21 JavaScript编译器（二）：V8的解释器和优化编译器

你好，我是宫文学。通过前一讲的学习，我们已经了解了V8的整个编译过程，并重点探讨了一个问题，就是**V8的编译速度为什么那么快。**

V8把解析过程做了重点的优化，解析完毕以后就可以马上通过Ignition解释执行了。这就让JavaScript可以快速运行起来。

今天这一讲呢，我们重点来讨论一下，**V8的运行速度为什么也这么快**，一起来看看V8所采用的优化技术。

上一讲我也提及过，V8在2008年刚推出的时候，它提供了一个快速编译成机器码的编译器，虽然没做太多优化，但性能已经是当时其他JavaScript引擎的10倍了。而现在，V8的速度又是2008年刚发布时候的10倍。那么，是什么技术造成了这么大的性能优化的呢？

这其中，**一方面原因，是TurboFan这个优化编译器，采用了很多的优化技术。**那么，它采用了什么优化算法？采用了什么IR？其优化思路跟Java的JIT编译器有什么相同点和不同点？

另一方面，最新的Ignition解释器，虽然只是做解释执行的功能，但竟然也比一个基础的编译器生成的代码慢不了多少。这又是什么原因呢？

所以今天，我们就一起把这些问题都搞清楚，这样你就能全面了解V8所采用的编译技术的特点了，你对动态类型语言的编译，也就能有更深入的了解，并且这也有助于你编写更高效的JavaScript程序。

好，首先，我们来了解一下TurboFan的优化编译技术。

TurboFan的优化编译技术

TurboFan是一个优化编译器。不过它跟Java的优化编译器要完成的任务是不太相同的。因为JavaScript是动态类型的语言，所以如果它能够推断出准确的类型再来做优化，就会带来巨大的性能提升。

同时，TurboFan也会像Java的JIT编译器那样，基于IR来运行各种优化算法，以及在后端做指令选择、寄存器分配等优化。所有的这些因素加起来，才使得TurboFan能达到很高的性能。

我们先来看看V8最特别的优化，也就是通过对类型的推理所做的优化。

基于推理的优化（Speculative Optimazition）

对于基于推理的优化，我们其实并不陌生。在研究Java的JIT编译器时，你就发现了Graal会针对解释器收集的一些信息，对于代码做一些推断，从而做一些激进的优化，比如说会跳过一些不必要的程序分支。

而JavaScript是动态类型的语言，所以对于V8来说，最重要的优化，就是能够在运行时正确地做出类型推断。举个例子来说，假设示例函数中的add函数，在解释器里多次执行的时候，接受的参数都是整型，那么TurboFan就处理整型加法运算的代码就行了。这也就是上一讲中我们生成的汇编代码。

```
function add(a,b){
    return a+b;
}

for (i = 0; i<100000; i++){
    if (i%1000==0)
        console.log(i);

    add(i, i+1);
}
```

```
70  4c8bc7          REX.W movq r8,rdi
73  41d1f8          sarl r8, 1
76  4c8bc9          REX.W movq r9,rcx
79  41d1f9          sarl r9, 1
7c  4503c1          addl r8,r9
```

但是，**如果不在解释器里执行，直接要求TurboFan做编译，会生成什么样的汇编代码呢？**

你可以在运行d8的时候，加上“–always-opt”参数，这样V8在第一次遇到add函数的时候，就会编译成机器码。

```
./d8 --trace-opt-verbose \
    --trace-turbo \
    --turbo-filter=add \
    --print-code \
    --print-opt-code \
    --code-comments \
```

```
    --always-opt \
    add.js
```

这一次生成的汇编代码，跟上一讲生成的就不一样了。由于编译器不知道add函数的参数是什么类型的，所以实际上，编译器是去调用实现Add指令的内置函数，来生成了汇编代码。

这个内置函数当然支持所有加法操作的语义，但是它也就无法启动基于推理的优化机制了。这样的代码，跟解释器直接解释执行，性能上没太大的差别，因为它们本质上都是调用一个全功能的内置函数。

```
55  488b5518        REX.W movq rdx,[rbp+0x18]  <-- 第1个参数，加载到rdx寄存器
59  488b4510        REX.W movq rax,[rbp+0x10]  <-- 第2个参数，加载到rax寄存器
5d  48be510f2408ae180000 REX.W movq rsi,0x18ae08240f51    ;; object: 0x18ae08240f51 <NativeContext[257]>
-- Inlined Trampoline to Add --                <-- 函数的上下文对象，加载到rsi寄存器
67  49ba8086b40a01000000 REX.W movq r10,0x10ab48680  (Add)    ;; off heap target
71  41ffd2          call r10                   <-- Add内置函数的地址，加载到r10寄存器
-- <not inlined:34> --                         <-- 调用Add内置函数
74  488be5          REX.W movq rsp,rbp
77  5d              pop rbp                    <-- 还原栈指针，准备退出函数
78  c21800          ret 0x18                   <-- 退出函数，返回
```

而推理式优化的版本则不同，它直接生成了针对整型数字进行处理的汇编代码：

```
70  4c8bc7          REX.W movq r8,rdi   <-- 第1个参数，加载到r8寄存器。注意，这里是从物理寄存器取参数。
73  41d1f8          sarl r8, 1          <-- 向右移1位。
76  4c8bc9          REX.W movq r9,rcx   <-- 第2个参数，加载到r9寄存器。
79  41d1f9          sarl r9, 1          <-- 向右移1位。
7c  4503c1          addl r8,r9          <-- 把r9加到r8上
```

我来给你解释一下这几行指令的意思：

- 第1行和第3行，是把参数1和参数2分别拷贝到r8和r9寄存器。**注意**，这里是从物理寄存器里取值，而不是像前一个版本一样，在栈里取值。前一个版本遵循的是更加保守和安全的调用约定。
- 第2行和第4行，是把r8和r9寄存器的值向右移1位。
- 第5行，是把r8和r9相加。

看到这里，你可能就发现了一个问题：**只是做个简单的加法而已，为什么要做移位操作呢？**实际上，如果你熟悉汇编语言的话，要想实现上面的功能，其实只需要下面这两行代码就可以了：

```
movq rax, rdi    #把参数1拷贝到rax寄存器
addq rax, rcx    #把参数2加到rax寄存器上，作为返回值
```

那么，多出来的移位操作是干什么的呢？

这就涉及到了V8的**内存管理机制**。原来，V8对象都保存在堆中。在栈帧中保存的数值，都是指向堆的指针。垃圾收集器可以通过这些指针，知道哪些内存对象是不再被使用的，从而把它们释放掉。我们前面学过，Java的虚拟机和Python对于对象引用，本质上也是这么处理的。

但是，这种机制对于基础数据类型，比如整型，就不太合适了。因为你没有必要为一个简单的整型数据在堆中申请内存，这样既浪费内存，又降低了访问效率，V8需要访问两次内存才能读到一个整型变量的值（第一次读到地址，第二次根据该地址到堆里读到值）。你记得，Python就是这么访问基础数据的。

V8显然不能忍受这种低效的方式。它采用的优化机制，是一种被广泛采用的技术，叫做**标记指针**（Tagged Pointer）或者**标记值**（Tagged Value）。《Pointer Compression in V8》这篇文章，就介绍了V8中采用Tagged Pointer技术的细节。

比如说，对于一个32位的指针值，它的最低位是标记位。当标记位是0的时候，前31位是一个短整数（简写为Smi）；而当标记位是1的时候，那么前31位是一个地址。这样，V8就可以用指针来直接保存一个整数值，用于计算，从而降低了内存占用，并提高了运行效率。

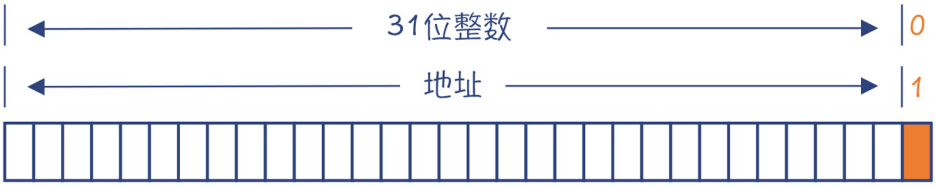


图1：用标记指针技术来表示一个整数

好了，现在你已经理解了V8的推理式编译的机制。那么，还有什么手段能提升代码的性能呢？

当然，还有基于IR的各种优化算法。

IR和优化算法

在讲Java的JIT编译器时我就提过，V8做优化编译时采用的IR，也是基于Sea of Nodes的。

你可以回忆一下Sea of Nodes的特点：合并了数据流图与控制流图；是SSA形式；没有把语句划分成基本块。

它的重要优点，就是优化算法可以自由调整语句的顺序，只要不破坏数据流图中的依赖关系。在Sea of Nodes中，没有变量（有时也叫做寄存器）的概念，只有一个个数据节点，所以对于死代码删除等优化方法来说，它也具备天然的优势。

说了这么多，那么要如何查看TurboFan的IR呢？一个好消息是，V8也像GraalVm一样，提供了一个图形化的工具，来查看TurboFan的IR。这个工具是**turbolizer**，它位于V8源代码的

tools/turbolizer目录下。你可以按照该目录下的README.md文档，构建一下该工具，并运行它。

```
python -m SimpleHTTPServer 8000
```

它实际启动了一个简单的Web服务。你可以在浏览器中输入“0.0.0.0:8000”，打开turbolizer的界面。

在运行d8的时候，如果带上参数“–trace-turbo”，就会在当前目录下输出一个.json文件，打开这个文件就能显示出TurboFan的IR来。比如，上一讲的示例程序add.js，所显示出的add函数的IR：

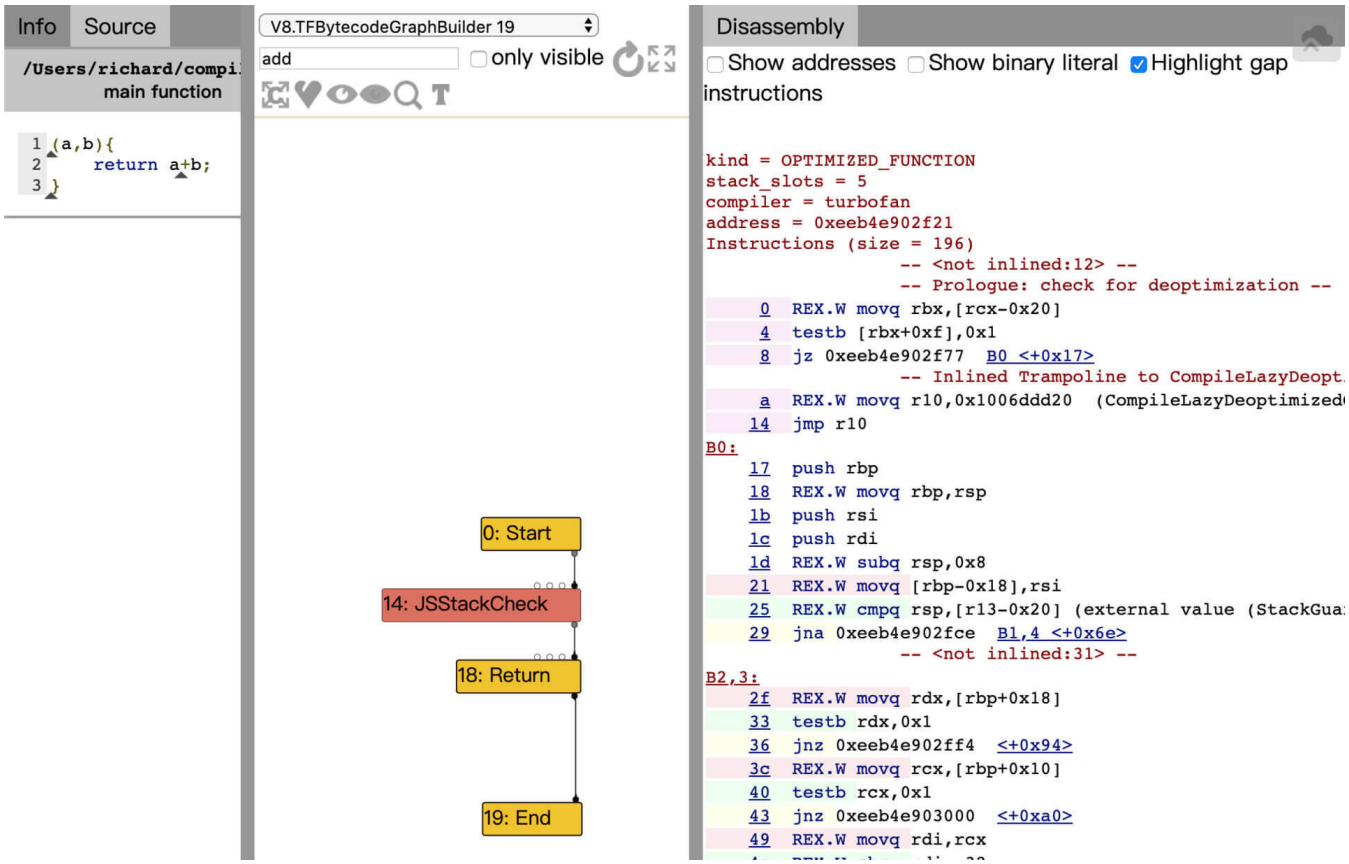


图2：在turbolizer中显示add函数的IR

界面中最左边一栏是源代码，中间一栏是IR，最右边一栏是生成的汇编代码。

上图中的IR只显示了控制节点。你可以在工具栏中找到一个按钮，把所有节点都呈现出来。在左侧的Info标签页中，还有一些命令的快捷键，你最好熟悉一下它们，以便于控制IR都显示哪些节点。这对于一个大的IR图来说很重要，否则你会看得眼花缭乱：

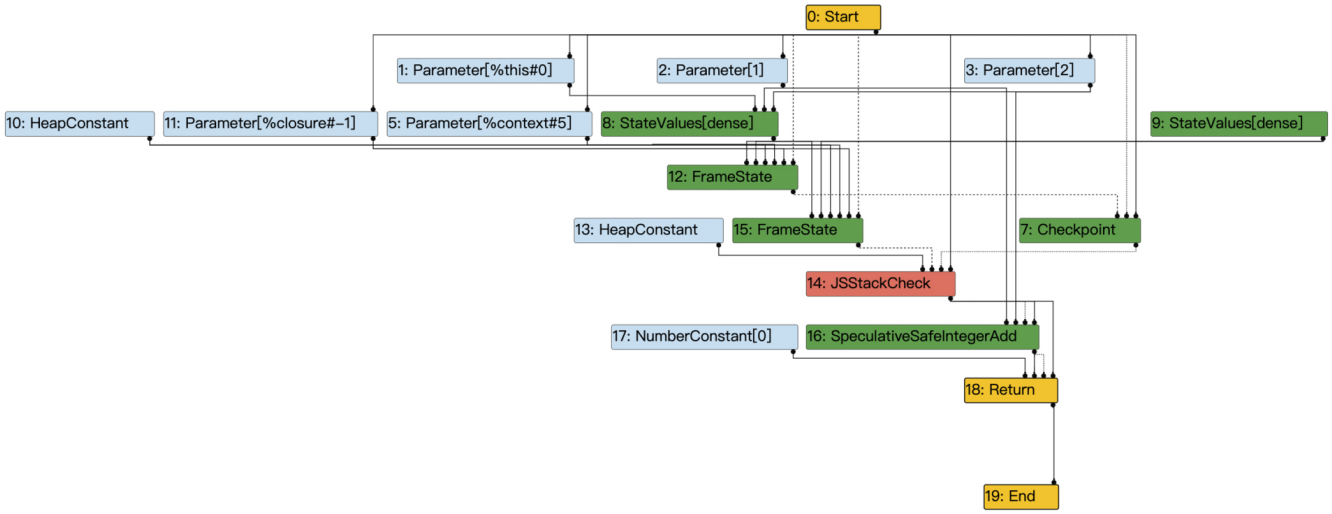


图3：完整展开的add函数的IR

在这个图中，不同类型的节点用了不同的颜色来区分：

- 黄色：控制流的节点，比如Start、End和Return；
- 淡蓝色：代表一个值的节点；
- 红色：JavaScript层级的操作，比如JSEqual、JSToBoolean等；
- 深蓝色：代表一种中间层次的操作，介于JavaScript层和机器层级中间；
- 绿色：机器级别的语言，代表一些比较低层级的操作。

在turbolizer的界面上，还有一个下拉菜单，里面有多个优化步骤。你可以挨个点击，看看经过不同的优化步骤以后，IR的变化是什么样子的。

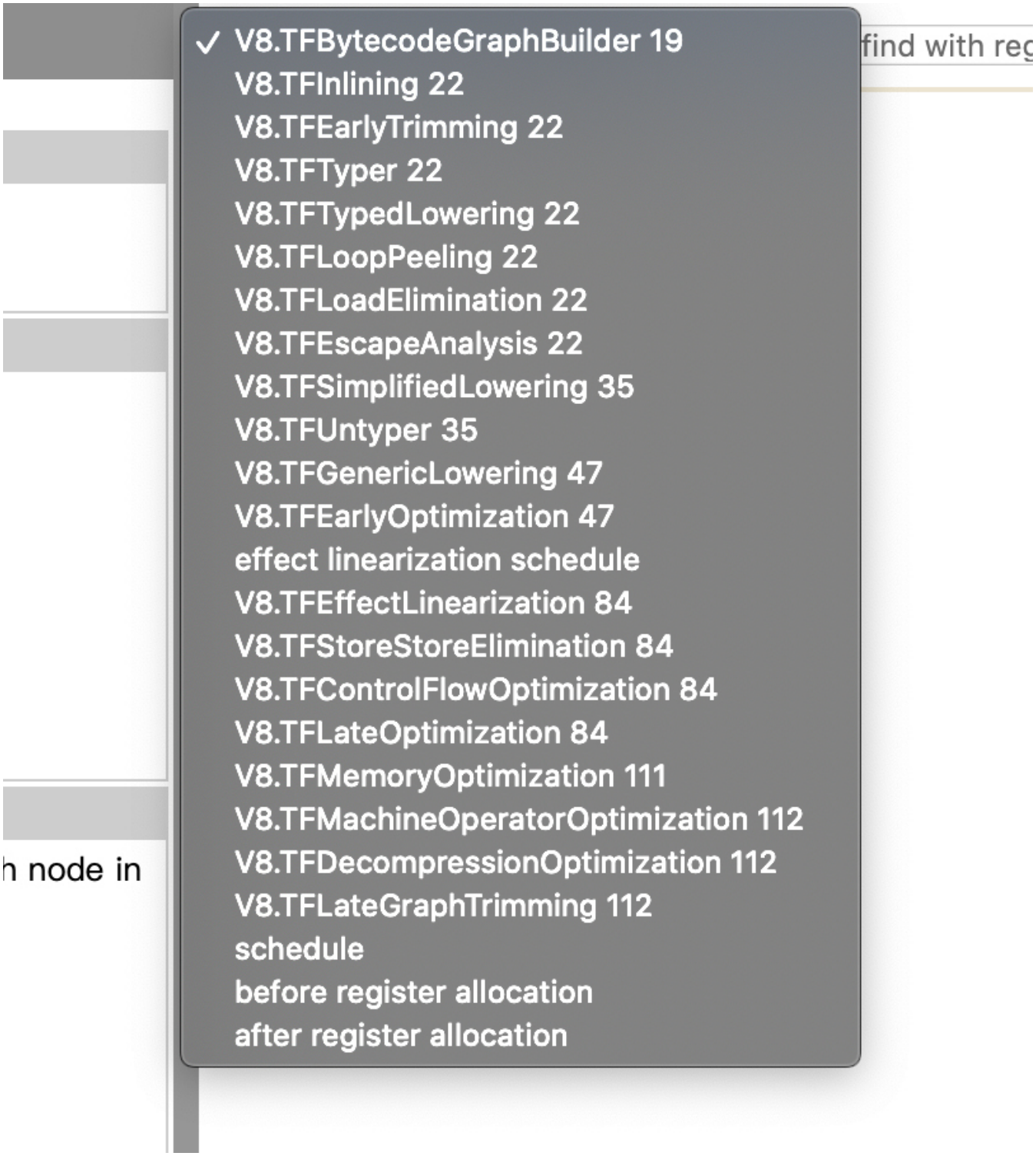


图4：TurboFan对IR的处理步骤

你可以看到，在第一步“v8.TFBytecodeGraphBuilder”阶段显示的IR中，它显示的节点还是有点儿多。我们先隐藏掉与计算功能无关的节点，得到了下面的主干。**你要注意其中的绿色节点**，这里已经进行了类型推测，因此它采用了一个整型计算节点：SpeculativeSafeIntegerAdd。

这个节点的功能是：当两个参数都是整数的时候，那就符合类型推断，做整数加法操作；如果不符合类型推断，那么就执行逆优化。

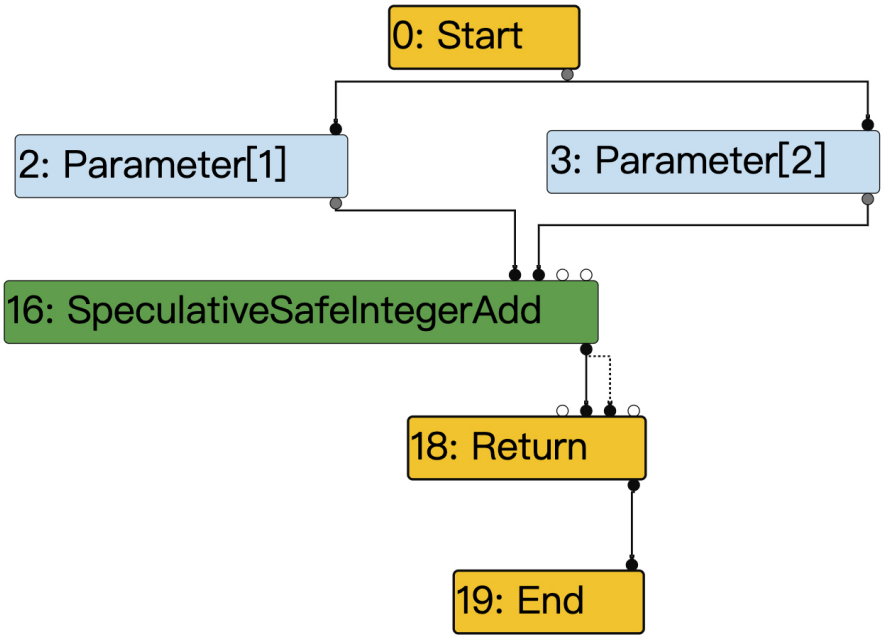


图5：v8.TFBytecodeGraphBuilder”阶段的IR

你可以再去点击其他的优化步骤，看看图形会有什么变化。

在v8.TFGenericLowering阶段，我们得到了如下所示的IR图，这个图只保留了计算过程的主干。里面增加了两个绿色节点，这两个节点就是把标记指针转换成整数的；还增加了一个深蓝色的节点，这个节点是在函数返回之前，把整数再转换成标记指针。

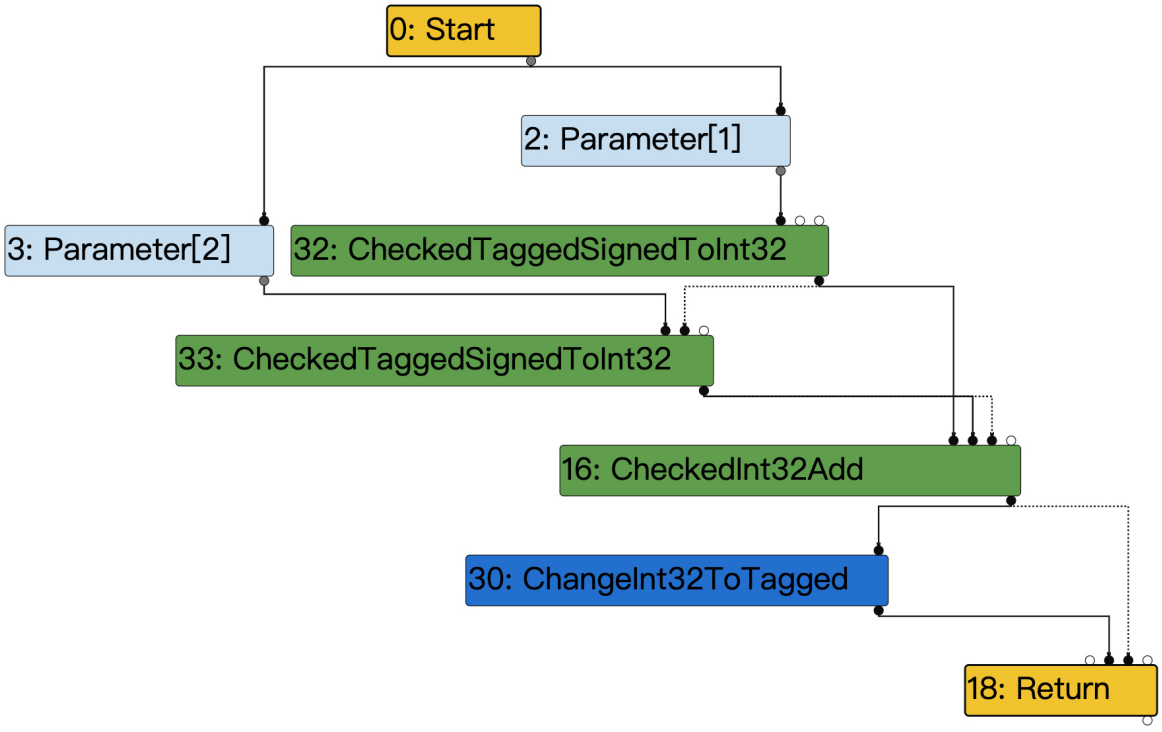


图6：v8.TFGenericLowering阶段的IR

在v8.TFLateGraphTrimming阶段，图中的节点增加了更多的细节，它更接近于具体CPU架构的汇编代码。比如，我们把前面图6中的标记指针，转换成32位整数的操作，就变成了两个节点：

- TruncateInt64ToInt32：把64位整型截短为32位整型；
- Word32Sar：32位整数的移位运算，用于把标记指针转换为整数。

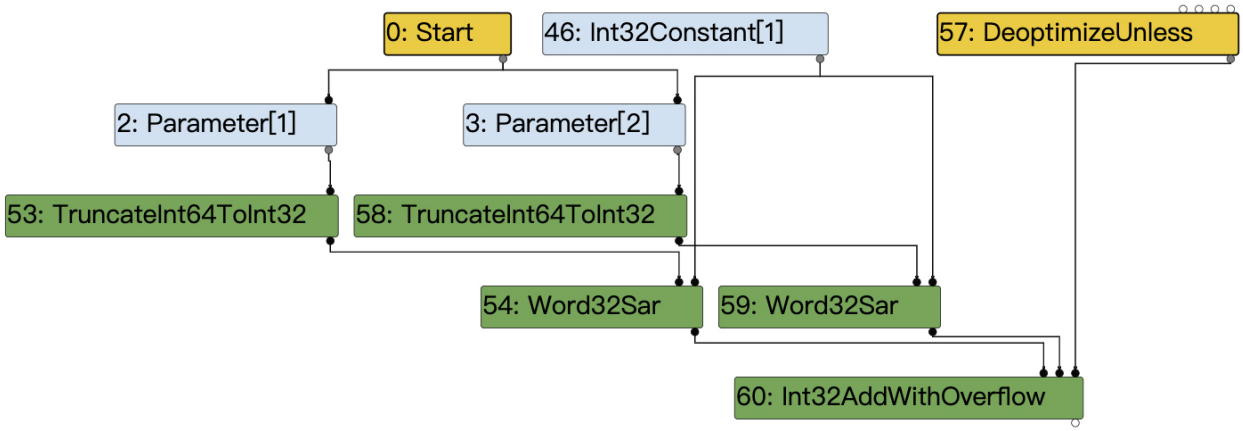


图7：v8.TFLateGraphTrimming阶段的IR

这三个阶段就形象地展示出了TurboFan的IR是如何Lower的，从比较抽象的、机器无关的节点，逐步变成了与具体架构相关的操作。

所以，基本上IR的节点可以分为四类：顶层代表复杂操作的JavaScript节点、底层代表简单操作的机器节点、处于二者之间做了一定简化的节点，以及可以被各个层次共享的节点。

刚才我们对V8做优化编译时，所采用的IR的分析，只关注了与加法计算有关的主干节点。你还可以用同样的方法，来看看其他的节点。这些节点主要是针对异常情况的处理。比如，如果发现参数类型不是整型，那么就要去执行逆优化。

在做完了所有的优化之后，编译器就会进入指令排序、寄存器分配等步骤，最后生成机器码。

现在，你就了解了TurboFan是如何借助Sea of Nodes来做优化和Lower的了。但我们还没有涉及具体的优化算法。**那么，什么优化算法会帮助V8提升性能呢?**

前面在研究Java的JIT编译器的时候，我们重点关注了内联和逃逸分析的优化算法。那么，对于JavaScript来说，这两种优化也同样非常重要，一样能带来巨大的优化效果。

我们先来看看**内联优化**。对于之前的示例程序，由于我们使用了“–turbo-filter=add”选项来运行代码，因此TurboFan只会编译add方法，这就避免了顶层函数把add函数给内联进去。而如果你去掉了这个选项，就会发现TurboFan在编译完毕以后，程序后面的运行速度会大大加

快，一闪而过。这是因为整个顶层函数都被优化编译了，并且在这个过程中，把add函数给内联进去了。

然后再说说**逃逸分析**。V8运用逃逸分析算法，也可以像Java的编译器一样，把从堆中申请的内存优化为从栈中申请（甚至使用寄存器），从而提升性能，并避免垃圾收集带来的消耗。

不过，JavaScript和Java的对象体系设计毕竟是不一样的。在Java的类里，每个成员变量相对于对象指针的偏移量都是固定的；而JavaScript则在内部用了**隐藏类**来表示对象的内存布局。这也引出V8的另一个有特色的优化主题：**内联缓存**。

那接下来，我就带你详细了解一下V8的隐藏类和内联缓存机制。

隐藏类（Shapes）和内联缓存（Inline Caching）

隐藏类，学术上一般叫做Hidden Class，但不同的编译器的叫法也不一样，比如Shapes、Maps，等等。

隐藏类有什么用呢？你应该知道，在JavaScript中，你不需要先声明一个类，才能创建一个对象。你可以随时创建对象，比如下面的示例程序中，就创建了几个表示坐标点的对象：

```
point1 = {x:2, y:3};
point2 = {x:4, y:5};
point3 = {y:7, x:6};
point4 = {x:8, y:9, z:10};
```

那么，V8在内部是怎么来存储x和y这些对象属性的呢？

如果按照Java的内存布局方案，一定是在对象头后面依次存放x和y的值；而如果按照Python的方案，那就需要用一个字典来保存不同属性的值。但显然用类似Java的方案更加节省内存，访问速度也更快。

所以，V8内部就采用了隐藏类的设计。如果两个对象，有着相同的属性，并且顺序也相同，那么它们就对应相同的隐藏类。

在上面的程序中，point1和point2就对应着同一个隐藏类；而point3和point4要么是属性的顺序不同，要么是属性的数量不同，对应着另外的隐藏类。

所以在这里，你就会得到一个**编写高性能程序的要点**：对于相同类型的对象，一定要保证它们的属性数量和顺序完全一致，这样才会被V8当作相同的类型，从而那些基于类型推断的优化才会发挥作用。

此外，V8还用到了一种叫做**内联缓存**的技术，来**加快对象属性的访问时间**。它的原理是这样的：当V8第一次访问某个隐藏类的属性的时候，它要到隐藏类里，去查找某个属性的地址相对于对象指针的偏移量。但V8会把这个偏移量缓存下来，这样下一次遇到同样的shape的时候，直接使用这个缓存的偏移量就行了。

比如下面的示例代码，如果对象o是上面的point1或point2，属性x的地址偏移量就是相同的，因为它们对应的隐藏类是一样的：

```
function getX(o){
  return o.x;
}
```

有了内联优化技术，那么V8只有在第一次访问某个隐藏类的属性时，速度会慢一点，之后的访问效率就跟Java的差不多了。因为Java这样的静态类型的代码，在编译期就可以确定每个属性相对于对象地址的偏移量。

好，现在你已经了解了TurboFan做优化的一些关键思路。接下来，我们再返回来，重新探讨一下Ignition的运行速度问题。

提升Ignition的速度

最新版本的V8已经不需要多级的编译器了，只需要一个解释器（Ignition）和一个优化编译器（TurboFan）就行。在公开的测试数据中，Ignition的运行速度，已经接近一个基线编译器生成的机器码的速度了，也就是那种没有做太多优化的机器码。

这听上去似乎不符合常理，毕竟，解释执行怎么能赶得上运行机器码呢？所以，这里一定有一些值得探究的技术原理。

让我们再来看看Ignition解释执行的原理吧。

在上一讲中你已经了解到，V8的字节码是很精简的。比如，对于各种加法操作，它只有一个Add指令。

但是我们知道，Add指令的语义是很复杂的，比如在ECMAScript标准中，就对加法的语义有很多的规定，如数字相加该怎么做、字符串连接该怎么做，等等。

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Let *lstr* be ? ToString(*lprim*).
    b. Let *rstr* be ? ToString(*rprim*).
    c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumeric(*lprim*).
9. Let *rnum* be ? ToNumeric(*rprim*).
10. If Type(*lnum*) is different from Type(*rnum*), throw a **TypeError** exception.
11. Let *T* be Type(*lnum*).
12. Return *T*::add(*lnum*, *rnum*).

图8：ECMAScript标准中加法操作的语义规则

这样的话，**V8在解释执行Add指令的时候，就要跳到一个内置的函数去执行，其他指令也是如此。这些内置函数的实现质量，就会大大影响解释器的运行速度。**

那么如果换做你，你会怎么实现这些内置函数呢？

选择1：用汇编语言去实现。这样，我们可以针对每种情况写出最优化的代码。但问题是，这样做的工作量很大。因为V8现在已经支持了9种架构的CPU，而要为每种架构编写这些内置功能，都需要敲几万行的汇编代码。

选择2：用C++去实现。这是一个不错的选择，因为C++代码最后编译的结果也是很优化的。不过这里也有一个问题：C++有它自己的调用约定，跟V8中JavaScript的调用约定是不同的。

比如，在调用C++的内置函数之前，解释器要把自己所使用的物理寄存器保护起来，避免被C++程序破坏，在调用完毕以后还要恢复。这使得从解释器到内置函数，以及从内置函数回到解释器，都要做不少的转换工作。你还要写专门的代码，来对标记指针进行转换。而如果要使

用V8的对象，那要处理的事情就更多了，比如它要去隐藏类中查找信息，以及能否通过优化实现栈上内存分配，等等。

那么，我们还有别的选择吗？

有的。你看，V8已经有了一个不错的优化编译器TurboFan。**既然它能产生很高效的代码，那么我们为什么不直接用TurboFan来生成机器码呢？**这个思路其实是可行的。这可以看做是V8编译器的一种自举能力，用自己的编译器，来生成自己内部要使用的内置函数的目标代码。

毕竟，TurboFan本来就是要处理标记指针、JavaScript对象的内存表示等这些问题。这个方案还省去了做调用约定的转换的工作，因为本来V8执行的过程中，就要不断在解释执行和运行机器码之间切换，V8内部对栈桢和调用约定的设计，就是要确保这种切换的代价最低。

在具体实现的时候，编写这些内置函数是用JavaScript调用TurboFan提供的一些宏。这些宏可以转化为TurboFan的IR节点，从而跟TurboFan的优化编译功能无缝衔接。

好了，分析到这里，你就知道为什么Ignition的运行速度会这么快了：**它采用了高度优化过的内置函数的实现，并且没有调用约定转换的负担。**而一个基线编译器生成的机器码，因为没有经过充分的优化，反倒并没有那么大的优势。

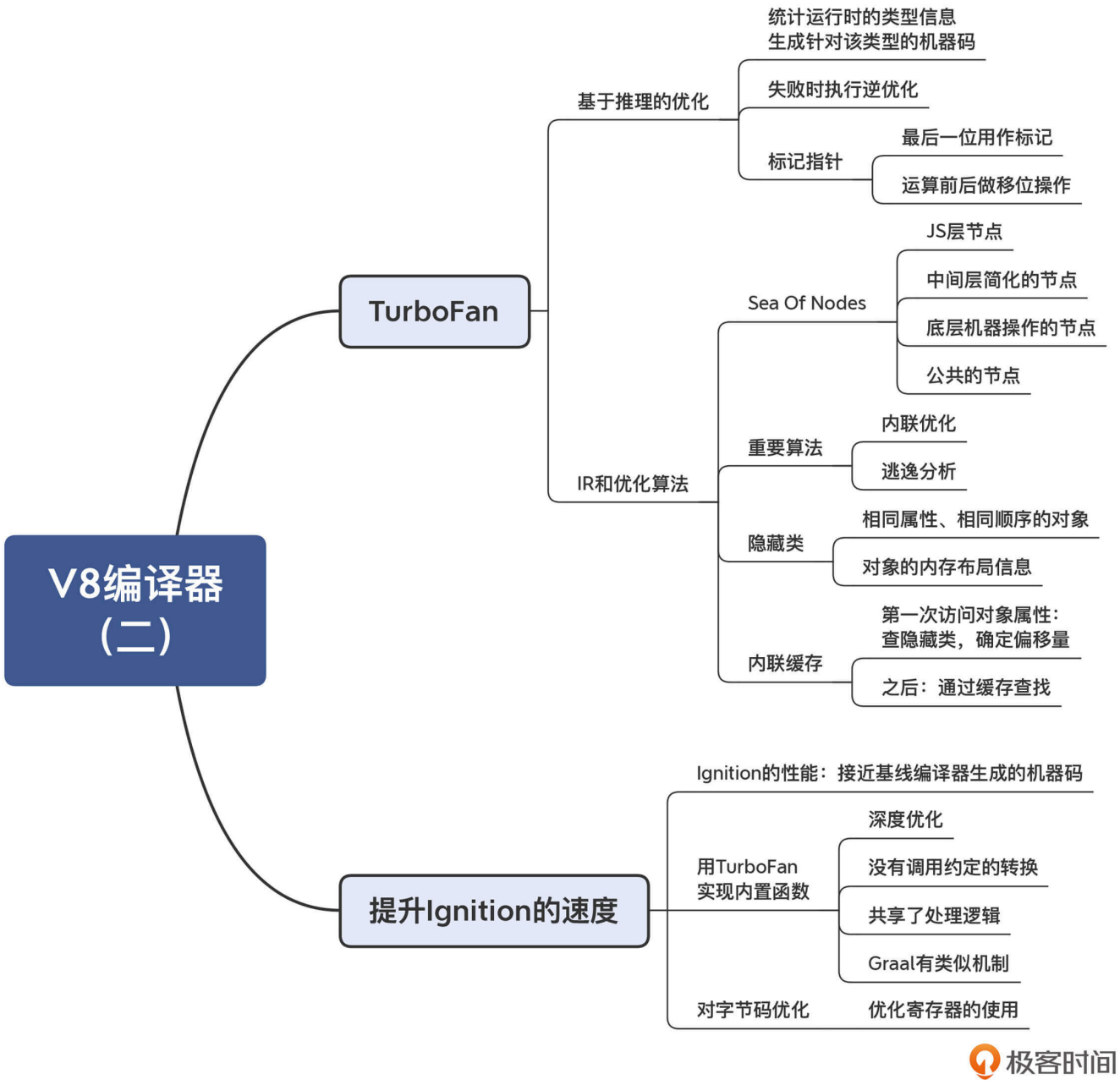
再补充一点，V8对字节码也提供了一些优化算法。比如，通过优化，可以减少对临时变量的使用，使得代码可以更多地让累加器起到临时变量的作用，从而减少内存访问次数，提高运行效率。如果你有兴趣在这个话题上去做深入研究，可以参考我在文末链接中给出的一篇论文。

课程小结

本讲，我围绕运行速度这个主题，给你讲解了V8在TurboFan和Ignition上所采用的优化技术。你需要记住以下几个要点：

- 第一，由于JavaScript是动态类型的语言，所以优化的要点，就是推断出函数参数的类型，并形成有针对性的优化代码。
- 第二，同Graal一样，V8也使用了Sea of Nodes的IR，而且对V8来说，内联和逃逸优化算法仍然非常重要。我在解析Graal编译器的时候已经给你介绍过了，所以这一讲并没有详细展开，你可以自己去回顾复习一下。
- 第三，V8所采用的内联缓存技术，能够在运行期提高对象属性访问的性能。另外你要注意的是，在编写代码的时候，一定要避免对于相同的对象生成不同的隐藏类。
- 第四，Ignition采用了TurboFan来编译内置函数，这种技术非常聪明，既省了工作量，又简化了系统的结构。实际上，在Graal编译器里也有类似的技术，它叫做Snippet，也是用自身的中后端功能来编译内置函数。所以，你会再次发现，多个编译器之间所采用的编译技术，是可以互相印证的。

这节课的思维导图我同样帮你整理出来了，供你参考和复习：



一课一思

我们已经学了两种动态类型的语言的编译技术：Python和JavaScript。那我现在问一个开脑洞的问题：如果你要给Python加一个JIT编译器，那么你可以从JavaScript这里借鉴哪些技术呢？在哪些方面，编译器会得到巨大的性能提升？

参考资料

1. V8的指针压缩技术：Pointer Compression in V8

2. 介绍V8基于推理的优化机制：An Introduction to Speculative Optimization in V8
3. 对Ignition字节码做优化的论文：Register equivalence optimization，我在GitHub上也放了一份拷贝

上一页 下一页