

## 14 | 编译器和解释器：V8是如何执行一段JavaScript代码的？

2019-09-05 李兵

《浏览器工作原理与实践》

课程介绍 >



讲述：李兵

时长 12:48 大小 11.73M



前面我们已经花了很多篇幅来介绍 JavaScript 是如何工作的，了解这些内容能帮助你从底层理解 JavaScript 的工作机制，从而能帮助你更好地理解和应用 JavaScript。

今天这篇文章我们就继续“向下”分析，站在 JavaScript 引擎 V8 的视角，来分析 JavaScript 代码是如何被执行的。

前端工具和框架的自身更新速度非常快，而且还不断有新的出现。要想追赶上前端工具和框架的更新速度，你就需要抓住那些本质的知识，然后才能更加轻松地理解这些上层应用。比如我们接下来要介绍的 V8 执行机制，能帮助你从底层了解 JavaScript，也能帮助你深入理解语言转换器 Babel、语法检查工具 ESLint、前端框架 Vue 和 React 的一些底层实现机制。因此，了解 V8 的编译流程能让你对语言以及相关工具有更加充分的认识。



要深入理解 V8 的工作原理，你需要搞清楚一些概念和原理，比如接下来我们要详细讲解的**编译器（Compiler）、解释器（Interpreter）、抽象语法树（AST）、字节码（Bytecode）、**

即时编译器（JIT）等概念，都是你需要重点关注的。

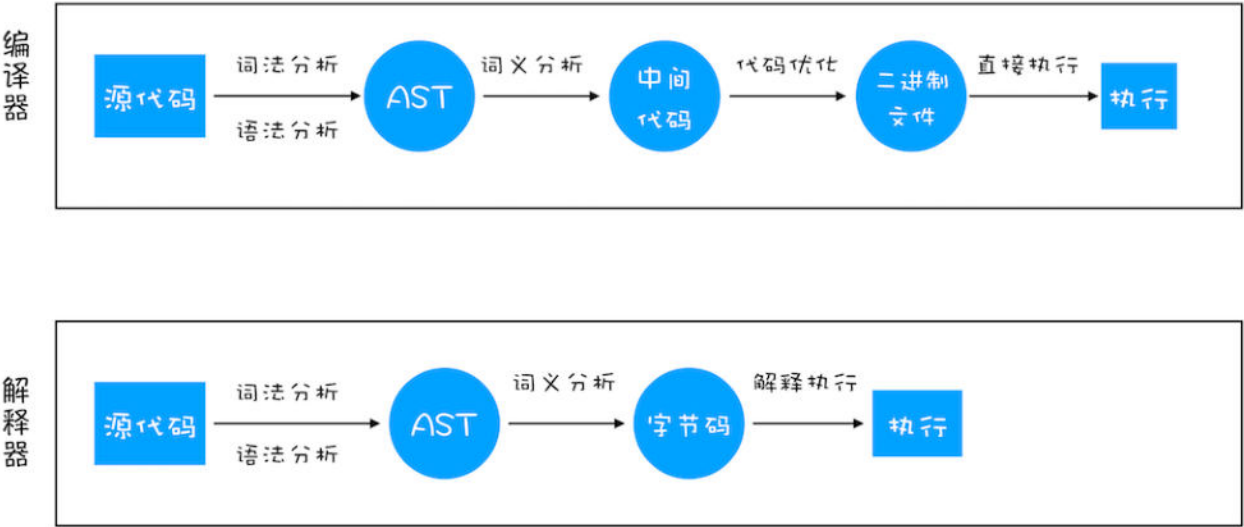
## 编译器和解释器

之所以存在编译器和解释器，是因为机器不能直接理解我们所写的代码，所以在执行程序之前，需要将我们所写的代码“翻译”成机器能读懂的机器语言。按语言的执行流程，可以把语言划分为编译型语言和解释型语言。

编译型语言在程序执行之前，需要经过编译器的编译过程，并且编译之后会直接保留机器能读懂的二进制文件，这样每次运行程序时，都可以直接运行该二进制文件，而不需要再次重新编译了。比如 C/C++、GO 等都是编译型语言。

而由解释型语言编写的程序，在每次运行时都需要通过解释器对程序进行动态解释和执行。比如 Python、JavaScript 等都属于解释型语言。

那编译器和解释器是如何“翻译”代码的呢？具体流程你可以参考下图：



编译器和解释器“翻译”代码

从图中你可以看出这二者的执行流程，大致可阐述为如下：

1. 在编译型语言的编译过程中，编译器首先会依次对源代码进行词法分析、语法分析，生成抽象语法树（AST），然后是优化代码，最后再生成处理器能够理解的机器码。如果编译成

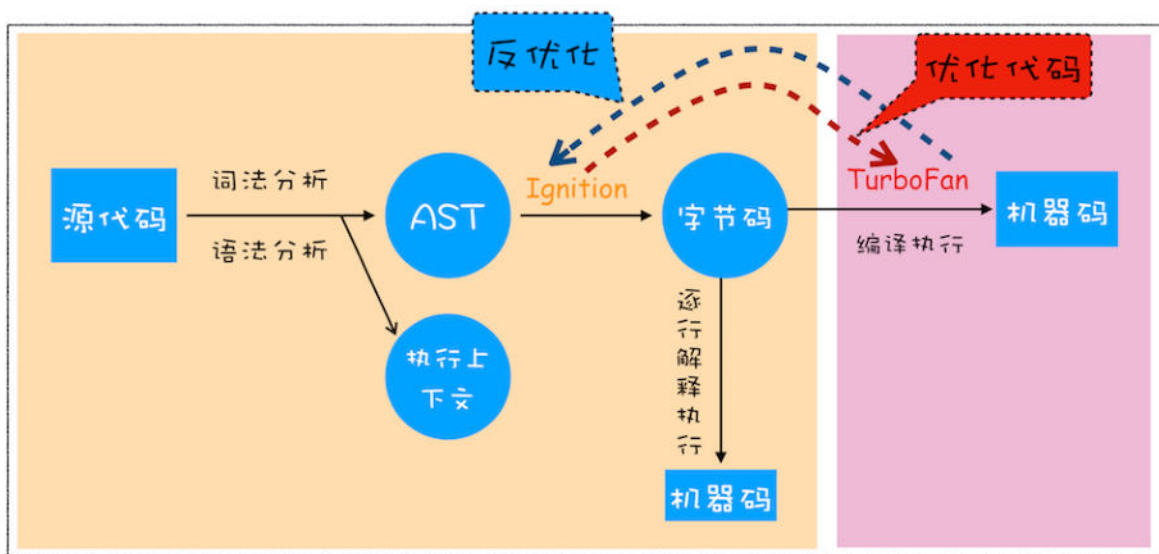


功，将会生成一个可执行的文件。但如果编译过程发生了语法或者其他的错误，那么编译器就会抛出异常，最后的二进制文件也不会生成成功。

2. 在解释型语言的解释过程中，同样解释器也会对源代码进行词法分析、语法分析，并生成抽象语法树（AST），不过它会再基于抽象语法树生成字节码，最后再根据字节码来执行程序、输出结果。

## V8 是如何执行一段 JavaScript 代码的

通过上面的介绍，相信你已经了解编译器和解释器了。那接下来，我们就重点分析下 V8 是如何执行一段 JavaScript 代码的。你可以先来“一览全局”，参考下图：



V8 执行一段代码流程图

从图中可以清楚地看到，V8 在执行过程中既有**解释器 Ignition**，又有**编译器 TurboFan**，那么它们是如何配合去执行一段 JavaScript 代码的呢？下面我们就按照上图来一一分解其执行流程。

### 1. 生成抽象语法树（AST）和执行上下文


将源代码转换为**抽象语法树**，并生成**执行上下文**，而执行上下文我们在前面的文章中已经介绍过很多了，主要是代码在执行过程中的环境信息。

那么下面我们就得重点讲解下抽象语法树（下面表述中就直接用它的简称 AST 了），看看什么是 AST 以及 AST 的生成过程是怎样的。



高级语言是开发者可以理解的语言，但是让编译器或者解释器来理解就非常困难了。对于编译器或者解释器来说，它们可以理解的就是 AST 了。所以无论你使用的是解释型语言还是编译型语言，在编译过程中，它们都会生成一个 AST。这和渲染引擎将 HTML 格式文件转换为计算机可以理解的 DOM 树的情况类似。

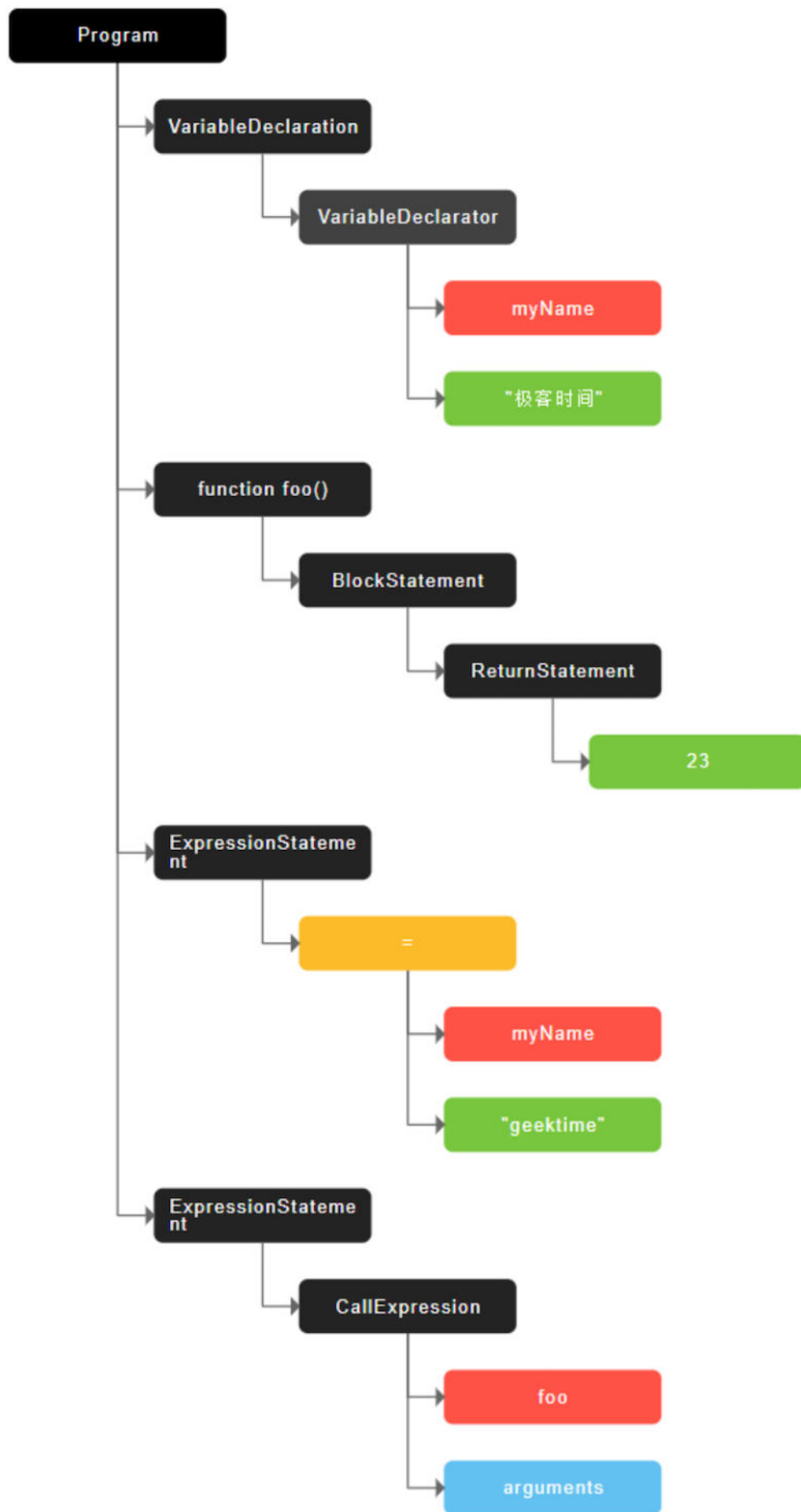
你可以结合下面这段代码来直观地感受下什么是 AST：

 复制代码

```
1 var myName = "极客时间"
2 function foo(){
3   return 23;
4 }
5 myName = "geektime"
6 foo()
```

这段代码经过 [🔗 javascript-ast](#) 站点处理后，生成的 AST 结构如下：





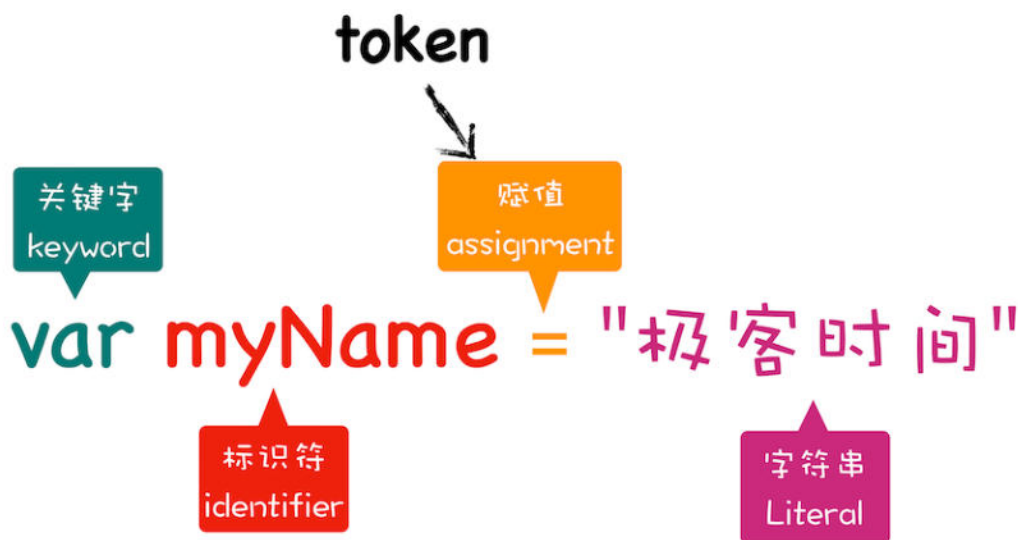
从图中可以看出，AST 的结构和代码的结构非常相似，其实你也可以把 AST 看成代码的结构化的表示，编译器或者解释器后续的工作都需要依赖于 AST，而不是源代码。

AST 是非常重要的—种数据结构，在很多项目中有着广泛的应用。其中最著名的一个项目是 Babel。Babel 是一个被广泛使用的代码转码器，可以将 ES6 代码转为 ES5 代码，这意味着你可以现在就用 ES6 编写程序，而不用担心现有环境是否支持 ES6。Babel 的工作原理就是先将 ES6 源码转换为 AST，然后再将 ES6 语法的 AST 转换为 ES5 语法的 AST，最后利用 ES5 的 AST 生成 JavaScript 源代码。

除了 Babel 外，还有 ESLint 也使用 AST。ESLint 是一个用来检查 JavaScript 编写规范的插件，其检测流程也是需要将源码转换为 AST，然后再利用 AST 来检查代码规范化的问题。

现在你知道了什么是 AST 以及它的一些应用，那接下来我们再来看下 AST 是如何生成的。通常，生成 AST 需要经过两个阶段。

**第一阶段是分词 (tokenize)**，又称为词法分析，其作用是将一行行的源码拆解成一个个 token。所谓 **token**，指的是语法上不可能再分的、最小的单个字符或字符串。你可以参考下图来更好地理解什么 token。



分解 token 示意图





从图中可以看出，通过 `var myName = “极客时间”` 简单地定义了一个变量，其中关键字“var”、标识符“myName”、赋值运算符“=”、字符串“极客时间”四个都是 token，而且它们代表的属性还不一样。

**第二阶段是解析（parse），又称为语法分析**，其作用是将上一步生成的 token 数据，根据语法规则转为 AST。如果源码符合语法规则，这一步就会顺利完成。但如果源码存在语法错误，这一步就会终止，并抛出一个“语法错误”。

这就是 AST 的生成过程，先分词，再解析。

有了 AST 后，那接下来 V8 就会生成该段代码的执行上下文。至于执行上下文的具体内容，你可以参考前面几篇文章的讲解。

## 2. 生成字节码

有了 AST 和执行上下文后，那接下来的第二步，解释器 Ignition 就登场了，它会根据 AST 生成字节码，并解释执行字节码。

其实一开始 V8 并没有字节码，而是直接将 AST 转换为机器码，由于执行机器码的效率是非常高效的，所以这种方式在发布后的一段时间内运行效果是非常好的。但是随着 Chrome 在手机上的广泛普及，特别是运行在 512M 内存的手机上，内存占用问题也暴露出来了，因为 V8 需要消耗大量的内存来存放转换后的机器码。为了解决内存占用问题，V8 团队大幅重构了引擎架构，引入字节码，并且抛弃了之前的编译器，最终花了将近四年的时间，实现了现在的这套架构。

那什么是字节码呢？为什么引入字节码就能解决内存占用问题呢？

**字节码就是介于 AST 和机器码之间的一种代码。但是与特定类型的机器码无关，字节码需要通过解释器将其转换为机器码后才能执行。**

理解了什么是字节码，我们再来对比下高级代码、字节码和机器码，你可以参考下图：





字节码和机器码占用空间对比

从图中可以看出，机器码所占用的空间远远超过了字节码，所以使用字节码可以减少系统的内存使用。

### 3. 执行代码

生成字节码之后，接下来就要进入执行阶段了。

通常，如果有一段第一次执行的字节码，解释器 Ignition 会逐条解释执行。到了这里，相信你已经发现了，解释器 Ignition 除了负责生成字节码之外，它还有另外一个作用，就是解释执行字节码。在 Ignition 执行字节码的过程中，如果发现有热点代码（HotSpot），比如一段代码被重复执行多次，这种就称为**热点代码**，那么后台的编译器 TurboFan 就会把该段热点的字节码编译为高效的机器码，然后当再次执行这段被优化的代码时，只需要执行编译后的机器码就可以了，这样就大大提升了代码的执行效率。

V8 的解释器和编译器的取名也很有意思。解释器 Ignition 是点火器的意思，编译器 TurboFan 是涡轮增压的意思，寓意着代码启动时通过点火器慢慢发动，一旦启动，涡轮增压介入，其执行效率随着执行时间越来越高效率，因为热点代码都被编译器 TurboFan 转换了机器码，直接执行机器码就省去了字节码“翻译”为机器码的过程。

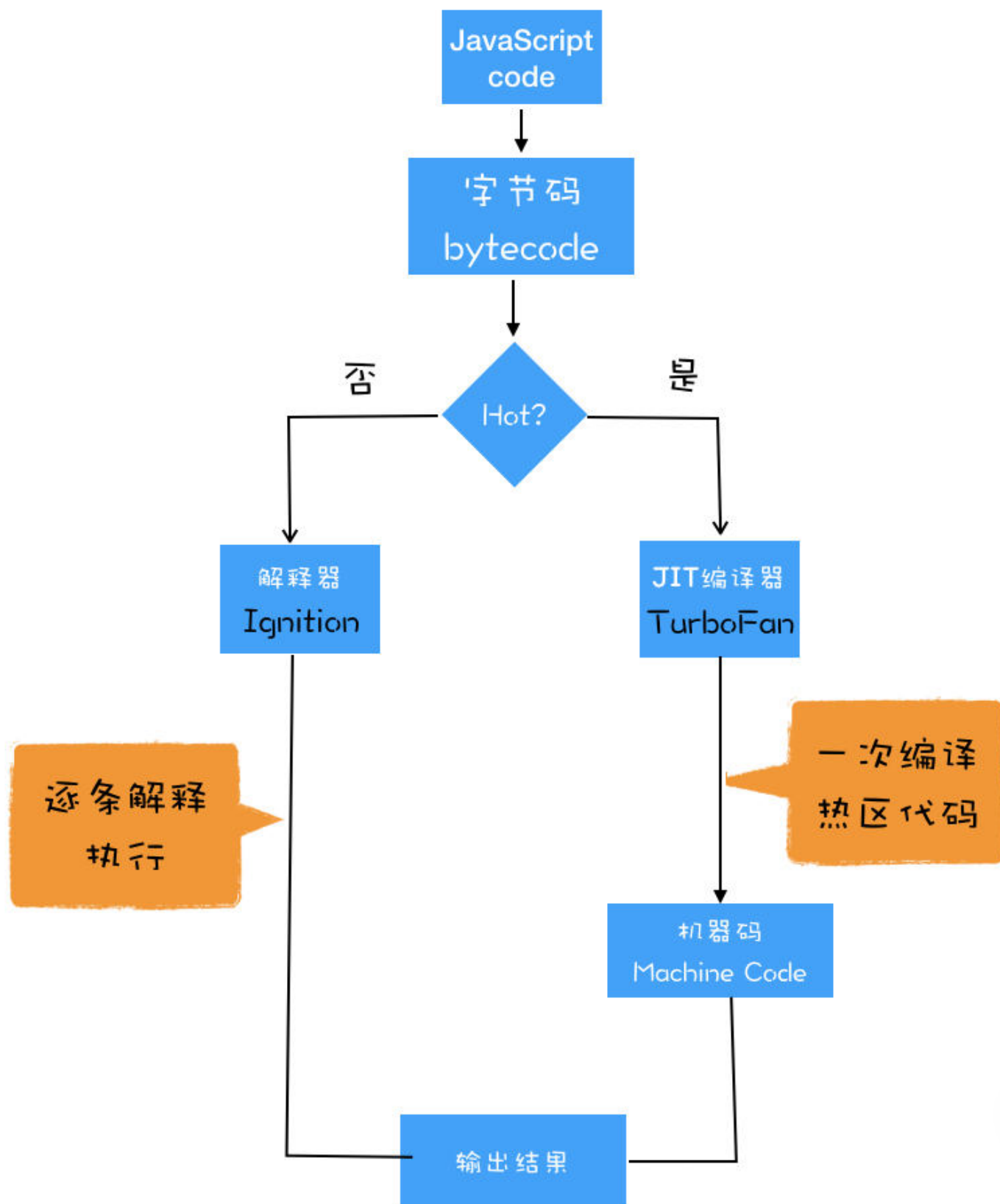
其实字节码配合解释器和编译器是最近一段时间很火的技术，比如 Java 和 Python 的虚拟机也都是基于这种技术实现的，我们把这种技术称为**即时编译（JIT）**。具体到 V8，就是指解释器 Ignition 在解释执行字节码的同时，收集代码信息，当它发现某一部分代码变热了之后，TurboFan 编译器便闪亮登场，把热点的字节码转换为机器码，并把转换后的机器码保存起来，以备下次使用。





对于 JavaScript 工作引擎，除了 V8 使用了“字节码 +JIT”技术之外，苹果的 SquirrelFish Extreme 和 Mozilla 的 SpiderMonkey 也都使用了该技术。

这么多语言的工作引擎都使用了“字节码 +JIT”技术，因此理解 JIT 这套工作机制还是很有必要的。你可以结合下图看看 JIT 的工作过程：



## JavaScript 的性能优化

到这里相信你现在已经了解 V8 是如何执行一段 JavaScript 代码的了。在过去几年中，JavaScript 的性能得到了大幅提升，这得益于 V8 团队对解释器和编译器的不断改进和优化。

虽然在 V8 诞生之初，也出现过一系列针对 V8 而专门优化 JavaScript 性能的方案，比如隐藏类、内联缓存等概念都是那时候提出来的。不过随着 V8 的架构调整，你越来越不需要这些微优化策略了，相反，对于优化 JavaScript 执行效率，你应该将优化的中心聚焦在单次脚本的执行时间和脚本的网络下载上，主要关注以下三点内容：

1. 提升单次脚本的执行速度，避免 JavaScript 的长任务霸占主线程，这样可以使得页面快速响应交互；
2. 避免大的内联脚本，因为在解析 HTML 的过程中，解析和编译也会占用主线程；
3. 减少 JavaScript 文件的容量，因为更小的文件会提升下载速度，并且占用更低的内存。

## 总结

好了，今天就讲到这里，下面我来总结下今天的内容。

- 首先我们介绍了编译器和解释器的区别。
- 紧接着又详细分析了 V8 是如何执行一段 JavaScript 代码的：V8 依据 JavaScript 代码生成 AST 和执行上下文，再基于 AST 生成字节码，然后通过解释器执行字节码，通过编译器来优化编译字节码。
- 基于字节码和编译器，我们又介绍了 JIT 技术。
- 最后我们延伸说明了下优化 JavaScript 性能的一些策略。

之所以在本专栏里讲 V8 的执行流程，是因为我觉得编译器和解释器的相关概念和理论对于程序员来说至关重要，向上能让你充分理解一些前端应用的本质，向下能打开计算机编译原理的大门。通过这些知识的学习能让你将很多模糊的概念关联起来，使其变得更加清楚，从而拓宽视野，上升到更高的层次。



## 思考时间

最后留给你个思考题：你是怎么理解“V8 执行时间越久，执行效率越高”这个性质的？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 31 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 垃圾回收：垃圾数据是如何自动回收的？

下一篇 15 | 消息队列和事件循环：页面是怎么“活”起来的？

## 学习推荐

# JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



不将就

2019-09-05

重复看之前的文章，受益良多，在此表示感谢！

不过有几个疑问，老师有空的解答下哈！

问题一：渲染进程里的input标签上传图片，通过与浏览器主进程通信，主进程读取硬磁盘图片数据返回给渲染进程，渲染进程里的js发起ajax请求，是通过浏览器主进程去调用网络进程发起请求，还是渲染进程可以直接调用网络进程发起请求？

问题二：请求长时间处于pending状态或者脚本执行死循环，这时刷新或前进后退页面不响应，刷新或前进后退页面是属于浏览器主进程的UI交互行为，为什么渲染进程里的js引擎执行会影响到主进程？

问题三：

```
function fn(){
```

```
var a =10
```

```
function f1(){
```

```
console.log(a)
```

```
};
```

```
function f2(){
```

```
console.log('f2')
```

```
};
```

```
f2();
```

```
};
```

```
fn();
```

我在函数f2里打断点，当执行到函数f2时，chrome里显示Closure:{a:10},如果把这个原因解释为在fn函数里会预扫描f1函数，那我现在把fn2函数和调用都注释了，现在执行fn函数时不产生Closure，为什么就不预扫描f1函数了？这是为什么？



作者回复:

第一个问题:

xmlhttprequest 可以直接走网络进程, 不需要浏览器进程介入

第二个问题:

因为前进或者后退也需要执行当前页面脚本啊, 比如要执行beforeunload事件, 执行的时候页面没响应了, 所以前进后退也就失效了

第三个问题:

你把f2注释了, 当执行fn函数时, 照样会预扫描f1, 照样会产生闭包, 只不过当fn执行结束之后, 闭包的内容没有外部引用, 那么下次垃圾回收直接把比闭包的内容回收掉

共 6 条评论 >

👍 71



**Rapheal**

2019-09-09

老师, 编译的基本单位是一段JS代码 (内联JS) 或者一个JS文件吗(还是以当前调用栈将要执行函数为单位) ?

作者回复: 全局代码, 或者函数 !

比如下载完一个js文件, 先编译这个js文件,但是js文件内定义的函数是不会编译的。

等调用到该函数的时候, Javascript引擎才会去编译该函数!

共 5 条评论 >

👍 38



**钟钟**

2019-09-08

执行时间越长, 执行效率越高。是因为更多的代码成为热点代码之后, 转为了机器码来执行吗?

作者回复: 是的

共 5 条评论 >

👍 35



**舔命难违**

2020-03-04

“V8 执行时间越久, 执行效率越高”, 难怪我电脑开机越久就越卡.....

**Hurry**

2019-09-07

从本文中明确的应该是在写代码的时候，如何让代码易于被 TurboFan 优化，减少反优化，老师提到的 hiddenClass 等我觉得大家还是有必要了解一下，大家可以尝试使用 node 加选项 `--trace-opt` 跑代码体验一下 TurboFan 如何做优化，就会有很直观的感受 <https://github.com/hjzheng/performance-test/blob/master/v8/addFunction.js>



15

**GY**

2019-09-27

前面第7和第12讲，变量提升说js的执行过程，是有编译过程的，变量提升就发生在编译过程，经过编译后，会生成两部分内容，执行上下文和可执行代码，但是在这一讲中，却并没有编译过程，在AST生成后，解释器就开始执行生成字节码执行了，这几讲的内容有点互相冲突，那么详细的过程到底是怎样的呢

我在查看其它资料，出现了预编译这个名词，这个又怎么解释呢  
希望能解答下

作者回复：

你可以把JavaScript的编译看成了部分：

第一部分从一段JavaScript代码编译到字节码，然后解释器解释执行字节码！

第二部分深度编译，将活跃的字节码编译成二进制，然后直接执行二进制。

无论哪个阶段都需要编译。

共 3 条评论 &gt;

9

**Geek\_Jamorex**

2019-09-05

我想提一个问题，V8解析后的字节码或热节点的机器码是存在哪的，是以缓存的形式存储的么？和浏览器三级缓存原理的存储位置比如内存和磁盘有关系么？  
最近面试有被问到，没答上来。。希望老师回答，十分感谢~

共 10 条评论 &gt;

7

**小兵**

2019-09-05

避免大的内联脚本，因为在解析 HTML 的过程中，解析和编译也会占用主线程；这句话可以理解为解析HTML代码的时候需要解析内联代码，而放到js文件的时候不需要吗？





另外思考题应该是执行越久，热点代码越多，即时编译的作用越大。

作者回复: 只要是同步脚本都会阻塞，这里我可能没说清楚。

我的表达的以上是同步脚本尽量小，尽量能内联。

其它的尽量采用异步脚本，如使用async和defer。

共 5 条评论 >

👍 6



李懂

2019-09-05

怎么都需要字节码文件，为啥，JavaScript不像java一样先编译为字节码，这样执行效率不就高了么！

作者回复: 你可以认为WebAssembly就是，WebAssembly经过TuboFan处理下就能执行了



👍 6



Marvin

2019-09-05

我理解，V8执行越久，被编译成机器码的热点代码就越多，所以整体执行效率就越高。如果是这样的话，那么V8内存占用也会越来越多，会面临的问题会和

作者回复: 引入了字节码，就有弹性空间了，可以在内存和执行速度之间做调节。

相比之前的V8，将JS代码全部编译成字节码，这种模式就没有协商的空间了！

共 2 条评论 >

👍 5



Bazinga

2019-11-28

总结说：V8 依据 JavaScript 代码生成 AST 和执行上下文，再基于 AST 生成字节码，然后通过解释器执行字节码，通过编译器来优化编译字节码。但是第二节生成字节码那一段说：解释器 Ignition 就登场了，它会根据 AST 生成字节码，并解释执行字节码。还有即时编译（JIT）技术那张图片，看起来也是先生成字节码 再经过解释器 。所以字节码是解释器生成的吗？我都看懵了，求解答。

作者回复: 流程是这样的：



v8先生成ast!

然后ignition根据ast生成字节码。

在然后ignition解释执行字节码。

所以ignition生成了字节码并解释执行字节码。

共 2 条评论 >

👍 4



**westfall**

2019-10-23

字节码最终也会转成机器码来执行的吧？因为最终都是cpu来执行，cpu只能执行机器码

作者回复: 是的



👍 4



**阳仔**

2021-08-23

面试被问到：js 在编译过程中，会做一定的优化，那么日常开发，应该怎么利用这个优化，提升代码质量

共 1 条评论 >

👍 2



**Geek\_panda**

2021-07-20

老师还在吗，想请教2个问题

1. v8生成执行上下文是根据源码生成还是根据ast来生成呢？

2. 解释器执行字节码时是不是也需要将他转成机器码，如果是的话，那他是不是也会通过TurboFan编译器编译

@李兵 老师

共 3 条评论 >

👍 2



**crown**

2020-05-15

V8刚开始执行代码的时候，都是通过ignition解释器来逐行解析字节码的，这样性能会比较慢。当执行一段时间过后，ignition可以捕获到经常被执行的到的字节码。这些字节码就会被作为热代码交给turbofan编译成为机器码。后续可以直接使用机器码，而机器码的执行效率优于字节码。当V8执行越久，使用量高的字节码都被编译为机器码。故V8执行越久，效率越高



👍 2





sugar

2020-01-22

老师您好，我曾想过不用babel typescript等的ast而是自己开发一个c++项目，引入v8利用他的ast来做一些代码转换工作，这样可以基于c的很多机制做更多多线程方面的优化。后发现这对于v8来说是不可能的，因为v8是一部分一部分解析js的，v8为什么采用这样的机制呢？另外这方面如果想自己动手拿v8做些事儿 老师有什么推荐的资料 或书籍可以看看吗？

共 2 条评论 >



2



七月有风

2019-12-07

老师,你好，node 的 JavaScript 引擎是 V8, ReactNative 和 Android webview 的 JavaScript 引擎是V8引擎吗？

共 2 条评论 >



2



刹那

2019-10-22

想到一个问题，可以把代码预先编译成字节码吗？这样浏览器下载了就能直接运行

共 4 条评论 >



2



Geek\_177f82

2019-09-16

问一个基础的问题。希望老师解答。编译器编译后的二进制文件，与解析器解析后机器码是一个东西吗？

共 2 条评论 >



2



Angus

2019-09-05

V8执行越久，被编译成机器码的热点就越多，这些机器码帮助字节码可以直接执行而不再使用解释器逐行执行，这相当于浏览器缓存，提高了执行性能。这些生成的机器码也会带来内存占用升高的问题，这里应该会有一个权衡措施吧，根据已占用的内存权衡如何判定是热点并生成机器码保存。

作者回复: 是的，可以实现很多策略来权衡不同系统的情况



2

