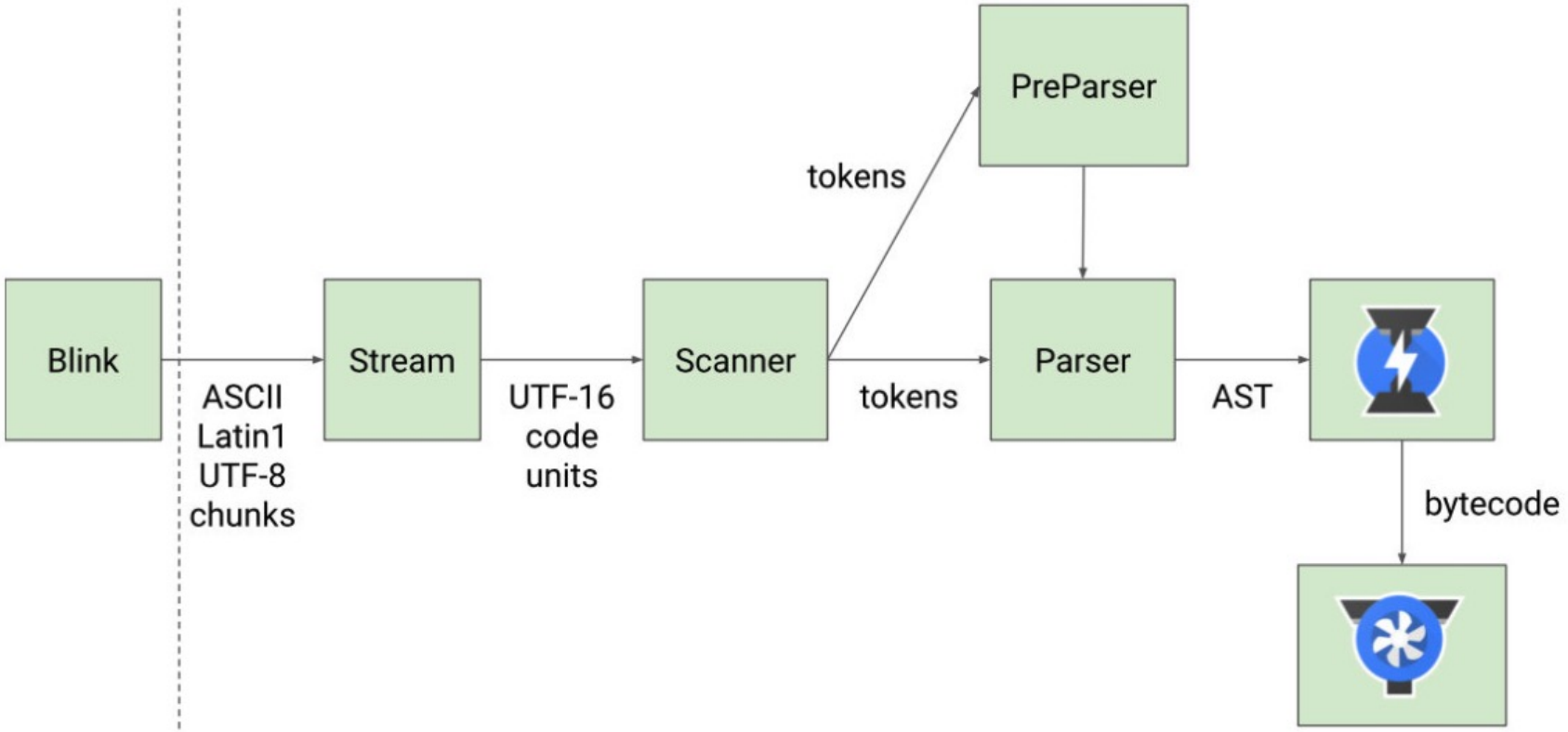
图例的详细解析

- **解析 (Parse)**：JavaScript 代码首先被解析器处理，转化为抽象语法树（AST）。
  - 这是代码编译的初步阶段，主要转换代码结构为V8内部可进一步处理的格式。
- **AST**：抽象语法树（AST）是源代码的树形表示，用于表示程序结构。之后，AST 会被进一步编译成字节码。
- **Ignition**：Ignition 是 V8 的解释器，它将 AST 转换为字节码。
  - 字节码是一种低级的、比机器码更抽象的代码，它可以快速执行，但比直接的机器码慢。
- **字节码**（Bytecode）：字节码是介于源代码和机器码之间的中间表示，它为后续的优化和执行提供了一种更标准化的形式。
  - 字节码是由 Ignition 生成，可被直接解释执行，同时也是优化编译器 TurboFan 的输入。
- **TurboFan**：TurboFan 是 V8 的优化编译器，它接收从 Ignition 生成的字节码并进行进一步优化。
  - 比如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。
  - 当然还会包括很多其他的优化手段，如死代码消除（Dead Code Elimination）等，总是V8有很多手段可以提高代码执行效率。
- **机器码**：经过 TurboFan 处理后，字节码被编译成机器码，即直接运行在计算机硬件上的低级代码。
  - 这一步是将 JavaScript 代码转换成 CPU 可直接执行的指令，大大提高了执行速度。
- **运行时优化**：在代码执行过程中，V8 引擎会持续监控代码的执行情况。
  - 如果发现之前做的优化不再有效或者有更优的执行路径，它会触发去优化（Deoptimization）。
  - 去优化是指将已优化的代码退回到优化较少的字节码状态，然后重新编译以适应新的运行情况。

V8三大模块的核心分析（GC垃圾回收后续讲解）

- **V8引擎本身的源码非常复杂，大概有超过100w行C++代码，通过了解它的架构，我们可以知道它是如何对JavaScript执行的：**
- **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；**
  - 将代码转化成AST树是一个非常常见的操作，比如在Babel、Vue源码中都需要进行这样的操作；
  - 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；**
  - 比如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去优化为ByteCode（ Deoptimization ），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，就会去优化的转换成字节码；
  - TurboFan的V8官方文档：https://v8.dev/blog/turbofan-jit

V8引擎的解析图（官方）

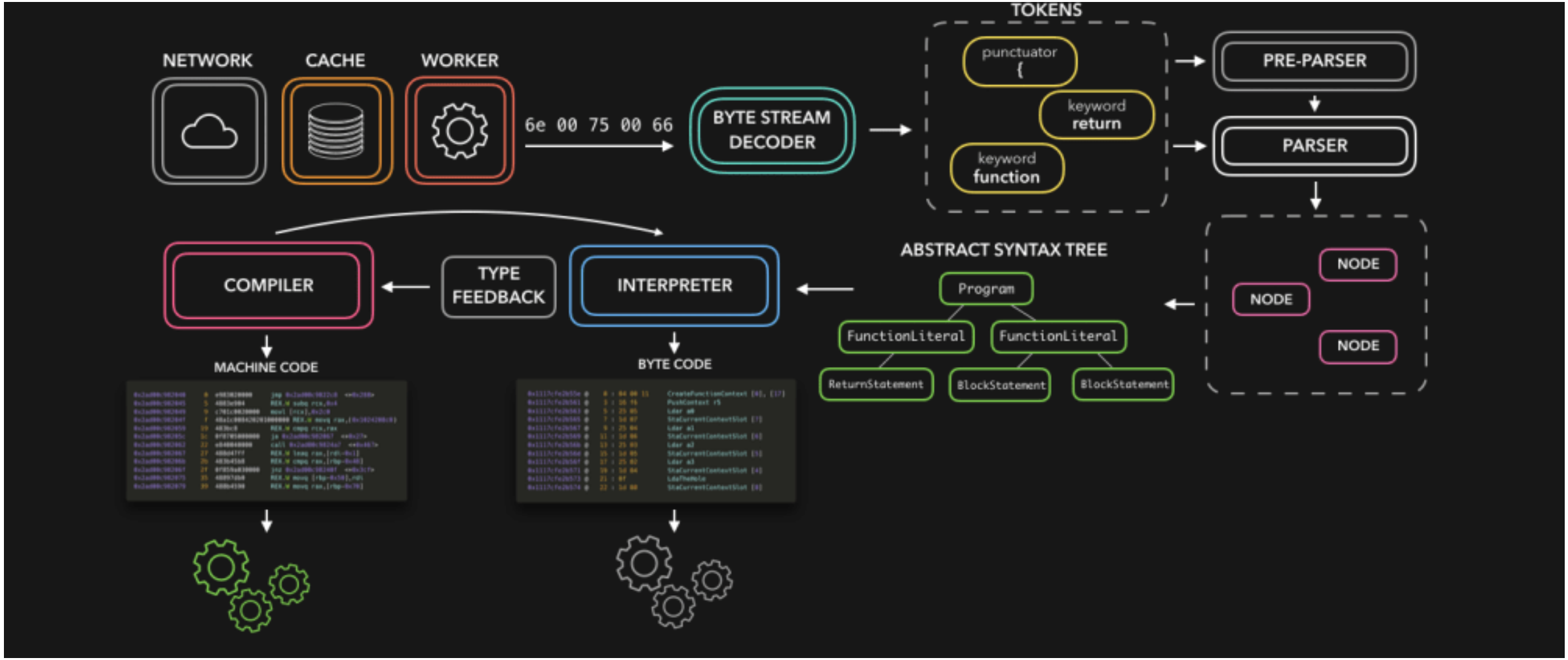


- **词法分析（英文lexical analysis）**
  - 将字符序列转换成token序列的过程。
  - token是**记号化**（tokenization）的缩写
  - **词法分析器**（lexical analyzer，简称lexer），也叫**扫描器**（scanner）
- **语法分析（英语：syntactic analysis，也叫 parsing）**
  - **语法分析器也可以称之为**parser。

V8执行的细节分析

- **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树结构；
    - ✓ PreParser称之为预解析；
- **为什么需要预解析PreParser呢？**
  - 预解析一方面的作用是快速检查一下是否有语法错误，另一方面也可以进行代码优化。
  - 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）并且可以执行字节码，之后的过程就是代码的执行过程（后续会详细分析）。

V8引擎的解析图


NETWORK
CACHE
WORKER
6e 00 75 00 66
BYTE STREAM DECODER
TOKENS
punctuator
{
keyword
return
keyword
function
PRE-PARSER
PARSER
NODE
NODE
NODE
ABSTRACT SYNTAX TREE
Program
FunctionLiteral
FunctionLiteral
ReturnStatement
BlockStatement
BlockStatement
COMPILER
TYPE FEEDBACK
INTERPRETER
MACHINE CODE
BYTE CODE