第5章

程 序 分 析

摘要：程序分析是计算机科学中通过系统化方法研究程序行为、结构、性能或安全性的技术，旨在理解、验证或优化软件。它通过自动化或半自动化的工具，帮助开发者发现代码缺陷、提升效率、保障安全，并辅助复杂系统的维护。程序分析是编译器设计、软件工程和安全领域的重要基础。本章拟重点介绍常见的各种程序分析方法，包括控制流分析、数据流分析、抽象解释、符号执行、污点分析、关联关系分析、依赖关系分析、因果关系分析、修改影响分析、修改传播分析等，旨在向读者展示丰富多彩的程序分析方法。深入的学习需要结合推荐阅读完成。

5.1 概　　述

程序分析(Program Analysis)是软件缺陷检测的主流方法之一，它通过自动化或半自动化的工具，帮助开发者发现代码缺陷、提升效率、保障安全，并辅助复杂系统的维护。例如，通过**控制流分析**可以发现程序代码、算法中隐含的死循环、死锁、条件冲突等方面的缺陷；通过**数据流分析**可以发现变量定义和使用方面的缺陷；通过**修改影响分析**可以确定缺陷的影响范围；通过**依赖关系分析**可以进行缺陷根源分析和定位；通过**因果关系分析**可以发现代码中存在的因果关系缺陷、逻辑关系缺陷等。

实际上，程序分析主要有以下五大方面的作用：①发现缺陷，识别潜在的逻辑错误、内存泄漏、安全漏洞等；②优化性能，分析代码执行效率，定位瓶颈(如冗余计算、低效算法)；③验证正确性，确保程序行为符合预期(如并发程序的线程安全性)；④辅助重构，分析代码依赖关系，支持模块化或架构调整；⑤安全审计，检测恶意代码或合规性问题(如未加密的敏感数据传输)。

根据是否需要运行被分析的程序，程序分析被划分成静态分析(Static Analysis)和动态分析(Dynamic Analysis)两种技术。实际应用中，静态分析适合在早期开发阶段快速发现代码问题，动态分析适合在测试阶段验证程序行为和性能，而现代程序分析工具(如 Coverity、KLEE 等)通常会混合使用静态分析和动态分析两种技术，即混合分析(Hybrid Analysis)技术，通过符号执行(Symbolic Execution)等提升程序分析效果。下面分别介绍一下静态分析、动态分析和混合分析。

(1) 静态分析：在不实际运行程序的情况下，通过分析代码、结构或文档来检

查程序的行为和属性。简单来说就是只分析程序代码本身。除了能够检查并排除程序中的安全漏洞和错误之外，静态分析的思想还可以被用于代码编译器来优化代码。为了实现代码优化，软件工程师需要借助多种方法来进行静态分析，其中最重要的方法之一就是流分析技术。流分析技术是一种比较传统的编译器优化技术，使用流分析技术可以在确定一个指定程序的相对路径的同时，保证程序内容的真实性。流分析技术大体上分为数据流分析和控制流分析，从逻辑关系上而言，控制流分析要先于数据流分析，起着先导性的作用。

（2）动态分析：在程序运行过程中，通过监控其行为和输出来分析程序的实际表现。动态分析方法可以有效地挖掘由软件实现缺陷导致的软件漏洞、检测由软件隐藏功能导致的软件后门，以及检测由恶意代码攻击引起的软件运行状态异常等等。动态分析的常用方法主要包括动态执行监控、符号执行、Fuzz 测试等。

（3）混合分析：一种结合静态分析和动态分析的技术，旨在提高程序分析的精度和效率。这种方法首先通过静态分析获取初步的程序不变性信息，然后对静态分析结果中不确定的部分进行动态分析，通过观察程序运行时各个对象的状态变化进行验证和补充。

静态分析和动态分析是两种互补的程序分析技术，各有各的优势，它们的比较如表 5.1 所示。

表 5.1　静态分析和动态分析比较

维　　度	静态分析（Static Analysis）	动态分析（Dynamic Analysis）
执行时机	不运行程序（编译前）	运行程序（测试/运行时）
覆盖率	理论全覆盖（所有代码路径）	依赖测试用例（实际执行路径）
检测能力	语法错误、潜在漏洞、代码风格问题	运行时的错误、性能问题、内存泄漏
误报率	较高	较低
工具举例	ESLint、SonarQube	V Valgrind、GNU Debugger、Fuzzing 工具

在软件定义中我们知道：**程序＝指令＋数据**，意味着正常的程序中只有指令（即用程序语言写的代码）和数据两种东西，其中指令控制程序的执行，形成控制程序执行的序列，又称控制流（Control Flow）；数据是程序的处理对象，就是流动在程序中的各种变量，它们的取值随着被定义、被使用而不停地在发生变化，形成数据流（Data Flow）。由于程序中只有控制流和数据流这两种流，控制流分析（CFA）和数据流分析（DFA）显然是最根本的两种程序分析技术，它们是任何其他程序分析技术的前提和基础。为此，我们先介绍控制流分析、数据流分析，然后再介绍其他的程序分析技术，包括抽象解释（Abstract Interpretation）、符号执行、污点分析（Taint Analysis）、关系分析（Relation Analysis）、依赖分析（Dependency Relationship Analysis）、因果分析（Cause-Effect Analysis）、修改影响分析、修改传播分析（Change Propagation Analysis，CPA）等。

5.2　控制流分析

5.2.1　什么是控制流分析

控制流分析（CFA）是程序分析中的一种技术，主要用于研究程序代码的执行顺序和逻

辑路径。它通过分析程序中的控制结构(如条件分支、循环、函数调用等),推断程序在运行时的可能行为,从而支持优化、错误检测或安全验证等任务。

在控制流分析过程中,有两个重要的核心概念:控制流结构和控制流图。其中,控制流结构包括:顺序执行、条件分支(如 if-else、switch)、循环(如 for、while)和函数调用(可能涉及跨函数的控制流跳转、过程间分析)。常见的控制流结构图如图 5.1 所示。

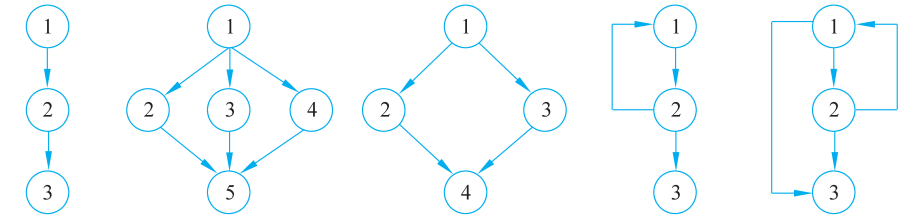

图 5.1 常见的控制流结构图

控制流图(Control Flow Graph,CFG)也叫控制流程图,由弗朗西斯·艾伦(Frances E. Allen)于 1970 年提出的,是程序分析中程序的一种图形化表示方法,用于描述程序在执行过程中可能的控制流路径。CFG 通过节点(基本块)和边(控制转移)展示代码的结构,广泛应用于编译器优化、软件测试、漏洞检测等领域。CFG 的基本块(Basic Block)是一个满足以下条件的连续执行的代码序列:①单一入口,只能从块的第一条指令进入;②单一出口,只能从块的最后一条指令离开(如跳转、分支或函数返回);③内部无分支,块内没有跳转指令或分支目标。CFG 的边(Edge)表示基本块之间的控制转移关系,例如:①顺序执行,块 A 执行后无条件进入块 B;②条件分支,块 A 末尾是 if 语句,根据条件跳转到块 B 或块 C;③循环结构,块 A 末尾跳转到自身或之前的块,形成循环。

CFG 的主要作用包括:①程序分析,检测死代码、不可达路径、循环复杂度等;②编译器优化,如常量传播、死代码消除、循环展开;③测试覆盖,生成测试用例以覆盖所有分支或路径;等等。

5.2.2 控制流图构造

正常情况下,控制流图的构造包含以下几个步骤。

步骤 1:划分基本块(下面是一个示例代码片段)。

```
1   //基本块 1
2   a = 1;
3   b = 2;
4   if (a > b) goto L1;
5
6   //基本块 2
7   c = a + b;
8   goto L2;
9
10  //基本块 3(标签 L1)
11  L1:
```

```
12  c = a - b;
13
14  //基本块 4(标签 L2)
15  L2:
16  return c;
```

(1) 确定入口指令：①函数的起始指令；②跳转指令的目标标签(如 L1：)；③紧跟在跳转指令后的指令(如 if (x) goto L1 的下一条指令)。

(2) 划分基本块：从入口指令开始，按顺序添加指令，直到遇到跳转指令或分支目标指令。

步骤 2：连接基本块(构建边)。①顺序执行：若块 A 的最后一条指令不是跳转语句，则块 A 的下一个块是块 B(按代码顺序)。②条件分支：若块 A 末尾是 if (cond) goto X else goto Y，则添加两条边：A → X 和 A → Y。③无条件跳转：若块 A 末尾是 goto X，则添加一条边 A → X。④函数调用与返回：通常将函数调用视为顺序执行，但需注意调用后的返回点(可能需特殊处理)。

步骤 3：处理复杂结构。①循环：块末尾跳转到之前的块，形成环(如 while 循环)。②switch-case：每个 case 对应一个分支边。③异常处理：需用额外边表示异常跳转路径。

从程序的特点来看，控制流分析可分为两大类：**过程内控制流分析**和**过程间控制流分析**。过程内控制流分析是对函数内部的程序运行流程的分析，过程间控制流分析是对函数之间调用关系的分析，其中更为重要的是过程内的控制流分析。无论是过程内控制流分析，还是过程间控制流分析，控制流图都是一种有效的手段。

5.2.3　过程内控制流分析

1. 什么是过程内控制流分析

过程内控制流分析(Intra-Procedural Control Flow Analysis)简称控制流分析(CFA)，是针对单个函数(或过程)内部的控制流路径进行静态分析的技术。它通过构建函数或过程的 CFG，识别代码的执行路径、分支、循环等结构，为编译器优化、代码质量检测和安全分析提供基础。

CFA 的主要作用包括：①理解代码执行逻辑，帮助明确函数内代码的执行顺序(如条件分支、循环、跳转)，识别所有可能的执行路径(例如 if-else 的不同分支)。②支持程序分析与优化，包括发现不可达代码(Dead Code)或冗余逻辑、辅助编译器优化(如循环展开、常量传播)，以及发现潜在问题(如无限循环、空指针解引用)等。

2. 过程内控制流分析流程

CFA 主要包含三个主要步骤。

步骤 1：构建控制流图(CFG)。①将函数拆分为基本块，每个块是连续执行的指令序列；②用边连接基本块，表示控制转移(如顺序执行、条件分支、循环)。图 5.2 给出了一段程序的控制流图示例。

步骤 2：分析控制流路径。①分支路径：识别所有条件分支的可能走向(例如 if 的真/假分支)。②循环结构：检测 for、while 等循环的入口和出口。③异常处理：跟踪 try-catch 块中的异常传播路径(如 Java 的异常处理)。

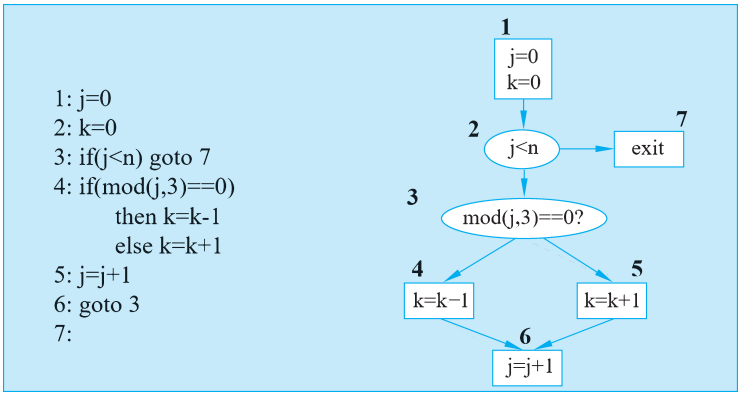

图 5.2 控制流图示例

步骤 3：标记特殊节点。①入口节点：函数的起始基本块。②出口节点：函数的返回或终止块(如 return、throw)。③交汇点：多个分支汇聚的位置(如 if-else 后的共同代码)。

3. 典型应用场景

CFA 应用范围很广，其中典型的应用场景包括编译器优化、静态代码检测和测试覆盖率分析等。

(1) 编译器优化：可以进行如下编译器优化任务。①死代码消除：删除永远无法执行到的代码。②循环优化：判断循环是否可展开或并行化。③常量折叠：分析条件分支是否总为固定值(如 if (true) {…})。

(2) 静态代码检测：可以进行如下检测任务。①检测不可达路径：可以发现无法进入的分支(如 if (false) {…})。②检测资源泄漏：可以检查函数中是否存在未释放的资源(如未关闭的文件句柄)。③检测安全漏洞：可以识别可能被攻击者利用的控制流(如未校验的输入导致代码注入)。

(3) 测试覆盖率分析：可以生成测试用例覆盖所有分支(分支覆盖)或路径(路径覆盖)，也可以统计已执行和未执行的代码块(如使用 JaCoCo 工具)。

CFA 是理解单个函数逻辑的核心技术，通过构建 CFG 和路径分析，为优化代码、提高安全性和生成测试用例提供关键支持，尽管面临间接跳转和路径爆炸等挑战，结合静态分析与动态测试，仍能显著提升代码质量与可靠性。

5.2.4 过程间控制流分析

1. 什么是过程间控制流分析

过程间控制流分析(Interprocedural Control Flow Analysis，ICFA)是在多个函数或方法之间追踪控制流路径的分析技术。它通过分析函数调用关系、参数传递和返回值，构建跨函数(跨过程)的控制流模型，以理解程序整体的执行逻辑。与 CFA(仅关注单个函数)不同，ICFA 需要处理函数调用、递归、多态等复杂场景，是全局程序分析的核心技术。

ICFA 的两个核心概念包括：①跨函数控制流，追踪函数调用链(如 A()→B()→C())，分析调用前后的代码执行路径；②参数与返回值影响，判断函数调用的输入参数如何影响被调用函数的执行逻辑。

ICFA 的主要作用包括：①识别跨函数的安全漏洞(如数据竞争、内存泄漏)；②优化全

局代码(如函数内联、跨过程常量传播)；③支持程序理解(如逆向工程中还原调用关系)。

与 CFA 不同的是,ICFA 难度更大、复杂度更高,面临的挑战更多,典型的挑战如下。

(1) 函数调用复杂性：①直接调用,如 A()显式调用 B(),静态可解析；②间接调用,如函数指针、虚方法(C++/Java)、反射(Java/Python),需动态或上下文敏感分析(Context-Sensitive Analysis)；③递归调用：需处理循环调用链(如 A() → A())的终止条件。

(2) 路径爆炸：多个函数组合调用可能导致路径数量指数级增长。例如,函数 A()调用 B() 和 C(),每个函数内部又有分支。

(3) 上下文敏感性：同一函数在不同调用位置的上下文(如参数、全局变量)可能不同,需区分不同调用场景。例如,B(x = 1)和 B(x = 2)的执行路径可能完全不同。

2. 过程间控制流分析流程

步骤 1：构建调用图(Call Graph)。调用图的节点为函数,边表示调用关系(如 A → B)。构建方法包括：①静态分析,基于语法解析(如识别 call 语句),但对间接调用可能不精确；②动态分析,通过运行时插桩(如 gprof)记录实际调用链,但依赖测试用例覆盖；③混合分析,结合静态推断和动态反馈(如机器学习预测高频调用路径)。

步骤 2：构建过程间控制流图(Interprocedural CFG，ICFG)。ICFG 的结构是将多个函数的 CFG 通过调用边(Call Edges)和返回边(Return Edges)连接起来。其中,调用边指从调用点(如 A()中的 call B())到被调用函数入口(B()的入口块),返回边指从被调用函数出口(B() 的 return)返回到调用点的下一指令(A()中 call B() 之后)。图 5.3 表示一种过程间控制流图,其中,节点表示基本模块(或程序语句),实线仍然表示控制流边,圆点虚线表示调用边,短虚线表示返回边。

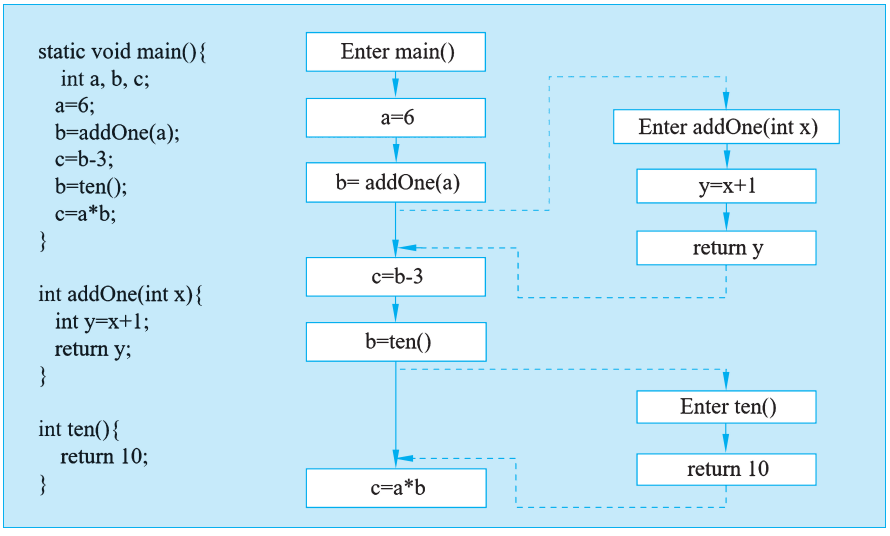

图 5.3　过程间控制流图示例

步骤 3：上下文敏感分析。目的是区分同一函数在不同调用上下文中的行为。实现方式包括：①调用栈记录,为每个函数调用分配唯一标识(如调用链 A()→B()→C()的上下文)；②摘要(Summary),为函数生成输入/输出影响的抽象表示(如参数 $x>0$ 时返回值 y 的范围)。

3. 典型应用场景

ICFA 应用范围很广，其中典型的应用场景包括全局死代码消除、数据流分析、安全漏洞检测和性能优化等。

（1）全局死代码消除：发现未被任何函数调用的代码（如未使用的私有方法）。

（2）数据流分析：跨函数追踪变量传播（如全局变量在多个函数间的修改）。

（3）安全漏洞检测：①内存泄漏，检查资源分配（malloc()）和释放（free()）是否跨函数匹配；②SQL 注入，追踪用户输入从 readInput()到 executeQuery() 的路径。

（4）性能优化：①内联优化，将高频调用的短函数内联到调用点，减少调用开销；②跨过程常量传播，若某函数参数始终为常量，直接替换为常量值。

ICFA 通过追踪跨函数调用关系，扩展了程序分析的覆盖范围，是程序化、安全保障和逆向工程的关键技术。尽管 ICFA 面临间接调用和路径爆炸等挑战，结合上下文敏感分析和现代工具（如 LLVM、CodeQL），仍能有效解决复杂系统中的全局性问题。

5.2.5 过程内控制流分析和过程间控制流分析比较

CFA 和 ICFA 是程序分析中的两种不同方法，它们在范围、复杂性、应用场景等方面存在显著差异。以下是对两者的详细比较。

（1）从分析范围的角度对比：①CFA 仅关注单个函数（过程）内部的执行路径，目标是构建函数内的控制流图（CFG），分析条件分支、循环、异常处理等结构，识别不可达代码或潜在错误。例如：分析函数中的 if-else、for 循环或 return 语句的路径可能性。②ICFA 的范围跨越多个函数，分析函数调用（如 A() → B() → C()）及其相互影响，目标是构建整个程序的控制流图（CFG），考虑函数调用的上下文（参数传递、返回值、副作用等）。例如：追踪变量在多个函数间的状态变化（如全局变量、参数传递导致的空指针问题）。

（2）从复杂度与挑战的角度对比：①CFA 的复杂度低，仅需处理单个函数的线性或分支结构，无须考虑外部调用。面临的挑战是局部优化（如循环展开、死代码删除）容易实现，但无法处理跨函数依赖。②ICFA 的复杂度高，需处理递归、动态分发（如虚函数、函数指针）、多线程等场景。面临的挑战包括：调用图构建，即确定所有可能的调用目标（如动态语言中的函数指针）；上下文敏感性，即区分不同调用位置的影响（如递归的不同层次）；性能开销，即全局分析可能导致指数级时间或空间复杂度。

（3）从应用场景角度对比：①CFA 主要用于编译器优化，含局部死代码消除、寄存器分配、基本块重排序；静态代码检查，即检测函数内的逻辑错误（如除零、未初始化变量）；提升代码覆盖率，生成函数内执行路径的测试用例。②ICFA 主要用于全局优化（内联展开、跨函数常量传播、逃逸分析）、安全漏洞检测（追踪跨函数的数据流，如 SQL 注入、缓冲区溢出）、程序理解（可视化函数调用关系，分析系统级行为）。

（4）从性能与精度权衡角度对比：①CFA 的优点是快速、低开销，适合实时或大规模代码的初步检查；缺点是可能遗漏跨函数问题（如参数传递导致的空指针）。②ICFA 的优点是全面性高，能发现复杂交互导致的问题；缺点是资源消耗大，需在精度与效率间权衡（如上下文敏感或不敏感）。

CFA 和 ICFA 的对比如表 5.2 所示。

表 5.2　CFA 和 ICFA 的对比

维　　度	过程内控制流分析(CFA)	过程间控制流分析(ICFA)
分析范围	单个函数	跨函数调用
复杂度	低	高(需处理递归、动态绑定等)
适用场景	局部优化、简单错误检测	全局优化、安全漏洞分析
性能	高效	取决于资源
精度	受限于函数边界	更全面,但依赖上下文处理策略

实际应用中,二者常结合使用:先通过 ICFA 确定关键调用链路,再针对特定函数进行深度 CFA。

5.3　数据流分析

5.3.1　什么是数据流分析

数据流分析(DFA)是程序分析中的一种技术,旨在追踪程序中数据的定义、使用和传播过程,分析变量或表达式在程序执行时的可能状态(如值、生命周期、依赖关系等)。它通过静态或动态方法,推导数据在程序中的流动路径,从而支持代码优化、错误检测和安全验证。

数据流(Data Flow)是一组有序的有起点和终点的由变量的定义及使用产生的序列,包括输入流和输出流两种。其中,①变量定义(DEF):将数据存储起来,存储单元的内容改变。②变量使用(USE):将数据取出来,存储单元的内容不变。

数据流图(Data Flow Graph, DFG):程序的数据流图也称为 DEF-USE 图,它勾画了程序中变量在不同基本块间的定义和使用流。

(1) 用 DEF[i]表示在基本块 i 中定义的变量集合。程序中的变量声明、赋值语句、输入语句和传址调用都可以用来定义变量。

(2) 用 USE[i]表示在基本块 i 中有使用的变量集合。其中,C-USE[i] 表示在基本块 i 中计算使用(C-USE)的变量集合;P-USE[i]表示在基本块 i 中谓词使用(P-USE)的变量集合。变量的 C-USE 表示该变量被用在赋值语句表达式、下标表达式、输出语句中,或者被当做参数传递给调用函数;变量的 P-USE 表示该变量被用在条件表达式中(如 if 和 while 语句)。

5.3.2　数据流图构造

正常情况下,数据流图的构造包含以下几个步骤。

步骤 1:计算 DFG 中每个基本块 i 的 DEF[i]、C-USE[i]和 P-USE[i]。

步骤 2:将节点集 N 中的每个节点 i(每个节点对应 DFG 中的一个基本块)与 DEF[i]、C-USE[i]和 P-USE[i]关联起来。

步骤 3:针对每个具有非空 P-USE 集并且在条件 C 处结束的节点 i,如果条件 C 为真时执行的是边(i, j),C 为假时执行的是边(i, k),分别将边(i, j)和(i, k)与 C、! C 关联起来。

DEF-USE 对：勾画了变量的一次特定的定义和使用。我们只关心两种类型的 DEF-USE 对：一种是由定义及其 C-USE 构成的 DEF-USE 对，另一种是由定义及其 P-USE 构成的 DEF-USE 对，分别用集合 DCU 和 DPU 来描述这两类 DEF-USE 对。

DEF-clear 路径：假设变量 x 在节点 i 中定义（记作 $D_i(x)$），在节点 j 中使用，考虑路径 p =（i，n1，n2，…，nk，j），k≥0，路径 p 从节点 i 开始，结束于节点 j，并且节点 i、j 在子路径 n1，n2，…，nk 中未出现，如果变量 x 没有在子路径 n1，n2，…，nk 中被重定义，称 p 是变量 x 的 DEF-clear 路径。在这种情况下，也称 x 在节点 i 处的定义，即 $D_i(x)$ 在节点 j 处是活跃的。

常见的变量定义和使用缺陷包括：①变量被定义了，但从来没有被使用；②使用的变量没有被定义；③变量在使用之前被多次定义。

通过数据流分析和数据流测试可以发现这些数据流缺陷。

数据流测试：根据代码中变量的使用情况进行的测试，主要关注软件中的数据定义和使用。数据流测试的详细内容参见本书第 7 章。

下面介绍常见的数据流分析，包括可到达定义分析、变量活性分析、可用表达式分析、不可达路径分析和过程间数据流分析等。

5.3.3 过程内数据流分析

过程内数据流分析（Intra-procedural Data Flow Analysis）简称**数据流分析（DFA）**，是一种静态程序分析技术，专注于在**单个函数或过程内部**追踪数据的定义、使用和传播，以推断程序执行时的数据状态变化。它不涉及跨函数调用的分析（如参数传递、返回值或全局变量影响），仅关注当前函数或过程内的控制流和数据流关系。

DFA 的主要作用包括：①变量状态推断：分析变量在函数内的定义（赋值）和使用情况，例如：确定变量在某一节点是否已被初始化，或者检测变量是否在未被定义前就被使用（如未初始化错误）。②优化与验证：支持编译器优化（如删除冗余赋值、常量传播），发现潜在缺陷（如内存泄漏、不可达代码）。③路径敏感分析：根据控制流分支（如 if/else）推断不同路径下的数据状态。

DFA 面临的挑战与局限性有：①循环处理：循环结构可能导致数据流方程需要多次迭代才能收敛。例如：循环中变量的重复赋值需通过迭代确定最终状态。②路径敏感性不足：若分析不考虑具体分支条件（如 if(x>0)），结果可能过于保守。可以结合符号执行或路径敏感分析进行改进。③指针与别名问题：若函数内存在指针操作，需额外分析指针指向关系。例如：*p = 10 可能修改多个变量的值。

一些典型的 DFA 应用场景包括：①编译器优化：删除死代码（如未被使用的变量赋值），常量折叠（如将 2 + 3 替换为 5）。②静态代码检查：检测未初始化变量、冗余代码，发现潜在错误（如除零风险）。③安全分析：追踪敏感数据（如密码）在函数内的传播路径，防止泄露。

DFA 的主要类型包括：**可到达定义分析、变量活性分析、可用表达式分析、不可达路径分析**以及**常量传播**。其中，常量传播（Constant Propagation）推断变量是否为常量值，并替换变量为常量以优化代码。例如：下面的 C 代码中存在常量传播现象。

```
1   const int a = 100;
2   int b = a * 2;                                  //可优化为 int b = 200;
```

1. 可到达定义分析

可到达定义(Reaching Definition)：简单地讲，分析某个变量的定义(如赋值语句)能到达哪些使用点。准确地讲，在节点 p 的一个定义 d，到节点 q 是可达的，当且仅当这个定义在从 p 到 q 的所有路径上不会被重新定义(即被 Kill 了)。例如，在图 5.4 中，节点 1 处定义了 j = 0 和 k = 0，对节点 6 来说，j 的这个定义是可到达的，k 的这个定义就不是可到达的，因为 k 在节点 4 或者节点 5 被重新定义了，在节点 1 的定义被 Kill 了。图 5.4 中节点编号和节点编号是不一致的。

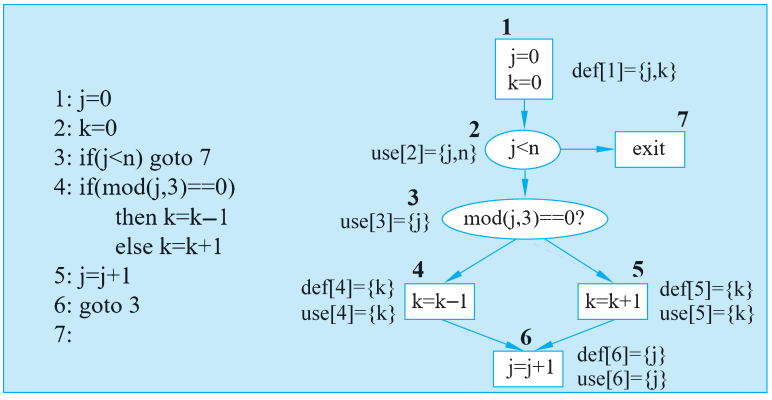

图 5.4　可到达定义示例

2. 变量活性分析

变量活性分析用于判断变量在某个节点是否可能被后续代码使用。具体来讲，如果节点 p 处的变量 v 可以在控制流图中以 p 为起始点的某条路径中被使用，则称其在节点 p 上是活跃的(Live)，否则是死的(Dead)。也可以说，程序在某个节点时，如果存在一段程序可执行，稍后读取一个变量而不在其间写入该变量，则该变量在这个节点是活跃的。例如，在图 5.4 中，变量 j 在节点 1 被定义，在节点 3 被使用，而其间没有重新定义，因此称变量 j 在节点 4(及节点 3 处)是活跃的。而在节点 6 被重新定义了，之前在节点 1 定义的变量 j 在节点 6 之后就是死的。

3. 可用表达式分析

可用表达式(Available Expressions)是编译原理和数据流分析中的一个重要概念，主要用于优化编译器中的公共子表达式消除(Common Subexpression Elimination，CSE)。它的核心作用是识别程序中哪些表达式(例如 a + b 或 x * y)在某个节点(如基本块入口/出口)是**可用的**(即之前已被计算且未被修改)，从而避免重复计算。

也就是说，在程序执行到某个位置时，如果一个表达式的结果在之前的所有路径上已经被计算过，并且其操作数在之后未被修改，则该表达式在此位置是**可用的**。可用表达式具有两个重要性质：①可用性。表达式的结果可以直接复用，无须重新计算；②安全性：复用不会导致错误(例如操作数未被修改)。

下面是一个可用表达式的代码示例，对语句 6 来说，a + b 就是可用表达式。