2022年4月22日 / #JAVASCRIPT

深入理解 JavaScript 的 V8 引擎

luojiyin

原文：How JavaScript Works: Under the Hood of the V8 Engine，作者：Ilya Lyamkin

今天我们来看看 JavaScript 的 V8 引擎，弄清楚 JavaScript 到底是如何执行的。

在之前的文章中，我们了解了浏览器的结构，并对 Chromium 有了一个高度的概述。让我们回顾一下，这样我们就可以准备好在这里深入研究了。

背景

Web 标准 是浏览器实施的一套规则。它们定义并描述了万维网。

W3C 是一个为网络开发制定标准的国际社区。他们确保每个人都遵循相同的准则，不必支持几十个完全不同的环境。

而浏览器中最重要的两个部分是 JavaScript 引擎和渲染引擎。

Blink 是一个渲染引擎，负责整个渲染管道，包括 DOM 树、样式、事件和 V8 集成。它解析 DOM 树，解决样式问题，并确定所有元素的视觉几何。

在通过动画帧不断监测动态变化的同时，Blink 在屏幕上绘制内容。JS 引擎是浏览器的一个重要部分--但我们还没有进入这些细节。

JavaScript 引擎

JavaScript 引擎将 JavaScript 编译成本地机器代码并执行。每个主要的浏览器都开发了自己的 JS 引擎。谷歌的 Chrome 使用 V8，Safari 使用 JavaScriptCore，Firefox 使用 SpiderMonkey。

我们将特别使用 V8，因为它在 Node.js 和 Electron 中使用，但其他引擎也是以同样的方式构建的。

每个步骤都将包括一个负责该步骤的代码链接，因此你可以熟悉代码库，并在本文之后继续研究。

我们将使用 GitHub 上的 V8 镜像，因为它提供了一个方便和知名的 UI 来浏览代码库。

准备代码

V8 需要做的第一件事是下载源代码。这可以通过网络、缓存或 service workers 来完成。

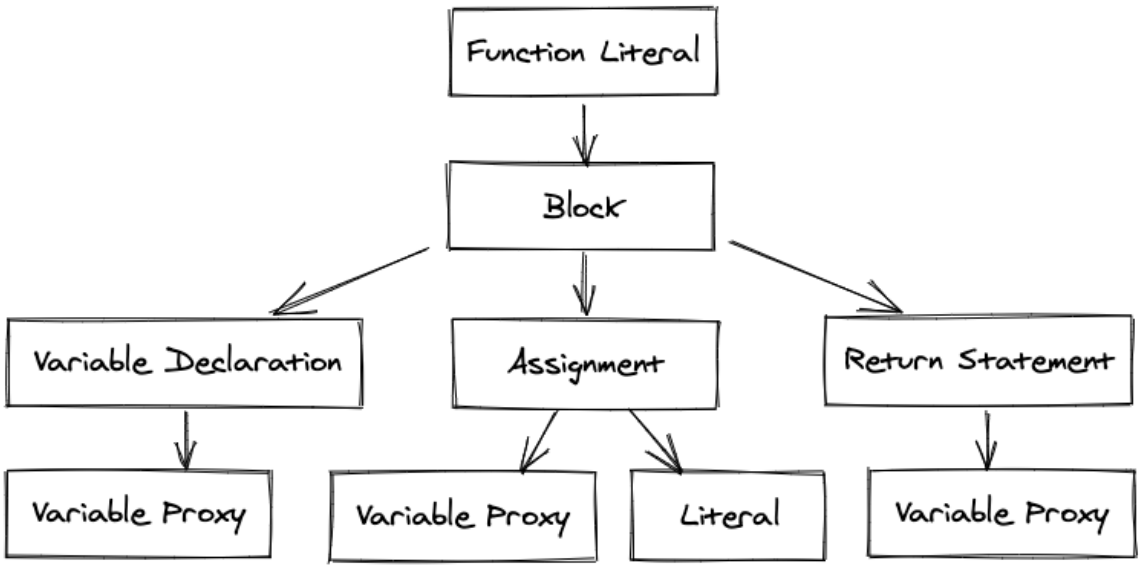
扫描器接收 JS 文件并将其转换为已知的标记列表。在 keywords.txt文件中有一个所有 JS 标记的列表。

解析器识别它并创建一个 抽象语法树（AST）：源代码的树状表示。树上的每个节点都表示代码中出现的一个结构。

让我们看一下一个简单的例子：

```
function foo() {
    let bar = 1;
    return bar;
}
```

这段代码将产生以下树状结构：



AST 树的例子

1. 定义 `foo` 函数。
2. 声明 `bar` 变量。
3. 将 `1` 分配给 `bar` 。
4. 从函数中返回 `bar` 。
   你还会看到 `VariableProxy` --一个将抽象变量连接到内存中某个地方的元素。解决 `VariableProxy` 的过程被称为 **范围分析（Scope Analysis）** 。

在我们的例子中，这个过程的结果是所有 `VariableProxy` 都指向同一个 `bar` 变量。

The Just-in-Time（JIT）paradigm（即时编译）

一般来说，为了使你的代码能够执行，编程语言需要被转化为机器代码。对于如何以及何时发生这种转换，有几种方法。

最常见的转换代码的方法是进行超前编译。它的工作原理：在编译阶段，代码在程序执行之前就被转化为机器代码了。许多编程语言都采用这种方法，如C++、Java 和其他语言。

在表格的另一边，是解释型：每一行代码都将在运行时执行。这种方法通常被动态类型语言（如 JavaScript 和 Python）采用，因为在执行之前不能知道确切的类型。

因为提前编译可以一起评估所有代码，它可以提供更好的优化并最终生成更高性能的代码。另一方面，解释型语言更容易实现，但它通常

（JIT）编译的新方法。它最好地结合了解释和编译。

在使用解释（interpretation）作为基础方法的同时，V8 可以检测到比其他函数更频繁使用的函数，并使用以前执行的类型信息对其进行编译。

然而，类型有可能会发生变化。我们需要对已编译的代码进行去优化，转而返回到解释法（之后，我们可以在得到新的类型反馈后重新编译函数）。

让我们更详细地探讨一下 JIT 编译的每个部分。

Interpreter（解释器）

V8 使用一个叫做 Ignition 的解释器。最初，它接受一个抽象的语法树并生成字节码。

字节码指令也有元数据，如源行位置，以便将来进行调试。一般来说，字节码指令与 JS 的抽象内容相匹配。

现在让我们以我们的例子为例，为它手动生成字节码：

```
LdaSmi #1 // write 1 to accumulator
Star r0   // read to r0 (bar) from accumulator
Ldar r0   // write from r0 (bar) to accumulator
Return    // returns accumulator
```

累加器避免了推送和弹出堆栈顶部的需要。它也是许多字节代码的隐含参数，通常保存操作的结果。隐式地返回累加器。

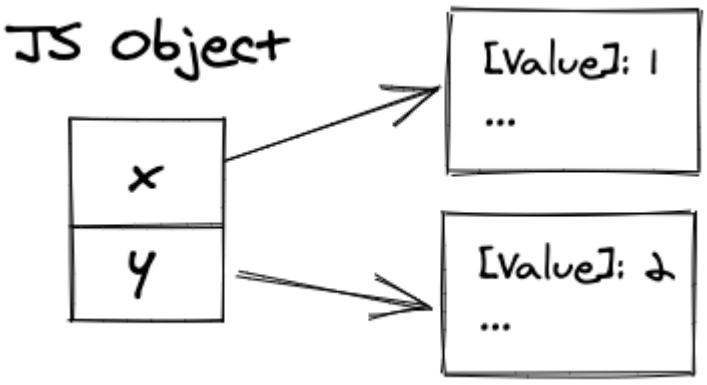
你可以查看 在相应的源代码中 所有可用的字节码。如果你对其他 JS 概念（如循环和async/await）如何在字节码中呈现感兴趣，我觉得通过这些 测试用例来阅读是很有用的。

Execution（执行）

生成后，Ignition 将使用一个以字节码为关键的处理程序表来解释这些指令。对于每个字节码，Ignition 可以查找相应的处理程序函数，并使用提供的参数执行它们。

正如我们之前提到的，执行阶段还提供关于代码的类型反馈。让我们来弄清楚它是如何被收集和管理的。

首先，我们应该讨论如何在内存中表示 JavaScript 对象。在一个天真的方法中，我们可以为每个对象创建一个字典，并将其链接到内存中。



保存对象的第一种方法

为了解决这个问题，V8 使用 **Object Shapes**（或内部映射 Maps internally）和内存中的值向量将对象的结构与值本身分开。

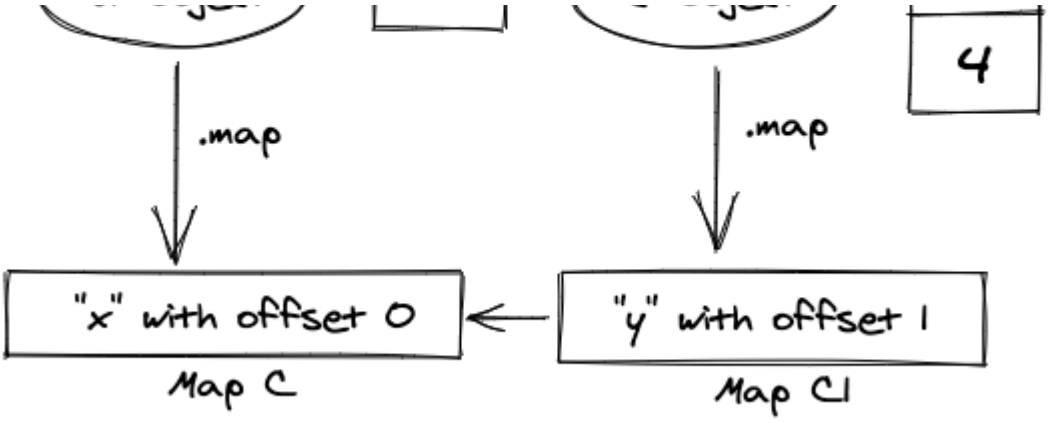
例如，我们创建一个对象字面：

```
let c = { x: 3 };
let d = { x: 5 };
c.y = 4;
```

在第一行，它将产生一个 shape `Map[c]`，其属性为 `x`，偏移量为 0。

在第二行中，V8 将为一个新的变量重新使用相同的 shape。

在第三行之后，它将为属性 `y` 创建一个新的 shape `Map[c1]`，偏移量为1，并创建一个与之前 shape `Map[c]` 的引用。



object shapes 的例子

在上面的例子中，你可以看到每个对象都可以有一个指向 object shape 的链接，对于每个属性名称，V8 可以在内存中找到一个值的偏移。

Object shapes 本质上是链接列表。因此，如果你写 `c.x`，V8 会去到列表的头部，在那里找到 `y`，移动到连接的 shape，最后它得到 `x` 并从中读取偏移。然后它将进入内存向量并返回其中的第一个元素。

你可以想象，在一个大的网络应用中，你会看到大量的 shapes。同时，在链接列表中搜索需要线性时间，使得属性查找成为一个一个非常耗费时间的操作。

为了解决 V8 中的这个问题，你可以使用**在线缓存 Inline Cache (IC)**。它记住了在哪里可以找到对象的属性的信息，以减少查找的次数。

你可以把它看作是你代码中的一个监听点：它跟踪一个函数中所有的 _CALL_、_STORE_和_LOAD_事件，并记录所有经过的 shapes。

```
function load(a) {
    return a.key;
}
```

对于上述函数，反馈向量将是这样的：

```
[{ slot: 0, icType: LOAD, value: UNINIT }];
```

这是一个简单的函数，只有一个 IC，它的类型是 `LOAD`，值是 `UNINIT`。这意味着它是未初始化的，我们不知道接下来会发生什么。

让我们用不同的参数调用这个函数，看看 Inline Cache 会有什么变化。

```
let first = { key: 'first' }; // shape A
let fast = { key: 'fast' }; // the same shape A
let slow = { foo: 'slow' }; // new shape B
```

在第一次调用 `load` 函数后，我们的内联缓存（inline cache）将得到一个更新的值：

这个值现在变成了单态的，这意味着这个缓存只能解析为 shape A。

在第二次调用后，V8 将检查 IC 的值，它会看到它是单态的，并且与 `fast` 变量的 shape 相同。所以它将迅速返回偏移量并解析它。

第三次，其形状与存储的 shape 不同。所以 V8 将手动解决它，并将其值更新为多态状态，有两个可能的 shape 的数组。

```
[{ slot: 0, icType: LOAD, value: POLY[(A, B)] }];
```

现在我们每次调用这个函数时，V8 需要检查的不仅仅是一个 shape，而是在几种可能性中进行迭代。

为了使代码更快，你可以用相同的类型初始化对象，并且不对它们的结构做太多的改变。

注意：你可以记住这一点，但如果导致代码重复或代码性能降低，就不要这样做。

内联缓存还可以跟踪它们被调用的频率，以决定它是否是优化编译器的好候选者——Turbofan。

Compiler（编译器）

Ignition 只能让我们走到这里。如果一个函数变得足够热（译者注：调用频繁），它将在编译器中被优化，Turbofan，以使其更快。

正如我们之前看到的，类型反馈并不能保证它在未来不发生变化。

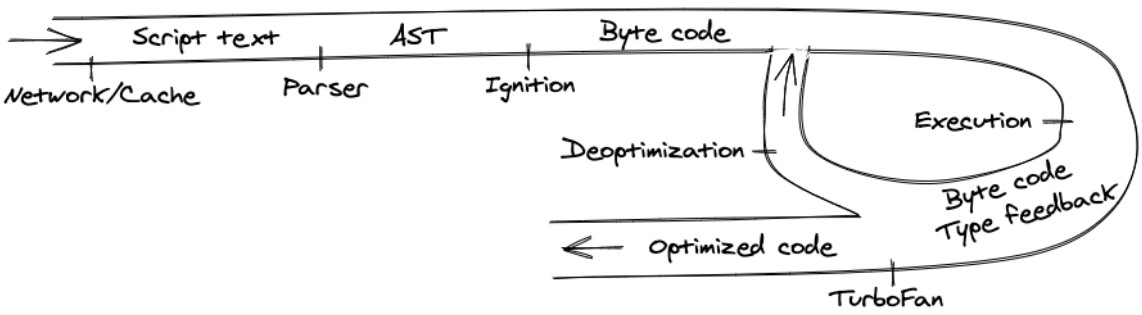
例如，Turbofan 基于一些加法总是加整数的假设来优化代码。

但如果它收到的是一个字符串，会发生什么？这个过程被称为**去优化**，我们扔掉优化的代码，回到解释的代码，恢复执行，并更新类型反馈。

总结

在这篇文章中，我们讨论了 JS 引擎的实现以及 JavaScript 如何执行的具体步骤。

总结一下，让我们从头看一下编译管道。



V8 概述

我们将一步一步地看下去：

1. 这一切都始于从网络中获取 JavaScript 代码。
2. V8 解析源代码并将其转化为抽象语法树（AST）。

4. 在这一点上，引擎开始运行代码并收集类型反馈。
5. 为了使它运行得更快，字节码可以和反馈数据一起被发送到优化编译器。优化编译器在此基础上做出某些假设，然后产生高度优化的机器代码。
6. 如果在某些时候，其中一个假设被证明是不正确的，优化编译器就会取消优化，并回到解释器中。

就是这些了！如果你对上面某个特定的阶段有任何疑问，或者想了解更多的细节，你可以潜心研究源代码，或者在 Twitter 上联系我。

深入阅读

- life of a script 来自谷歌的视频
- A crash course in JIT compilers 来自 Mozilla
- 很好的解释 Inline Caches in V8
- 深入了解 Object Shapes

luojiyin

阅读 更多文章。

07795461）。

我们的使命：帮助人们免费学习编程。我们通过创建成千上万的视频、文章和交互式编程课程——所有内容向公众免费开放——来实现这一目标。学员在世界各地自发成立数千个 freeCodeCamp 学习小组。

所有给 freeCodeCamp 的捐款都将用于我们的教育项目，购买服务器和其他服务，以及聘用员工。

你可以点击此处免税捐款。

精选文章

版权条例