

## 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

过程间优化指的是利用函数调用的上下文信息优化函数调用和代码运行。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。编译器也支持一些自动化的内联优化分析。本节主要介绍一种通用的过程间优化问题: 内联优化, 以及一类特殊的内联优化问题: 尾递归。

### 10.1 内联优化分析

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。并非所有的内联都是有优化收益的, 下面我们对影响内联优化收益的因素进行讨论。

#### 10.1.1 收益分析

内联产生优化收益的主要原因有两点:

- 减少函数调用开销;
- 带来新的过程内优化可能。

函数调用开销的优化效果与参数个数以及是否有返回值正相关, 具体可表示为以下形式:

$$Cost = C(jmp_1) + \sum_i^n C(par_i) + C(jmp_2) + C(retval)$$

其中,  $C(jmp_1)$  和  $C(jmp_2)$  分别为跳转到 callee 以及跳回到 caller 的开销;  $\sum_i^n C(par_i)$  为参数传递开销,  $n$  为参数的个数;  $C(retval)$  为返回值传递开销。因此, 如果 callee 没有参数和返回值, 则内联优化收益比较一般。另外, 如果 callsites 在循环内部, 则该内联会在运行时带来更为明显的优化效果。

只有当函数有参数或返回值时内联才能带来新的优化可能。具体来说, 参数可以为 callee 优化带来可能, 例如参数为常量时通过常量传播进行优化; 类似的, 返回值为常量时可以为 caller 中 callsites 后续的代码带来优化可能。

#### 10.1.2 副作用分析

函数内联可能的副作用有三点, 其中前两点最为关键:

- 增大代码体积: 如果同一个 callee 在多个 callsites 出现, 内联该函数会导致代码体积膨胀。这是因为 callee 的代码会被复制到多个调用点, 增加了重复代码。另外, 如果一个 callee 为 public API, 内联该 callee 后不能将该 callee 删除, 因此会增加重复代码。
- 增加指令读取开销: 函数内联将 callee 的代码复制到 caller 中, 通常会导致 caller 体积增大, 可能引发不同程度的 cache miss 问题。尤其是当 callee 中包含循环时, 内联使得原来的循环被拆分到不同的 cache line 中, 增加引发 cache miss 的频率。

- 加剧寄存器分配负担：一般认为，将 callee 的寄存器值复制到 caller 内部会导致更复杂的寄存器干扰，需要使用更多寄存器。但实际上，如果处理得当，这种影响几乎可以忽略不计。具体来说，如果 callsite 之前赋值的寄存器在调用后仍需使用，可能会导致被调用方缺少可用寄存器，从而需要将寄存器值溢出到内存。此时，只需按照函数调用约定选择相同的寄存器溢出即可。

结合上述因素，一个函数越小，则将其内联带来副作用的概率越小。其它情况下，内联是否会带来正优化收益取决于很多因素，难以给出确切的建议 [1]。编译器一般不会主动对循环和递归调用（尾递归除外）进行内联。

## 10.2 内联优化算法

### 10.2.1 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。

### 10.2.2 调用图预处理

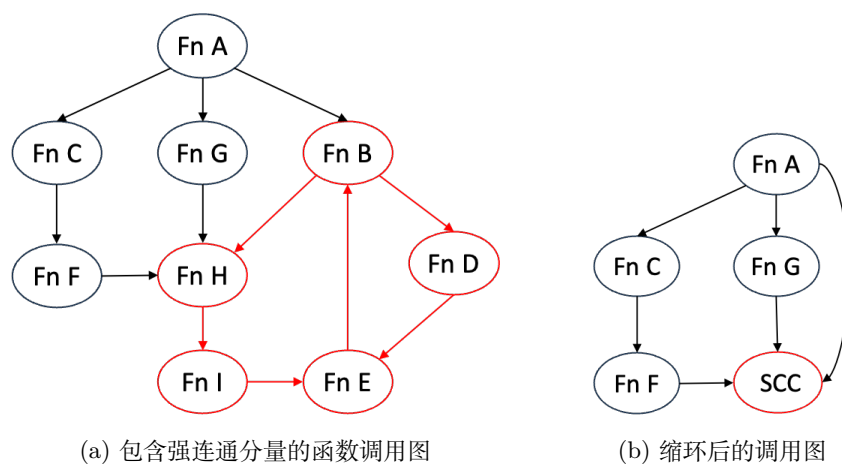


图 10.1: 调用图预处理

函数调用图与控制流图不同，其中的循环不一定是自然循环。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或对其进行单独内联。

有向图的强连通分量检测可以采用经典的 Tarjan 算法 [2]（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

```

1:  $t \leftarrow 0$ ; // time of arrival
2: procedure VISIT( $v$ )
3:    $Arrive[v] \leftarrow t$ ; // 记录每个节点的到达时间
4:    $NextArrive[v] \leftarrow t$ ; // 记录下一跳的最早到达时间
5:    $t \leftarrow t + 1$ ;
6:    $S.push(v)$ ;
7:   for each  $n$  in  $v.next()$  do
8:     if  $Arrive[n] == 0$  then
9:       Visit( $n$ );
10:     $NextArrive[v] \leftarrow \min(NextArrive[v], NextArrive[n])$ ;
11:   else if  $s.contains(n)$  then
12:      $NextArrive[v] \leftarrow \min(NextArrive[v], Arrive[n])$ ;
13:   end if
14: end for
15: if  $NextArrive[v] == Arrive[v]$  then // 找到强联通分量
16:    $scc \leftarrow \text{pop } S \text{ until } v$ ;
17:    $SCC.add(scc)$ ;
18: end if
19: end procedure

```

### 10.2.3 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。具体的收益和开销可以通过启发式算法计算获得，也可以针对不同 callsites 内联后实际的收益评测获得。

---

**算法 2** 贪心式内联优化算法

---

```
1:  $S \leftarrow \emptyset$ ; // 记录可以被内联的函数调用
2:  $C \leftarrow 0$ ; // 记录内联代价
3: procedure SEARCHINLINE( $v$ )
4:   for each  $e$  in  $E$  do
5:     if inlineable( $w$ ) then // 排除不可内联的函数调用，如间接调用
6:       BenefitEstimation( $e$ );
7:        $S.insert(e)$ ; // 基于收益排序
8:     end if
9:   end for
10:  for each  $e$  in  $S$  do
11:     $cost \leftarrow CostEstimation(e)$ ;
12:     $C \leftarrow C + cost$ ;
13:    if  $C > budget$  then // 排除不可内联的函数调用，如间接调用
14:       $S.remove(e)$ ; // 基于收益排序
15:    end if
16:  end for
17: end procedure
```

---

## 10.3 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。以阶乘为例，代码 10.1 实现了一个尾递归版本，而代码 10.2 则非尾递归。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: TeaPL 代码：尾递归形式的阶乘算法

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

代码 10.2: TeaPL 代码：非尾递归形式的阶乘算法

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 IR 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。由于无 phi 指令形式的 IR 更容易反应出最终的优化效果，此处我们采用非完全 SSA 形式进行演示。

```
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
    %n3 = sub i32 %n2, 1
    %r3 = mul i32 %n2, %r2
    ; 优化前: %t1 = call i32 @fac(i32 %n3, i32 %r3)
    ; 优化前: ret i32 %t1
    store i32 %n3, %n
    store i32 %n2, %r
    br %bb1
}
```

代码 10.3: 尾递归消除

将尾递归转化为循环后，可以进一步设计算法对这种循环进行优化。例如，代码 10.4 将代码 10.3 中 bb1 的内容复制到 bb3 中，从而消除循环体内针对变量 %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
```

```

%n3 = sub i32 %n2, 1
%r3 = mul i32 %n2, %r2
store i32 %r3, %r
%t1 = icmp lt i32 %n3, 2;
br i1 %t1 label %bb2, label %bb3
}

```

代码 10.4: 尾递归优化

## 练习

1. 将下列代码翻译为 C 或 Rust 语言，并对比在内联和非内联时的性能表现。

```

fn callee(a[]: int, l:int) {
    let i = 0;
    while (i<l) {
        a[i] = a[i] + 1;
        i = i + 1;
    }
}

fn caller() {
    let i = 0;
    let a[1000]:int = {0};
    while (i<1000) {
        callee(a);
    }
}

```

代码 10.5: TeaPL 代码：内联可能加剧指令读取开销

2. 宏的实现方法与内联有很多相似之处，在编译预处理阶段对宏进行字符串替换。对比这两种方式的差别，请举例说明。

```

macro_rules! foo(i)
    i + i;
}

fn main() {
    foo!(10); // 替换为 10 + 10;
    foo!(f(x)); // 替换为 ?
}

```

代码 10.6: TeaPL 代码：宏

## Bibliography

- [1] Inline Functions, <https://isocpp.org/wiki/faq/inline-functions#inline-and-perf>.
- [2] Robert Tarjan. “Depth-first search and linear graph algorithms.” SIAM journal on computing 1, no. 2 (1972): 146-160.