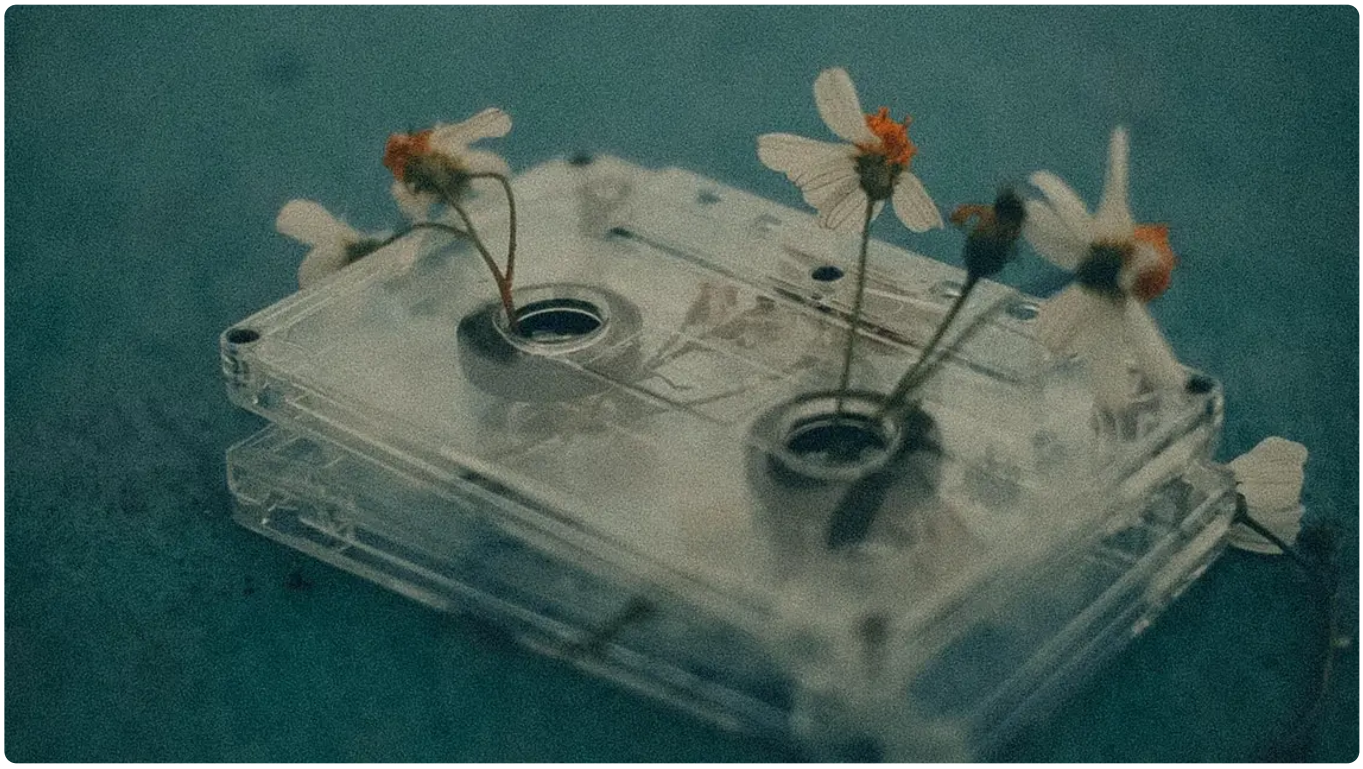
41｜后端优化：生成LIR和指令选择

2021-11-17 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 18:23 大小 16.85M

你好，我是宫文学。

前面几节课中，我们讨论的主要是中端的优化。中端优化是跟具体硬件关系并不大，但由于我们还要生成针对具体 CPU 的汇编代码或机器码，所以做完中端优化之后，我们还要针对具体 CPU 的特性来做一些优化，也就是后端优化。

其实，我们已经接触过一些后端优化技术了。比如，之前我们已经讲过寄存器分配算法、尾调用和尾递归的优化，这些基本上都属于后端优化。不过，那个时候，我们是从 AST 直接生成汇编代码，然后在这个过程中做一些后端优化的。

在第三部分优化篇中，我们引入了新的、基于图的 IR，进行了很多与硬件无关的优化。用这个基于图的 IR 来做后端优化，效果又怎样呢？接下来，我们就要修改以前的生成汇编代

码的逻辑，改成从这个 IR 来生成汇编代码，并在这个过程中做一些优化。

今天这一节课，我就带你从中端过渡到后端，看看如何实现后端优化，并生成目标代码。这其中包括 IR 的 Lower、生成 LIR 和指令选择，以及寄存器分配、指令重排序、窥孔优化和汇编代码的生成，等等工作。不过有些知识点我之前已经讲过了，还有一些知识点不是我们这门课的重点，我就不再展开讲了，重点帮你贯穿一下整个过程。

首先，我们先了解一下给 IR 做 Lower 的过程。

Lower 过程

HIR 要经过一系列 Lower 过程，最后变成 LIR。我先举一个例子，让你理解一下在 Lower 过程中会发生什么事情。这是一个简单的例子，它只实现了给 mammal 对象 weight 字段赋值的功能：

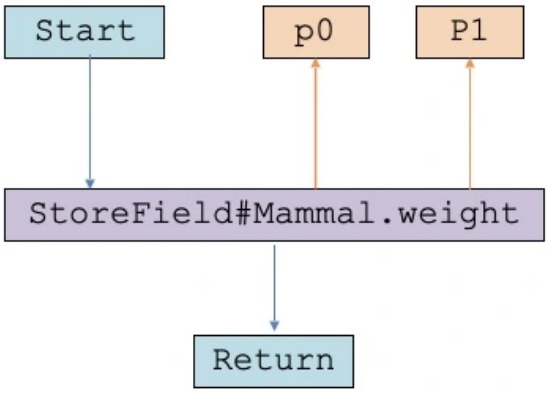
复制代码

```
function accessField(mammal:Mammal, weight:number){
    mammal.weight = weight;
}
```

针对对象属性的赋值，通常编译器要生成写内存的指令。这是因为，对象通常使用的是在堆里申请的内存。而且，由于一个对象可能会由多个线程访问，所以只有把对象属性写到内存里，另一个线程才能访问到更新后的属性。

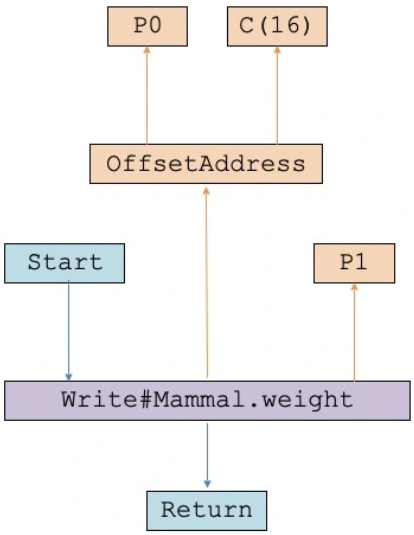
当然，我们前一节课也说过，如果这个对象并没有逃逸，那就是另一种情况了。我们先假设该对象是逃逸的，那么我们要给对象属性赋值，首先就需要进行**写内存**的操作。

对于这个简单的场景，一开始这个程序的 IR 是下面这样：



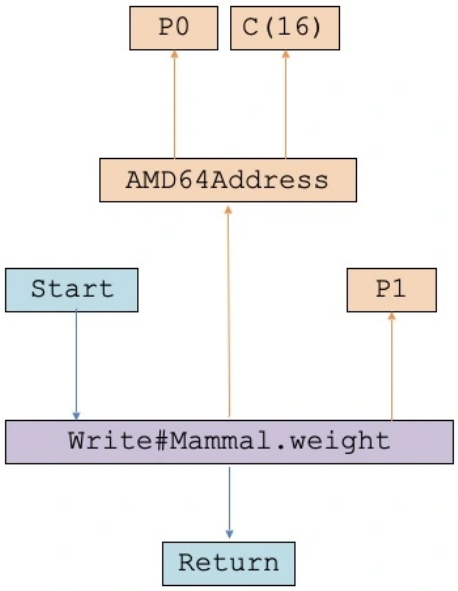
这里，我们使用了一个抽象度比较高的节点，叫做 StoreField。它接受两个输入：一个输入是对象的引用，也就是对象地址；第二个输入是 weight 属性的值。

在编译的过程中，这个 IR 会被做 Lower 处理。StoreField 节点会被一个 Write 节点代替，Write 节点是一个写内存的操作。而内存地址呢，用 OffsetAddress 表示，也就是一个基地址加上一定的偏移量。基地址就是对象的地址，偏移量是对象头的大小。在 PlayScript 的设计中，它是 16 个字节。Lower 过一次的 IR 图是这样的：



到目前为止，这个 IR 还是跟具体 CPU 无关的。因为按照这张 IR 图，无论针对什么 CPU，你都可以通过在某个地址的基础上加上一个偏移量，来获得新的地址。

然后这个 IR 还会进一步被 Lower，让地址的表示方式更贴近 x86-64（或 AMD64）架构的具体寻址方式。我们曾经学过 x86-64 的寻址方式，它的完整形式包括基地址、偏移量、下标值、元素字节数等多个参数。但这里我们只需要它的简化方式，也就是基地址加上一个偏移量就行。进一步 Lower 的 IR 变成了这样：



到了这一步，我们的 IR 已经变得跟具体 CPU 架构相关了。接下来，我们就把它彻底转化成 LIR 的格式。

生成 LIR 和指令选择

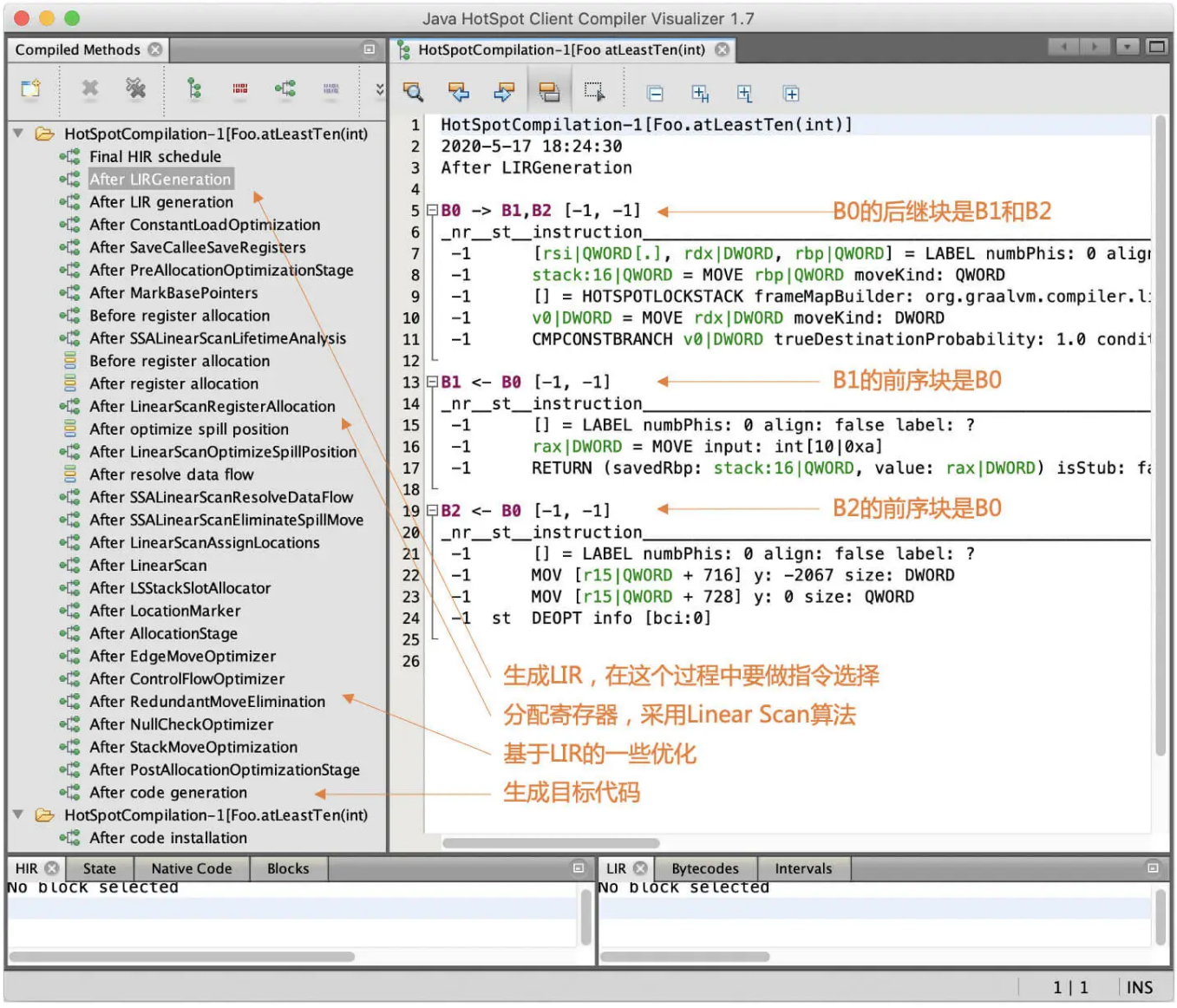
那 LIR 又是什么样子的呢？你可以思考一下，如果你来设计编译器，应该如何设计 LIR 呢？

LIR 的目的，是进行机器相关的优化，并最后生成汇编代码。所以，**大部分 LIR 的设计，都是跟汇编代码是同构的**。也就是说，LIR 是由一条条指令构成的，指令是放在基本块中的，而基本块之间存在跳转关系。

从这个意义上说，我们之前生成汇编代码的时候，已经设计过这样的 LIR。而 Graal、LLVM、Go 语言的 gc 编译器，在生成汇编代码或机器码之前也都有类似的 LIR 设计。这个数据结构看上去仍然是基于 CFG 的，但我们目前已经不需要分析它的控制流和数据流，并调整里面的代码了。这些工作，我们在中端优化的时候都已经完成了。**现在，基于这个**

LIR，我们关心的主要是指令选择、寄存器分配和指令重排序（或者叫做指令调度）这样的话题。

由于我们的 IR 设计借鉴了 Graal 编译器，那么同样的，我们继续跟着 Graal 看看它是怎么处理 LIR 的。图中是 Graal 编译器中生成的 LIR 的例子，你可以通过它建立对 LIR 的直观感觉：



这张图中一行行的文本，是为了显示 LIR 中内容，便于调试，实际上的 LIR 都是内存里的一条条的指令对象。在这个图中，你还能看到 Graal 编译器的后端处理过程，包括生成 LIR、寄存器分配，一直到生成目标代码。**我们自己实现的编译器，也需要完成类似的功能。**

好了，我们现在已经理解了 LIR 是什么样子了。那我们现在就从 HIR 生成 LIR，并在这个过程中进行指令的选择，这又可以分成几项子任务。

首先，我们要把 **HIR 中的不同节点分配到不同的基本块**中，这被叫做调度算法（Schedule）。

我们在 39 节已经介绍过，由于很多数据流节点是浮动的，我们可以自由地选择在什么时候进行计算。但我们在生成汇编代码之前，还是要把确定这些数据节点的计算时机，因此要把它们分配到具体的基本块中。

而且，我们需要基于一些规则来完成这个分配工作，比如对于循环无关的代码，我们会提到循环外边；而基于控制流来求值的代码，比如 if 语句的两个不同分支的代码，我们尽量分配到这两个分支对应的基本块中。制定这些规则的出发点，是尽可能地提升程序的性能，但其实并不能完全保证。在比较 AOT 和 JIT 时，我们已经讲过这点了。

在划分好基本块以后，我们再做第二项工作，**把 IR 图转化成 LIR 的指令，并在这个过程中进行指令的选择。**

如果细讲起来，指令选择有两层含义，而这两层含义的工作经常是一起实现的。指令选择的第一层含义，是把抽象的运算，准确地 Lower 到硬件的具体指令上。比如说，我们可以从比较抽象的层次，对整数和浮点数都执行加法运算。但到了 CPU 层面，整数的加法指令和浮点数的加法指令就不一样了。其他指令，比如比较运算、数据拷贝的指令，也是跟数据类型和所采用的指令集相关的，编译器要确定出正确的指令。

指令选择的第二层含义，指的是相同的功能，可以用不同的指令组合来实现，而我们要尽量选择让整体性能最优的那组指令。这实际上是一个最优化问题。

关于指令选择的算法，我在《编译原理之美》的 29 节做过一些理论性的介绍，在《编译原理实战课》的 16 节，我也介绍过 Graal 编译器的具体实现。这门课，我也会参考 Graal 的思路，做一个比较简化的实现。

在我看来，要理解指令选择，除了学习算法，更重要的是**要了解很多具体的指令选择场景**。下面，我们就以 x86-64 架构的指令来举几个例子，帮助你建立直观理解。经过这些讲解后，你就能理解那些抽象的算法到底在说些什么了。

第一个例子，是经常出现在 if 语句中的条件跳转指令：

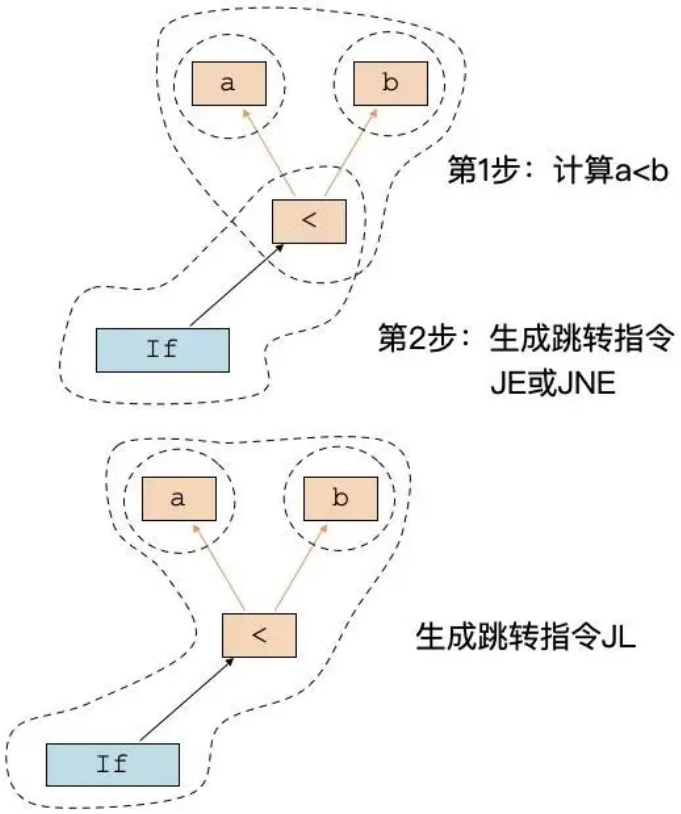
```
if(a>b){
  //somecode
}
else{
  //some other code
}
```

复制代码

回忆一下，我们在这门课的第 9 节、为字节码虚拟机生成字节码的时候，if 条件和跳转相关的字节码是分两步来生成的：第一步，处理 if 条件，计算条件表达式"a<b"，并生成 1 或 0 两个值，代表 true 和 false；第二步，处理 if 节点，根据 < 节点的值来生成跳转指令。跳转指令使用 JE 或 JNE 就行了，也就是比较 if 条件是不是 1。

用这个方式生成指令比较简单。算法上说，就是对每个 AST 节点依次进行处理。像字面量、变量这样的节点，我们会返回一个 Operand。而对于计算性节点，我们就要生成指令，并把指令运行的结果作为 Operand 来返回。

不过，大部分 CPU 或虚拟机都提供了更丰富的条件跳转指令，比如 JL 指令就可以用于在 a<b 的时候做跳转，而 JG 指令就可以用于在 a>b 的时候跳转。这个时候，我们需要同时处理 if 和 < 号两个节点，确定采用什么指令，你可以看一下这张示意图：



这就是指令选择算法的特点，我们需要一次性地考虑 AST 或 IR 中的多个节点，并生成合适的指令，只要最后算法确实覆盖了所有节点就行。

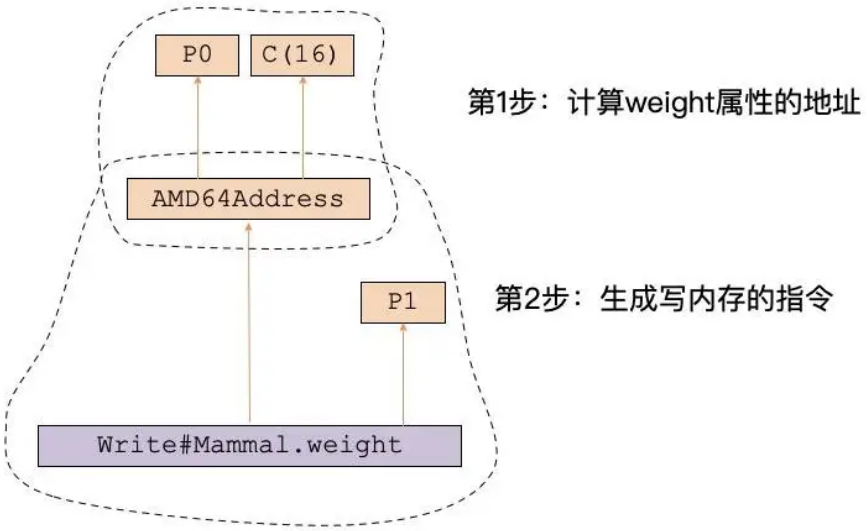
第二类经常需要做指令选择的情况，是对内存的访问。我还是用这节课开头这个、给 mammal 对象的 weight 属性赋值的例子来做说明：

复制代码

```
function accessField(mammal:Mammal, weight:number){
    mammal.weight = weight;
}
```

在生成指令的时候，我们需要在对象的基地址的基础上，添加一个偏移量，获得 weight 属性的地址，然后再给这个地址赋值。

要完成这个操作，我们有两个办法。第一个办法是分成两步来生成指令，第一步是先计算出 weight 属性的地址，第二步是往内存地址写 weight 的值：

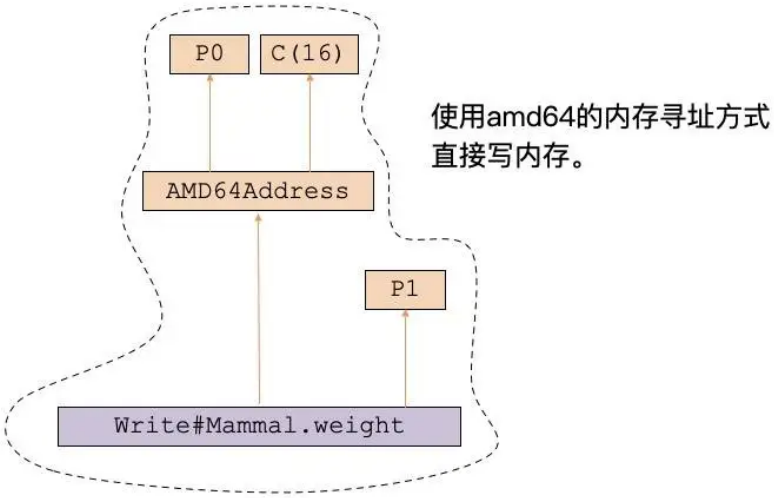


这两步对应的 LIR 相当于下面两条代码：

复制代码

```
add $16, p0    #把p0,也就是对象的地址加上偏移量
mov p1, (p0)   #把p1赋给p0指向的内存地址
```

不过，我们还有第二个方法来生成指令，这就是直接使用 x86-64 的寻址方式，用一条指令就能完成地址计算和写内存这两个操作。我们的指令选择算法需要一次性处理 Write 和 AMD64Address 两个节点：



生成的 LIR 相当于下面的一条代码：

```
1 mov p1, 16(p0)    #把p1赋给p0指向的内存地址再加16的偏移量。
```

复制代码

通过我举的这两个场景，你大概应该明白指令选择的工作原理了。在生成 LIR 的过程中，类似的场景还有很多。这里的共同点，都是**因为目标 CPU 往往能用更简洁的方式，一次性完成两项甚至多项运算工作，从而达到节省指令、提高效率的目标。**

那在生成了 LIR 以后，编译器接下来还要做哪些工作呢？

后续优化工作

首先，是我们已经学习过的**寄存器分配工作**。这项工作，我们之前已经基于前一个版本的 LIR 做过了。在升级 LIR 之后，我们只需要适当做一些完善就行了。

在我们新的 LIR 中，一开始变量还都是用名称表示的，或者说它们是逻辑寄存器。而寄存器分配算法，是要把它们映射成物理寄存器。并且，当物理寄存器不够的时候，需要把某

个寄存器的值 spill 到栈桢中来腾出寄存器。而调用其他函数的时候，还需要把 Caller 保护的寄存器保护起来。在 Callee 中，也有需要保护的寄存器。所有这些寄存器的管理工作，都是在寄存器分配算法中完成的。

再接下来呢，有些编译器会使用一个**代码重排序**的算法，通过调整指令的执行顺序，在不改变程序运行结果的情况下，利用 CPU 的流水线功能，提升程序运行的效率。

这是一个可选的功能，而且我在《编译原理之美》第 30 节课具体分析过，在这里我就不再展开了，如果你有兴趣实现这个功能，可以参考那一节课。不过我注意到，现在很多编译器都没有实现这个功能，这有两个原因。一个原因，是现代 CPU 在硬件层面上已经有很好的乱序执行能力了，所以编译器层面上的优化带来的收益不高。

第二个原因，是对于并发执行的程序，改变指令的执行顺序需要考虑更多的影响。比如在单线程的情况下，改变指令顺序不影响计算结果，但在多线程的情况下，就可能导致计算结果的不一致。

最后呢，我们通常还会**基于 LIR 再做一些优化工作**。其中一项常用的技术，叫做**窥孔优化**。窥孔优化是什么意思呢？我还是通过举例子来说明。

如果你仔细阅读了我们当前编译器生成的字节码或者汇编代码，会发现里面有一些像是废话的代码。你可以看看下面这个示例代码：

复制代码

```
# some code
call _foo
movsd %xmm0, %xmm1    #把返回值拷贝到xmm1
# 一些代码，并没有修改xmm0的值
movsd %xmm1, %xmm0    #把xmm1的值拷贝到xmm0，作为函数返回值
```

这里，在调用 foo 函数之后，我首先把返回值从 xmm0 拷贝到了一个新的寄存器 xmm1。这么做的原因，是因为 xmm0 有可能在调用函数之前，分配给了一个别的变量，而在调用函数之后，需要把该变量从内存恢复回 xmm0。所以，为了防止返回值被破坏，我们通常需要把它赋给一个临时变量，而寄存器分配算法会给这个临时变量分配单独的寄存器，比如 xmm1。

但实际的代码执行过程中，后面并没有代码来修改 xmm0 的值。而在返回当前函数的时候，我们又把 xmm1 拷贝回 xmm0，作为函数的返回值。

你用肉眼就可以看出，这里存在着优化的机会，也就是后面的两个 movsd 指令都是多余的，可以去掉。

不仅我们课程里生成的汇编代码和字节码有这样的瑕疵，你如果不用优化参数来调用 clang 或 gcc 编译器，生成的汇编代码中也都有很多这样的冗余指令。

所谓窥孔优化，就可以用于处理这种场景。**它的原理，是通过一个窗口扫描 LIR、汇编代码或者是机器码，每次扫描 n 行，这 n 行就是窗口的大小。**程序可以分析这个窗口里的代码有没有什么冗余的代码，或者其他可以优化的机会，并进行优化。比如，如果一个窗口能够覆盖我们前面的示例代码，那就可以发现其中的问题。这个窗口可以沿着代码不断的滑动，从而发现所有代码中的优化机会。

不过，如果编译器在前序的工作中处理得越好，留给后面窥孔优化的机会就越少。就像我刚才举的例子中，如果我生成指令和做寄存器分配的算法更聪明一些，就可以不用在寄存器之间来回倒腾数据了，也就不需要后面来做这种窥孔优化了，这个优化算法同样是个可选项。在后续迭代的 PlayScript 项目中，如果遇到了需要窥孔优化的场景，我会再加上它。

在基于 LIR 做完所有的优化以后，最后一个环节就是**生成目标代码**。

这是一个比较直接的转换过程，并不复杂，在我们之前的 asm_x86-64.ts 的代码中就有实现。不过，我们是生成文本的汇编代码，而很多编译器，特别是 JIT 编译器，是直接生成机器码来运行。生成机器码相比生成汇编代码其实并没有什么特别的难点，我们只需要额外做一点翻译工作就好了。

对于 JIT 编译器而言，生成机器码反倒省去了程序的静态链接的工作。它并不需要像静态编译那样，把所有函数都连续存储在内存的文本区，再计算出每个函数的入口地址，方便操作系统加载到内存并运行。在使用 JIT 的虚拟机里，每个函数的地址都可以在运行时查询获得。

课程小结

今天这节课，我们把中端优化之后，编译器的后续工作过了一遍，把之前我们已经讲过的与后端有关的功能也串联了一下，让你产生一个清晰的、全局的认知。我希望你记住下面的重点：

首先，你需要对 IR 的 Lower 过程有直观的认识。在 Lower 的过程中，IR 的节点会越来越与具体硬件上的实现相关。你可以记住这节课举的给对象赋值的例子。在实现编译器的过程中，你自己就会发现更多这样的场景。

第二，LIR 通常可以跟具体 CPU 架构的汇编代码直接对应。它是由指令构成的，指令放在基本块中，基本块之间有跳转。

第三，指令选择算法，通常表现为一次性匹配多个 IR 节点，发现其中的模式，并用一条指令实现多个功能，这样就能减少指令的数量，实现性能的提升。

第四，编译器后端的优化还包括寄存器分配、指令重排序和窥孔优化等。寄存器分配是必须的，而指令重排序和窥孔优化是可选的。

思考题

如果要理解指令选择，这个关键就是要知道一些实际的场景。我在这节课中举了两个场景，之前我还曾提过 lea 指令的场景。

那么，你还知道有哪些具体的指令选择的场景吗？比如，我在《编译原理实战课》的 16 节，给出了一个代码链接：AMD64NodeMatchRules，里面是 Java 的 Graal 编译器所实现的一些指令选择的规则。如果你阅读过这些编译器的代码，或者是你自己也实现过一些指令选择的场景，可以在留言区分享一下。

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。

上一篇 40 | 中端优化第3关：一起来挑战过程间优化

下一篇 42 | 到这里，我们的收获和未尽的工作有哪些？

精选留言 (1)

写留言

奋斗的蜗牛

2021-11-17

太牛了，感觉打开了新世界

展开