10　过程间优化

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 掌握内联优化
- 掌握尾递归优化

过程间优化指的是对函数调用进行优化，利用函数调用的上下文信息优化函数调用和代码运行。本节主要介绍一种通用的过程间优化问题：内联，以及一类特殊的内联问题：尾递归。

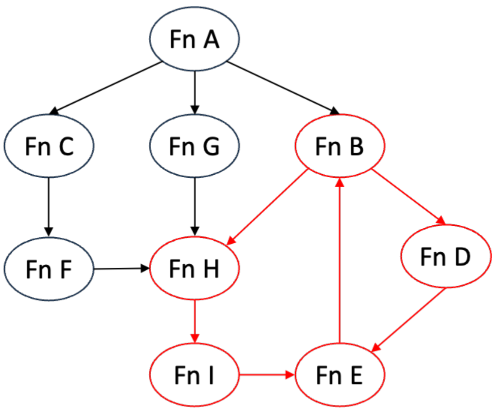
10.1　内联优化

内联指的是将 callee 的函数体复制到 caller 中，并使得程序等价的代码转换方法。内联的好处是可以减少函数调用开销，也会带来一些新的过程内优化可能。其缺点是会增大代码体积和编译时间。因此，内联并非一定越多越好。在现代编程语言中，程序员可以对需要内联的函数手动标注，编译器会根据标注情况进行内联。

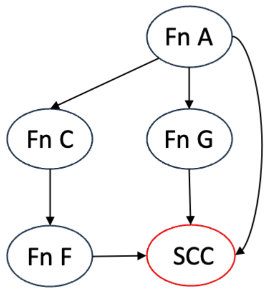
10.1.1　问题建模

本节讨论一种自动内联问题，即编译器如何在没有内联标注的情况下自动选取内联的 callsites，达到最优的内联优化效果。由于函数的调用关系可以表示为有向有环图，该问题相当于如何在图上选取特定的边进行内联，使得在预算有限的情况下收益最大。

10.1.2　调用图预处理



(a) 函数调用图



(b) 强连通分量缩环后的调用图

图 10.1: 基于 GVN 的优化

解决内联问题一般先需要对函数调用图进行预处理，消除其中递归调用引入的环或强连通分量。以图 10.1a为例，其中包含一个强连通分量 B-D-E-H-I。由于内联涉及选取顺序（bottom-up 或 top-down）问题，一般不对强连通分量进行内联。

有向图的强连通分量检测可以采用经典的 Tarjan 算法 [1]（算法 1）。下面以图 10.1a为例分析 Tarjan 算法的运算过程。

表 10.1: 应用 Tarjan 算法检测图 10.1a中的强联通分量。

步骤	**Stack**	**Time**	**ArriveTime, NextArriveTime**									**SCC**
			A	B	C	D	E	F	G	H	I	
1	A	1	1,1									
2	A, C	2	1,1		2,2							
3	A, C, F	3	1,1		2,2			3,3			*	
4	A, C, F, H	4	1,1		2,2			3,3		4,4		
5	A, C, F, H, I	5	1,1		2,2			3,3		4,4	5,5	
6	A, C, F, H, I, E	6	1,1		2,2		6,6	3,3		4,4	5,5	
7	A, C, F, H, I, E, B	7	1,1	7,7	2,2		6,6	3,3		4,4	5,5	
8	A, C, F, H, I, E, B, H	8	1,1	7,4	2,2		6,6	3,3		4,4	5,5	
9	A, C, F, H, I, E, B, D	8	1,1	7,4	2,2	8,8	6,6	3,3		4,4	5,5	
10	A, C, F, H, I, E, B, D, E	8	1,1	7,4	2,2	8,6	6,6	3,3		4,4	5,5	
11	A, C, F, H, I, E, B	8	1,1	7,4	2,2	8,6	6,6	3,3		4,4	5,5	
12	A, C, F, H, I, E	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,5	
13	A, C, F, H, I	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	
14	A, C, F, H	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D
15	A, C, F	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D, F
16	A, C	8	1,1	7,4	2,2	8,6	6,4	3,3		4,4	5,4	H-I-E-B-D, F, C
17	A, G	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C
18	A, G, H	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C
19	A, G	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G
20	A, B	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G
21	A	2	1,1	7,4	2,2	8,6	6,4	3,3	2,2	4,4	5,4	H-I-E-B-D, F, C, G, A

算法 1 Tarjan 强联通分量检测算法

$t \leftarrow 0$; // time of arrival
procedure $\textsc{Visit}(v)$
    $Arrive[v] \leftarrow t$; // 记录每个节点的到达时间
    $NextArrive[v] \leftarrow t$; // 记录下一跳的最早到达时间
    $t \leftarrow t + 1$;
    S.push(v);
    for each n in v.next() **do**
        if $Arrive[n] == 0$ **then**
            Visit(n);
            $NextArrive[v] \leftarrow \min(NextArrive[v], NextArrive[n])$;
        else if s.contains(n) **then**
            $NextArrive[v] \leftarrow \min(NextArrive[v], Arrive[n])$;
        end if
    end for
    if $NextArrive[v] == Arrive[v]$ **then** // 找到强联通分量
        $scc \leftarrow$ pop S until v;
        SCC.add(scc);
    end if
end procedure

10.1.3 贪心式内联优化算法

从有向无环图中选取边进行内联的问题可建模为背包问题，属于 NP-hard 问题。算法 2介绍了一种基于贪心方法求解的思路。该方法首先对所有边的内联收益进行排序，然后优先选取收益最大、且不超过总预算的边进行内联。

算法 2 贪心式内联优化算法

$S \leftarrow \emptyset$; // 记录可以被内联的函数调用
$C \leftarrow 0$; // 记录内联代价
procedure SEARCHINLINE(v)
    for each e in E **do**
        if inlineable(w) **then** // 排除不可内联的函数调用，如间接调用
            BenefitEstimation(e);
            S.insert(e); // 基于收益排序
        end if
    end for
    for each e in S **do**
        $cost \leftarrow$ CostEstimation(e);
        $C \leftarrow C + cost$;
        if $C > budget$ **then** // 排除不可内联的函数调用，如间接调用
            S.remove(e); // 基于收益排序
        end if
    end for
end procedure

10.2 尾递归优化

如果函数返回语句之前的最后一条指令是调用自己，则称为尾递归调用。代码 10.1展示了一个尾递归调用的例子，而代码 10.2则不是尾递归。尾递归是一种特殊的递归调用，可以专门对其进行内联优化。

```
fn fac(n:int, r:int) -> int {
    if (n < 2) {
        ret r;
    }
    else {
        ret fac(n-1, n*r);
    }
}
```

代码 10.1: 尾递归形式的阶乘算法 TeaPL 代码

```
fn fac(n:int) -> int {
    if (n < 2) {
        ret 1;
    }
    else {
        ret n * fac(n-1);
    }
}
```

代码 10.2: 非尾递归代码

尾递归调用内联的过程称为尾递归消除。与一般内联不同，该过程无需拷贝函数体，在递归调用处将参数保存到原参数变量中并且跳转到函数入口即可。代码 10.3以中间代码形式展示了尾递归调用代码 10.1的递归调用消除方式。该方法的本质是使用循环替换递归调用，可以进一步设计算法对这种循环进行优化。如，代码 10.4将代码 10.3中 bb1 的内容复制到 bb3 中，从而消除针对变量%n 的一次冗余的 store 和 load。

```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
  %n = alloca i32
  %r = alloca i32
  store i32 %n0, i32* %n
  store i32 %r0, i32* %r
  br label %bb1
bb1:
  %n1 = load i32, i32* %n
  %t0 = icmp slt i32 %n1, 2
  br i1 %t0, label %bb2, label %bb3
bb2:
  %r1 = load i32, i32* %r
  ret i32 %r1
bb3:
  %n2 = load i32, i32* %n
  %r2 = load i32, i32* %r
  %t1 = sub i32 %n2, 1
  %t2 = mul i32 %n2, %r2
  ; 优化前: %t3 = call i32 @fac(i32 %t1, i32 %t2)
  ; 优化前: ret i32 %t3
  store i32 %t1, %n
  store i32 %t2, %r
  br %bb1
}
```

代码 10.3: 尾递归消除

```
define i32 @fac(i32 %n0, i32 %r0) {
bb0:
  %n = alloca i32
  %r = alloca i32
  store i32 %n0, i32* %n
  store i32 %r0, i32* %r
  br label %bb1
bb1:
  %n1 = load i32, i32* %n
  %t0 = icmp slt i32 %n1, 2
  br i1 %t0, label %bb2, label %bb3
bb2:
  %r1 = load i32, i32* %r
  ret i32 %r1
bb3:
  %n2 = load i32, i32* %n
  %r2 = load i32, i32* %r
  %t1 = sub i32 %n2, 1
  %t2 = mul i32 %n2, %r2
```

```
  store i32 %t1, %n
  store i32 %t2, %r
  %t3 = icmp lt i32 %n3, 2;
  br i1 %t3 label %bb2, label %bb3
}
```

代码 10.4: 尾递归优化

Bibliography

[1] Robert Tarjan. "Depth-first search and linear graph algorithms." SIAM journal on computing 1, no. 2 (1972): 146-160.