# 第十一章 代 码 优 化

# 11.1 概述

经过前面各章所介绍的各个编译阶段，我们已经可以把源程序变换成目标代码了，但这样生成的目标代码效率未必是最高的。

代码优化（code optimization)

目的：提高目标代码运行效率 { 时间效率（减少运行时间）；空间效率（减少内存容量）}

原则：进行优化必须严格遵循“不能改变原有程序语义”原则。

# 优化的分类：

★从优化的层次，与机器是否有关，分为：

- 独立于机器的优化：即与目标机无关的优化，通常是在中间代码上进行的优化。
- 与机器有关的优化：充分利用系统资源（指令系统，寄存器资源）。

★从优化涉及的范围，又分为：

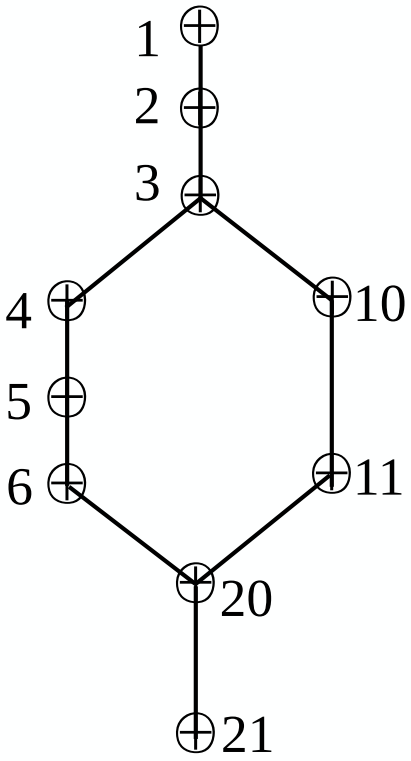
- 局部优化：是指在基本块内进行的优化。
- 循环优化：对循环语句所生成的中间代码序列上所进行的优化。
- 全局优化：顾名思义，跨越多个基本块的全局范围内的优化。因此它是指在非线性程序段上（包括多个基本块, GOTO, 循环）的优化。需要进行全局控制流和数据流分析，比较复杂。

# [定义]基本块（basic block )

满足以下三个条件的程序段，称为基本块:

- 只有一个入口和一个出口，且语句为顺序执行的程序段。
- 它使得所有转向该基本块的入口都是该基本块的第一条语句。
- 如果块中任一语句被执行，则该块内的所有语句也将被执行（无分支），且执行次数一样（无循环）。

# 例：一个FORTRAN语言例子




```
1.   FACTOR = A( I ) + 2.0 * B( I )
2.   EXP1 = ABS( FACTOR )
3.   IF ( KEY.NE.0 ) GO TO  10
4.   BASE = 2.0
5.   FACTOR = FACTOR ** 2
6.   GO TO  20
10.  BASE = 10.0
11.  FACTOR = SQRT( FACTOR )
20.  Q = ( BASE ** EXP ) * FACTOR
21.  RETURN
```

基本块

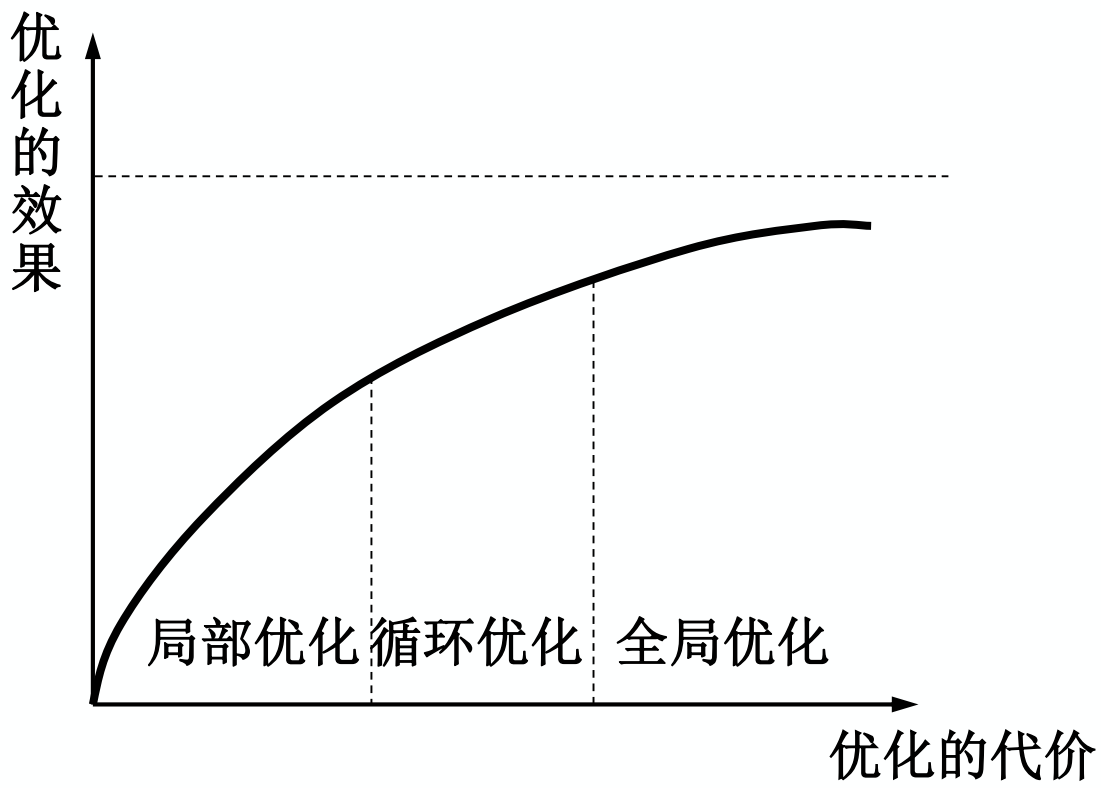
基本块

基本块

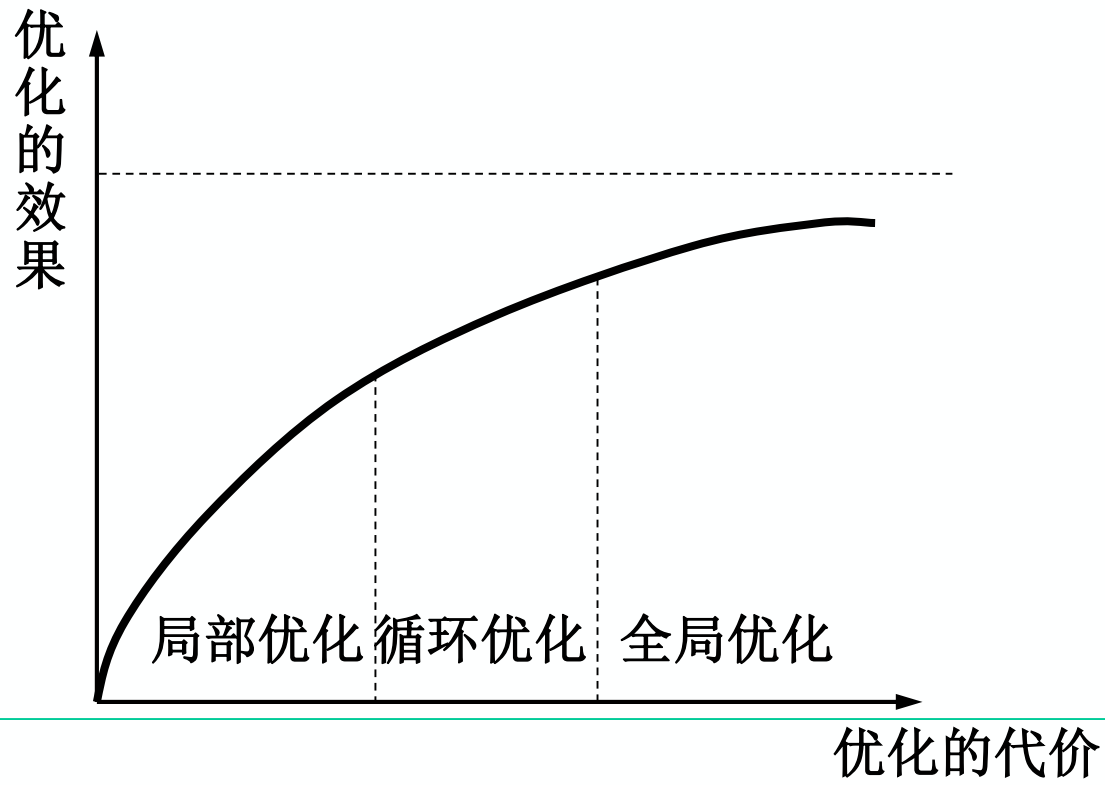
基本块

（先编号，后画图→决定基本块）

优化所花费的代价和优化产生的效果可用下图表示：



- 图的左部表示只要做些简单的处理，便能得到明显的优化效果。这相当于局部优化。
- 若要进一步提高优化效果，就要逐步付出更大的代价——循环优化。
- 全局优化进一步提高。



# 为什么要优化?

- 有的大型计算程序一运行就要花上几十分钟，甚至好几小时。这时为优化即使付出些代价也是值得的。
- 另外，程序中的循环往往要占用大量的计算时间，所以为减少循环执行时间所进行的优化对减少整个程序的运行时间有很大的意义。——尤其有实时要求的程序，如市场决策，供需及求益的平衡。
- 对于像学生作业之类的简单小程序(占机器内存、运行速度均可接受)，或在程序的调试阶段，花费许多代价去进行一遍又一遍的优化就毫无必要。

# 11.2 优化的基本方法和例子

在这一节中，我们将介绍一些有代表性的优化处理方法和例子。实际的优化应在中间代码或目标代码上进行。但为了便于说明，这儿我们用源程序形式举例。

## （1）利用代数性质(代数变换)

- 编译时完成常量表达式的计算，整数类型与实型的转换。

例: a := 5 + 6 + x → a := 11 + x

又如: 设 x 为实型，x := 3 + 1 可变换成x := 4.0

- 下标变量引用时，其地址计算的一部分工作可在编译时预先做好(运行时只需计算“可变部分”即可)。

例如：二维数组定义为：

Array[ $l_1 : u_1$, $l_2 : u_2$ ]

令$d_i = u_i - l_i + 1$, $i = 1, 2$ //每一维的尺寸

假如数组元素按行存放，则Array[ $i_1, i_2$ ]的位置为

$D = a + ((i_1 - l_1)\, d_2 + (i_2 - l_2)) * m$ // m为每个元素所占大小

$= a - (l_1 d_2 + l_2) * m + (i_1 d_2 + i_2) * m$

= CONSPART + VARPART

● 运算强度削弱：用一种需要较少执行时间的运算代替另一种运算，以减少运行时的运算强度(时、空开销)。

如　　x ** 2 → x * x

3 * x → x + x + x

8 * x ，　4 * x　等换成左移运算

x / 2 ，　x / 16　等换成右移运算

x := x + 1　　变为INC x指令

x / 5 → x * 0.2 等

利用机器硬件所提供的一些功能，如左移、右移操作做乘法或除法，具有更高的代码效率。

## （2）复写(copy)传播

如 x := y 这样的赋值语句称为复写语句。由于 x 和 y 值相同，所以当满足一定条件时，在该赋值语句下面出现的 x 可用 y 来代替。

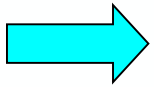
例如:

```
x := y ;
u := 2 * x ;
v := x + 1 ;
```

若以后的语句中不再用到 x 时，则上面的 x := y 可删去。

若上例中不是x := y而是 x := 3。则复写传播变成了常量传播。即

```
x := y;
u := 2 * x;
v := x + 1;
```



```
x := 3;
u := 2 * x;
v := x + 1;
```

u := 6;  v := 4;

又如  $t_1 := y / z;$  $x := t_1;$

若这里 $t_1$为暂时(中间)变量，以后不再使用，则可变换为

x := y / z;

此外常量传播，引起常量计算，如:

pi = 3.14159  r = pi / 180.0

此时: pi = 3.14159  r = 0.0174532  （常量计算）

## （3）删除公共子表达式

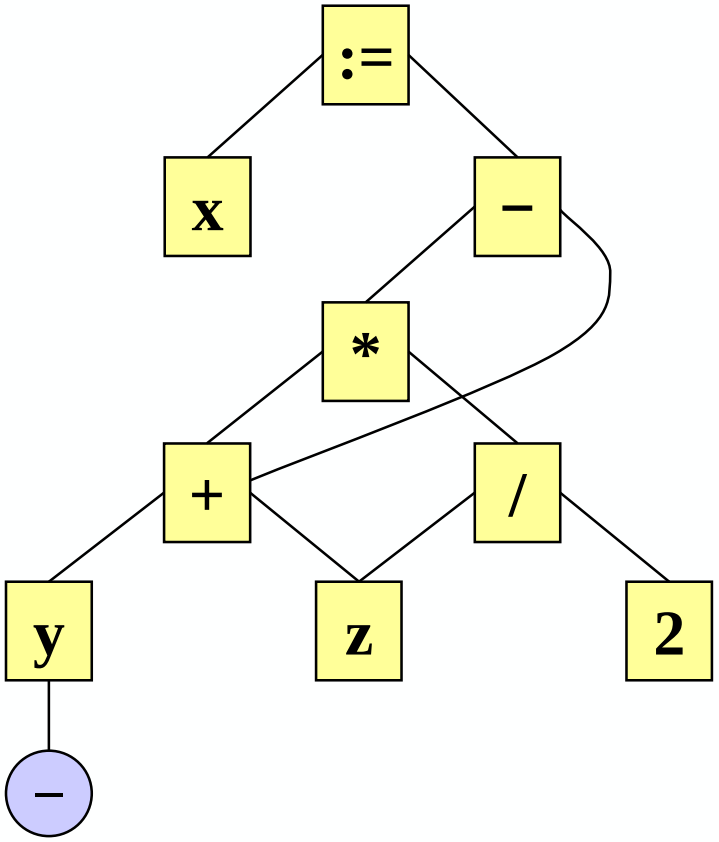
具有相同值的子表达式在两个以上地方出现时，称它为公共子表达式(common subexpression)。

例如：x := ( -y + z ) * z / 2 - ( -y + z )

其中：( -y + z )是公共子表达式。

我们可用DAG (directed acyclic graph，有向无循环图)来表示具有公共子表达式的抽象语法树。

**x := ( -y + z ) * z / 2 - ( -y + z )**




显然，对于公共子表达式只要计算1次即可。

## （4）删除冗余代码

**冗余代码就是毫无实际意义的代码，又称死代码 (dead code)或无用代码(useless code)。**

**例如:** **x := x + 0;** **x := x * 1;** 等

又例:

```
FLAG := TRUE
IF  FLAG  THEN…
  …
ELSE…
```

FLAG永真

另外在程序中为了调试常有如下:

if debug then … 的语句。

但当debug为false时，then后面的语句便永远不会执行。这就是可删去的冗余代码。

(可用条件编译 #if DEBUG 程序编写，而源代码中还应留着)

## （5）循环优化

经验告诉我们：“程序运行时间的80%是由仅占源程序20%的部分执行的”。——二八定律(Pareto Principle)

这20%的源程序就是循环部分，特别是多重循环的最内层的循环部分。因此减少循环部分的目标代码对提高整个程序的时间效率有很大作用。

```
for  i = 1  to  10
  for   j = 1  to  100
      x := x + 0 ;
      y := 5 + 7 + x ;
```

优化一条，少10*100次运算

除了对循环体进行优化，还有专用于循环的优化

## a) 循环不变式的代码外提

不变表达式：不随循环控制变量改变而改变的表达式或子表达式。

如：FOR I := $E_1$ STEP $E_2$ TO $E_3$ DO

BEGIN

S := 0.2 * 3.1416 * R

P := 0.35 * I

V := S * P

……

不变表达式可外提

不能外提!

如　　**while　…　do**

**x := …（b * b - 4.0 * a * c）…**

若a, b, c的值在该循环体中不改变时，则可将循环不变式移到循环之外，即变为:

$t_1$ **:= b * b - 4.0 * a * c**

**while　…　do**

**x := …(** $t_1$**) …**

从而减少计算次数——也称为频度削弱。

## b) 循环展开

循环展开是一种优化技术。它将构成循环体的代码(不包括控制循环的测试和转移部分)，重新产生许多次(这可在编译时确定)，而不仅仅是一次。以空间换时间!

例：PL/1中的初始化循环

```
DO  I = 1  TO  30
    A[ I ] = 0.0
END
```

展开

```
    I := 1
L1: IF  I > 30  THEN
    GOTO  L2
    A[ I ] = 0.0
    I = I+1
    GOTO  L1
L2:
```

代码5条语句
共执行5*30
条语句

```
A[1] = 0.0
A[2] = 0.0
……
A[30] = 0.0
```

30条语句
（指令）执行
也是30条语句

## 说明：

- 循环一次执行5条语句才给一个变量赋初值。展开后，一条语句就能赋一个值，运行效率高。
- 优化在生成代码时进行，并不是修改源程序。
- 必须知道循环的初值、终值及步长。
- 但非所有展开都是合适的。如上例中循环展开后节省了测试和转移语句：2*30=60语句。

完成具体业务逻辑的代码 / 辅助性代码

∴增加29条省60条

但若循环体中不是一条而是40条，则展开将有40*30=1200条，但省的仍是60条，就不算优化了。

∴判断准则：1. 主存资源丰富
处理机时间昂贵
2. 循环体语句越少越好

→ 循环展开有利（大型机）

```
DO    I = 1   TO   30
      A[ I ] = 0.0
END
```

实现步骤：

1. 识别循环结构，确定循环的初值、终值和步长。
2. 判断。以空间换时间是否合算来决定是否展开。
3. 展开。重复产生循环体所需的代码个数。

比较复杂：

∴在对空间与时间进行权衡时，还可以考虑一种折衷的办法，即部分展开循环。如上例展为：

```
DO    I = 1   TO   30   STEP   3
      A[ I ] = 0.0
      A[ I + 1 ] = 0.0
      A[ I + 2 ] = 0.0
END;
```

空间只多二条，但省了20次测试时间（只循环10次）

## c) 归纳变量的优化和条件判断的替换

归纳变量(induction variable): 在每一次执行循环迭代的过程中，若某变量的值固定增加（或减少）一个常量值，则称该变量为归纳变量(induction variable)。即若当前执行循环的第 j 次迭代，归纳变量的值应为c * j + c′, 这里 c 和 c′ 都是循环不变式。

例:
```
for  i := 1  to  10  do
    a[i] := b[i]  +  c[i]
```

```
for  i:= 1    to    10    do
     a[i] := b[i]  +  c[i]
```

```
1)  i := 1
2)  labb:
3)  if  i > 10   goto   labe
4)  t1 :=  4 * i
5)  t2 :=  b [ t1 ]
6)  t3 :=  4 * i
7)  t4 :=  c [ t3 ]
8)  t5 :=  t2 + t4
9)  t6 :=  4 * i
10) a [ t6 ] := t5
11)  i := i + 1
12) goto   labb
13) labe:
```

优化

```
1)  u := 4
2)  labb:
3)  if u >40   goto   labe
4)  tb := b [ u ]
5)  tc := c [ u ]
6)  t  :=  tb + tc
7)  a[ u ] := t
8)  u := u + 4
9)  goto   labb
10) labe:
```

中间变量$t_1$，$t_3$，$t_6$都是归纳变量。

$t_1 := 4 * i$，　$t_3 := 4 * i$，　$t_6 := 4 * i$

## d) 其它循环优化方法

- 把多重嵌套的循环变成单层循环。
- 把 n 个相同形式的循环合成一个循环等。

对于循环优化的效果是很明显的。某FORTRAN 77 编译程序，在进行不同级别的优化后所得的目标代码指令数为：

| 优化级别 | 循环内的指令数（包括循环条件判断） |
|---|---|
| 0（不优化） | 21 |
| 1（局部优化） | 16 |
| 2（循环优化） | 6 |
| 3（全局优化） | 5 |

## （6） in_line 展开

把过程（或函数）调用改为in_line展开可节省许多处理过程（函数）调用所花费的开销。

如： **procedure m( i , j : integer;  max : integer);**

**begin  if  i  > j  then  max := i  else  max := j  end;**

若有过程调用 **m ( k , 0, max );**

则内置展开后为:

**if  k > 0  then  max := k  else  max := 0;**

省去了函数调用时参数压栈，保存返回地址等指令。这也仅仅限于简单的函数。

## （7）其他， 如控制流方法

如　　BR　　L　　无条件转移

…　　——为不可达代码

L :

又如：转移到转移指令的指令

BR　L1　　BR　L2

…　　优化　　…

L1: BR　L2　　L1: BR　L2

还有:

$BR_{CC}$ L1 当条件CC成立，转到L1

BR L2

L1: ….

可改进为:

$BR'_{CC}$ L2 当条件不能成立时，转到L2

L1: …

此外还有其它方法。且随着软件技术的飞速发展，不断会有新的优化方法推出。

练习：作常数合并优化的表达式属性翻译文法及语义动作程序（中间代码——栈式抽象机指令）。

例： **A := 2 + 3 + C**

**A := 5 + C**

# 总结：

优化分为两大类

- 与机器无关的优化，即独立于机器的（中间）代码优化
  - 局部优化：（一个入口，一个出口，线性）——基本块
    - 方法：常数合并；冗余子表达式的削除等
  - 循环优化：对循环语句所生成的中间代码序列上所进行的优化
    - 方法：循环展开；频度削弱；循环不变表达式的外提；强度削弱
  - 全局优化：顾名思义，跨越多个基本块的全局范围内的优化。因此它是在非线性程序段上（包括多个基本块, GOTO循环）的优化。
- 与机器有关的优化，即目标代码上的优化（与具体机器有关）