V8核心模块：解析器，解释器，优化编译器

> **V8是怎么执行JavaScript代码的?**

当V8执行JavaScript源码时，首先解析器会把源码解析为抽象语法树（Abstract Syntax Tree），解释器再将AST翻译为字节码，一遍解释一遍执行。

在此过程中，解释器会记特定代码片段的运行次数，如果代码运行次数超过某个阈值，那么该段代码就被标记为热代码，并将运行信息反馈给优化编译器。

优化编译器根据反馈信息，优化并编译字节码，最终生成优化后的机器码，当该段代码再次执行时，不用再次解释，提升了效率。

这种在运行时编译代码的技术称之为JIT(即时编译)，通过JIT可以极大提升JavaScript代码的执行性能。

简单介绍每个的作用

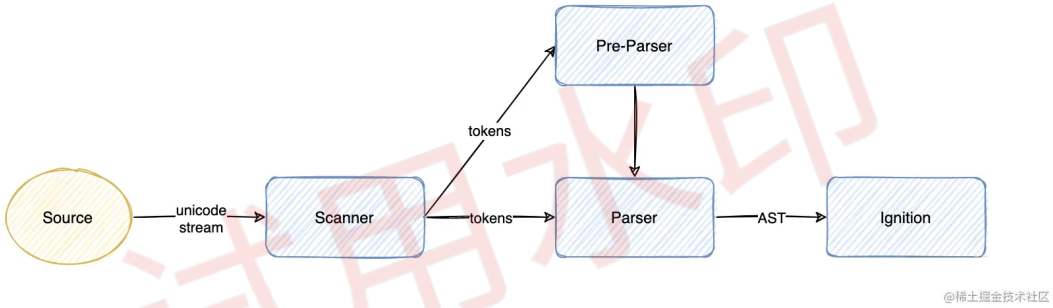
- 解析器将 JavaScript 源码解析为 AST，解析过程分为词法分析和语法分析，V8 通过预解析提升解析效率；
- 解释器 `Ignition` 根据 AST 生成字节码并执行。这个过程中会收集执行反馈信息，交给 TurboFan 进行优化编译；

- `TurboFan` 根据 Ignition 收集的反馈信息，将字节码编译为优化后的机器码，后续 Ignition 用优化机器码代替字节码执行，进而提升性能。

> ### 解析器（Parser）如何把源码转成AST?

让V8执行我们的编写的源码，就要将源码装换成V8能理解的格式。V8会先把源码解析为抽象语法树（AST），这是用来表示源码的树形结构的对象，这个过程称之为**解析（Parsing）**。

解析和编译过程的性能非常重要，V8只有等编译完成后才能运行代码。解析过程如下图：



分为两部分：

1. 词法分析：将字符流转换为tokens，字符流就是我们编写的一行行代码，token是指语法上不能再分割的最小单位（可能是单个字符，也可能是字符串）图中的Scanner就是V8的**词法分析器**。
2. 语法分析：根据语法规则，将tokens组成一个有嵌套层级的AST，在此过程，如果源码不符合语法规范，解析过程就会终止，并抛出语法错误。图中的Parser和Pre-Parse都是V8的**语法分析器**。

词法分析

在V8中，Scanner负责接收Unicode字符流，并将其解析为tokens，提供给解析器使用。比如 var a = 1;，这行代码经过词法分析后的tokens是这样：

```
[
 {
 "type": "Keyword",
 "value": "var"
 },
 {
 "type": "Identifier",
 "value": "a"
 },
 {
 "type": "Punctuator",
 "value": "="
 },
 {
 "type": "Numeric",
 "value": "1"
 },
 {
 "type": "Punctuator",
 "value": ";"
 }
]
```


包含了五个tokens：

- 关键字 var
- 标识符 a
- 赋值运算符 =
- 数值 1
- 分隔符；

语法分析

接下来，V8的解析器会通过语法分析，根据tokens生成AST，var a = 1; 这行代码生成的AST的JSON结构如下所示：

```json
{
 "type": "Program",
 "start": 0,
```

```
  "end": 10,
  "body": [
  {
  "type": "VariableDeclaration",
  "start": 0,
  "end": 10,
 "declarations": [
 {
 "type": "VariableDeclarator",
 "start": 4,
 "end": 9,
 "id": {
 "type": "Identifier",
 "start": 4,
 "end": 5,
 "name": "a"
 },
 "init": {
 "type": "Literal",
 "start": 8,
 "end": 9,
 "value": 1,
 "raw": "1"
 }
 }
 ],
 "kind": "var"
 }
 ],
 "sourceType": "module"
}
```


对生成AST内容感兴趣的可以在 astexplorer.net/ 中查看。

但是，对于一份JavaScript源码，如果所有的源码在执行前都要解析才能执行，那会面临一下问题。

- **代码执行时间变长**：一次性解析所有代码，必然会增加代码的运行时间。
- **消耗更多内存**：解析完的AST，以及根据AST编译后的字节码都会存放在内存中，必然会占用更多内存空间。
- **占用磁盘空间**：编译后的代码会缓存在磁盘上，占用磁盘空间。 所以，现在主流JavaScript引擎都实现了延迟解析（Lazy Parsing）。

延迟解析

延迟解析的思想：在解析过程中，对于不是立即执行的函数，只进行**预解析（Pre Parser）**,只有当函数调用时，才对函数进行全量解析。

进行预解析时，只验证函数语法是否有效，解析函数声明、确定函数作用域。不生成AST，实现预解析的是Pre-Parser解析器。

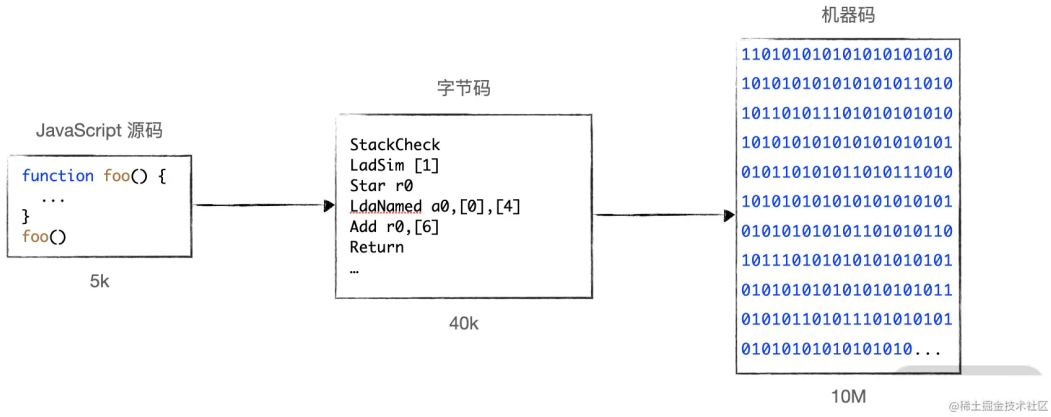
```js
function foo(a, b) {
 var res = a + b;
 return res;
}
var a = 1;
var c = 2;
foo(1, 2);
```

由于 Scanner 是按字节流从上往下一行行读取代码的，所以 V8 解析器也是从上往下解析代码。当 V8 解析器遇到函数声明 foo 时，发现它不是立即执行，所以会用 `Pre-Parser 解析器`对其预解析，过程中只会解析函数声明，不会解析函数内部代码，不会为函数内部代码生成 AST。

然后 `Ignition 解释器`会把 AST 编译为字节码并执行，解释器会按照自上而下的顺序执行代码，先执行 var a = 1; 和 var c = 2; 两个赋值表达式，然后执行函数调用 foo(1, 2) ，这时 Parser 解析器才会继续解析函数内的代码、生成 AST，再交给 `Ignition 解释器`编译执行。

> **解释器（Ignition）如何将 AST翻译为字节码并执行?**

在V8架构的演进中，浅谈V8脚本引擎的工作原理（二.V8脚本引擎的演进）提到的V8为了解决内存占用问题，引入了字节码。如图，通常一个几十KB的文件，转换为机器码可能就是几十兆，这回消耗巨大内存。



什么是字节码

V8的字节码是对机器码的抽象，语法与汇编有些类似，我们可以把V8字节码看做一个个指令，这些指令组合到一起实现我们编写的功能，采用和物理CPU相同的计算模型进行设计。JavaScript源码的任何功能都可以等价转换成字节码的组合。字节码生成其实是遍历树的过程，V8定义了几百个字节码，可以在V8解释器头文件中查看到所有字节码bytecodes.h。

解释器在执行字节码时，主要使用通用寄存器和累加寄存器，函数参数和局部变量都保存在通用寄存器中r0,r1，累加寄存器用于保存中间结果（accumulator register）。

举例说明字节码执行流程。 首先定义一个含有三个形参的函数f，函数功能就是对参数进行计算，并返回值。

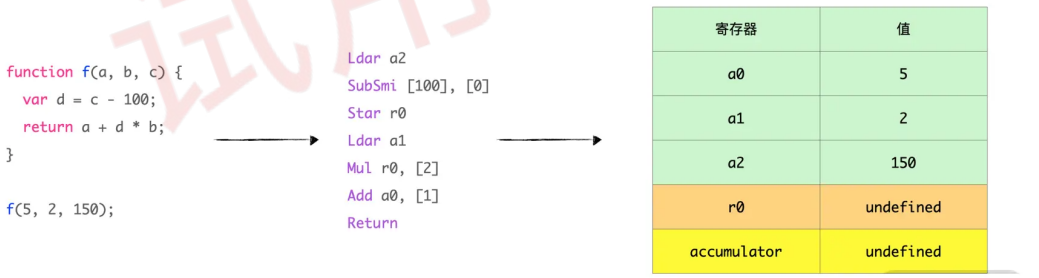
```js
// index.js
function f(a, b, c) {
 var d = c - 100;
 return a + d * b;
}
f(5, 2, 150);
```

假设我们以实参5、2、150调用函数，则解释器会把函数编译为字节码。

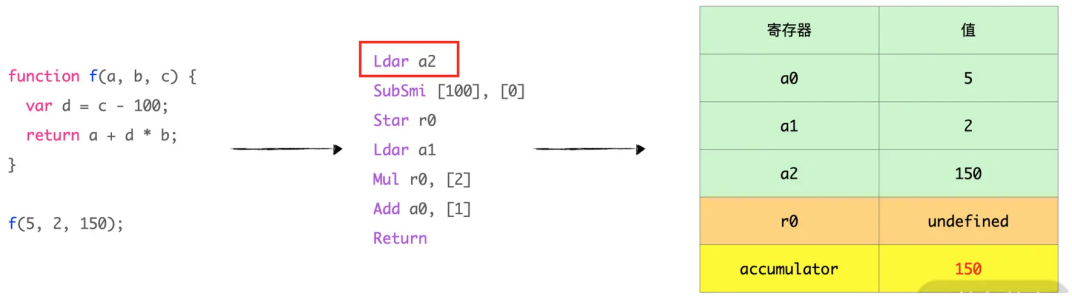
可以通过node --print-bytecode index.js 来查看JavaScript文件生成的字节码。(会生成非常多，取了最后一段重要的)

```
$ node --print-bytecode index.js
...
[generated bytecode for function: f (0x242cd33a35b9 <SharedFunctionInfo f>)]
Parameter count 4
Register count 1
Frame size 8
 32 S> 0x242cd33a3e06 @ 0 : 25 02 Ldar a2
 34 E> 0x242cd33a3e08 @ 2 : 41 64 00 SubSmi [100], [0]
 0x242cd33a3e0b @ 5 : 26 fb Star r0
 43 S> 0x242cd33a3e0d @ 7 : 25 03 Ldar a1
 56 E> 0x242cd33a3e0f @ 9 : 36 fb 02 Mul r0, [2]
 52 E> 0x242cd33a3e12 @ 12 : 34 04 01 Add a0, [1]
 60 S> 0x242cd33a3e15 @ 15 : aa Return
Constant pool (size = 0)
Handler Table (size = 0)
Source Position Table (size = 14)
0x242cd33a3e19 <ByteArray[14]>
```

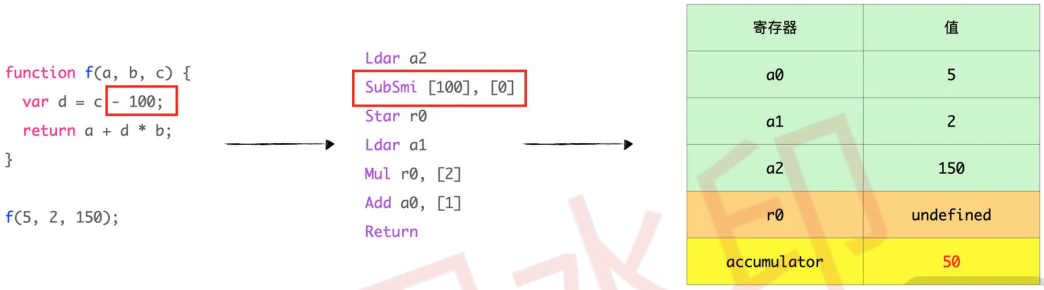
当解释器执行代码时，会把参数分别加载到a0、a1、a2寄存器上（图中accumulator 表示累加寄存器）然后逐行执行字节码。



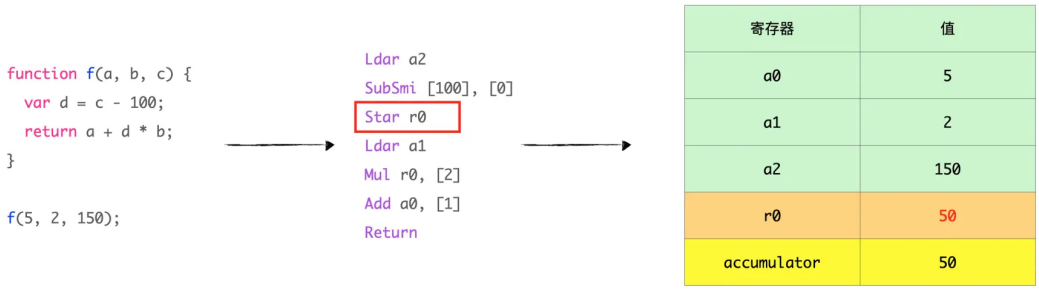
- 读取形参 c 并做出计算
- Ldar a2: Ldar表示将寄存器的值加载到累加器的操作。a2是加载的数值，加载后，accumulator的值为150。



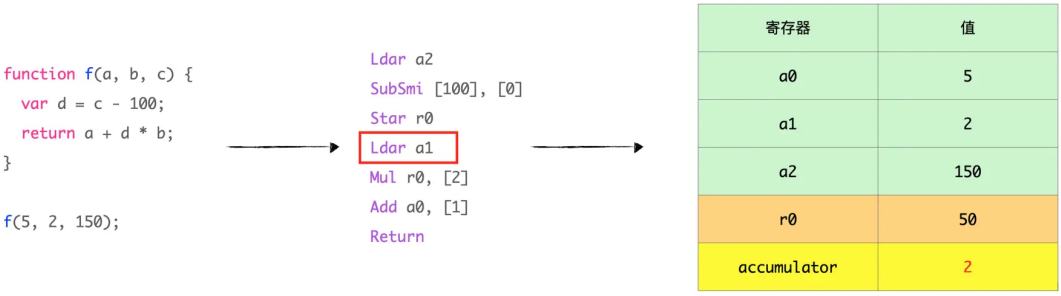
- 将计算结果放入累加寄存器中。
- SubSmi [100], [0]：SubSmi [100] 表示将累加寄存器的值减少 100，这时 accumulator 的值就变为了 50，[0] 反馈向量 (FeedBack Vector) 的索引，反馈向量记录了函数在执行过程中的一些关键的中间数据。



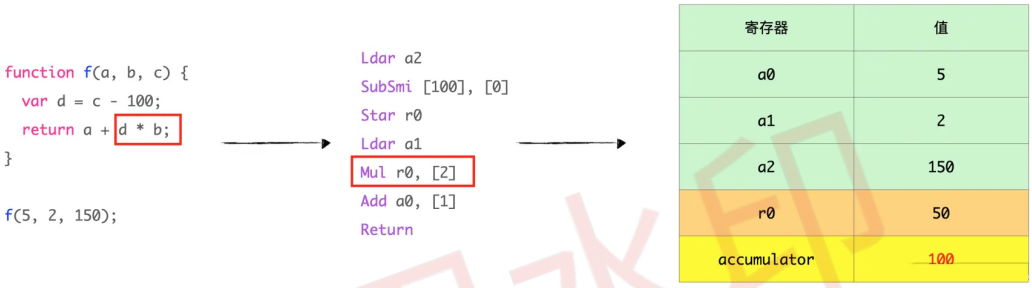
- 将累加寄存器的值放到r0中临时记录 也是变量b的值
- Star r0：表示把累加器中的值保存到寄存器 r0 中，这时 r0 的值就变为了 50。



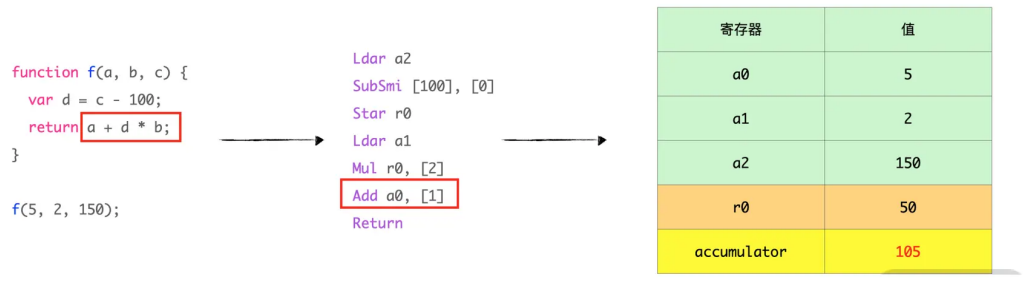
- 读取`a + d * b`语句，先执行`d * b`
- Ldar a1:表示将寄存器a1的值加载到累加寄存器，这时accumulator的值变为2。



- 继续执行`d * b`的第二个动作
- Mul r0,[2]：Mul r0 表示将accumulator的值与 r0 寄存器的值相乘，并把结果再次放入累加寄存器，其中 [2] 同样是反馈向量，执行完毕后，accumulator 的值就变为了 100。



- 执行`a + 100`的动作
- Add a0, [1]：Add a0 表示将累加寄存器的值与 a0 寄存器的值相加，并将结果再次放入累加寄存器，这时 accumulator 的值就变为了 105。



- Return：表示结束当前函数的执行，并返回累加寄存器中的值，函数执行结果是 105。

这是解释器执行字节码的简单过程（中间省略了对AST遍历翻译成的字节码的环节），但依旧需要对字节码转换为机器码，CPU只识别机器码。

看似多了一层字节码的转换感觉效率低了，但相比于机器码，字节码的优势是更方便进行性能优化，最主要是由优化编译器编译热点代码。基于字节码的优化的架构性能远超直接转为机器码的架构性能。

前面提到Ignition解释器在执行的过程中，会标记重复执行的热点代码交给TurboFan生成效率更高的机器码，接下来看看TurboFan是如何工作的。

> ### 优化编译器（TuboFan）的工作原理

V8在提示JavaScript性能方面做了很多优化工作,其中最主要的有内联和逃逸分析两种算法。

内敛（inlining）

```js
function add(x, y) {
 return x + y;
}
function three() {
 return add(1, 2);
}
```

首先我们定义了一个求和函数 `add` ，函数有两个参数 x 和 y，然后定义了函数 three ，并在函数 three 中调用 `add` 函数。

如果不经优化，直接编译该段代码，则会分别生成两个函数的机器码。但为了进一步提升性能，TurboFan优化编译器首先会对以上两个函数进行内联，然后再编译。

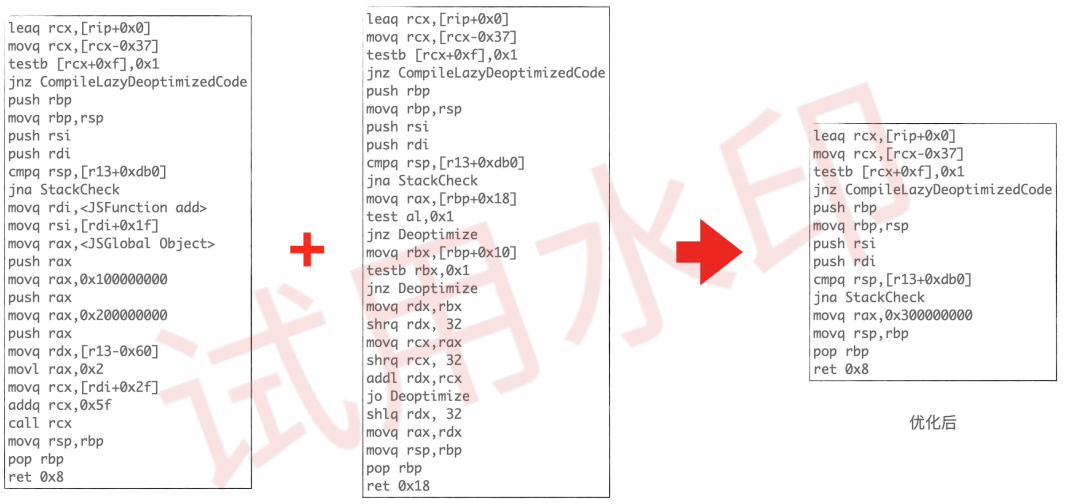
由于函数 `three` 内部的行为就是求 1 和 2 的和，所以上面的代码就等价于下面的：

```js
function three_add_inlined() {
 var x = 1;
```

```
  var y = 2;
  var add_return_value = x + y;
  return add_return_value;
}
```

更进一步，由于函数 `three_add_inlined` 中 x 和 y 的值都是确定的，所以 `three_add_inlined` 还可以进一步优化，直接返回结果 3：

```
function three_add_const_folded() {
  return 3;
}
```



通过内联，可以降低复杂度、消除冗余代码、合并常量，并且内联技术通常也是逃逸分析的基础。那什么又是逃逸分析呢？

逃逸分析（Escape Analysis）

分析对象的生命周期是否仅限于当前函数。

```
class Point {
  constructor(x, y) {
  this.x = x;
  this.y = y;
```

```
  }

  distance(that) {
  return Math.abs(this.x - that.x)
  + Math.abs(this.y - that.y);
  }
}

function manhattan(x1, y1, x2, y2) {
  const a = new Point(x1, y1);
  const b = new Point(x2, y2);
  return a.distance(b);
}

```

我们定义了一个 `Point` 类，用于表示某个点的坐标，类中有个 `distance` 方法，用来计算两个点之前的曼哈顿距离。

然后我们在 `manhattan` 函数中 new 了两个点， a 和 b ，并计算 a b 的曼哈顿距离。`TurboFan` 首先会通过内联，将 `manhattan` 函数转换为下面这样的函数：

```
function manhattan_inlined(x1, y1, x2, y2) {
  const a = {x:x1, y:y1};
  const b = {x:x2, y:y2};
  return Math.abs(a.x - b.x)
  + Math.abs(a.y - b.y);
}

```

再接下来就会对 `manhattan_inlined` 中的对象进行逃逸分析。什么样的对象会被认为是“未逃逸”的呢？主要有以下几个条件：

- 对象在函数内部定义；
- 对象在作用域函数内部，如：没有被返回、没有传递应用给其他函数等。

在 `manhattan_inlined` 中，变量 a b 都是函数内的普通对象，所以它们都是“未逃逸”对象。那么我们就可以对函数中的对象进行替换，使用标量替换掉对象：

```
```

```
function manhattan_scalar_eplacement(x1, y1, x2, y2) {
 var a_x = x1;
 var a_y = y1;
 var b_x = x2;
 var b_y = y2;
 return Math.abs(a_x - b_x)
 + Math.abs(a_y - b_y);
}
```

这样函数内就不再有对象定义，取而代之的是 a_x a_y b_x b_y ，且直接来源于函数参数。

这样做的好处是，我们可以直接将变量加载到寄存器上，不再需要从内存中访问对象属性了，提升了执行效率的同时还减少了内存使用。