Node.js V8 内部结构：说明性入门指南

瓦尔丹·格里戈里安 (vardanator) · 跟随
出版于 代码突发
阅读时间 12 分钟 · 2017 年 11 月 10 日

当您从 C++ 切换到更“友好”的语言时，您会有一种被这种新语言、其环境或运行时或其他东西（无法告诉您）愚弄的奇怪感觉。在 C++ 中，你必须思考低层（我的意思是低层），深入思考，你必须了解[至少一些]低层细节。事实上，您必须了解这些细节才能了解您的代码如何工作或可能工作（然后在未来几年了解更多，但仍然对掌握这种语言有多么困难感到困惑）。服务器端 JavaScript Node.js 非常适合后端，实际上，到目前为止，我对 Node.js 的体验只是 REST API 开发（那些结构良好的 URL 以及相应的处理程序，如果是这样，则发送该，否则发送该，主要是常规的东西）。Node.js 以其异步性吸引了我的心，而我内心的那部分，那个为 C++ 的伟大而尖叫的部分，总是低声念叨着那些神奇的话语，“但是如何呢？它是如何工作的？”。

作为 C++ 的粉丝（这并不意味着我对它了解得足够多），我喜欢 Node.js 基于**libuv**（用 C 编写）和**V8**（大部分是用 C++ 编写）的想法。所以挖起来很容易。简单易行，对吧？十分简单..

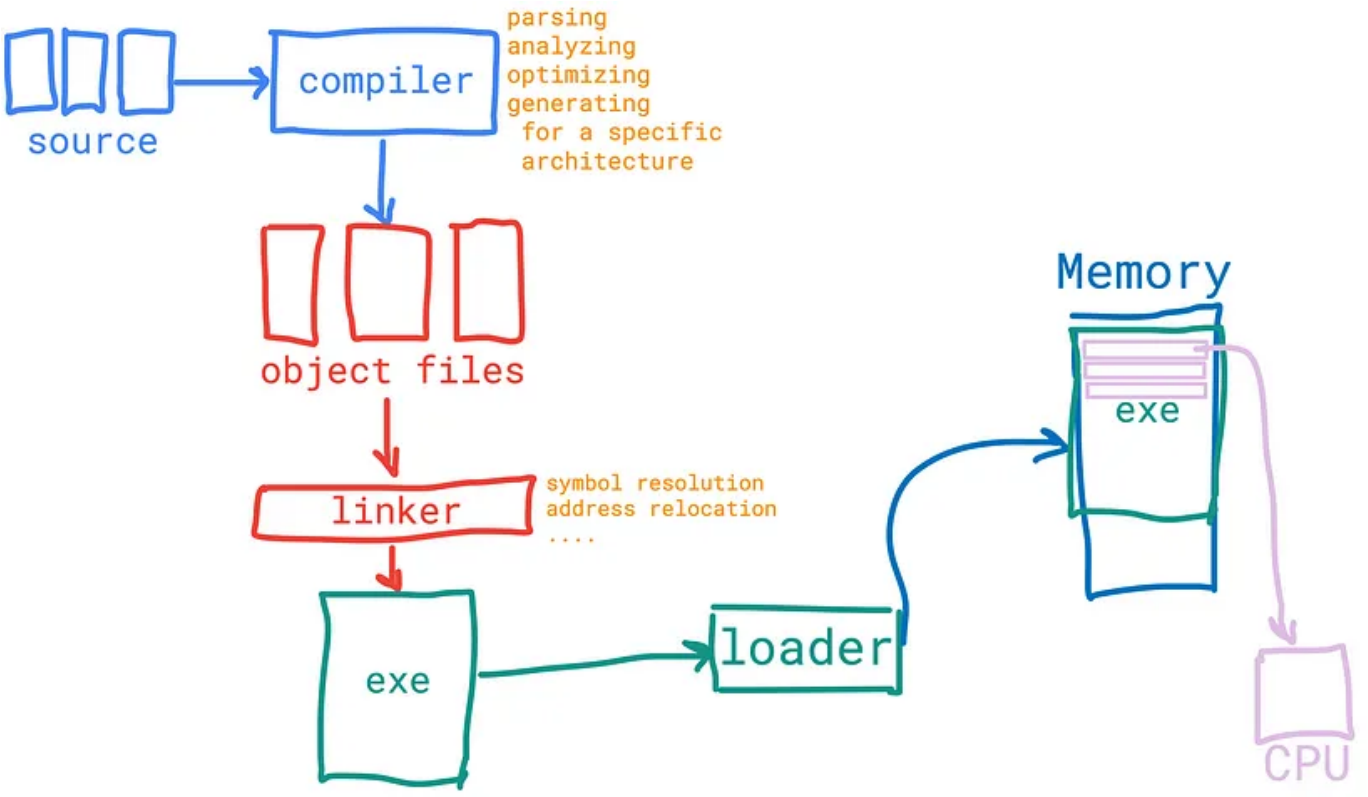
我们将研究（某种程度上）JavaScript V8 引擎。

> *V8*是开源的*JavaScript*执行引擎，为*JS*代码提供高性能的执行环境。

JS 引擎确实有很多，我只是喜欢研究 Google 工程师生产的产品。这是一种品牌名称，你看，引擎，它不是 V8，而是 Google 的 V8。为什么我们要深入研究？好吧，除了挖掘一些概念验证技术确实很有趣之外，它还可以帮助我们编写更好的代码。

首先，我们需要比较一下C++的编译过程和Node.js的解释过程（从现在开始我将交替使用JS和Node.js）。起初，我认为 JS 是一种“简单”的脚本语言，由单个解释器完成日常翻译工作。

在C++中，我们得到源代码，编译器将源代码单元编译成目标文件。当然，当一个称为预处理器的特殊程序（我只是将其视为编译器的一部分）使源代码“准备好”进行编译时，有一个预处理阶段。因此源代码被编译成目标代码。目标代码单元。其中很多。（假设我们有一个足够大的项目）。



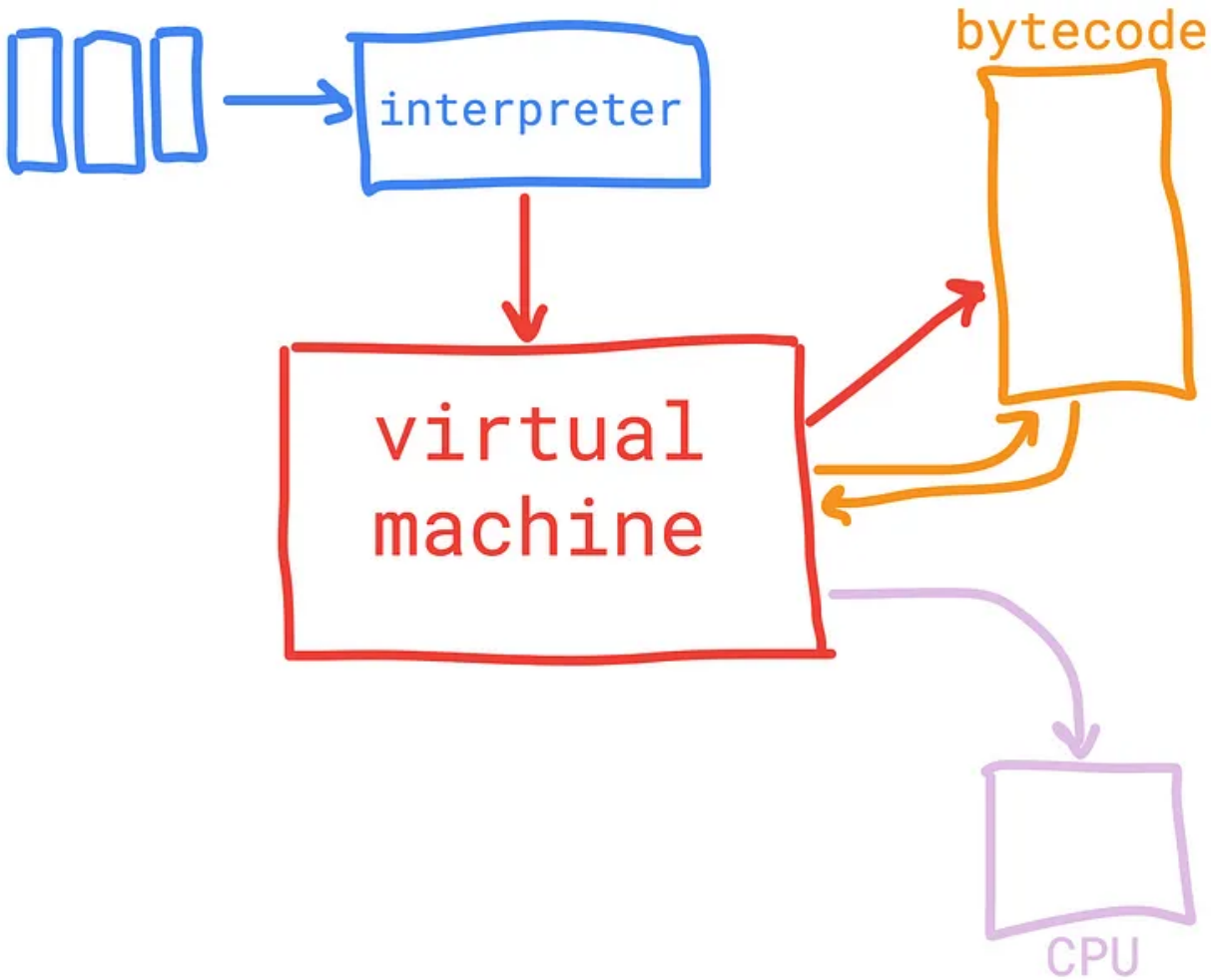
AoT编译和执行（简单抽象）

这个过程相当复杂，想要深入了解编译/翻译技术/技术的人，建议阅读Dragon 书（我还没读过，我什至没有开始，虽然我已经读过内容部分、关于部分、致谢部分，嗯，就像我所有的书一样）。在编译过程中，编译器生成目标代码作为其输出，一些相当优化的目标代码（是的，机器代码）。一些开发人员坚持认为编译器比开发人员本身优化并生成更好的代码。我无法反驳这一点。

因此，链接器就出现了，它是另一个工具，旨在组合这些目标文件，进行一些地址重定位、符号解析等（很容易找到有关链接器的更多信息，真的建议这样做）。链接器的意

义非常重要，它将生成的目标代码单元粘合到一个大文件中，即可执行文件。该文件可以作为程序运行。程序。必须有人运行该程序。不是用户，用户只是通过双击相应的图标来启动。还有一个工具，**loader**。加载程序顾名思义，就像“快点”一样，加载程序加载。可执行文件的内容必须由操作系统加载到虚拟内存中（加载器是操作系统的一部分）。所以加载器只是将我们的可执行文件内容复制到虚拟机中，将CPU相应的指令指针寄存器设置为启动例程（主函数？），瞧，程序开始运行。（对硬过程最抽象的描述）。因此，没有回溯，代码已经编译，代码是完全机器代码，为特定机器架构生成。没有解释，CPU只是一一执行指令（机器代码指令）（好吧，假设我们有一个直接的CPU）。这就是为什么人们开始将这个编译命名为 AoT 编译。提前 (AoT)，就像“现在不需要编译它，但是，是的，你提前完成了”。与此相反的是流行的 JIT 编译。及时 (JiT)，就像“需要时编译”。这就像耐克的口号，Just Compile It！

在我看来，这就是即时编译（又名解释编译，或者你可能会说互编译）的工作原理。

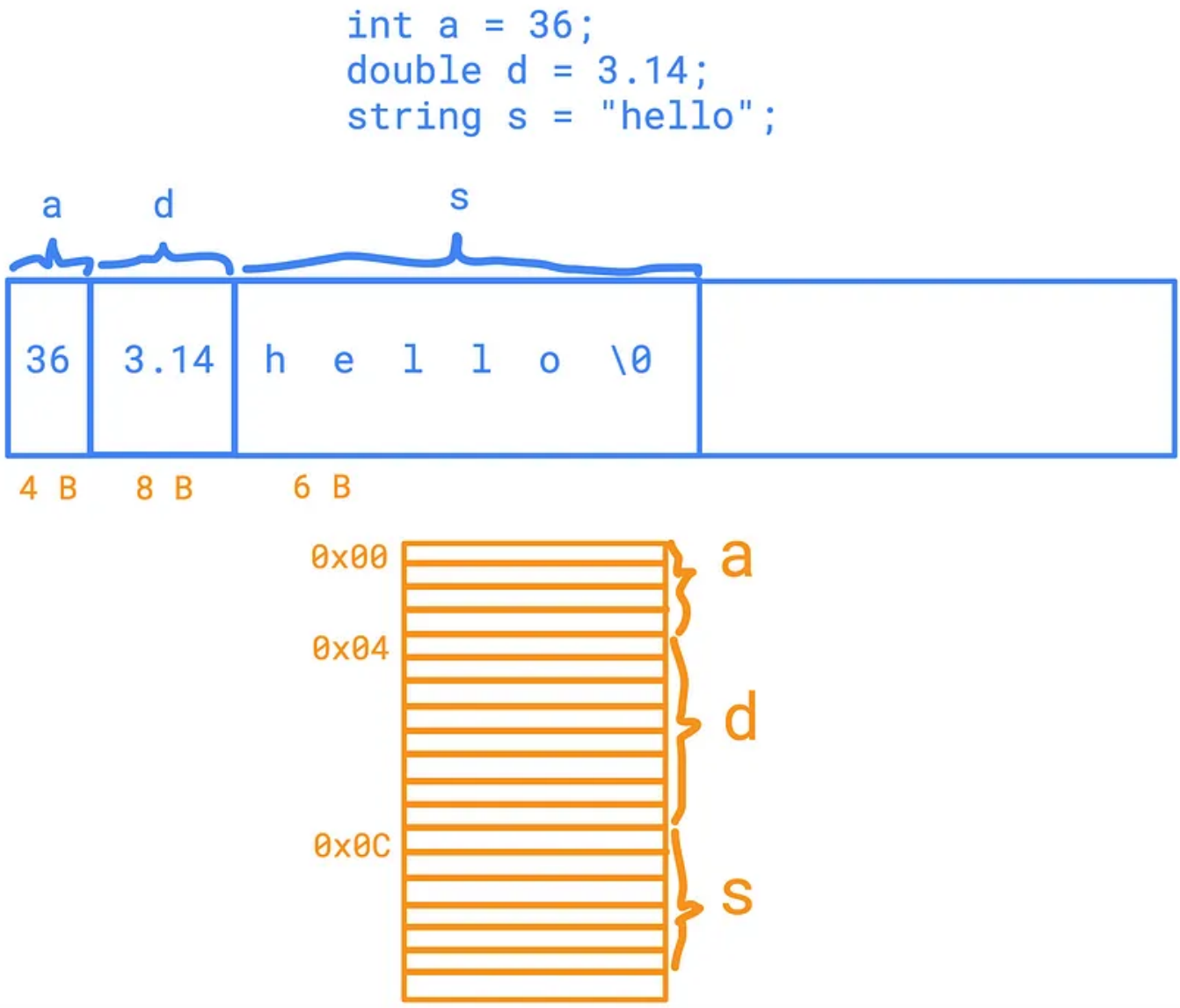


及时编译

同样，我们有一个源代码（这里没有预处理，不允许使用 C#），并且源代码正在由解释器解释。还记得快点吗？快点快点，翻译翻译。解释器是一个旨在帮助虚拟机的工具。虚拟机是安装在用户计算机上并运行我们编写的代码的程序。在这种特殊情况下，“用户的计算机”是我们的后端服务器，该服务器运行我们的 Node.js 代码。所以解释器不会生成机器代码。它不需要了解不同的架构/机器/平台（Intel、AMD、Windows、Linux 等）。解释器只需要了解虚拟机，而虚拟机必须了解架构/机器/平台。所以解释器生成“代码”，一些虚拟机可以理解的“中间代码”，虚拟机必须生成实际的机器代码来运行程序。我认为这只是将较高级别的代码逐行翻译为较低级别的代码并执行每个代码。事实证明，解释器和虚拟机做了一些非常困难的事情。为什么这样？为什么不直接编译成机器码呢？这是一个很长的故事，一个又长又另一个的故事。简短的答案是“平台”。有这么多... 为什么这样？为什么不直接编译成机器码呢？这是一个很长的故事，一个又长又另一个的故事。简短的答案是“平台”。有这么多... 为什么这样？为什么不直接编译成机器码呢？这是一个很长的故事，一个又长又另一个的故事。简短的答案是“平台”。有这么多...

切入正题，解释不好，编译好。嗯，更实际的是，解释速度很慢，编译后的程序执行速度更快。但 JavaScript 还有许多其他“坏”的事情。首先，弱类型。C++ 是严格类型语言，JavaScript 是弱类型语言。这意味着什么？这意味着我们可以自由。

严格的类型是好的，至少对于编译器本身来说是这样。编译器必须严格知道变量对应的类型。为什么？因为编译器要生成代码。它必须将这些变量放置在正确的位置，为它们分配固定大小的框，并尽可能优化它们。编译器无法将 4 字节整数放入 20 字节框中，只是因为它可以转换为某个字符串。



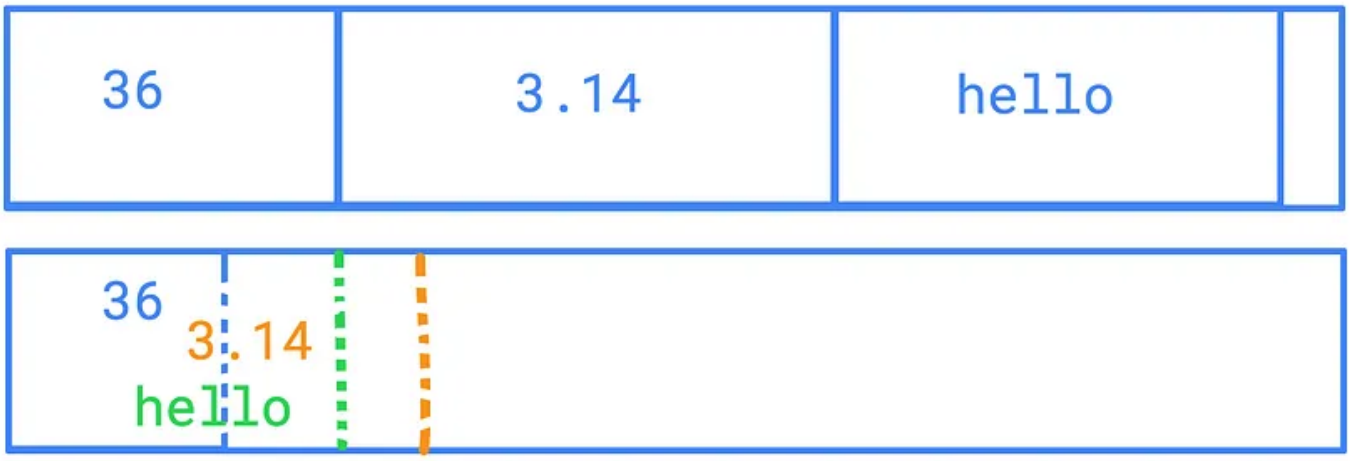
严格类型有助于编译器生成正确的代码

编译器将变量准确地放置在它们“应该”的位置。

相反，你可以在 JavaScript 中编写类似的东西，而不会产生任何无能的感觉（一点点）。

```
var some = "这是一个字符串"; // 声明一个字符串
some = 36; // 动态更改类型
some = 3.14;
```

```
var a = 36;
a = 3.14;
a = "hello";
```



弱类型会减慢执行速度

解释器应该如何解释上面的变量'a'（有时我交替使用解释器、虚拟机和执行引擎）？是否应该将“a”放在一个大（以防万一）盒子中？是否应该为每个“a”类型存储隐藏变量？解释器（或虚拟机）为支持这些“语言怪癖”所做的任何事情都会减慢代码的执行速度。

还有更多即将到来.....

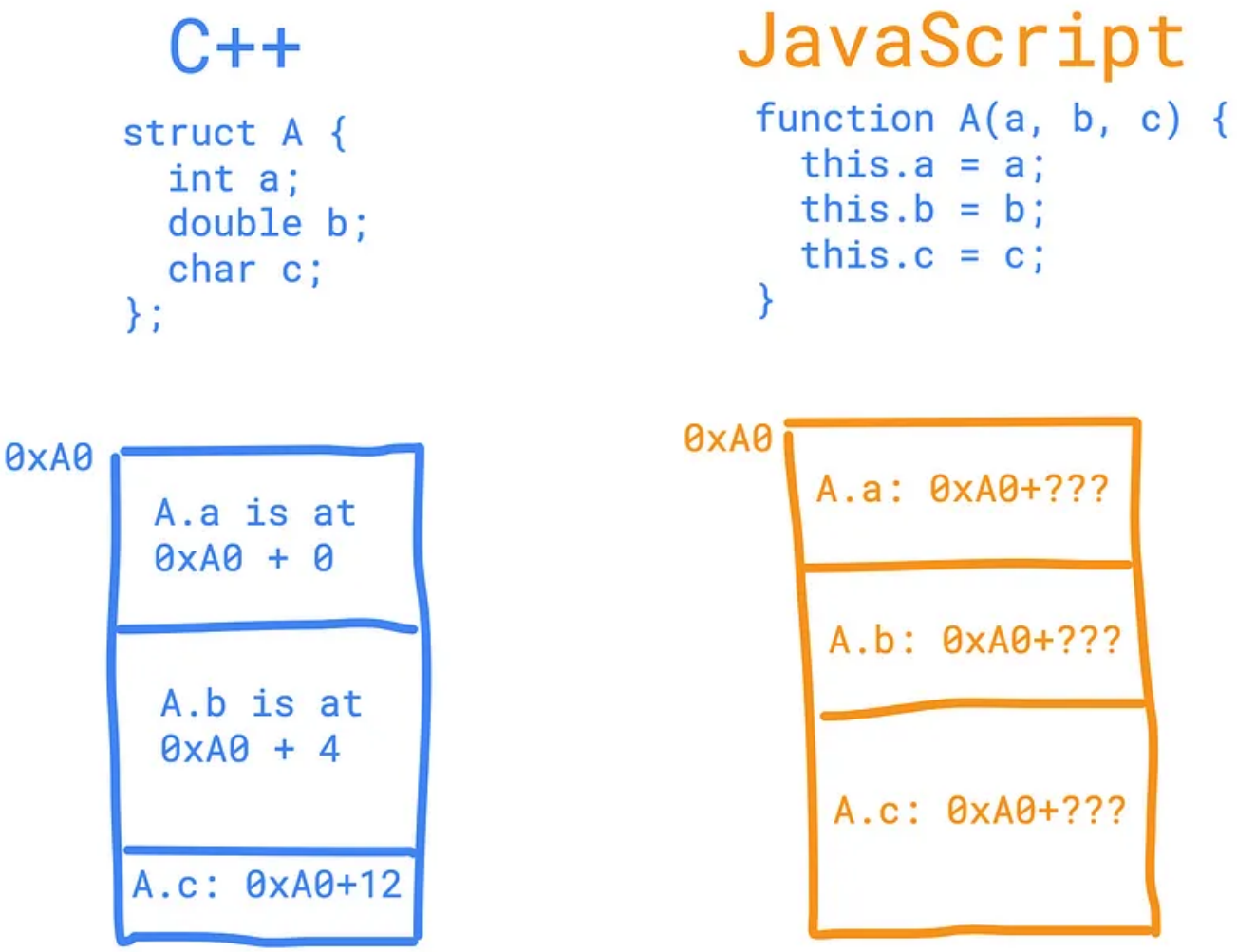
那些可以动态地、即时地“改变”其属性的对象又如何呢？见下文。

```
function A(a, b, c) {
  this.a = a;
  this.b = b;
  this.c = c;
}

let obj = new A();
let obj2 = new A(1, 2, 3);
let obj3 = new A("str", 3.14, 20);
obj.b = 45;// change on the fly
// add on the fly
obj3.new_prop = "some str";
delete obj2.b; // remove on the fly
```

对虚拟机进行压力测试

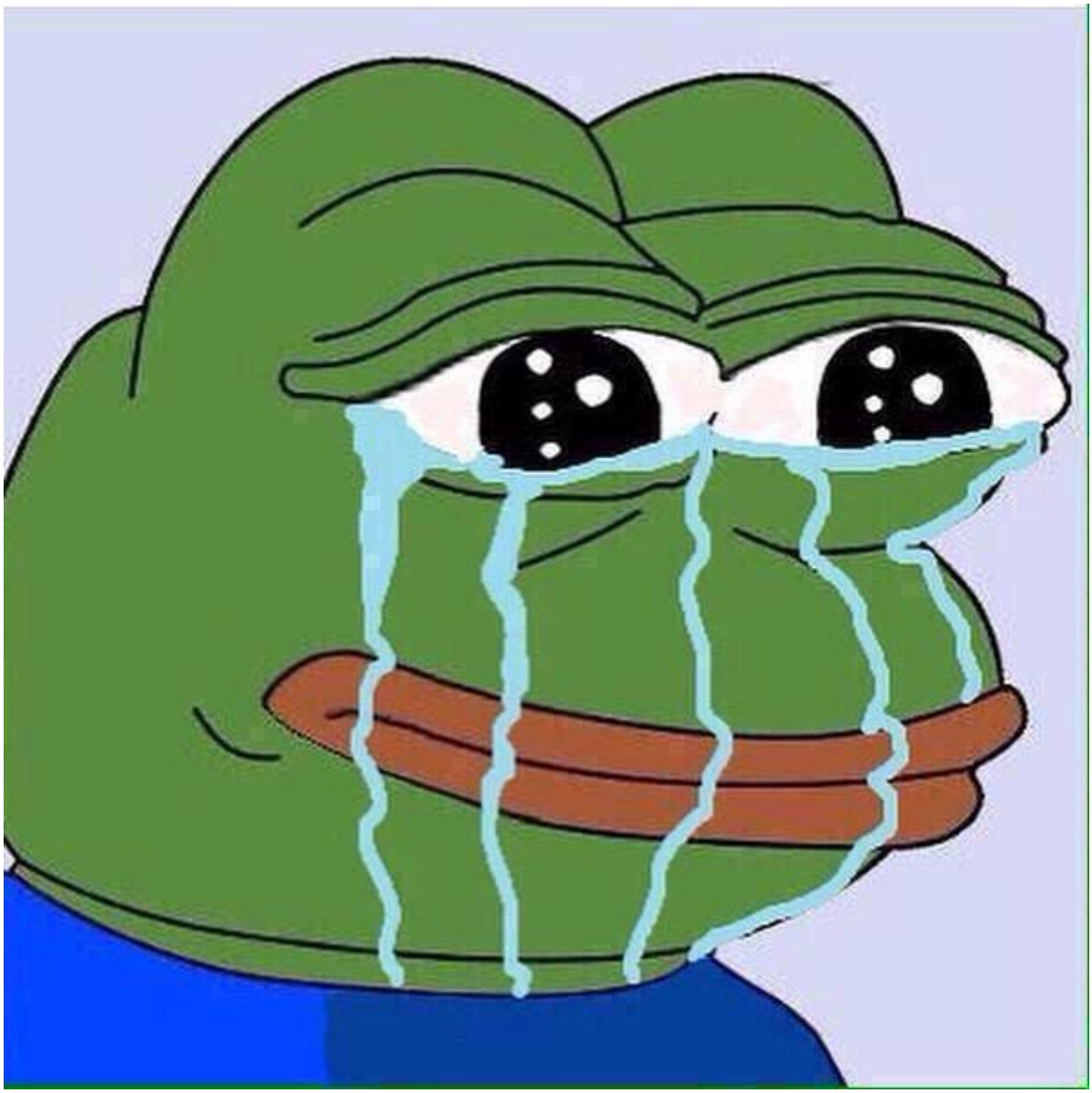
我们关心的只是对象的布局。你看，C++编译器生成对象的布局很流畅，而JavaScript的“执行引擎”则在思考上苦苦挣扎，如何“ frakin ”做到这一点？



对象内存布局

我将这些插图称为“伪插图”，因为有时它们包含一些无效但不重要的细节。在上图中我们看到，在 C++ 中，A 的实例被“平滑”地布局，“a”属性占用 4 个字节（只是一个假设，就像我所说的伪图）并从地址 0xA0 开始。对象本身（A 的实例）从地址 0xA0 开始，因此第一个属性“a”被放置在地址 0xA0 + 相应的偏移量处（在本例中为 0，作为“a”属性，是实例）。因此，由于“a”占用 4 个字节，因此下一个（第二个）属性“b”从 0xA0 + 4 开始（“a”属性占用的大小）。“c”从 0xA0 + 12 开始（加上“a”和“b”使用的大小）。我们忽略了一个调整因素（在这种情况下不是主要情况）。

现在看看同一张插图的右侧。JS 应该如何放置 A 的实例及其属性？它应该如何猜测类型，如果它们改变怎么办？它应该做什么？它会去哪里？

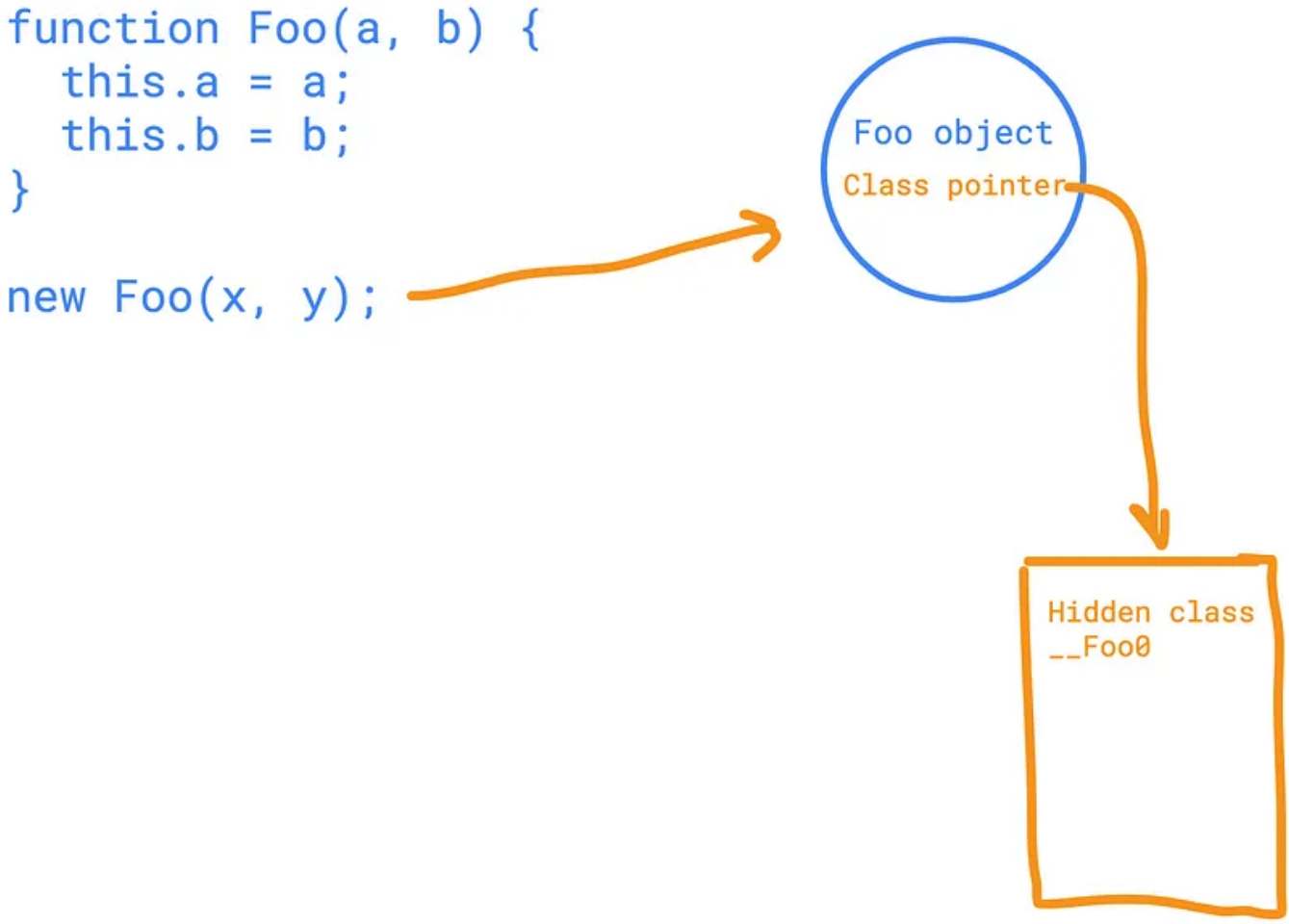


V8，尝试尽可能快地执行你的代码

最后，我们进入正题，隐藏类。V8引入隐藏类的想法只是为了让代码执行得更快。比属性的“字典查找”更快。这是通过为您创建的每个对象实例以及更多对象实例生成隐藏类来完成的。例如，当我们有这样的代码时。

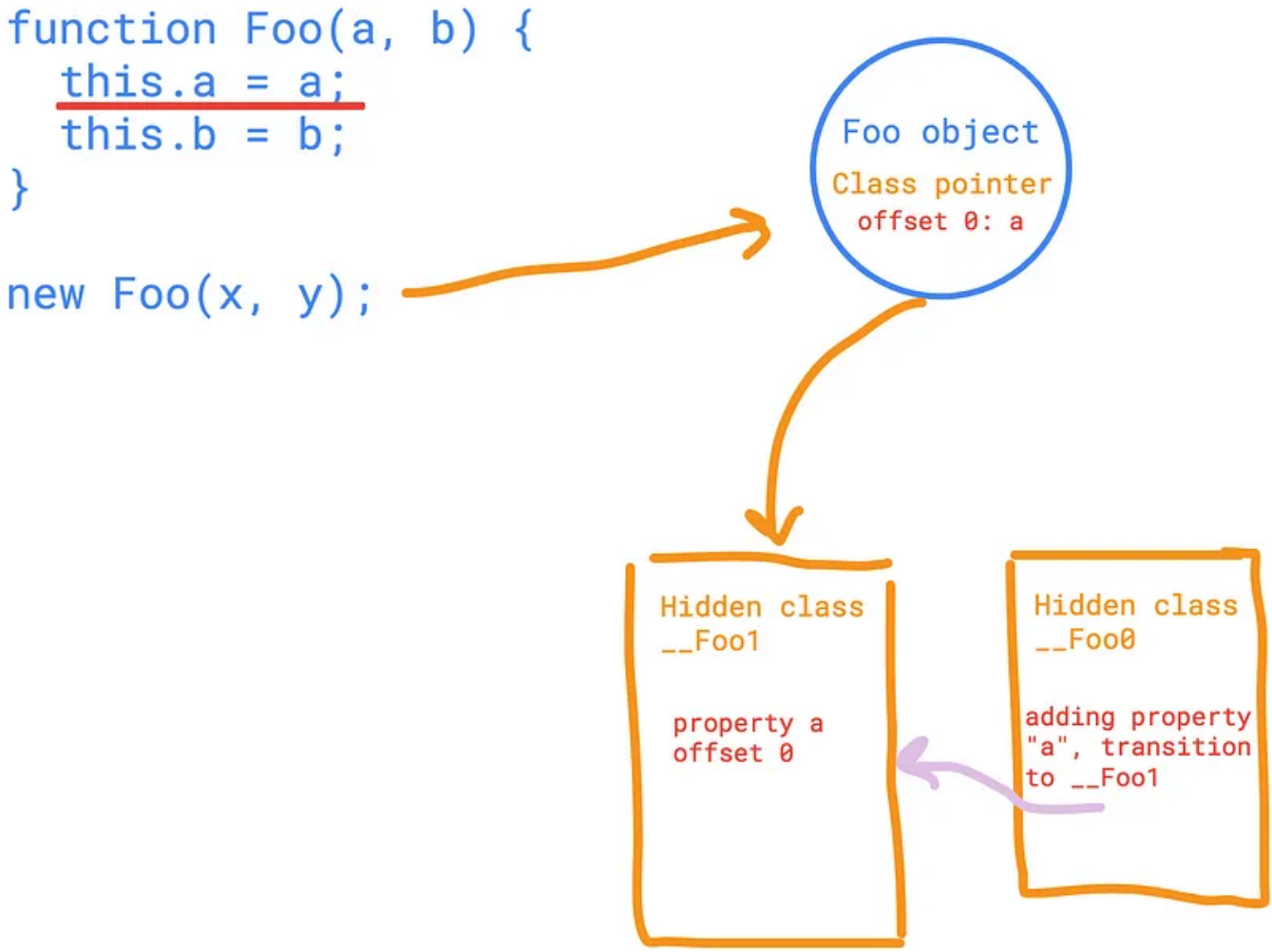
```
函数 Foo(a, b) {
  this.a = a;
  这个.b = b;
}
```

我们看到一个具有两个属性的“结构”，而 V8 看到了.....类中的痛苦。这是因为执行 `new Foo(x, y);` V8 必须创建几个隐藏类。



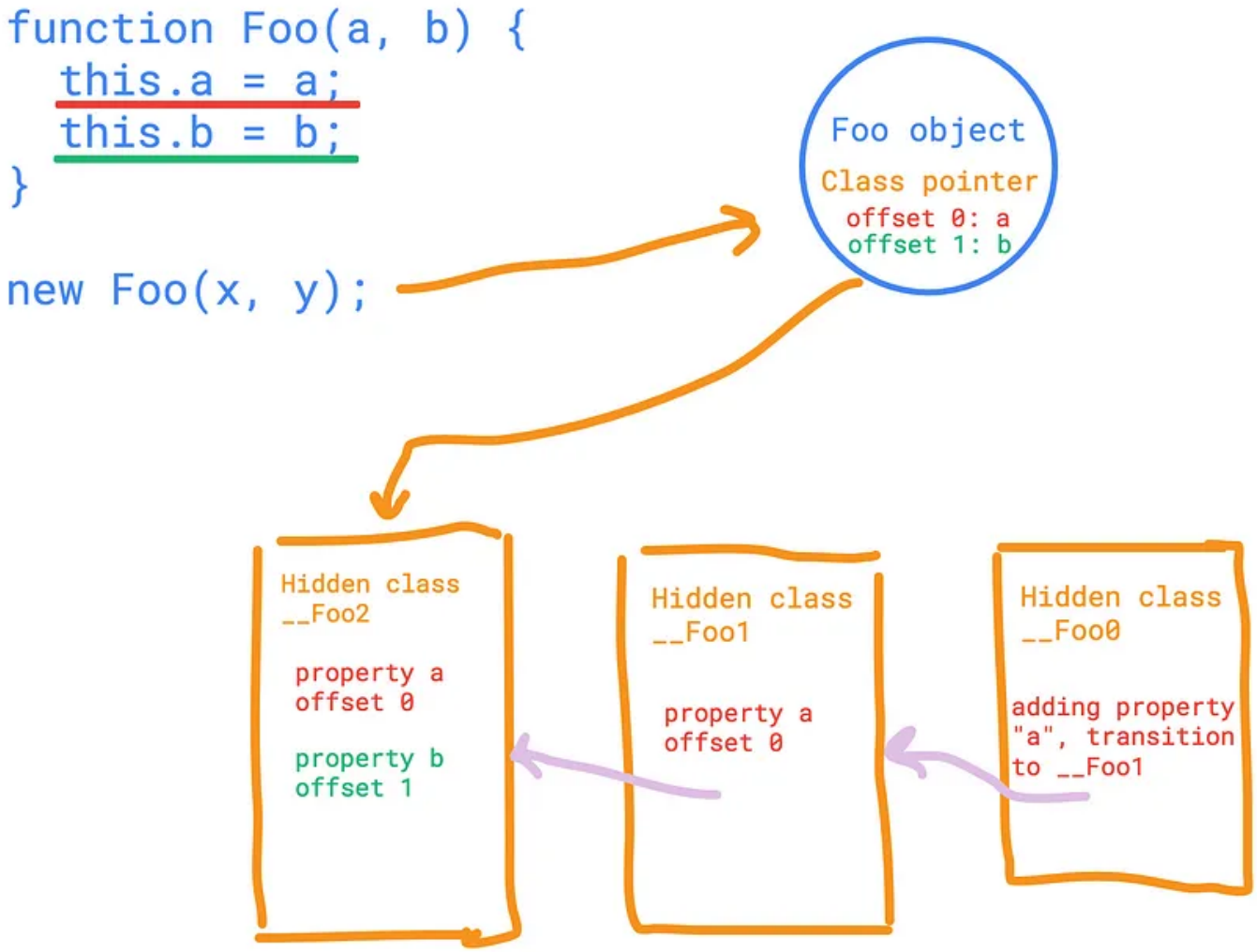
V8为对象创建一个空的隐藏类

当执行 Foo 的第一行时，V8 从隐藏类 __Foo0 转换到隐藏类 __Foo1。



执行第一行将第一个属性添加到对象中，并进行隐藏类转换

相同的逻辑适用于第二行 () 的执行 `this.b = b;` 。



添加第二个属性会进行另一个转换，到 __Foo2

这个概念没问题。当 V8 执行 `new Foo(x, y)` 时，它会创建一个隐藏类。一个空的隐藏类。C++ 中类似这样的东西。

```
结构 __Foo0 {};
```

当V8执行第一行时，例如（创建）分配属性“a”，它会生成另一个隐藏类，并进行所谓的“转换”到__Foo0隐藏类的“较新版本”。

```
struct __Foo1 {
  输入 a; //
};
```

我故意未指定“a”属性的类型（您可能会猜测，根据实际类型，将生成另一个隐藏类，这对 V8 来说很困难，是吧？）。

最后，执行最后一行会产生另一个第三个隐藏类，如下所示。

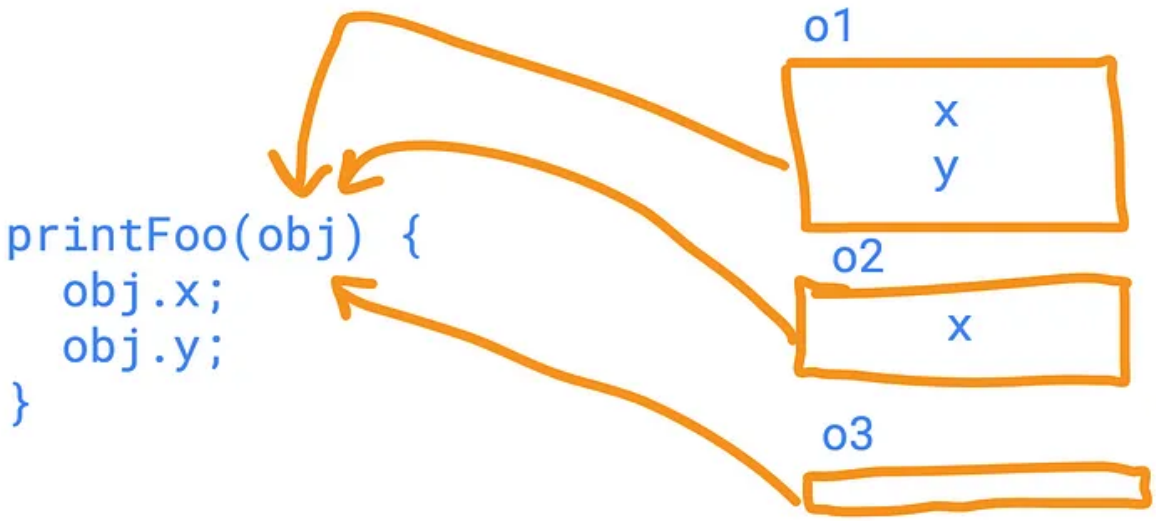
```
struct __Foo2 {
  输入 a; // 我的偏移量为 0，我是第一个属性，是的
  Type b; // 我的偏移量为 1，我是第二个并且不高兴
};
```

为什么要这样做呢？只是因为它比简单的字典查找更快。V8 可以像对待 C++ 对象一样对待我们的 JS 对象，具有适当平滑的内存布局，每个属性都有相应的偏移量。这可能会增加代码大小的一些开销，但确实会使代码运行得更快。至少比以前快了。

隐藏类对于内联缓存也很有帮助，内联缓存是 V8 中实现的另一个“技巧”（在其他地方发明的）。我无法提供有关内联缓存的更多信息，这是我刚刚开始深入研究的主题，但这里有一个示例代码，其中应用了相应内联缓存的热函数会有意义。

```
let o1 = new Foo(x, y);
let o2 = new Foo(x);
let o3 = new Foo();

printFoo(o1);
printFoo(o2);
printFoo(o3);
// printFoo considered as a "hot" function
```



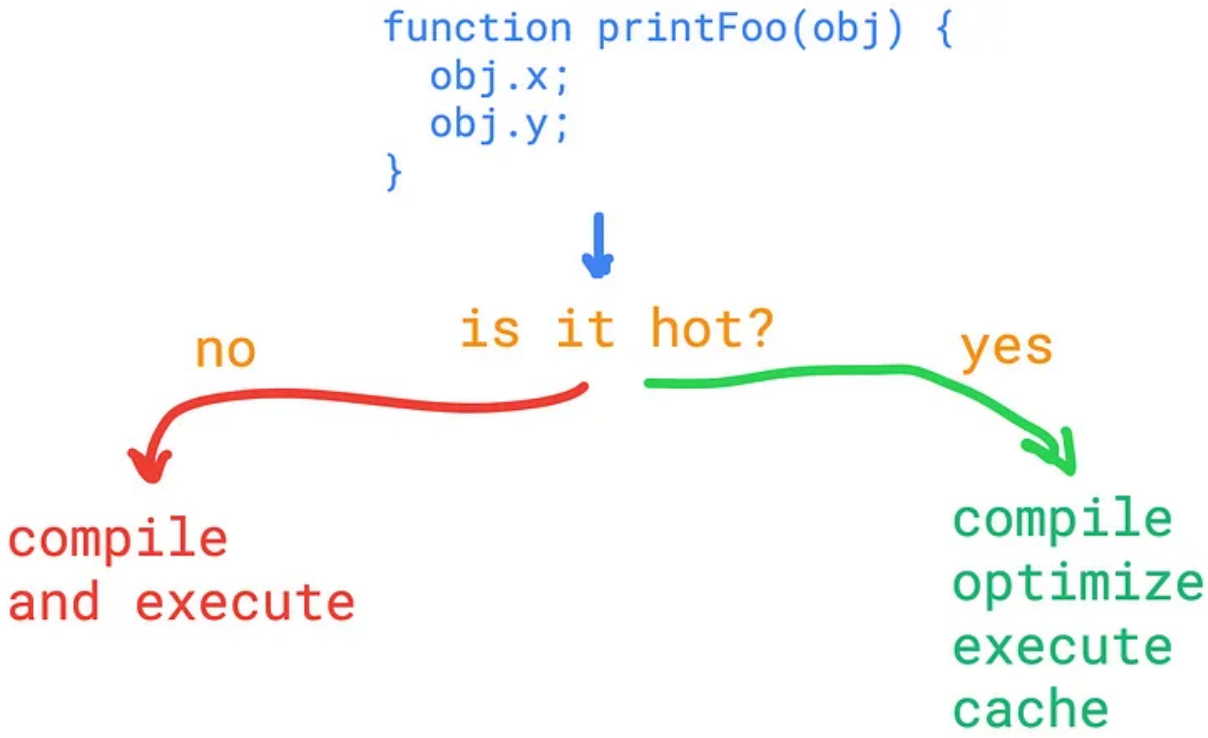
热点函数优化，Inline Caching超抽象介绍

我们有 `printFoo` 函数，采用之前讨论过的 Foo 类型的对象。所以上面的代码调用了 printFoo 3 次。使用相同的“声明”类型的对象，但正如我们已经看到的，具有不同的隐藏类。这里的技巧是 V8 根据输入参数的“种类”来优化函数调用。当一个函数被调用多次（例如超过 2 次）时，V8 开始考虑将其标记为“热”函数。“热”函数应该尽快优化，主要是因为如果该函数被多次调用，并且该函数进行了一些加载，那么该加载将被“应用”与该函数被调用的次数一样多。听起来很简单。因此，通过优化这些“热门”函数，V8 试图减少代码执行时间。实际的优化是通过分析函数调用并收集有关传递参数“种类”的所需数据来实现的。我们通过 `obj` ，然后我们访问 `x` 和 `y` 的属性 `obj` 。所以如果所有传递的对象的隐藏类都相同，V8就会缓存对properties对应内存位置的调用 `x` （ `y` 我这里可能说错了，要警惕）。在上面的代码中，我们传递了不同隐藏类的对象。所以如果我们做这样的事情：

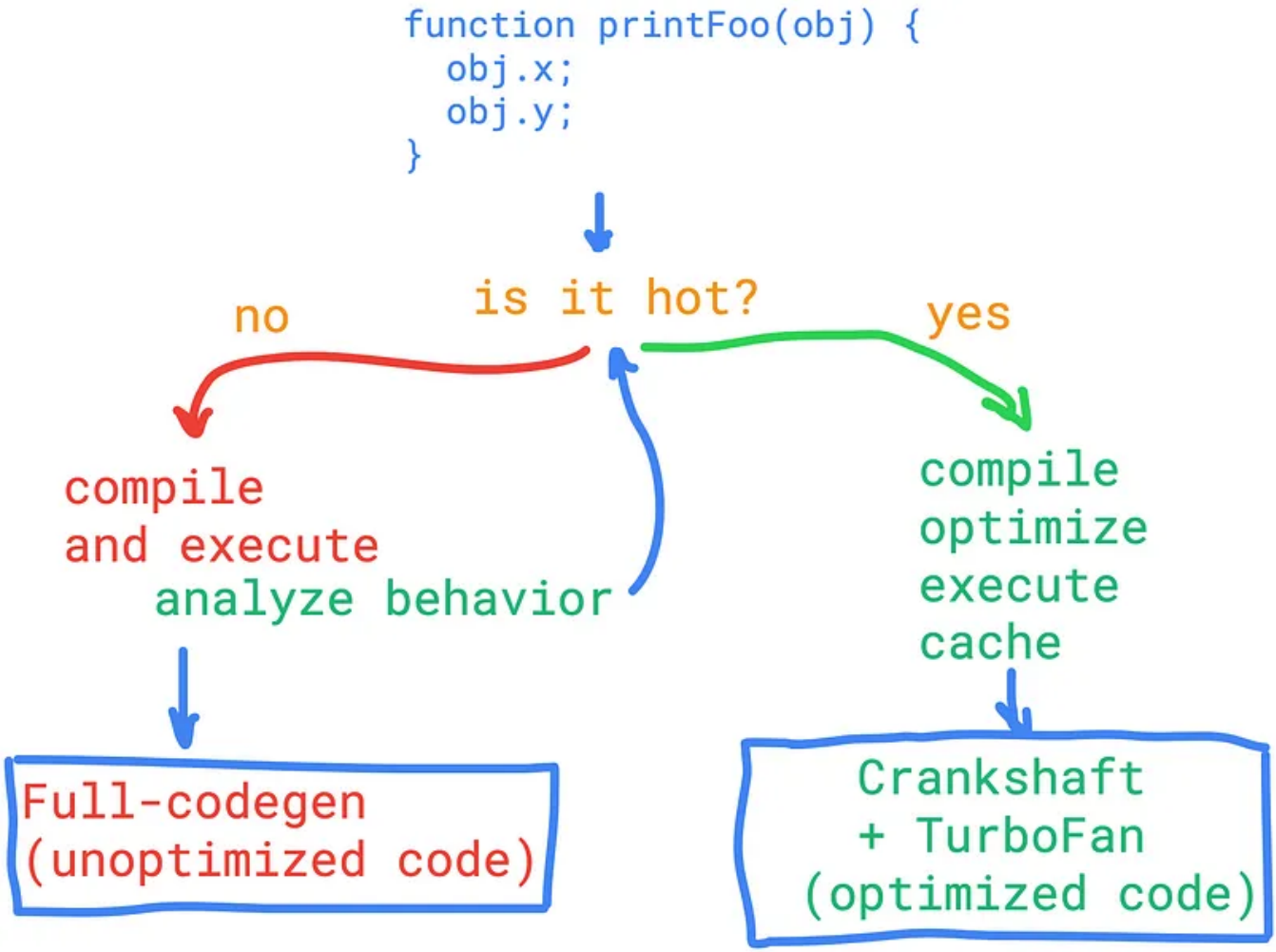
```
printFoo({a: 1, b: 2});
printFoo({a: 2, b: 3});
```

```
printFoo({a: 44, b: 55});
```

那么这些调用将被 V8 成功优化。（这是一些非常人为的例子，但它阐明了这个想法）。



上图显示了简单的想法，但是“热吗？”是如何实现的？检查工作？事实证明，V8 在调用函数时会进行一些行为分析。

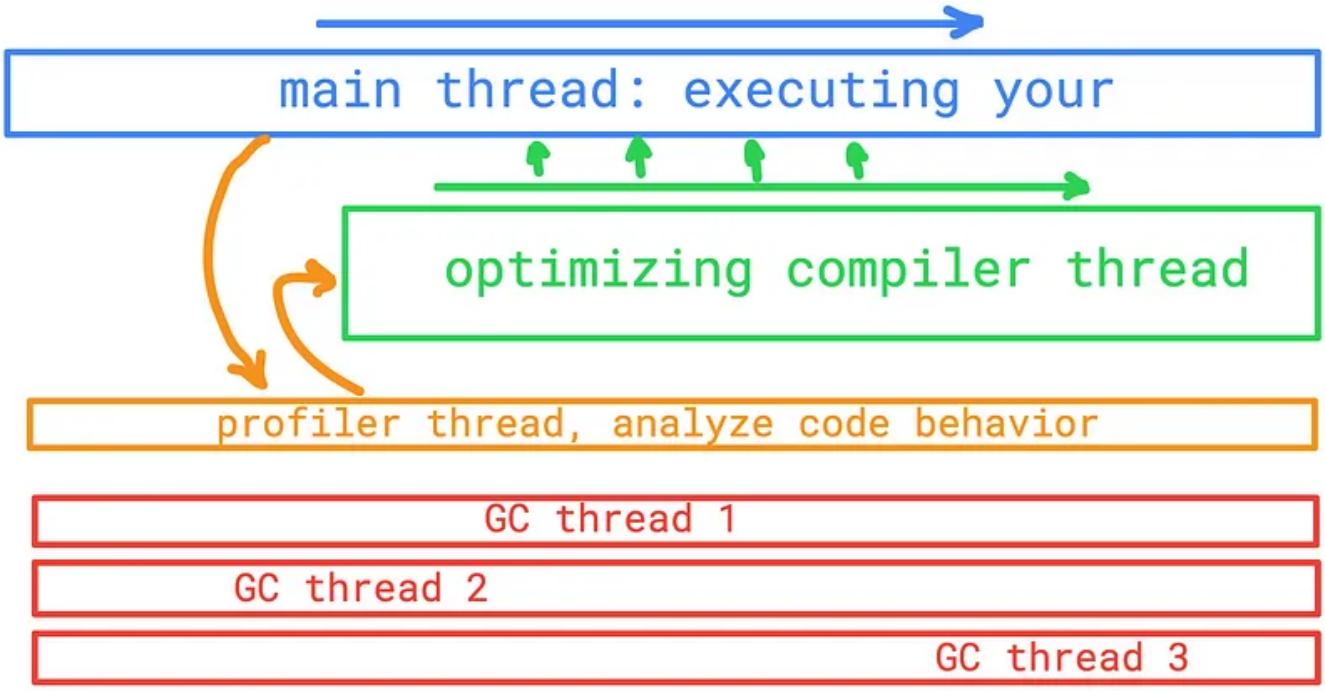


优化与未优化的代码执行

对于每个可疑函数调用，都会更新该特定函数的一些分析数据。在下次打电话“热吗？”之前 检查完成。因此，如果函数很热，那么代码执行就会切换到优化过程。优化是由另一个编译器完成的，它生成优化的机器代码并用优化的代码更新相应的“执行上下文”。

我们在上图中看到，两个不同的编译器有不同的名称。用于未优化机器代码编译器的完整代码生成器和作为优化编译器的 Crankshaft+TurboFan。实际上，它只是 Crankshaft，正如我从 V8 的博客文章中了解到的那样，Crankshaft 不足以扩展。它正在被 TurboFan 取代，并且已经或很快（还不确定）曲轴将从 V8 中完全移除。然后是点火。Ignition 是 V8 的解释器，旨在取代 Full-codegen 未优化的编译器。

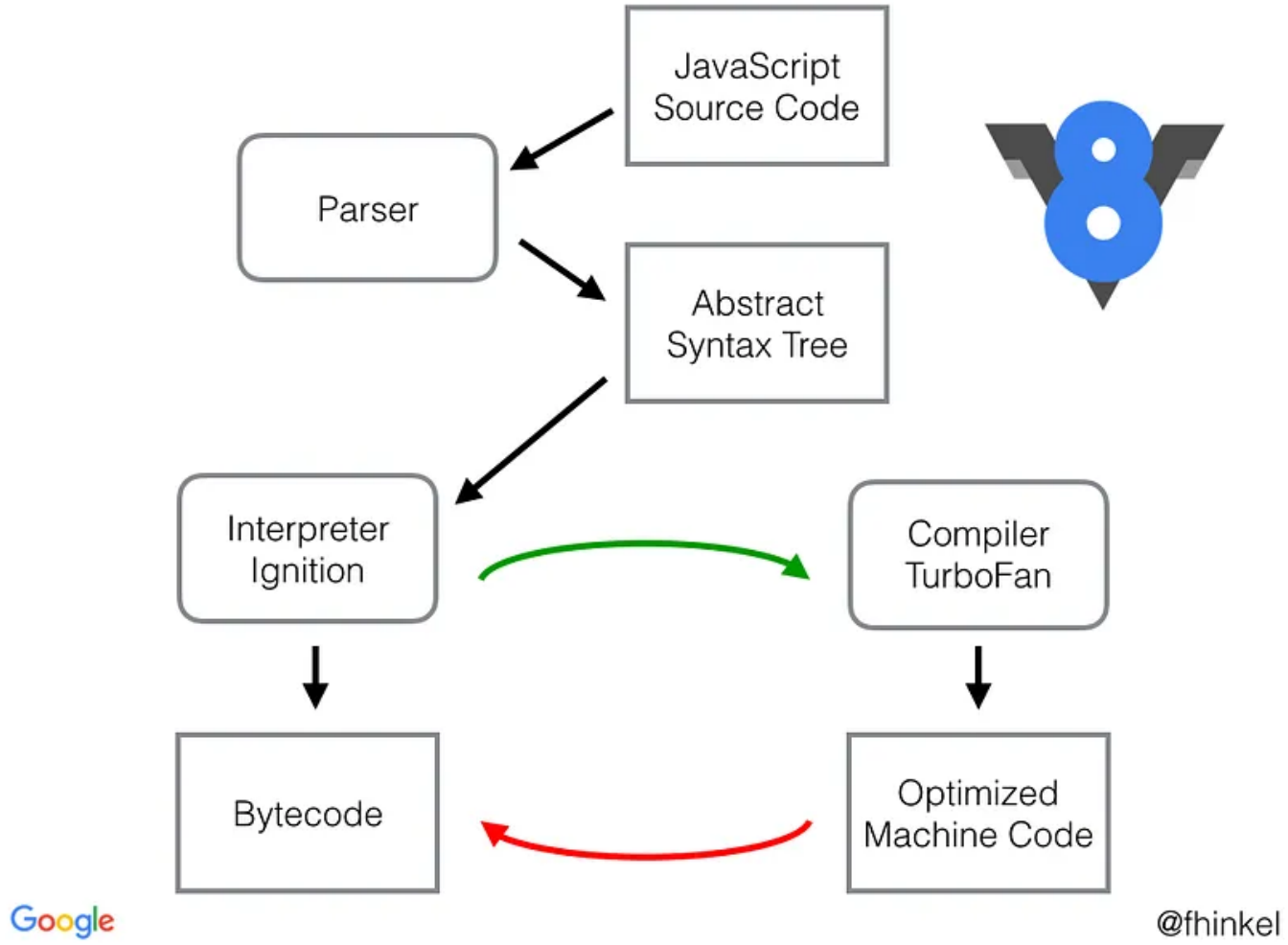
到目前为止，我们得到了这个双编译器的想法。V8 从我们的源 JS 生成一些未优化的机器代码并执行它。在此执行期间，V8 分析代码行为并收集一些元数据。然后，基于此元数据，V8 生成优化的机器代码，使该优化版本作为当前运行的版本。所有提到的这些都无法在单个线程上完成。V8 使用不同的线程来管理这项庞大的工作。



不同线程执行代码

有主线程，用于尽快运行我们的代码。我所说的“尽快”是指需要时间来解析代码，需要时间来生成相应的机器代码，然后主线程开始运行它。除了主线程之外，还有另一个线程，图中标记为“优化编译器线程”。我在插图中添加了“分析器线程”，只是为了使图片完整。还有一些垃圾收集器线程只是为了将所有线程保留在一个插图中。如图所示，主

文章中获取了它）。



V8 编译器管道

如上图所示，解析器将 JS 代码解析为抽象语法树，然后由 Ignition（解释器）进行解释。Ignition 生成能够逐条指令执行的字节码（编译为未优化的机器代码）。在代码执行过程中，TurboFan 会努力处理源代码（与 Ignition 并肩工作）。

网络上有大量信息，其中一些信息非常值得阅读/观看（跟我一起阅读它们，我刚刚开始深入研究 V8 实现）：

v8/v8

v8 - V8 git 存储库的官方镜像

github.com