

## 29 | 目标代码的生成和优化（一）：如何适应各种硬件架构？

2019-10-30 宫文学 来自北京

《编译原理之美》



在编译器的后端，我们要能够针对不同的计算机硬件，生成优化的代码。在 [@23 讲](#)，我曾带你试着生成过汇编代码，但当时生成汇编代码的逻辑是比较幼稚的，一个正式的编译器后端，代码生成部分需要考虑得更加严密才可以。

那么具体要考虑哪些问题呢？**其实主要有三点：**

指令的选择。同样一个功能，可以用不同的指令或指令序列来完成，而我们需要选择比较优化的方案。

寄存器分配。每款 CPU 的寄存器都是有限的，我们要有效地利用它。

指令重排序。计算执行的次序会影响所生成的代码的效率。在不影响运行结果的情况下，我们要通过代码重排序获得更高的效率。

我会用两节课的时间，带你在这三点问题建立直观认识，然后，我还会介绍 LLVM 的实现策略。这样一来，你会对目标代码的生成，建立比较清晰的顶层认知，甚至可以尝试去实现自己的算法。


接下来，我们针对第一个问题，聊一聊为什么需要选择指令，以及如何选择指令。

## 选择正确的指令

你可能会问：我们为什么非要关注指令的选择呢？我来做个假设。


如果我们不考虑目标代码的性能，可以按照非常机械的方式翻译代码。比如，我们可以制定一个代码翻译的模板，把形如 “ $a := b + c$ ” 的代码都翻译成下面的汇编代码：

```
1  mov b, r0  //把b装入寄存器r0
2  add c, r0  //把c加到r0上
3  mov r0, a  //把r0存入a
```

 复制代码


那么，下面两句代码：

```
1  a := b + c
2  d := a + e
```

 复制代码

将被机械地翻译成：

```
1  mov b, r0
2  add c, r0
3  mov r0, a
4  mov a, r0
5  add e, r0
6  mov r0, d
```

 复制代码


你可以从上面这段代码中看到，第 4 行其实是多余的，因为 r0 的值就是 a，不用再装载一遍了。另外，如果后面的代码不会用到 a（也就是说 a 只是个临时变量），那么第 3 行也是多余的。

这种算法很幼稚，正确性没有问题，但代码量太大，代价太高。所以我们最好用聪明一点儿的算法来生成更加优化的代码。**这是我们要做指令选择的原因之一。**

**做指令选择的第二个原因是**，实现同一种功能可以使用多种指令，特别是 CISC 指令集（可替代的选择很多，但各自有适用的场景）。

对于某个 CPU 来说，完成同样的任务可以采用不同的指令。比如，实现 “ $a := a + 1$ ”，可以生成三条代码：

```
1 mov a, r0
2 add $1, r0
3 mov r0, a
```

 复制代码


也可以直接用一行代码，采用 inc 指令，而我们要看看用哪种方法总体代价最低：

```
1 inc a
```

 复制代码

第二个例子，把 r0 寄存器置为 0，也可以有多个方法：

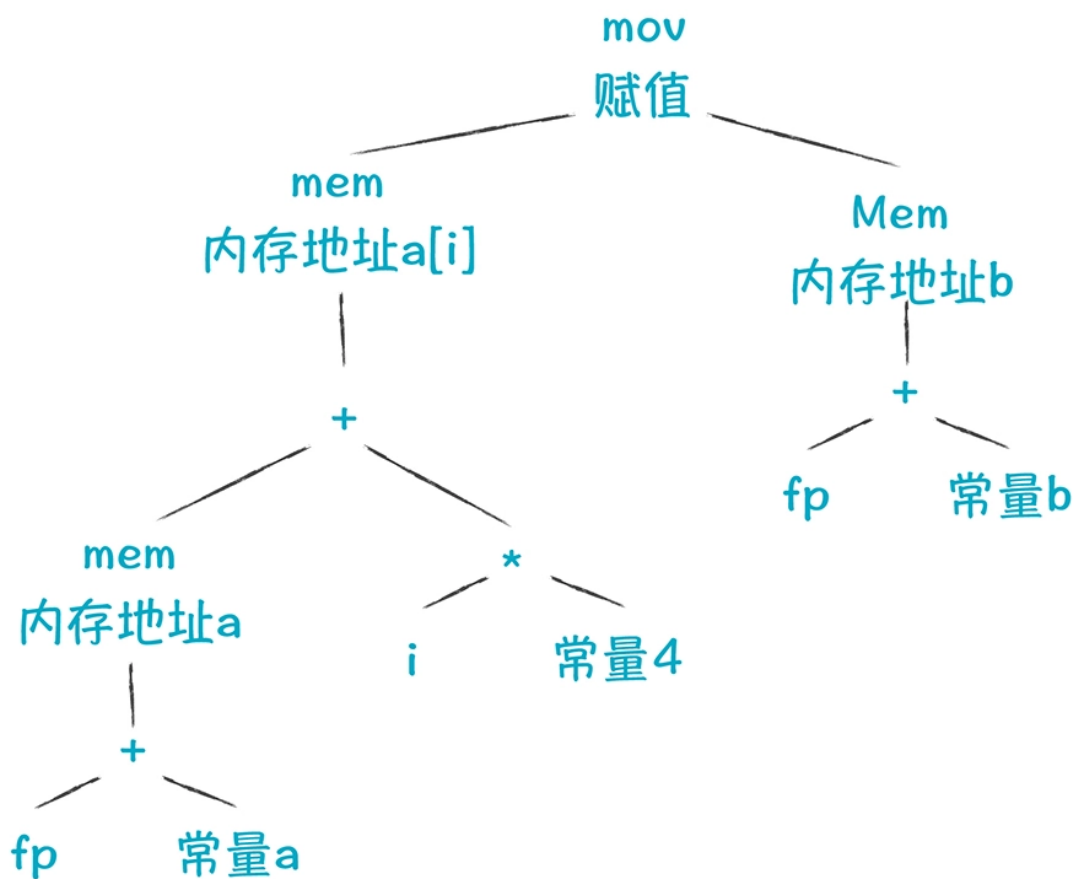
```
1 mov $0, r0    //赋值为立即数0
2 xor r0, r0    //异或操作
3 sub r0, r0    //用自身的值去减
4 ...
```

 复制代码

再比如， $a * 7$  可以用  $a \ll 3 - a$  实现：首先移位 3 位，相当于乘 8，然后再减去一次  $a$ ，就相当于乘以 7。虽然用了两条指令，但是，可能消耗的总的时钟周期更少。

**在这里我想再次强调一下**，无论是为了生成更简短的代码，还是从多种可能的指令中选择最优的，我们确实需要关注指令的选择。那么，我们做指令选择的思路是什么呢？目前最成熟的算法都是基于树覆盖的方法，我通过一个例子带你了解一下，**什么是树覆盖算法**。

$a[i] = b$  这个表达式的意思是，给数组  $a$  的第  $i$  个元素赋值为  $b$ 。假设  $a$  和  $b$  都是栈里的本地变量， $i$  是放在寄存器  $ri$  中。这个表达式可以用一个 AST 表示。



你可能觉得这棵树看着像 AST，但又不大像，那是因为里面有 `mem` 节点（意思是存入内存）、`mov` 节点、栈指针 (`fp`)。它可以算作低级 (low-level) AST，是一种 IR 的表达方式，有时被称为结构化 IR。这个 AST 里面包含了丰富的运行时的细节信息，相当于把 LLVM 的 IR 用树结构来表示了。你可以把一个基本块的指令都画成这样的树状结构。

基于这棵树，我们可以翻译成汇编代码：

```

1 load M[fp+a], r1 //取出数组开头的地址，放入r1，fp是栈帧的指针，a是地址的偏移量
2 addi 4, r2       //把4加载到r2
3 mul ri, r2       //把ri的值乘到r2上，即i*4，即数组元素的偏移量，每个元素4字节
4 add r2, r1       //把r2加到r1上，也就是算出a[i]的地址
5 load M[fp+b], r2 //把b的值加载到r2寄存器
6 store r2, M[r1]  //把r2写入地址为r1的内存

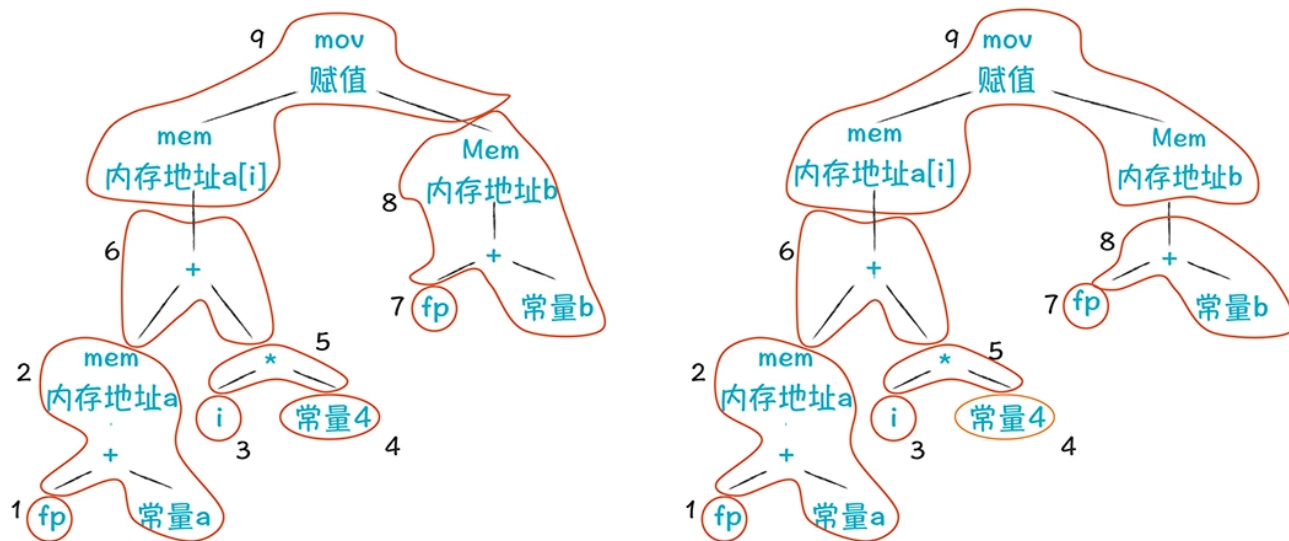
```

在这里，我用了一种假想的汇编代码，跟 LLVM IR 有点儿像，但更简化、易读：

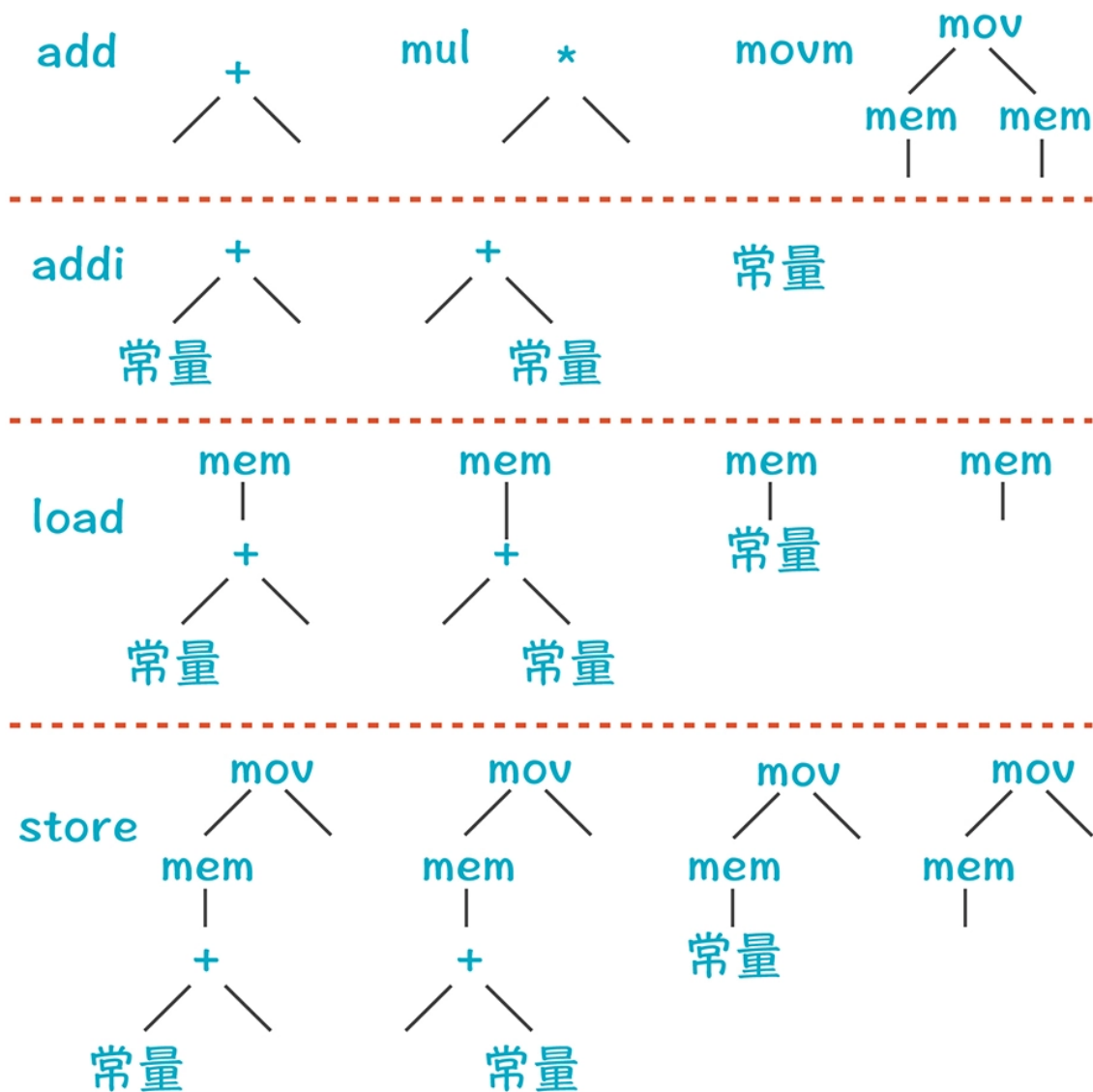
汇编代码	含义
load	从内存装载到寄存器
store	从寄存器保存回内存
add	从源寄存器加到目的寄存器
addi	常量加到目的寄存器
mul	从源寄存器乘到目的寄存器
movm	从一个内存地址拷贝到另一个内存地址
M[x]	引用地址为x的内存
fp	栈顶指针

**注意**，我们生成的汇编代码还是比较精简的。如果采用比较幼稚的方法，逐个树节点进行翻译，代码会很多，你可以手工翻译试试看。

用树覆盖的方法可以大大减少代码量，其中用橙色的线包围的部分被形象地叫做**一个瓦片 (tiling)**，那些包含了操作符的瓦片，就可以转化成一条指令。每个瓦片可以覆盖多个节点，所以生成的指令比较少。



那我们是用什么来做瓦片的呢？原来，每条机器指令，都会对应 IR 的一些模式（Pattern），可以表示成一些小的树，而这些小树就可以当作瓦片：

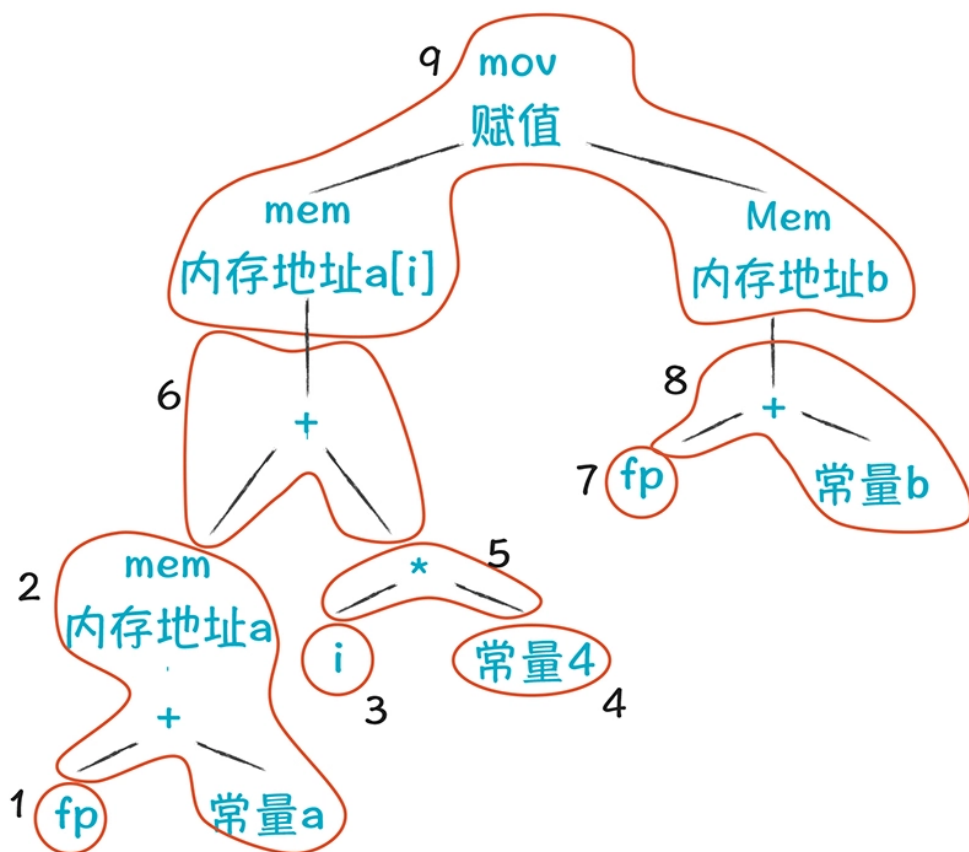


我们的算法可以遍历 AST，遇到上面的模式，就可以生成对应的指令。以 **load 指令为例**，它有几个模式：任意一个节点加上一个常量就行，这相当于汇编语言中的间接地址访问；或者 mem 下直接就是一个常量就行，这相当于是直接地址访问。最后，地址值还可以由下级子节点计算出来。

所以，从一棵 AST 生成代码的过程，就是用上面这些小树去匹配一棵大树，并把整个大树覆盖的过程，所以叫做树覆盖算法。2、4、5、6、8、9 这几个节点依次生成汇编代码。

要注意的是，覆盖方式可能会有多个，比如下面这个覆盖方式，相比之前的结果，它在 8 和 9 两个瓦片上是有区别的：





生成的汇编代码最后两句也不同：

[复制代码](#)

```
1 load M[fp+a], r1 //取出数组开头的地址，放入r1，fp是栈帧的指针，a是地址的偏移量
2 addi 4, r2 //把4加载到r2
3 mul ri, r2 //把ri的值乘到r2上，即i*4，即数组元素的偏移量，每个元素4字节
4 add r2, r1 //把r2加到r1上，也就是算出a[i]的地址
5 addi fp+b, r2 //把fp+b的值加载到r2寄存器
6 movm M[r2], M[r1] //把地址为r2到值拷贝到地址为r1内存里
```

你可以体会一下，这两个覆盖方式的差别：

对于瓦片 8 中的加法运算，一个当做了间接地址的计算，一个就是当成加法；

对于根节点的操作，一个翻译成从 store，把寄存器中的 b 的值写入到内存。一个翻译成 movm 指令，直接在内存之间拷贝值。至于这两种翻译方法哪种更好，比较总体的性能哪个更高就行了。



到目前为止，你已经直观地了解了为什么要进行指令选择，以及最常用的树覆盖方法了。当然了，树覆盖算法有很多，比如 Maximal Munch 算法、动态规划算法、树文法等，LLVM 也有自己的算法。

**简单地说一下 Maximal Munch 算法。**Maximal Munch 直译成中文，是每次尽量咬一大口的意思。具体来说，就是从树根开始，每次挑一个能覆盖最多节点的瓦片，这样就形成几棵子树。对每棵子树也都用相同的策略，这样会使得生成的指令是最少的。注意，指令的顺序要反过来，按照深度优先的策略，先是叶子，再是树根。这个算法是 Optimal 的算法。

Optimal 被翻译成最佳，我不太赞同这种翻译方法，翻译成“较优”会比较合适，它指的是在局部，相邻的两个瓦片不可能连接成代价更低的瓦片。覆盖算法除了 Optimal 的还有 Optimum 的，Optimum 是全局最优化的状态，就是代码总体的代价是最低的。

关于其他算法的细节在本节课就不展开了，因为根据我的经验，在学指令选择时，最重要的还是建立图形化的、直观的理解，理解什么是瓦片，如何覆盖会得到最优的结果。

接下来，我们继续探讨开篇提到的第二个问题：寄存器分配。

## 分配寄存器

寄存器优化的任务是：最大程度地利用寄存器，但不要超过寄存器总数量的限制。

因为我们生成 IR 时，是不知道目标机器的信息的，也就不知道目标机器到底有几个寄存器可以用，所以我们在 IR 中可以使用无限个临时变量，每个临时变量都代表一个寄存器。

现在既然要生成针对目标机器的代码，也就知道这些信息了，那么就要把原来的 IR 改写一下，以便使用寄存器时不超标。

那么寄存器优化的原理是什么呢？**我用一个例子带你了解一下。**

下图左边的 IR 中，a、d、f 这三个临时变量不会同时出现。假设 a 和 d 在这个代码块之后成了死变量，那么这三个变量可以共用同一个寄存器，就像右边显示的那样：

$a := b + c$

$r1 := r2 + r3$

$d := a + e$



$r1 := r1 + r4$

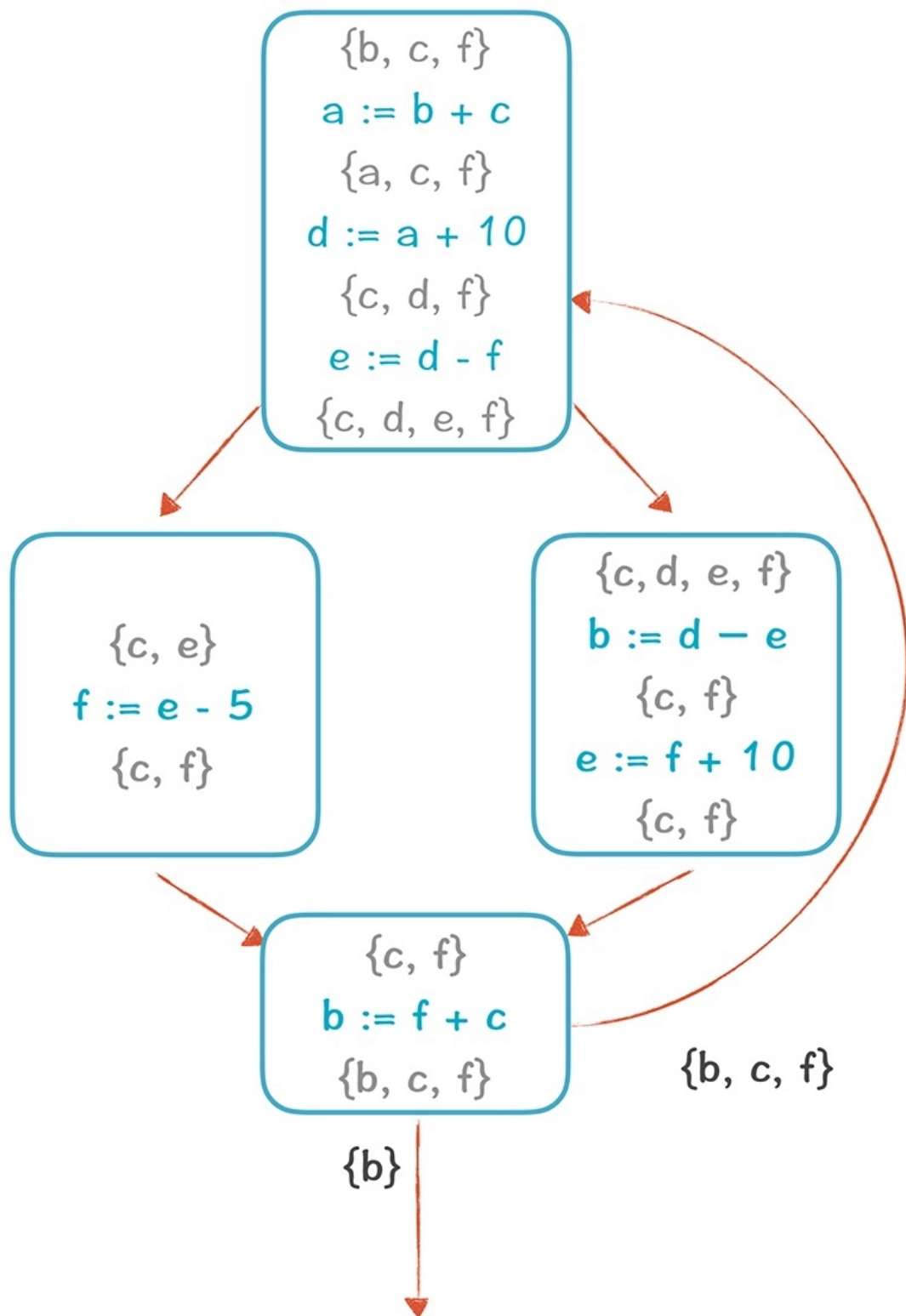
$f := d + 10$

$r1 := r1 + 10$

实际上，这三行代码是对“ $b + c + e + 10$ ”这个表达式的翻译，所以  $a$  和  $d$  都是在转换为 IR 时引入的中间变量，用完就不用了。这和在第 23 讲，我们把 8 个参数以及一个本地变量相加时，只用了一个寄存器来一直保存累加结果，是一样的。

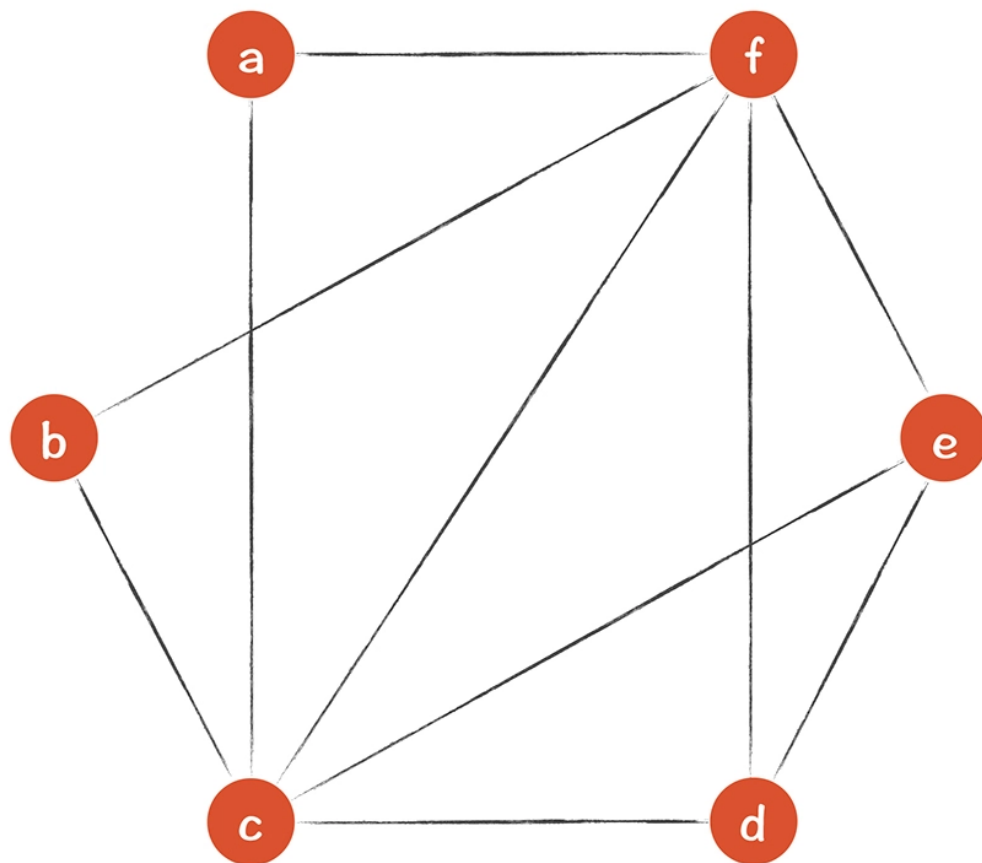
所以，通过这个例子，**你可以直观地理解寄存器共享的原则**：如果存在两个临时变量  $a$  和  $b$ ，它们在整个程序执行过程中，最多只有一个变量是活跃的，那么这两个变量可以共享同一个寄存器。

在第 27 讲和第 28 讲中，你已经学过了如何做变量的活跃性分析，所以你可以很容易分析出，在任何一个程序点，活跃变量的集合。然后，你再看一下，哪些变量从来没有出现在同一个集合中就行。**看看下面的这个图：**



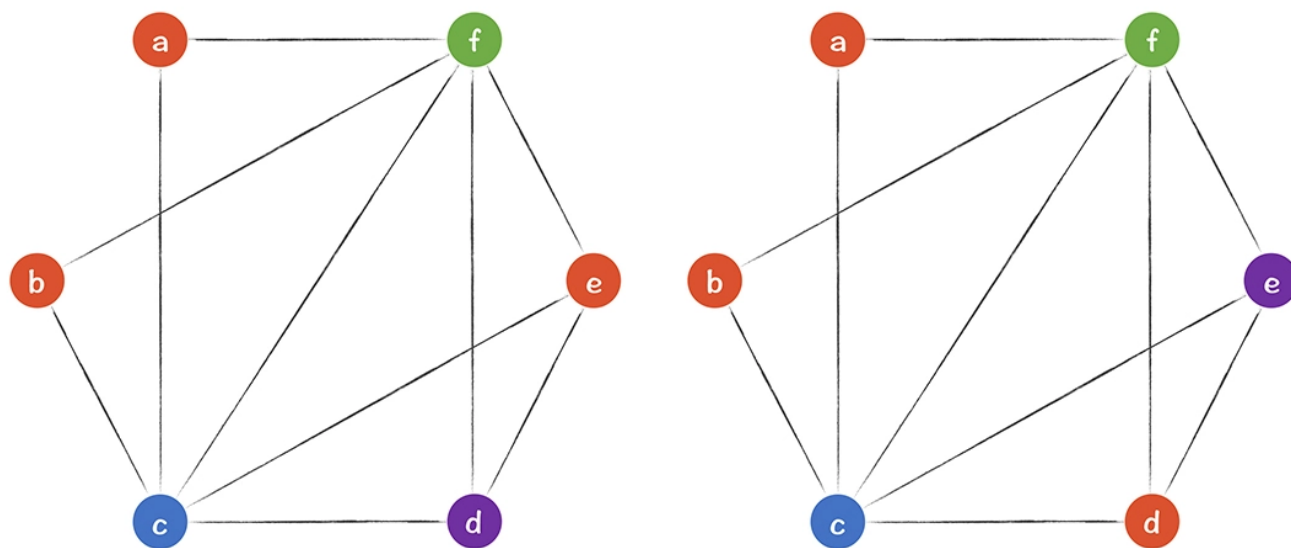
上图中，凡是出现在同一个花括号里的变量，都不能共享寄存器，因为它们在某个时刻是同时活跃的。那 a 到 f，哪些变量从来没碰到过呢？我们再画一个图来寻找一下。

下图中，每个临时变量作为一个节点，如果两个变量同时存在过，就画一条边。这样形成的图，叫做寄存器干扰图 (Register Interference Graph, RIG)。在这张图里，凡是没有连线的两个变量，就可以分配到同一个寄存器，例如，a 和 b，b 和 d，a 和 d，b 和 e，a 和 e。



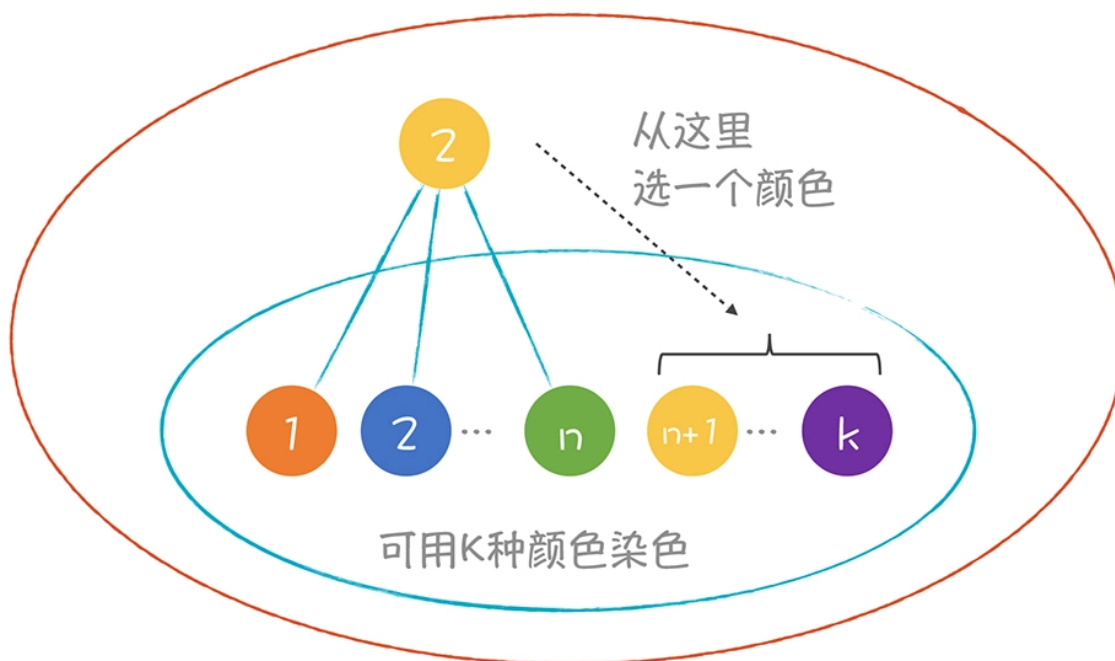
**那么问题来了：**针对这个程序，我们一共需要几个寄存器？怎么分配呢？

**一个比较常用的算法是图染色算法：**只要两个节点之间有连线，节点就染成不同的颜色。最后所需要的最少颜色，就是所需要的寄存器的数量。我画了两个染色方案，都是需要 4 种颜色：



不过我们是手工染色的，那么如何用算法来染色呢？假如一共有 4 个寄存器，我们想用算法知道寄存器是否够用？**应该如何染色？**

染色算法很简单。如果想知道  $k$  个寄存器够不够用，你只需要找到一个少于  $k$  条边的节点，把它从图中去掉。接着再找下一个少于  $k$  条边的节点，再去掉。如果最后整个图都被删掉了，那么这个图一定可以用  $k$  种颜色来染色。

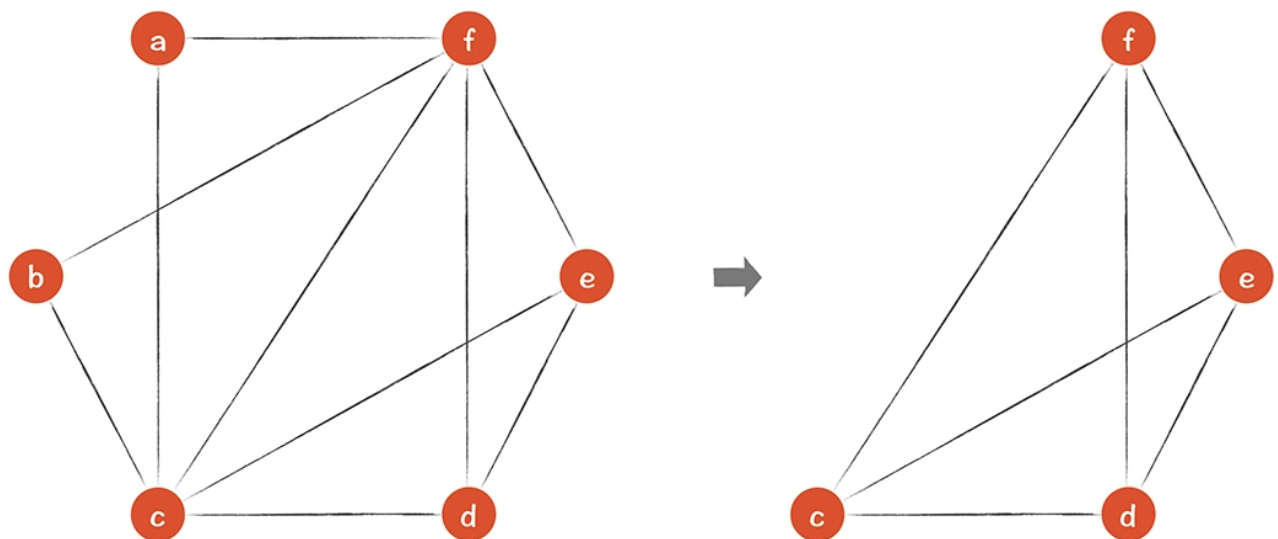


**为什么呢？** 因为如果一个图（蓝色边的）是能用  $k$  种颜色染色的，那么再加上一个节点，它的边的数量少于  $k$  个，比如是  $n$ ，那么这个大一点儿的图（橙色边的）还是可以用  $k$  种颜色染色的。道理很简单，因为加进来的节点的边数少于  $k$  个，所以一定能找到一个颜色，与这个点的  $n$  个邻居都不相同。

所以，我们把刚才一个个去掉节点的顺序反过来，把一个个节点依次加到图上，每加上一个，就找一个它的邻居没有用的颜色来染色就行了。整个方法简单易行。

但是，如果所需要寄存器比实际寄存器的数量多，该怎么办呢？当然是用栈了。这个问题就是寄存器溢出（Register Spilling），溢出到栈里去，我在 [@21 讲](#) 关于运行时机制时提到过，像本地变量、参数、返回值等，都尽量用寄存器，如果寄存器不够用，那就放到栈里。另外再说一下，无论放在寄存器里，还是栈里，都是活动记录的组成部分，所以活动记录这个概念比栈帧更广义。

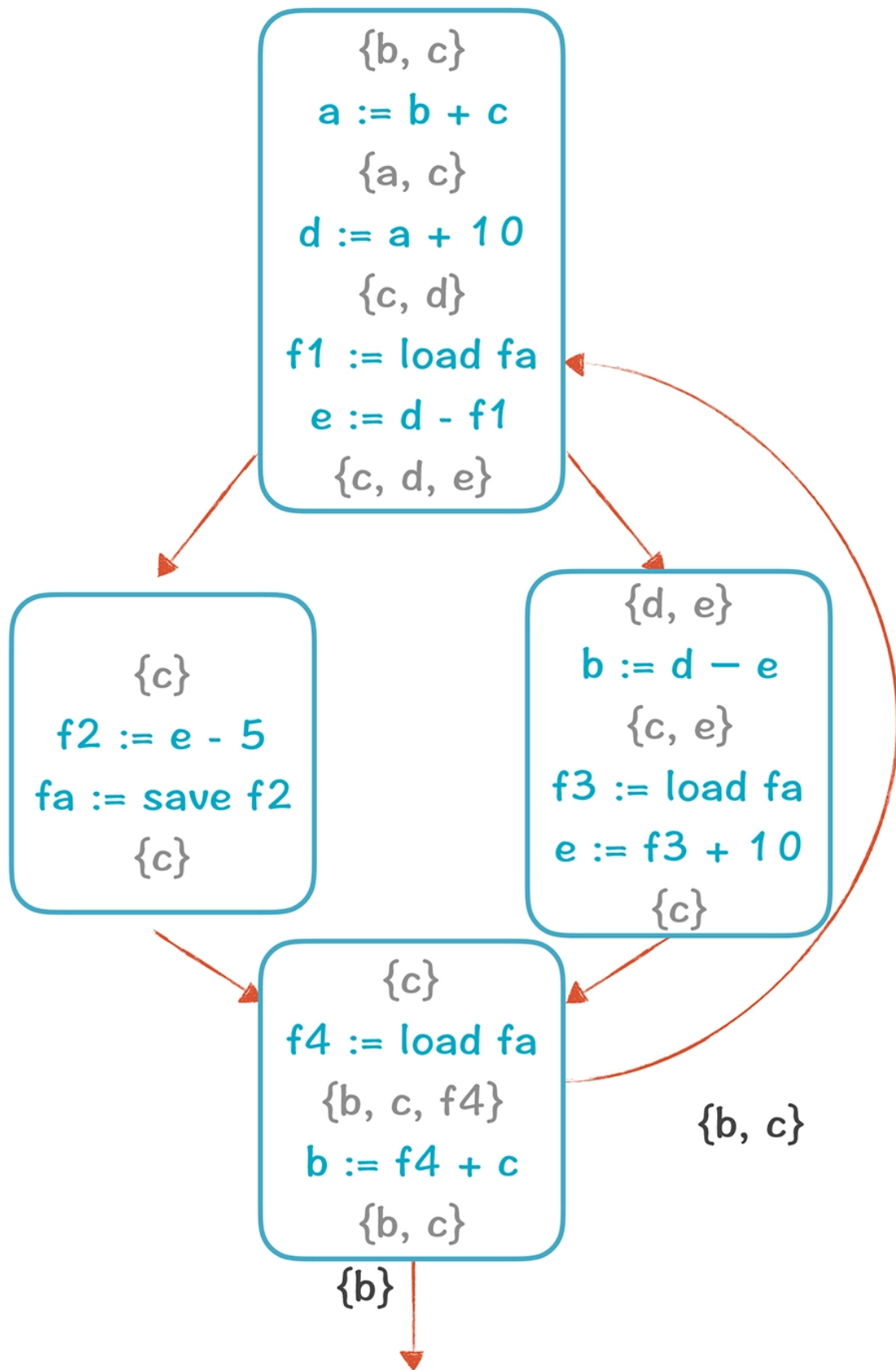
**还是拿上面的例子来说，** 如果只有 3 个寄存器，那么要计算一下 3 个寄存器够不够用。我们先把  $a$  和  $b$  从图中去掉：



这时你发现，剩下的 4 个节点，每个节点都有 3 个邻居。所以，3 个寄存器肯定不够用，必须要溢出一个去。我们可以选择让  $f$  保存在栈里，把  $f$  去掉以后，剩下的  $c, d, e$  可以用 3 种颜色成功染色。

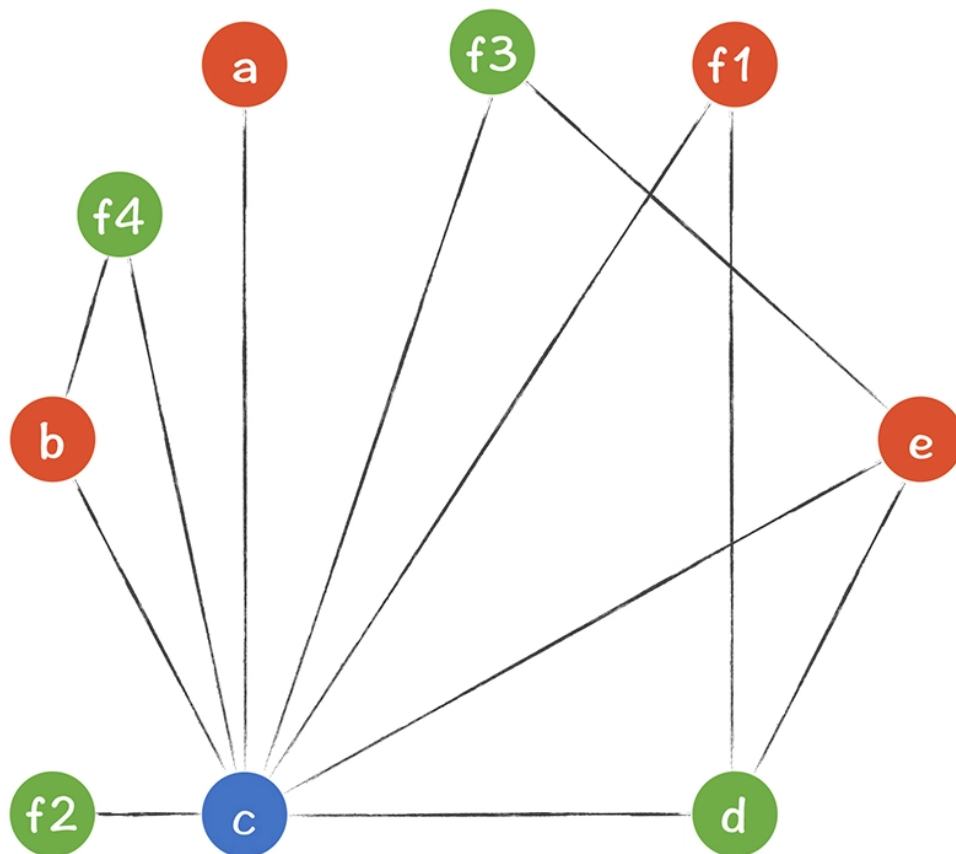
这就结束了吗？当然没有。f 虽然被保存到了栈里，但每次使用它的时候，都要 load 到一个临时变量，也就是寄存器中。每次保存 f，也都要用一个临时变量写入到内存。所以，我们要把原来的代码修改一下，把每个使用 f 的地方，都加上一条 load 或 save 指令，以便在使用 f 的时候把 f 放到寄存器，用完后再写回内存。**修改后的 CFG 如下：**





因为原来有 4 个地方用到了  $f$ ，所以我们引入了  $f1$  到  $f4$  四个临时变量。这样的话，总的临时变量反而变多了，从 6 个到了 9 个。不过没关系，虽然临时变量更多了，但这几个临时变量

的生存期都很短，图里带有 f 的活跃变量集合，比之前少多了。所以，即使有 9 个临时变量，也能用三种颜色染色，如下图所示：



最后，在选择把哪个变量溢出的时候，你实际上是要有所选择的。你最好选择使用次数最少的变量。在程序内循环中的变量，就最好不要溢出，因为每次循环都会用到它们，还是放在寄存器里性能更高。

目前为止，代码生成中的第二项重要工作，分配寄存器就概要地讲完了。我留给你一段时间消化本节课的内容，在下一讲，我会接着讲指令重排序和 LLVM 的实现。

## 课程小结

目标代码生成过程中有三个关键知识点：指令选择、寄存器分配和指令重排序，本节课，我讲了前两个，期望能帮你理解这两个问题的实质，让你对指令选择和寄存器分配这两个问题建立直观理解。这样你再去研究不同的算法时，脑海里会有这两个概念的顶层的、图形化的认识，事半功倍。与此同时，本节课我希望你记住几个要点如下：

相同的 IR 可以由不同的机器指令序列来实现。你要理解瓦片为什么长那个样子，并且在脑子里建立用瓦片覆盖一棵 AST 的直观印象，最好具备多种覆盖方式，从而把这个问题由抽象变得具象。

寄存器分配是编译器必须要做的一项工作，它把可以使用无限多寄存器的 IR，变成了满足物理寄存器数量的 IR，超出的要溢出到内存中保管。染色算法是其中一个可行的算法。

## 一课一思

关于指令选择，你是否知道其他的例子，让同一个功能可以用不同的指令实现？欢迎在留言区分享你的经验。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

© 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。

## 精选留言 (9)



沉淀的梦想

2019-11-01

最近几讲有种在学《算法导论》的感觉，感觉学编译原理真的能够帮助我们贯通整个计算机科学，涉及到的东西好多

作者回复：是。你总结得很对。编译原理要把计算机组成、数据结构、算法、操作系统，以及离散数学中的某些知识都用上。

比如，我们讲到指令选择、寄存器分配和指令排序，都是NP Complete的。这个时候，如果提前已经知道什么是NP Complete，那么马上就对这类算法有概念，也马上想到可以用什么样的方式来对待这类问题。

再比如，编译原理中很多算法都是基于树和图的，所以对这两个数据结构的理解也会有帮助。

至于计算机组成原理，跟后端技术的关联很密切。

程序运行时的环境，内存管理等内容，则跟操作系统有关。



6



**Allen\_Go**

2022-09-11 来自广东

感觉跟不上了，那就先过一遍，再啃几遍慢慢消化了，还是有信心拿下的。



**linfei**

2022-07-03

按那个算法，能把图删空确实说明该图可以用k种颜色着色，但如果不能删空却不说明它不能用k种颜色着色。例如四个节点，按照1-2-3-4-1这样的方式连接起来，每个节点都有2条边，但它是可以用2种颜色来着色的。



**if...else...**

2021-10-24

每篇都能学到新知识



**lion\_fly**

2020-12-24

染色算法，看起来很像数学中的四色问题

作者回复: 有共同点。

另一个同学也说过，跟约束求解有相似性。只不过，编译器中的算法特别强调速度，所以一般不用那种通用的求解器。



**Geek\_89bbab**

2020-05-06

像  $f1 := \text{load } fa$  这个指令，fa也需要一个寄存器存储吧，f1也需要一个寄存器存储。它们可以共用一个寄存器。



**瓜瓜**

2020-02-07

但是，如果需要寄存器比实际寄存器的数量少，该怎么办呢

-----  
这个是不是写错了??

作者回复: 是的，文稿写错了，刚好写反了！感谢你帮忙检查出来;-D



写点啥呢

2019-11-05

请问老师，对于需要通过栈保存寄存器溢出的变量，在使用的时候是不是还是要占用一个寄存器呀？比如文章中的最后例子，硬件是3个寄存器约束，溢出一个变量作为临时变量，但是后段代码生成的时候，是不是其实还是需要4个寄存器（load/save指令都需要一个寄存器的）？

作者回复: 不是的。

对于溢出到内存里的变量，在读（load）和写（store）的时候，确实要使用一个寄存器。但是使用的时间很短。所以你看看我配的图，之前很多个集合里都有f。溢出到内存以后，含有f1、f2、f3、f4的集合，反而少了。这就导致再次分配寄存器的时候，3个就够了。文稿里有这个推导过程，你仔细看一下。



ʘ(●°▽°●)ʘ

2019-10-30

老师，已经跟不上了... 还是好希望我们最后有没有类似研究一下实现一下图查询的sql，如gsq l标准。或者js2019的开源大佬的实现类的成果

作者回复: 后端需要更多对计算机组成等知识的理解，确实会有点挑战。

后端算法的特点也不一样，往往都是NP-Complete的，不追求最优解，次优解就挺好。这方面在思维上也要适应一下。

相对来说，前端比较纯粹。基本上把逻辑搞清楚就行了，而且肯定有最优解。

图查询？有同事用过node4j。我本人并没有深入研究过。不过，从编译原理的角度，这都是不同的应用领域。语言的部分，编译技术可以解决，它只是个接口，是个封装；落地机制部分，要运用每个领域的知识，比如关系数据库的原理、图数据库的原理。

