# 编译原理核心：循环优化入门教程

## 引言：为何要优化循环？

在软件性能优化的世界里，有一个广为人知的“90/10法则”：一个程序90%的执行时间，往往消耗在仅仅10%的代码上[1]。而这关键的10%代码，绝大多数情况下都潜藏在程序的循环结构中。正如教学材料中所指出的，循环在计算机程序中无处不在（pervasive），它们是程序执行时间的“热点”（hot spots），因此对循环进行优化是提升程序性能最关键、最有效的手段[3]。

循环优化的技术多种多样，涵盖了从简单的代码移动到复杂的结构变换，例如循环不变式上提（Loop invariant hoisting）、归纳变量削减（Induction variable reduction）、循环展开（Loop unrolling）、循环合并（Loop fusion）等等[3]。这些技术的共同目标是减少循环体内的计算量、降低控制开销或改善内存访问模式。对循环体内部指令的任何微小改进，其效果都会随着循环的成千上万次迭代而被急剧放大。这使得编译器设计者愿意投入大量精力去开发复杂的循环分析与优化技术，因为这是投入产出比最高的性能提升策略。

对于初学者而言，直接深入所有这些高级技术可能会感到困惑。因此，本教程将遵循浙江大学课程PPT的框架[3]，聚焦于循环优化的两大基石：

1. 如何精确地识别循环？ 这需要我们理解一些看似抽象但至关重要的图论概念，如支配结点和自然循环。
2. 如何执行最基础且最有效的优化？ 我们将深入学习循环不变式上提（Loop-Invariant Code Motion, LICM），这是几乎所有现代编译器都会实现的经典优化。

通过掌握这些基础，你将能理解编译器是如何“看待”和“改造”循环的，为进一步学习更高级的编译技术打下坚实的基础。

## 第一部分：理解循环的本质——支配结点与自然循环

在编译器开始优化之前，它必须首先能准确无误地识别出程序中的循环。这并不像人类阅读代码那样简单地寻找for或while关键字。编译器需要一种不依赖于具体语法、能够处理各种复杂控制流（包括goto、break、continue）的通用方法。这种方法建立在对程序控制流图（Control-Flow Graph, CFG）的严谨分析之上。

### 1.1 编译器的“循环”视角

在编译器的世界里，“循环”不是一个模糊的概念，而是一个具有严格图论定义的结构。这种定义专为优化目的而设计，其核心思想是确保循环有一个唯一的入口[1]。

根据PPT的定义，一个控制流图中的循环是满足以下三个条件的结点集合S，其中包含一个特殊的“首结点”（header）h[3]：

1. 从S中的任何一个结点出发，都存在一条路径能够回到首结点h。这保证了集合内的结点能够形成“循环”。
2. 从首结点h出发，存在一条路径能够到达S中的任何一个结点。这保证了首结点是循环的一部分，并且能够控制整个循环体。
3. （核心）除了通往首结点h的边之外，不存在任何从S外部结点指向S内部结点的边。这意味着首结点是进入该循环的唯一入口。

让我们通过图示来理解这个定义的精髓[3]。在图(c)和(d)中，结点1是唯一的入口，所有进入循环的控制流都必须先经过它，这符合定义。然而，在图(a)和(b)中，虽然存在环路，但外部结点可以直接跳转到环内的非首结点（如b或c），这导致了多个入口。这种结构在编译器看来不是一个“循环”，因为它给优化带来了极大的困难。

为什么“单入口”如此重要？因为这个唯一的首结点h为编译器提供了一个清晰的“把手”（handle）[3]。后续的许多优化，比如我们即将学习的循环不变式上提，需要将某些计算从循环内部移动到循环开始之前。有了唯一的入口，编译器就知道应该把这些代码放在哪里，从而保证优化既安全又有效。

这种严格的定义也引出了可规约图（**Reducible Graphs**）和 不可规约图（**Irreducible Graphs**）的概念[3]。大多数高级语言（如Java）通过结构化的控制流（如

if-then-else, while, for）自然地生成可规约图，其中的所有环路都满足我们对循环的定义。然而，像C/C++中不受限制的goto语句，则可能产生包含多入口环路的不可规约图，这会使许多标准的循环优化算法失效。

**1.2** 寻找循环的“把手”：支配结点

为了找到满足单入口定义的循环，编译器引入了一个更为基础和强大的概念——支配（**Dominance**）。

支配结点的定义与计算

一个结点d支配（dominates）另一个结点n，当且仅当从程序的入口结点到结点n的每一条可能的执行路径都必须经过d [3]。这个定义形式化了程序控制流中“必经之路”的概念。需要注意的是，每个结点都支配它自身。

为了找出所有结点的支配者集合，编译器通常采用一种基于数据流分析的迭代算法 [3]。该算法的核心是以下数据流方程，其中

D[n]表示支配结点n的结点集合，pred[n]表示n的所有前驱结点集合：

D[s0]=s0D[n]=n∪(p∈pred[n]∩D[p])for n優=s0
算法的执行过程如下 [8]：

1. 初始化：将入口结点s0的支配集设为{s0}。对于所有其他结点n，将其支配集D[n]初始化为一个包含所有结点的全集（这是一个保守的初始假设）。
2. 迭代：反复遍历所有结点（除入口外），根据上述方程更新它们的支配集。每次更新都是取其所有前驱结点支配集的交集，再加上结点自身。
3. 终止：当一轮完整的遍历没有对任何结点的支配集产生改变时，算法达到不动点（**fixed point**），计算结束。

让我们通过PPT中的例子 [3] 来看看这个过程。下表清晰地展示了支配集的迭代计算过程：

表格**1**：支配集**D[n]**的迭代计算过程

| 结点 (n) | 初始化 D[n] | 迭代 1 | 迭代 2 (不动点) |
|---|---|---|---|
| 1 (入口) | {1} | {1} | {1} |
| 2 | {1,2,3,4} | {1,2} | {1,2} |
| 3 | {1,2,3,4} | {1,3} | {1,3} |

| | | | |
|---|---|---|---|
| 4 | {1,2,3,4} | {1,4} | {1,4} |

这个表格将抽象的公式D[n] = {n} U (∩ D[p])转化为一个具体、可跟踪的计算过程，让初学者可以一目了然地看到数据流分析中“迭代至不动点”的全过程，极大地降低了理解门槛。

直接支配结点与支配树

在结点n的所有支配者中，存在一个“最接近”n的支配者，我们称之为n的直接支配结点（**Immediate Dominator, idom**）[3]。除了入口结点，每个结点都有且仅有一个直接支配结点。

这种唯一的父子关系，使得我们可以将整个程序的支配关系组织成一棵支配树（**Dominator Tree**）[3]。在这棵树中，每个结点的父结点就是它的直接支配结点，而它的所有祖先结点就是它的所有支配者。从控制流图到支配树的转换，是编译器分析中的一次关键抽象。它将复杂的、网状的控制流关系，提炼成了一个简单的、层次化的树形结构。在这棵树上，判断

d是否支配n，就从一个复杂的图遍历问题，简化成了一个高效的树上祖先查询问题。这为后续快速识别循环提供了极大的便利。

**1.3** 识别真正的优化对象：自然循环

有了支配关系和支配树，我们就可以精确地定义和识别用于优化的循环结构了。

回边与自然循环

识别循环的关键在于找到回边（**Back Edge**）。一条从结点n指向结点h的边被称为回边，当且仅当h支配n（即h dom n）[3]。直观上，回边就是一条从图中某处“向后跳”到其必经之路上的边，这正是循环行为的明确信号。

每一个回边n -> h都唯一地定义了一个自然循环（**Natural Loop**）[2]。这个自然循环由两部分组成[3]：

1. 循环的首结点h。
2. 所有能够到达n，并且路径上不经过h的结点集合。

因此，寻找自然循环的算法可以通俗地描述为 [2]：

1. 计算控制流图的支配关系。
2. 找出所有的回边n -> h。
3. 对于每一个回边，从n开始，在反向的控制流图上进行遍历（但不能越过h）。所有能被遍历到的结点，再加上h本身，就构成了这个回边所对应的自然循环。

从PPT的例子中我们可以看到 [3]，回边

10 -> 5定义了由结点{5, 8, 9, 10}构成的自然循环。同时，一个首结点可能对应多个自然循环（如果有多条回边指向它），在这种情况下，编译器通常会将它们合并成一个更大的循环进行分析。

嵌套循环与循环嵌套树

当一个自然循环的所有结点被完全包含在另一个自然循环内，且它们的的首结点不同时，我们就称之为嵌套循环（**Nested Loop**）[3]。不包含任何其他循环的循环被称为

最内层循环（**Innermost Loops**），它们通常是优化的首要目标，因为它们的执行频率最高。

为了获得程序循环结构的全貌，编译器会构建一棵循环嵌套树（**Loop-Nest Tree**）[3]。这棵树的结点代表程序中的各个循环，父子关系表示循环的嵌套。树的根可以看作是整个函数体，而树的叶子结点则对应着最内层循环。这个宏观视图可以指导编译器按照从内到外的顺序，系统地进行循环优化。

**1.4** 为优化做准备：循环前置首结点

许多循环优化，特别是我们接下来要讨论的代码移动，需要在循环正式开始前插入一些“准备”代码。如果一个循环的首结点有多个前驱（例如，一个goto和一个顺序执行流都指向循环头），那么将准备代码插入到哪里就成了一个问题。在每个前驱都插入一份副本会造成代码冗余，且难以维护 [3]。

为了解决这个问题，编译器引入了一个简单的技巧：为循环创建一个前置首结点（**Loop Preheader**）[3]。这是一个专门为优化而创建的新基本块，它的唯一后继是原循环的首结点。所有原来指向循环首结点的边，现在都重定向到这个前置首结点。这样一来，无论循环有多少个入口路径，编译器都有了一个唯一的、干净的“准备区”来插入优化代码，极大地简化了后续的变换操作。

## 第二部分：核心优化实践——循环不变式上提

掌握了如何识别循环后，我们就可以开始真正的优化了。循环不变式上提（Loop-Invariant Code Motion, LICM），也常被称为代码外提（Hoisting），是最经典和最普遍的循环优化之一。它的核心思想是：将那些在每次循环中都计算出相同结果的“不变”代码，从循环体内移动到循环体外，只计算一次。

### **2.1** 找出循环中的**“不动”**代码：循环不变式

一个在循环中结果始终不变的表达式，被称为循环不变式（**Loop Invariant**）。一个形如x := v1 op v2的赋值语句是循环不变的，当且仅当它的所有操作数（v1, v2）满足以下条件之一[3]：

1. 操作数是常量。
2. 所有能到达该语句的操作数定义都来自于循环的外部。
3. 循环内部只有一个定义能到达该操作数，且该定义本身也是一个循环不变式。

注意，这是一个递归定义。编译器通过迭代的方式来找出所有的不变式：首先，将所有操作数都是常量或定义在循环外的语句标记为不变式；然后，利用这些已找到的不变式，去发现更多新的不变式；如此反复，直到没有新的不变式可以被发现为止。

让我们分析一下PPT中的例子[3]：

```
L1: i := i + 1
    b := 7
    t := a + b
```

```
  *i := t
  if i < N goto L1
```

- i := i + 1：不是不变式，因为i在每次迭代中都会改变。
- b := 7：是不变式，因为操作数7是一个常量。
- t := a + b：假设a的定义在循环外，而b刚刚被我们确定为不变式。因此，t := a + b也是不变式。
- *i := t：不是不变式，因为地址i在每次迭代中都在变化。

**2.2** 安全第一：循环不变代码移动的**“三大纪律”**

找到不变式只是第一步。将它移出循环看似简单，但稍有不慎就可能改变程序的原始逻辑，导致错误。正如PPT所强调的：“Just because code is loop invariant doesn't mean we can move it!” [3]。为了保证优化的绝对正确性，一次安全的循环不变式上提必须同时满足以下三个严格的条件 [3]。

1. 支配条件 **(Dominance Condition)**：包含不变式赋值语句d: t <- expr的基本块，必须支配所有在循环外使用变量t的循环出口。
   - 为何需要？ 这个条件防止了在某些执行路径上引入不必要的计算和错误的最终结果。考虑一个while循环，如果循环条件一开始就不满足，程序会直接跳到循环之后。如果在原程序中，不变式赋值语句d在这种情况下不会被执行，但优化后却被无条件地移动到了循环之前，那么当程序从一个d不支配的出口退出时，变量t的值就会被错误地改变，从而影响后续代码的正确性 [3]。
2. 唯一性条件 **(Uniqueness Condition)**：在循环中，被赋值的变量t只能有这唯一一个定义。
   - 为何需要？ 如果变量t在循环中被多次赋值（即使这些赋值都是不变的），上提其中一个定义就会破坏原有的程序逻辑。例如，循环中可能先有t = a + b，后有t = c + d。如果只上提了前者，那么在原程序中本应使用c + d结果的地方，现在会错误地使用a + b的结果 [3]。
3. 前置首结点条件 **(Pre-header Condition)**：被赋值的变量t在循环的前置首结点处不能是活跃的（**live-out**）。换句话说，在进入循环之前，t的旧值不能被需要。
   - 为何需要？ 这个条件为了防止破坏从循环外传入的数据。如果变量t在进入循环时是活跃的，说明它的值是从外部传入的，并且很可能在循环的第一次迭代中就会被使用。此时，如果我们将一个新的不变赋值t :=...上提到循环之前，就会在t的旧值被使用之前，过早地将其覆盖，导致第一次迭代出错 [3]。

这三个条件共同构成了一个完备的“安全网”，确保代码移动在逻辑上与原程序等价，体现

了所有编译器优化的黄金法则：在不改变程序语义的前提下提升性能[11]。

### 2.3 实战演练：安全上提案例分析

现在，让我们综合运用这“三大纪律”，来分析PPT[3]中的几个具体案例。下表提供了一个结构化的分析框架：

表格2：循环不变式上提安全性分析

| 案例[3] | 支配条件 | 唯一性条件 | 前置首结点活跃性条件 | 结论 | 详细原因 |
|---|---|---|---|---|---|
| 案例 1 | ✅ | ✅ | ✅ | 安全 (Correct) | 赋值语句支配唯一的循环出口，t在循环内只有一次定义，且在循环开始前不活跃。 |
| 案例 2 | ❌ | ✅ | ✅ | 不安全 (Incorrect) | 赋值语句不支配循环的初始入口（当i>=N时直接跳到L2）。如果循环不执行，t的值会被错误改变。 |
| 案例 3 | ✅ | ❌ | ✅ | 不安全 (Incorrect) | 循环内对t有两次定义。上提其中一个会破坏原有的交替赋值逻辑。 |
| 案例 4 | ✅ | ✅ | ❌ | 不安全 (Incorrect) | t在循环入口是活跃的，其初始值在第一次迭代中被使用。上提会过早覆盖该值。 |

下面是对每个案例的详细解读：

- 案例 **1 (**正确且高效**)**：这是最理想的情况。不变式t <- a + b所在的基本块是循环体内唯一的块，因此它支配所有出口。t在循环内只被定义了一次。并且，在循环开始前，t的初始值0没有在循环外被使用，因此它在前置首结点不活跃。所有条件满足，可以安全上提。
- 案例 **2 (**违反支配条件**)**：这是一个典型的while循环结构。如果循环条件i < N一开始就为假，程序将直接跳转到L2，此时x应该使用t的初始值0。但优化后的代码将t <- a + b无条件地移到了循环之前，导致t的值被改变。即使循环一次都不执行，x也会错误地得到a + b的值。
- 案例 **3 (**违反唯一性条件**)**：循环中存在两个对t的定义：t <- a + b和t <- 0。虽然a + b是循环不变的，但这次上提会完全改变程序的行为。原程序中M[i]和M[j]存储的是不同的值，而上提后它们可能存储相同的值，这显然是错误的。
- 案例 **4 (**违反前置首结点活跃性条件**)**：这是最微妙的一个例子。t的初始值0在循环的第一次迭代中被M[i] <- t这条语句使用了。因此，t在循环入口是活跃的。如果我们将t <- a + b上提，这个赋值会发生在t的旧值被使用之前，从而导致第一次迭代时M[i]被赋予了错误的值a + b而不是0。

## 总结

本教程深入探讨了循环优化的两个基本但至关重要的方面：循环的识别与循环不变式上提。通过学习，我们可以得出以下核心结论：

1. 优化的基础是精确分析：编译器为了安全、有效地进行优化，不能依赖于表面的源代码结构。它必须将程序转化为控制流图，并利用支配这一核心概念来形式化地定义回边和自然循环。这一系列从代码到图再到树的抽象过程，是进行高级程序分析和转换的基石。
2. 优化必须保证语义等价：循环不变式上提是一个强大的性能提升技术，其本质是降低高频代码的执行次数。然而，它的正确性依赖于对控制流和数据流的深刻理解。我们学习到的三大安全纪律——支配条件、唯一性条件和前置首结点活跃性条件——共同构建了一个严密的逻辑框架，确保优化在任何情况下都不会改变程序的原始语义。

编译优化是一门在追求极致性能和保证绝对正确性之间寻求精妙平衡的工程学科。本章所学的理论和算法，正是这门学科的基石。它们展示了编译器如何通过严谨的数学工具来理解和改造程序，最终为我们生成更小、更快、更高效的机器代码。

引用的著作


1. Optimizing Compilers - cs.Princeton, 访问时间为 六月 14, 2025, https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/loops.pdf
2. Control-flow analysis - CS [45]12[01] Spring 2022 - Cornell University, 访问时间为 六月 14, 2025, https://www.cs.cornell.edu/courses/cs4120/2022sp/notes.html?id=cflow
3. ch18 循环优化.pdf
4. Loop Optimization in Compiler Design - GeeksforGeeks, 访问时间为 六月 14, 2025, https://www.geeksforgeeks.org/loop-optimization-in-compiler-design/
5. Discuss the Loop Optimization in the compiler design - MindStick, 访问时间为 六月 14, 2025, https://www.mindstick.com/blog/303728/discuss-the-loop-optimization-in-the-compiler-design
6. I. What is a Loop?, 访问时间为 六月 14, 2025, https://www.cs.cmu.edu/afs/cs/academic/class/15745-f03/public/lectures/L7_handouts.pdf
7. Dominators and Natural Loops, 访问时间为 六月 14, 2025, https://www2.cs.arizona.edu/~collberg/Teaching/553/2011/Handouts/Handout-31.pdf
8. Lesson 5: Global Analysis & SSA - CS@Cornell, 访问时间为 六月 14, 2025, https://www.cs.cornell.edu/courses/cs6120/2020fa/lesson/5/
9. 8.6.4 Natural Loops, 访问时间为 六月 14, 2025, https://web.cs.wpi.edu/~kal/PLT/PLT8.6.4.html
10. Lecture Notes on Loop-Invariant Code Motion - André Platzer, 访问时间为 六月 14, 2025, https://symbolaris.com/course/Compilers12/17-loopinv.pdf
11. Code Optimization in Compiler Design - Tutorialspoint, 访问时间为 六月 14, 2025, https://www.tutorialspoint.com/compiler_design/compiler_design_code_optimization.htm