



下载APP



36 | 节点之海：怎么生成基于图的IR？

2021-11-05 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 20:50 大小 19.08M



你好，我是宫文学。

从今天这节课开始，我们就要学习我们这门课的最后主题，也就是优化篇。

在前面的起步篇和进阶篇，我们基本上把编译器前端、后端和运行时的主要技术点都过了一遍。虽然到现在，我们语言支持的特性还不够丰富，但基本上都是工作量的问题了。当然，每个技术点我们还可以继续深挖下去，比如我们可以在类型计算中增加泛型计算的内容，可以把上两节课的垃圾收集算法改成更实用的版本，等等。在这个过程中，你还需要不断克服新冒出来的各种技术挑战。不过，基本上，你已经算入了门了，已经把主要的知识脉络都打通了。



而第三部分的内容，是我们整个知识体系中相对独立、相对完整的一部分，也是我们之前屡次提起过，但一直没有真正深化的内容，这就是优化。

优化是现代语言的编译器最重要的工作之一。像 V8 和其他 JavaScript 虚拟机的速度，比早期的 JavaScript 引擎提升了上百倍，让运行在浏览器中的应用可以具备强大的处理能力，这都是优化技术的功劳。

所以，在这第三部分，我会带你涉猎优化技术中的一些基础话题，让你能够理解优化是怎么回事，并能够上手真正做一些优化。

那在这第一节课，我会带你总体了解优化技术的作用、相关算法和所采用的数据结构。接着，我会介绍本课程所采用的一个行业内前沿的数据结构，基于图的 IR，又叫节点之海，从而为后面具体的优化任务奠定一个基础。

那首先，我们先简单介绍一下与优化有关的背景知识。

有关优化的背景知识

如果我要把优化的内容和算法都大致介绍一下，可能也需要好几节课的篇幅。不过，我在《编译原理之美》的 [第 27 节](#) 和 [第 28 节](#)，对优化算法的场景和分类，做了一些通俗的介绍。对于优化算法，特别是基于数据流分析的优化算法，也做了一些介绍。

而在《编译原理实战课》中，我在 [第 14 节](#)、[第 15 节](#)、[第 21 节](#)、[第 23 节](#)、[第 24 节](#) 分别涉及了 Java、JavaScript、Julia 和 Go 语言的编译器中的优化技术。所以，我这里就不重复那些内容了，只提炼几个要点，重点和你说一下优化的目标、分类、算法，以及数据结构，让你做好讨论优化技术的知识准备。

优化的目标

优化工作最常见的目标，是提高代码运行的性能。在有些场景下，我们还会关注降低目标代码的大小、优化 IO 次数等其他方面。

优化工作的分类

优化技术的种类非常多，我们很难用一个分类标准把各种优化工作都涵盖进去。但通常，我们会按照几个不同的维度来进行分类。

从优化算法的作用范围（或者空间维度）来说，可以分为局部优化（针对基本块的优化）、全局优化（针对整个函数）和过程间优化（多个函数一起统筹优化）。

从优化的时机（也就是时间维度）来说，我们在编译和运行的各个阶段都可以做优化。所以 llvm 的主要发起人 Chris Lattner 曾经发表了一篇论文，主题就是**全生命周期优化**。在编译期呢，编译器的前端就可以做优化，比如我们已经做过一些常数折叠工作。在后端也可以做一些优化，比如我们前面讲过的尾递归和尾调用优化。

但大部分优化是发生在前端和后端中间的过渡阶段，这个阶段有时候也被叫做中端。除了这些，还有运行时的优化。对于 V8 这种 JIT 的引擎，在运行时还可以收集程序运行时的一些统计信息，对程序做进一步的优化编译，在某些场景下，甚至比静态编译的效果还好。

优化的算法

优化涉及的算法也有很多。比如，前面我们做常量折叠的时候，基本上遍历一下 AST，进行属性计算就行了，但在做尾递归和尾调用优化的时候，我们就需要基于栈帧的知识对生成的汇编代码做调整，这里面就涉及到了一些优化的算法。但其中最有用的，则是**控制流**和**数据流分析**。

对于数据流分析，我们已经讲过不少了。那控制流分析是怎么回事呢？**控制流分析的重点是分析程序跳转的模式**，比如识别出来哪些是循环语句、哪些是条件分支语句等等，从而找到可以优化的地方。

比如，如果一个循环内部的变量，是跟循环无关的。那我们就可以把它提到循环外面，避免重复计算该变量的值，这种优化叫做“循环无关变量外提”。比如下面的示例程序中，变量 c 的值跟循环是无关的，所以我们就没必要每次循环都去计算它了。而要实现这种优化，需要优化算法把程序的控制流分析清楚。

 复制代码

```
1 function foo(a:number):number{
2   let b = 0;
3   for (let i = 0; i < a; i++){
4     let c = a*a; //变量c的值与循环无关，导致重复计算！
5     b = i + c;
6   }
7   return b;
8 }
9
```

优化算法所依托的数据结构

针对中端的优化工作，我们最经常采用的数据结构是**控制流图**，也就是 CFG。在生成汇编代码的时候，我们已经接触过控制流图了。当时我们把代码划分成一个个的基本块，每个基本块都保存一些汇编代码，基本块之间形成控制流的跳转。控制流图的数据结构用得很广泛，比如 llvm 编译器就是基于 CFG 的，这也意味着像 C、C++、Rust、Julia 这些基于 llvm 的语言都受益于 CFG 数据结构。另外，虽然 Go 语言并不是基于 llvm 编译器的，但也采用了 CFG。

控制流图最大的优点，当然是能够非常清楚地显示出控制流来，也就是程序的全局结构。

而我们做数据流分析的时候，通常也要基于这样一个控制流的大框架来进行。比如，我们在做变量活跃性分析的时候，就是先分析了在单个的基本块里的变量活跃性，然后再扩展到基于 CFG，在多个基本块之间做数据流分析。

不过，虽然 CFG 的应用很普遍，但它并不是唯一用于优化的数据结构。特别是，像 Java 编译器 Graal 和 JavaScript 的 V8 引擎，都采用了另一种基于图的 IR。不过构成这个图的节点并不是基本块。我在这节课后面会重点介绍这个数据结构，并且说明为什么采用这个数据结构的原因。

刚才我挑重点介绍了与优化有关的背景知识。不过，我用短短的篇幅浓缩了太多的干货，你可能会觉得过于抽象。所以，我还是举几个例子更加直观地说明一下与优化有关的知识，借此我们也可以继续讨论下面关于 IR 的话题。

一个优化的例子


我们先来看这个代码片段，这段代码中，x 和 y 都被赋值成了 a+b。

```
1 x = a + b
2 y = a + b
3 z = y - x
```

[复制代码](#)


你用肉眼就能看出来，第二行代码是可以被优化的，因为 x 和 y 的值是一样的，所以在第二行代码中，我们就不需要再计算一遍 a+b 了，直接把 x 赋值给 y 就行。这种优化，叫

做“公共子表达式删除 (Common Subexpression Elimination) ”：

 复制代码

```
1 x = a + b
2 y = x
3 z = y - x
```

再接着看，其实第三行也是可以优化的。因为 y 是等于 x 的，所以 $z := x - x$ ，也就是 $z := 0$ 。这种优化方法，是把 y 的值传播到了第三行，所以就叫做“拷贝传播 (Copy Propagation) ”。

 复制代码

```
1 x = a + b
2 y = x
3 z = x - x //进一步可以优化成 z = 0
```

再进一步，我们假设这个代码片段后面跟着的代码，不需要再用变量 y 了，那我们就可以把第二行代码删除，这个方法就叫做“死代码删除 (Dead Code Elimination) ”。

为了实现上面这些优化工作，我们经常使用的就是数据流分析算法。比如说，使用我们之前学过的变量活跃性分析，我们就可以知道在第二行处，其实 y 是不活跃的，是死代码，可以删除。

同时，我们在优化算法中，还会经常使用一种叫做“使用 - 定义链 (Use-def chain) ”的技术，也就是在变量的定义和使用之间建立连接。从变量的定义，可以找到所有使用它的地方。反过来，在每个使用变量的地方，也可以找到它的定义。

我们用这种技术分析一下上面的第一个代码片段。在这段代码中， x 的定义使用到了 a 和 b ，而它自己又被 y 和 x 所使用，这样就构成了 use-def 链。

变量	使用的变量	被下面的变量使用
x	a, b	y, z
y	x	z
z	x, y	/



那 use-def 链有什么用途呢？我们通过 use-def 链知道了变量和定义之间的关系以后，实际上也就清楚了数据是怎么从一个变量流动到另一个变量的，这其实就是程序中的数据流。知道了这些之后，我们就能更容易地进行数据流分析，也更容易实现优化。

比如，既然 y 的定义是 x，那么 x 可以顺着 use-def 链往下传，传播到 z 的定义中，也就是第三行代码中，实现拷贝传播。

而且，用 def-use 链判断死代码也更容易。比如，在第三个代码片段中，是没有变量引用 y 的，所以我们就能断定用来定义 y 的第二行肯定是死代码了。

所以，use-def 链是一项很有用的技术，比如 llvm 等很多编译器都采用了它。

不过，为了更好地使用 use-def 链技术，更清晰地表达程序中的数据流，我们对于要使用的 IR 是有一定要求的，也就是要求 IR 是符合 SSA 格式的。

静态单赋值（SSA）格式的 IR

我们首先说说什么是 IR。我估计你应该知道 IR 的意思的。IR 是 Intermediate Representation 的缩写，字面意思是中间表达，也就是我们的程序在转变成目标代码之前的一些中间格式。

从广义角度来说，介于源代码和目标代码之间的各种中间格式，都可以叫做 IR。从这个意义上来说，AST 也可以看做是一种 IR。不过，当我们提到 IR 的时候，更多时候指的是它比较狭义的意思，也就是用于优化的中间格式。

IR 也是有很多种的。在上面的示例程序中，我使用的这种 IR 叫做“三地址代码”，这也是教科书中经常使用的一种 IR。这种 IR 的每个变量就是一个地址，比如 $x = a + b$ 中，赋值符号左边是一个地址，右边最多可以有两个地址。

那我刚才提到的，要求 IR 是符合 SSA 格式的，又是什么意思呢？

SSA 是 Static Single Assignment 的缩写，也就是**静态单赋值**。它的意思是，**在代码中每个变量只能被定义一次**。比如，如果我们之前定义 x 为 $a+b$ ，之后又定义它为 $c+d$ ，那这个 IR 就不符合 SSA 格式了。

 复制代码

```
1 x = a + b
2 ...
3 x = c + d
```

这说的也是单赋值，那静态单赋值中的静态又是什么意思呢？

其实，如果某个变量的赋值是出现在一个循环中，那么在程序运行的时候，这个变量可能被赋值了多次，这是程序的动态情况。而我们目前对代码所做的分析，都是静态分析。我们可以说，只要在代码中，变量只被赋值过一次，就是符合 SSA 的。

如果每个变量只被赋值过一次，那么用“定义”这个词汇就很准确了，因为每个变量都是被其他变量所唯一定义的，在使用过程中一直不变的。

不过，为什么要求 IR 必须是 SSA 格式的呢？因为 SSA 格式的 IR 会产生很多好处，使得优化算法更加简单。

比如在下面的代码片段中， a 一开始被赋值为 c ，后来被赋值为 d 。那么这个时候，虽然 x 和 y 的定义相同，但它们实际的值是不同的，因为 a 的定义发生了变化，所以这里我们就不能进行子表达式删除的优化了。如果我们仍然要用 use-def 链来保存定义和使用的关系的话，那就必须要把 a 的这两个值区分开，比如变成 $a1$ 和 $a2$ ，这样实际上也就变成了 SSA 格式。

 复制代码

```
1 a = c
```

```
2 x = a + b
3 a = d
4 y = a + b
```

我再用 TypeScript 举一个例子。在下面这个代码片段中，变量 `a` 被赋值了两次。在第一次，`a` 是一个整型数据，在第二次，它变成了字符串型。

[复制代码](#)

```
1 let a:number|string;
2 a = 2;           //a现在是整型
3 console.log(a+3); //打印出5
4 a = "2";         //a变成了字符串型
5 console.log(a+3); //打印出字符串"23"
```

在这里例子中，前后两个 `a` 的定义其实是完全不相关的，连类型都不一样，所以这里本来就应该写成两个变量才更合理。

好了，我想你大致应该明白了 SSA 的含义以及使用 SSA 的原因了。那我们再进一步，介绍一种先进的、用图来表示的 IR。

基于图的 IR：节点之海

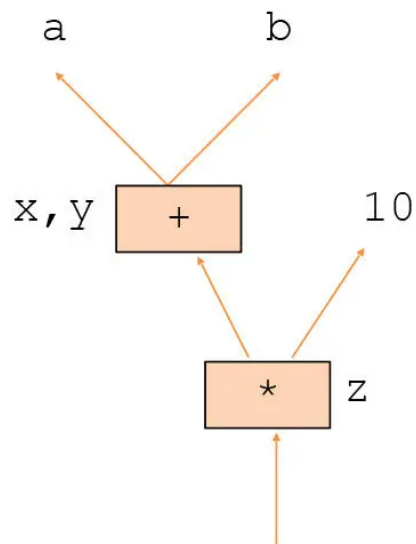
在前面的示例程序中，变量之间的定义 - 使用链是能够形成一张图的。如果再加上运算符，我们就可以用这张图来表达程序的逻辑了。

比如，这里有一个程序片段：

[复制代码](#)

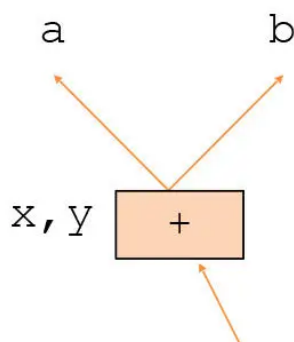
```
1 x = a + b
2 y = a + b
3 z = y * 10
```

如果我们把变量之间的数据流关系以及运算符画成图，就是下面的样子：



乍一看，你会觉得这是一颗倒过来的 AST。如果按照 AST 来看， z 的值确实就是 $(a+b)*10$ ，一点都没错。

这里你要注意的是，在生成这个图的时候，我们可以把相同的表达式合并。比如， x 和 y 的值都是 $a + b$ ，那么我们就用同一个子图来表示就好了，这样，我们自然而然地就实现了公共子表达式的删除的优化。从这一点上，你就能初步看到这种 IR 的优势了。



我们再接着看这个子图，你会发现，+ 号运算符有两条输出的边，代表它依赖这两条边所指向的 a 和 b 变量来提供值。还有一条输入的边，这里代表另一节点依赖这个节点数据。不过，也有的论文会把箭头反过来画，这样的话箭头方向代表的就是数据的流动方向，而不是数据依赖关系了，但它们表达的意思是一样的。

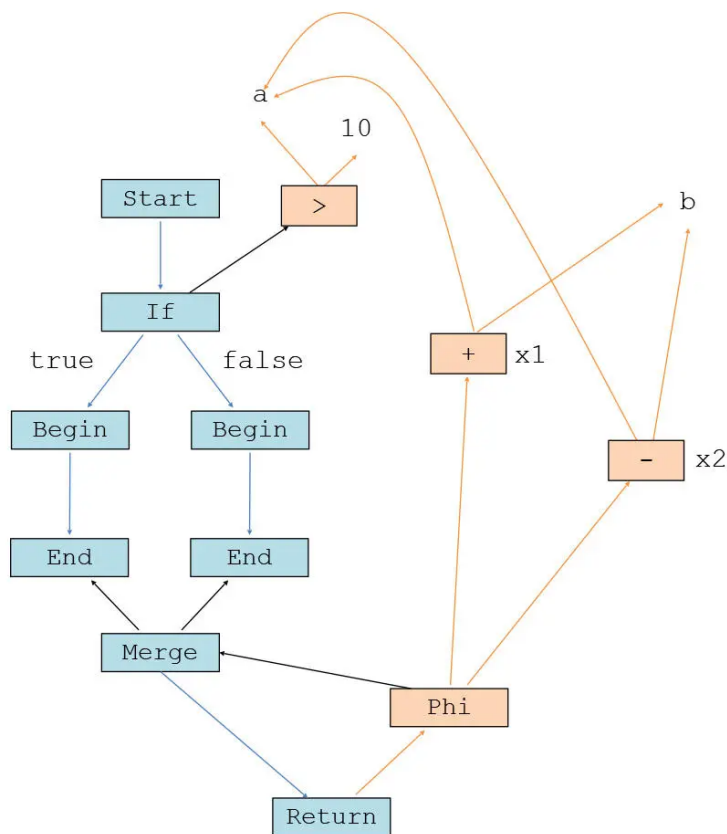
不过，目前我们画出来的图，只能用来表示数据流，表达像刚才这样的进行算术运算的直线式执行的逻辑，但不能表达控制流逻辑，比如 if 语句、for 循环语句和调用函数这些。不过，你现在已经学习过生成汇编代码了，所以你应该知道这些控制流逻辑都有一个共同的特点，就是它们会生成跳转指令，跳转到另外的代码地址去执行。

但我们仍然可以用图的方式来表达控制流逻辑。现在让我们先来分析一个带有 if 语句的简单的函数的例子，这个例子是用 TypeScript 写的：

[复制代码](#)

```
1 function foo(a:number, b:number):number{
2   let x:number;
3   if (a>10){
4     x = a + b;
5   }
6   else{
7     x = a - b;
8   }
9   return x;
10 }
```

把这个程序转化成图，是下面的样子：



我解释一下这张图。

首先，我们先看一下数据流的部分，也就是橙色线的和橙色节点的部分。你会看到，原来的变量 x ，现在要变成 $x1$ 和 $x2$ 这两个变量，这样才符合 SSA 的要求，每个变量只赋值一次。

但这个时候就有问题了，我们最后一条语句 “return x ” 中的 x ，到底是 $x1$ 还是 $x2$ 呢？

这显然取决于程序的控制流走得是哪个语句块。如果走得是 if 块呢，那么就使用 $x1$ ，否则就使用 $x2$ 。

所以，这里我们引入了一个 phi 节点。这个节点的作用，就是根据控制流提供的信息，来确定选择两个值中的哪一个。我在图中用黑线表示了从控制流中提供过来的信息。从这个例子中，你也可以看出，**phi 运算是 SSA 格式的 IR 中必然会采用的一种运算，用来从多个可能的数据流分支中选择一个值。**

那接下来我们再看看控制流，也就是程序中的蓝色节点和蓝色箭头的部分。

控制流是从 start 节点开始的，这也是进入函数的点。接着，在 if 语句那里，控制流会根据 if 条件为真还是假，形成两条分支。因此，if 节点需要从“ $a > 10$ ”这个节点获得条件表达式的计算结果。由于这个值是与控制流有关的，所以我也把这条箭头线也画成了黑色。

从 if 语句发出的两条控制流都以 begin 开头，以 end 结尾。如果每条分支里还有嵌套的 if 语句和循环语句，新的控制流节点就会出现在 begin 和 end 之间。这两条控制流在 merge 节点合并在一起，并且它们会给 merge 节点提供一个信息，说明控制流到底是从哪条分支过来的。这个信息又会给到数据流中的 phi 节点，告诉 phi 节点应该计算哪个数据流分支的数据。

到这里，整个图就完成了。这个图也忠实地体现了原来程序的逻辑。

其实这个图，就是当前 JavaScript 的 V8 编译器和 Java 的 Graal 编译器都在使用的一种 IR。我们后面的课程还会进一步解析这个 IR，也会再分析如何基于这个 IR 来编译程序。

最后说说“节点之海”这个别名。你可以看到，我们的示例程序是非常简单的，但都形成了由这么多的节点构成的图。可以想象，如果程序更复杂一点，节点就会更多。并且，我们在优化时还会把多个函数的图按照调用关系拼在一起，形成更大的图，就会更加令人眼花缭乱，所以这个 IR 也被叫做节点之海。好在，计算机的处理能力可比我们的眼睛强多了，完全能够基于这样的 IR 愉快地完成优化和编译工作。

课程小结

这节课到这里就结束了。今天这节课，我以非常紧凑的篇幅，介绍了与优化有关的背景知识，目的是帮助你把思路转移到优化这个主题上，开始一起思考与优化有关的技术。而今天这节课重点呢，显然是介绍 JavaScript 和 Java 编译器都在使用的一种基于图的 IR。这里，我希望你记住几个知识点。

首先，你需要记住优化技术的常用分类，比如按照空间维度，也就是优化的范围，可以分为本地优化、全局优化和过程间优化。从时间维度，优化技术可以贯穿整个程序的生命周期。

第二，从算法角度，优化算法也有很多，数据流分析算法仍然很有用。

第三，从数据结构角度，优化算法需要依托定义良好的数据结构和 IR。之前我们使用过的 CFG 是常用的、成熟的数据结构，而基于图的 IR 则是更前沿的、值得我们关注的一个数据结构。

第四，你还需要熟悉一些常见的优化场景，比如这节课提到的子表达式删除、拷贝传播、死代码删除、循环无关变量外提，等等。这样，你在思考与优化有关的技术的时候，会更容易联系实际。

第五，目前成熟语言的编译器里用于做优化的 IR 都是符合 SSA 格式的，它的好处是更容易形成 use-def 链、分隔开原本就应该是不同的变量、更有利于算法运行，等等。这节课介绍的基于图的 IR 也是符合 SSA 格式的，因为每个节点代表一个变量，所以每个节点当然只能静态赋值一次。你要注意，SSA 格式的 IR 遇到控制流的分支和合并时，需要一个 phi 运算帮助确定到底选择哪条数据流线路上的值。

第六、关于基于图的 IR，你目前只需要记住它能够同时表达数据流和控制流，并且它也能够像 AST 一样忠实地反映源代码的逻辑就好了，在后面的课程中我们还会继续深入了解它。

总结一遍以后，发现今天的知识点还真是挺密集的，我希望你能多看几遍，加深印象。

思考题

今天这节课，我们介绍了几个的简单的优化场景。你能不能再给我们分享一下你知道的优化场景和优化技术？多了解这些场景，会让我们的学习更加联系实际。

欢迎你把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

分享给需要的人，Ta 订阅后你可得 20 元现金奖励

 生成海报并分享


 赞 0  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 35 | 内存管理第2关：实现垃圾回收

精选留言 (2)

写留言

 **奋斗的蜗牛**
2021-11-05
叹服，老师的讲解直击重点，循序渐进
展开 ∨

 **奋斗的蜗牛**
2021-11-05

超赞，感谢老师的分享

展开 ∨