第3章

软件缺陷分析

软件开发过程复杂，通常经历需求分析、软件设计、代码实现和测试等过程，并最终在生产环境中部署运行。任何一个环节的错误都有可能导致软件运行不正确。本章关注代码实现层面的缺陷，首先介绍基于静态分析的缺陷分析技术，然后介绍基于深度学习和大模型的缺陷分析技术，最后介绍缺陷案例挖掘与分析技术。

3.1 基于静态分析的缺陷分析

静态分析是程序分析的主要方法之一，可以在不执行程序的情况下收集程序行为信息，然后基于程序行为分析程序的性质。因此，静态分析可用于判定程序是否满足某种性质，特别是检查代码是否存在特定的缺陷。

具有一定规模的软件一般都会进行模块划分，因为模块之间存在复杂的交互关系。在本节的讨论中，我们将分析对象限定为一个函数，并假定这个函数有准确的输入输出规范作为判定代码是否正确的依据。

3.1.1 程序的状态和语义

程序静态分析方法可以分为基于语法的分析方法和基于语义的分析方法。使用静态分析来检测程序缺陷时，基于语法的分析方法漏报和误报都较多，因此，本节我们只考虑基于语义的分析方法。在 2.1 节中，我们简要介绍了常用的程序语义表示方式和典型的静态分析方法。本节为了准确表示和分析程序的语义，我们先介绍程序状态和相关概念。

如图 3-1 所示的简单 C 语言程序，实现了用 abs 函数计算变量 x 的绝对值。该函数的语句并非按顺序逐句执行。编程语言通常支持分支和循环等控制结构，使得程序执行过程中的下一条语句存在多种选择。例如，对于图 3-1 中的程序，在执行完语句 2 之后，下一条执行语句由 x 的值决定，即如果 x<0，那么程序将执行语句 3；否则，将执行语句 4。若 x= –1，该函数首先执行语句 2。由于 if 的判断条件为 x<0，而且执行语句 2 前 x=–1，因此计算出来

的判断条件为真，执行语句 3。此外，判断条件并不修改变量的值，所以执行语句 2 时 x=–1。执行完语句 3 之后，x=1。

```
1 unsigned int abs(int x) {
2   if (x<0)
3     x = -x;
4   return x;
5 }
```

图 3-1　用 abs 函数计算变量 x 的绝对值程序示例

根据对图 3-1 程序示例的分析可以得出，程序的输入值一旦确定，程序执行的语句及其执行的顺序也是确定的。程序中变量的值决定了执行语句的结果以及下一条待执行的语句。程序变量的值存储在内存里，我们将它们表示为一个集合，称作内存状态。当前执行的那条语句，用语句之前的标号表示，称为程序点，类似于处理器中的特殊寄存器 PC（Program Counter）。因此，有序对（标号, 内存状态）能够唯一刻画程序执行过程中的当前状态，即程序状态。

图 3-2 所示的 swap 函数可实现交换变量 x 和变量 y 的值的功能。

```
1 void swap(int &x, int &y) {
2   x = x + y;
3   y = x - y;
4   x = x - y;
5 }
```

图 3-2　交换变量 x 和变量 y 的值的函数

我们用标号 l_i 表示程序执行的第 i 条语句，p_i 表示执行语句前的内存状态，q_i 表示执行语句后的内存状态，那么一条语句的执行就可以用$\{p_i\}l_i\{q_i\}$来表示。很多时候，内存状态 p_i 称为前置条件，q_i 称为后置条件。对于顺序执行的语句，如图 3-2 中的语句 2～语句 4，执行语句 2 后，接下来执行语句 3；语句 2 的后置条件成为语句 3 的前置条件。如果执行的是条件分支语句，将由条件表达式来选择下一条执行语句。由于执行的两条语句在代码中并不连续，我们用程序状态转移$(l_i, p_i)\rightarrow(l_j, q_j)$来描述语句间的跳转，表示第 l_i 条语句在前置条件 p_i 下执行，内存状态变成 q_j 并跳转到语句 l_j。

给定初始状态，程序的一次执行可以表示为一个程序状态序列$\{s_i=(l_i, p_i)\}$*。特别地，程序状态序列的第一个状态 s_0 称为初始状态，而程序状态序列的最后一个状态 s_n 称为终止状态。如果我们只关心执行结果，也就是内存状态的变化，那么程序状态序列可以简化为内存状态序列$\{p_i\}$*。通常，给定初始状态，若程序状态序列有限，则程序会终止。有时候，程序会进入死循环，其状态序列是无限循环的，则程序不终止。从程序分析的角度看，程序不终止是一种不好的性质。在实际应用中，这种不终止的程序有很多，如操作系统和服务器进程。

特别地，我们把程序的一个状态序列，视为程序的一个特定行为（Behavior）。我们收集给定程序的所有状态序列，它们构成的集合就刻画了程序的行为。当然，这个集合可能是有限的，也可能是无限的。我们将在 3.1.2 小节中介绍如何分析这些状态序列。

3.1.2　抽象解释理论

很多缺陷会在程序的语法层面上呈现出一定的模式。基于此，一些检测方法利用这些模式来扫描目标程序，从而发现目标程序中的缺陷。由于只利用了程序的语法特性，这类检测方法的速度很快，但缺点是误报和漏报较多。

为了提高缺陷检测的准确率，一种基于程序数据流的分析方法被提出。这种方法不需要执行程序，其利用了程序语义，发现程序在执行时的一些特性。编译技术使用这种方法来发现代码翻译中的一些性质，如公共子表达式、定义–使用链和活跃变量等，从而生成优化的可执行程序代码。

这种分析方法有一个统一的框架，但针对特定性质的分析方法还需要根据特定性质来具体构造。1977 年，Cousot[1]提出了抽象解释（Abstract Interpretation），尝试建立一种程序分析的理论基础。经过研究人员近半个世纪的努力，抽象解释理论得到了扩充和完善[2, 3]，成为程序分析领域统一的理论基础[4]。Giacobazzi 和 Ranzato[5]介绍了抽象解释理论的发展历史，陈立前等人[6]介绍了抽象解释理论及其应用的最新进展。

抽象解释理论的目标是系统地计算出任意位置的程序状态。由于程序的表现形式都被程序设计语言的语法约束，因此我们定义一个简单的命令式语言 ToyC 来介绍抽象解释理论。

图 3-3 给出了 ToyC 的语法。它定义了整数类型的变量以及整数的一些二元运算；它还提供算术表达式之间的比较功能，比较结果是布尔值。

ToyC 具有 C 语言的一些典型特性，其支持指针类型、取变量地址的&运算符、分配内存的 malloc 操作。ToyC 还支持一些简单的语句：

- 赋值语句：把算术表达式计算出来的值，赋给左值变量。
- 分支语句：根据布尔表达式 B 的值，来确定是走 true 分支，还是走 false 分支。
- 循环语句：如果布尔表达式 B 的值为 true，则继续执行循环内的语句。
- skip 语句：空语句。结合分支语句和 skip 语句，可以实现只有 if 分支或者 else 分支的语句。
- 单参函数：定义只有一个参数的函数。

这里，表达式的求值没有副作用，只有赋值语句和间接赋值语句可修改变量的值，改变内存状态。布尔表达式的求值结果决定程序的执行路径。

为了用抽象解释理论来分析 ToyC 程序，我们定义下列符号用来表示程序的语义：

- 变量集合 X，由程序中出现的变量构成；
- 值域 V，表示变量取值范围，在 ToyC 中是整数集 Z；
- 内存状态集合 M，是从 X 到 V 的映射，$m(x)$表示从内存中读取 x 的值，$m[x \mapsto c]$表

示将内存中 x 的值更新为 c；

- 程序标号集合 L，$l \in L$ 指向特定的语句，也称为程序点；
- 程序状态集合 $S = L \times M$，是程序标号和内存状态的笛卡儿积，(l, m)表示程序点 l 的内存状态。

n ∈ V	整数值
x ∈ X	程序变量
A_{op} := + | - | * | …	算术二元运算
B_{op} := < | ≤ | == | …	比较运算
E :=	表达式
| n	常量
| x	变量
| E A_{op} E	二元运算
| &x	取变量地址
| malloc	分配内存
| *E	内存地址解引用
| f	函数
B :=	布尔表达式
| E B_{op} E	比较运算
C :=	语句
| skip	跳过
| C;C	顺序语句
| x = E	赋值语句
| *E = E	间接赋值
| if(B){C}else{C}	条件分支
| while(B){C}	循环语句
| E(E)	函数调用
| return	函数返回
F := f(x) = C	函数定义
P := F^+ C	程序

图 3-3　ToyC 的语法

基于这些符号，我们可以形式化地表示程序的执行。为了从语义上表示程序，抽象解释理论必须可以描述程序的每一次实际执行。因此，我们先分析程序的实际执行，并以它们为参照来判别抽象解释是否可靠。

我们用$[\![E]\!]$来表示表达式 E 的语义计算。给定内存状态 $m \in M$，$[\![E]\!](m)$ 表示在内存状态为 m 时，计算表达式 E 得到的值。对于图 3-2 所示的程序，我们有 $X = \{x, y\}$、$V = Z$、$L = \{1, 2, 3, 4, 5\}$，当 $l = 2$ 时，对应的程序语句是“x = x + y”。给定初始状态 $m_0 = \{x \mapsto 0, y \mapsto 1\}$ 时，我们来观察程序在每条语句执行前和执行后的状态。对于程序状态$(2, m_0)$，我们有 $[\![x + y]\!](m_0) = 1$，赋值后有 $m[x \mapsto 1]$，得到内存状态 $m_1 = \{x \mapsto 1, y \mapsto 1\}$；对于程序状态$(3, m_1)$，$[\![x - y]\!](m_1) = 0$，赋值后有 $m[y \mapsto 0]$，得到内存状态 $m_2 = \{x \mapsto 1, y \mapsto 0\}$；类似地，执行语句 4，

得到终止状态 $m_3=\{x\mapsto 1, y\mapsto 0\}$。

在特定程序点上，每个变量都存储具体的值，也称为具体元素（Concrete Element）。程序执行中出现的所有具体元素构成的集合，称为具体域（Concrete Domain）。内存状态可表示为程序变量的笛卡儿积，属于多个具体域的笛卡儿积。相应地，内存状态存储具体元素时也叫具体状态（Concrete State）。

给定内存状态集合 M，我们定义了 ToyC 基本语法单元的语义计算，但不包括地址运算和函数调用，因为这不影响我们对抽象解释理论的理解。令 $\mathcal{P}(M)$ 表示 M 的幂集，我们结构化地执行如图 3-4 所示的语义计算规则。其中，$F_B(M)$ 是过滤算子（Filter），可以根据语句中的条件 B 来选择满足要求的那些内存状态。

$$
\begin{aligned}
&[\![E]\!]: M \to V\\
&\qquad [\![n]\!](m) = n\\
&\qquad [\![x]\!](m) = m(x)\\
&\qquad [\![E_1 A_{\text{op}} E_2]\!](m) = A_{\text{op}}([\![E_1]\!](m), [\![E_2]\!](m))\\
&[\![B]\!]: M \to \{\text{true}, \text{false}\}\\
&\qquad [\![E_1 B_{\text{op}} E_2]\!](m) = B_{\text{op}}([\![E_1]\!](m), [\![E_2]\!](m))\\
&\qquad F_B(M) = \{m \in M \mid [\![B]\!](m) = \text{true}\}\\
&[\![C]\!]_{\mathcal{P}}: \mathcal{P}(M) \to \mathcal{P}(M)\\
&\qquad [\![\text{skip}]\!]_{\mathcal{P}}(M) = M\\
&\qquad [\![C_1; C_2]\!]_{\mathcal{P}}(M) = [\![C_2]\!]_{\mathcal{P}}([\![C_1]\!]_{\mathcal{P}}(M))\\
&\qquad [\![x = E]\!]_{\mathcal{P}}(M) = \{m[x \mapsto [\![E]\!](m)] \mid m \in M\}\\
&\qquad [\![\text{if}(B)\{C_1\}\text{else}\{C_2\}]\!]_{\mathcal{P}}(M) = [\![C_1]\!]_{\mathcal{P}}(F_B(M)) \cup [\![C_2]\!]_{\mathcal{P}}(F_{\neg B}(M))\\
&\qquad [\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}(M) = F_{\neg B}(\cup_{i\geqslant 0}([\![C]\!]_{\mathcal{P}} \circ F_B)^i(M))
\end{aligned}
$$

图 3-4　ToyC 的语义计算规则示例

分支语句的执行分两种情况。第一种情况是执行 if 分支，我们有 $[\![C_1]\!]_{\mathcal{P}}(F_B(M))$；第二种情况是执行 else 分支，我们有 $[\![C_2]\!]_{\mathcal{P}}(F_{\neg B}(M))$。显然，分支语句执行结束后的内存状态应该是两个分支分别执行后的状态的并集，因此有 $[\![C_1]\!]_{\mathcal{P}}(F_B(M)) \cup [\![C_2]\!]_{\mathcal{P}}(F_{\neg B}(M))$。

循环语句的执行比较复杂，可以归约为分支语句的多次执行。如果循环只执行了 i 次，说明前 i 次都执行了 if 分支，可以表示为 $([\![C]\!]_{\mathcal{P}} \circ F_B)^i(M)$；接下来，则执行 else 分支，于是有 $M_i = F_{\neg B}(([\![C]\!]_{\mathcal{P}} \circ F_B)^i(M))$。实际上，循环的执行可以分解为只执行 $i = 0,1,2\cdots$ 的集合。在循环执行结束后，类似于分支语句，我们需要把每次循环得到的状态求并集，于是有 $[\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}(M) = F_{\neg B}(\cup_{i\geqslant 0}([\![C]\!]_{\mathcal{P}} \circ F_B)^i(M))$。

我们前面给出了图 3-2 所示程序的一个具体状态序列，长度有限。如果程序中有循环或

递归函数，那么其状态序列会变长，甚至变成无限长的序列。实际上，任何程序都是语法单元的有限序列，无限序列只是因为一些程序点反复出现二产生的。我们把一个程序点上出现的所有具体状态用集合表示，程序状态序列就变成了有限序列。

将一个程序用状态序列进行有限表示后，我们面临一个新问题，即程序点上的具体状态集合会很大，可能无限。图 3-2 中，变量 x 存储整数集 Z 中一个特定的值。由于 x 和 y 的值域都用整数表示，所以初始状态有 $2^{32} * 2^{32} = 2^{64}$ 个；如果 x 的定义域用实数表示，则初始状态有无穷多个。

我们需要借助抽象集合论表示无限集，通常用一些性质来描述元素，使得集合的表示有限。类似地，我们先定义程序状态的抽象，然后有限地抽象表示程序。

定义 3.1（抽象化） 程序的具体状态可用一些逻辑性质表示，记作抽象性质或者抽象元素，这些性质构成的集合 A 就是对程序状态的抽象化。抽象域是一些抽象元素构成的集合。

抽象解释用抽象元素来表示内存状态，称为抽象状态。给定一个抽象状态，为了评判其是否表示对应的具体状态，我们需要完成抽象化的逆过程，即具体化（Concretization）。

定义 3.2（具体化） 给定集合 A 中的一个抽象元素 a，我们可以找到满足性质 a 的具体状态集合，这个过程称为具体化，记作 $\gamma(a)$。

对于图 3-2 中的程序，x 和 y 的值都是 int 类型的整数，我们用符号 ⊤ 来表示所有的取值。如果一个程序点不可能被执行到，其对应的状态就是空集，我们用符号 ⊥ 来表示。

为了分析图 3-2 中的程序，我们用 x = N 表示 x < 0，x = P 表示 x > 0。此外，还有 x = 0。我们构造一个符号抽象域 sign={⊥, N, 0, P, ⊤}，其中 ⊤ 表示所有的取值，⊥ 表示空集。程序中的运算基于具体值，当变量的值属于抽象域之后，对应的运算需要重新定义。以表达式 x + y 为例：

- 当 x = P，y = P 时，x + y > 0，因此表达式 x + y = P；
- 当 x = N，y = N 时，x + y < 0，因此表达式 x + y = N；
- 当 x = N，y = P 或者 x = P，y = N 时，x + y 与 x、y 的具体取值有关，例如，在 x = –2 且 y = 1、x = 2 且 y = –1、x = –2 且 y = 2 的情况下，x + y 会得到 N、P 和 0 三种情况。这时我们用 ⊤ 表示表达式 x + y 的抽象域。

在图 3-1 中，x 可以取任意的整数，用 ⊤ 表示。执行 if 语句时，x 被分为了 N、P 和 0 三种情况。当 x = N 时，执行 x = –x 后，根据运算规则有 x = P。基于符号抽象域，我们可以证明 abs 函数确实实现了计算变量 x 的绝对值的功能。

在图 3-5 所示的程序中，add5 函数把参数 x 加上常量 5 并返回。假定 x 的取值范围为[–4, 5]。如果用符号抽象域来表示 x，我们有 x = ⊤。如果用具体值来计算，我们得到的返回结果可用区间[1,10]表示，在符号抽象域中是 P。这说明，如果用不精确的抽象域表示变量，那么计算结果也不会精确。

```
1 int add5(int x) {
2   x = x + 5;
3   return x;
4 }
```

图 3-5　用 add5 函数计算 x+5 的程序示例

我们设计一个新的抽象域 I = [l, h]，满足 $l \leqslant h$，即用区间来表示变量的取值范围。由于整数的离散性，闭区间就可以满足我们的表示需求。由于值的表示发生了变化，所以我们需要重新定义一些运算。给定区间抽象域表示的变量 $x = [l_x, h_x]$，$y = [l_y, h_y]$，我们给出典型运算的定义：

- 加法：$x + y = [l_x + l_y, h_x + h_y]$；
- 减法：$x - y = [l_x - h_y, h_x - l_y]$；
- 乘法：乘法比较复杂，需要考虑边界符号不一样的情况。令 $l = \min\{ l_x * l_y, l_x * h_y, h_x * l_y, h_x * h_y \}$，$h = \max\{ l_x * l_y, l_x * h_y, h_x * l_y, h_x * h_y \}$，那么 $x * y = [l, h]$；
- 小于：$x < y$，当且仅当 $h_x < l_y$。

对于图 3-5 中的程序，如果使用区间抽象域，那么我们得到 x = [–4, 5]和 5 = [5, 5]，根据加法规则，有 x = [1, 10]。将其变换到符号抽象域，有 x = P，而非之前的 x = ⊤。与符号域相比，区间域表示的信息更加精确。这个例子说明：基于精确的抽象域表示，计算结果更加精确。但区间抽象域比符号抽象域的计算更加复杂，精确性需要付出更多计算代价。

具体域表示程序执行时可取的值，而抽象域是对具体域的一种描述。对于赋值语句“x = x + 5”，如果 x = 1 或 10，那么在具体域上，有 x = {1, 10}；用区间抽象域表示，则有 x = [1, 10]。与符号抽象域类似，我们观察到，区间抽象域引入了具体域中没有出现的元素。为了保证分析的可靠性，即程序的行为要被完整地表示，抽象域必须能表示具体域中的所有值。

既然抽象域中的元素是对特定具体状态集合的一种近似，那么我们如何来确定哪种近似更好呢？令 S 是具体状态的集合，a 是抽象元素。我们给出最优抽象的定义。

定义 3.3（最优抽象） 给定具体状态的集合 S，抽象元素 a 是 S 的最优抽象，当且仅当满足下列条件：

- $S \subseteq \gamma(a)$；
- 如果 a' 也满足 $S \subseteq \gamma(a')$，那么 $\gamma(a) \subseteq \gamma(a')$，即 a' 是比 a 更粗糙的近似。

显然，如果 S 存在最优抽象 a，那么 a 唯一。特别地，我们用函数 α 表示 S 的最优抽象。

在抽象解释程序时，我们只能逐渐计算一个程序点 l 的状态集合，换言之，状态集合 S 在动态变化。为了比较不同时刻的状态集合，我们接下来严格定义具体域。

定义 3.4（具体域） 我们用 $(C, \subseteq)$ 表示具体域，集合 C 表示程序的真实行为，$\subseteq$ 是偏序关系。如果 $x, y \in C$ 且 $x \subseteq y$，那么 x 比 y 具有更强的性质，由行为 x 可以推测出行为 y。

类似地，我们来正式定义抽象域和抽象关系。

定义 3.5（抽象域和抽象关系） 我们用 $(A, \sqsubseteq)$ 表示抽象域，$\sqsubseteq$ 是 A 上的偏序关系。$(A, \sqsubseteq)$ 是具体域 $(C, \subseteq)$ 的一个抽象，那么抽象关系 $\vDash \subseteq C \times A$ 满足

- 对任意的 $c \in C, a_0$，$a_1 \in A$，如果 $c \vDash a_0$ 且 $a_0 \subseteq a_1$，那么有 $c \vDash a_1$；
- 对任意的 $c_0, c_1 \in C$，$a \in A$，如果 $c_1 \vDash a$ 且 $c_0 \subseteq c_1$，那么有 $c_0 \vDash a$。

构建抽象关系，一般来说就是找一个函数，把抽象元素映射到一个满足其性质的最大程序行为集上。

定义 3.6（具体化函数） 具体化函数或者具体化，是一个映射 $\gamma : A \to C$，对任意的 $a \in A$，

有 $\gamma(a) \vDash a$ 并且 $\gamma(a)$ 是 C 中满足 a 的最大的元素。

显然，具体化关系可以用具体化函数来表示，因此我们有下列等价关系

$$\forall c \in C, a \in A, c \vDash a \Leftrightarrow c \subseteq \gamma(a)$$

由于函数 γ 把抽象元素映射到一个满足性质的最大具体元素上，所以 γ 满足单调关系。换句话说，抽象元素越大，表示的具体状态就越多，我们对程序的刻画就越不精确。

因此，我们需要定义抽象化函数来把具体元素映射到一个能表示它的最小抽象元素上。

定义 3.7（抽象化函数） 对任意的 $c \in C$， c 有最优抽象的充要条件是：

- 存在一个抽象元素 a，满足 $c \subseteq \gamma(a)$；
- 如果有 a' 满足 $c \subseteq \gamma(a')$，那么 $a \sqsubseteq a'$。

此外，最优抽象的存在具有唯一性。

我们已经严格定义了具体域和抽象域，以及抽象关系和它们之间的转换函数，把具体域和抽象域紧密地关联在一起。但在分析图 3-1 和图 3-2 所示程序的语义时，我们看到，基于具体域的计算和基于抽象域的计算具有很大的不同。那么，基于抽象域的计算能够描述程序基于具体域计算的真实行为吗？要满足这种性质，需要抽象域和具体域之间满足伽罗华连接。

定义 3.8（伽罗华连接） 伽罗华连接（Galois Connection）要求具体化函数 γ 和抽象化函数 α 之间满足

$$\forall c \in C, a \in A \quad \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a)$$

我们通常用下列形式来表示

$$(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} (A, \sqsubseteq)$$

伽罗华连接具有程序分析所需要的一些重要性质，例如：

- 抽象化函数 α 和具体化函数 γ 都满足单调性，相互间可比较的元素映射后也可比较；
- $\forall c \in C, c \subseteq \gamma(\alpha(c))$，这表示具体元素抽象化再具体化后，会得到一个更弱的结果；
- $\forall a \in A, \alpha(\gamma(a)) \subseteq a$，这表示抽象元素具体化再抽象化后，会得到一个更强的结果。

这些性质，很容易从伽罗华连接的定义中导出。

有了伽罗华连接，我们在程序分析时，经常会交替使用抽象化函数和具体化函数。但很多时候，程序的具体状态并没有最优抽象。这意味着，程序分析的结果并不精确，需要我们在不同的目标之间进行权衡。

有了伽罗华连接，我们可以基于具体域、抽象域，以及它们之间的转换函数来求解表达式并执行各种语句。为了区别具体域和抽象域上计算的表示，我们用带#的运算符表示在抽象域上的计算，例如，$[\![E]\!]^{\#}$ 表示 E 的抽象解释。

ToyC 的抽象语义如图 3-6 所示，其中 $n^{\#}$ 表示常数在抽象域上的计算，具体形式与选择的抽象域有关。一个抽象元素可以表示多个具体元素，但它们在布尔条件 B 上的性质可能不同。因此，抽象过滤算子的计算借助了两个函数 α 和 γ，我们先进行具体化，再用具体化的过滤算子求出满足性质的抽象元素，最后进行抽象化。

$$
\begin{aligned}
&[\![E]\!]^{\#}: A \to A\\
&\qquad [\![n]\!]^{\#}(M^{\#}) = n^{\#}\\
&\qquad [\![x]\!]^{\#}(M^{\#}) = M^{\#}(x)\\
&\qquad [\![E_1 A_{\mathrm{op}} E_2]\!]^{\#}(M^{\#}) = A_{\mathrm{op}}^{\#}([\![E_1]\!]^{\#}(M^{\#}), [\![E_2]\!]^{\#}(M^{\#}))\\
&[\![B]\!]^{\#}: M^{\#} \to \{\top, \mathrm{true}, \mathrm{false}, \bot\}\\
&\qquad [\![E_1 B_{\mathrm{op}} E_2]\!]^{\#}(M^{\#}) = B_{\mathrm{op}}^{\#}([\![E_1]\!](M^{\#}), [\![E_2]\!](M^{\#}))\\
&\qquad F_B^{\#}(M^{\#}) = \alpha(\{m \in \gamma(M^{\#}) \mid [\![B]\!](m) = \mathrm{true}\})\\
&[\![C]\!]_{\mathcal{P}}^{\#}: \mathcal{P}(M^{\#}) \to \mathcal{P}(M^{\#})\\
&\qquad [\![\mathrm{skip}]\!]_{\mathcal{P}}^{\#}(M^{\#}) = M^{\#}\\
&\qquad [\![C_1; C_2]\!]_{\mathcal{P}}^{\#}(MM^{\#}) = [\![C_2]\!]_{\mathcal{P}}^{\#}([\![C_1]\!]_{\mathcal{P}}^{\#}(M^{\#}))\\
&\qquad [\![x = E]\!]_{\mathcal{P}}^{\#}(M^{\#}) = M^{\#}[x \mapsto E^{\#}(M^{\#})]\\
&\qquad [\![\mathrm{if}(B)\{C_1\}\mathrm{else}\{C_2\}]\!]_{\mathcal{P}}^{\#}(M^{\#}) = [\![C_1]\!]_{\mathcal{P}}^{\#}(F_B^{\#}(M^{\#})) \sqcup^{\#} [\![C_2]\!]_{\mathcal{P}}^{\#}(F_{\neg B}^{\#}(M^{\#}))\\
&\qquad [\![\mathrm{while}(B)\{C\}]\!]_{\mathcal{P}}^{\#}(M^{\#}) = F_{\neg B}^{\#}((\sqcup_{i\geqslant 0}^{\#}(C_{\mathcal{P}}^{\#} \circ F_B^{\#})^i(M^{\#}))
\end{aligned}
$$

图 3-6　ToyC 的抽象语义

顺序执行的两条语句，相当于两个函数的抽象解释的复合。但这样的复合，并不是在任何条件下都成立的。若抽象域和具体域满足伽罗华连接，则由下面的复合近似定理来保证其成立。

定理 3.1（复合近似定理） 给定单调函数 $f_0, f_1 : \mathcal{P}(M) \to \mathcal{P}(M)$，如果 $f_0^{\#}, f_1^{\#} : A \to A$ 是 f_0, f_1 的抽象解释（上近似，Over-Approximation），且有 $f_0 \circ \gamma \subseteq \gamma \circ f_0^{\#}$ 和 $f_1 \circ \gamma \subseteq \gamma \circ f_1^{\#}$，那么 $f_0^{\#} \circ f_1^{\#}$ 也是 $f_0 \circ f_1$ 的抽象解释。

由于复合近似定理的存在，我们在抽象域上计算函数的复合时，就可以等价为计算两个抽象解释函数的复合，从而简化程序分析的过程。

处理分支语句时，在具体域上，我们简单地计算了两个分支上状态的并集；在抽象域上，假定 if 分支和 else 分支经过抽象解释，分别有抽象状态 a_{l} 和 a_{r}，因为抽象域 $(A, \sqsubseteq)$ 是一个偏序，$a_{\mathrm{l}} \cup a_{\mathrm{r}}$ 不一定存在，所以对 $(A, \sqsubseteq)$ 的构造提出了要求。从最优抽象的角度来看，我们要计算出包含 a_{l} 和 a_{r} 的最小上界 a，即 $a_{\mathrm{l}}, a_{\mathrm{r}} \sqsubseteq a$，这要求 $(A, \sqsubseteq)$ 至少是一个完备偏序（Complete Partial Order，CPO）。在图 3-6 中，$\sqcup^{\#}$ 就是求最小上界（Join）的算子。为了形式化地研究程序语义，Scott[7]构建了数学基础——域论（Domain Theory），相关内容可以阅读专著[8]。

虽然循环可以看作是多个分支的并集，但分支的个数可以是无穷大的。因此在具体域上计算循环语句的语义时，我们也没有考虑其可计算性。令 $M_n = \cup_{i=0}^{n} M^i$，$\{M_n\}$ 则是一个单调序列。如果 $(C, \subseteq)$ 中的集合 C 有限，那么经过有限次迭代，M_n 收敛到 C 的一个子集或其自身。但当 C 是无限集时，M_n 在有限步内并不能收敛。

如果不能在有限步内计算出程序的状态，我们就不可能去可靠地判定其是否具有某些性质。因此，我们引入了拓宽（Widening）算子，加速循环计算的收敛。

定义 3.9（Widening 算子） 给定抽象域 $(A,\sqsubseteq)$，Widening 算子 ∇ 代表一个二元运算，满足下列性质：

对任意的抽象元素 $a_0,a_1\in A$，有

$$\gamma(a_0)\cup\gamma(a_1)\subseteq\gamma(a_0\nabla a_1)$$

对于抽象元素序列 $\{a_n\}_{n\in N}$，计算新的序列 $\{a'_n\}_{n\in N}$，有

$$\begin{cases}a'_0=a_0\\ a'_{n+1}=a'_n\nabla a_n\end{cases}$$

假定 Widening 算子 ∇ 可以加速计算的收敛，计算

$$M^{\#}_{i+1}=M^{\#}_i\nabla([\![C]\!]^{\#}_{\mathcal{P}}\circ F^{\#}_B)^{i+1}(M^{\#})$$

如果有 $M^{\#}_{i+1}\sqsubseteq M^{\#}_i$，即可断定循环计算收敛了，记作 $M^{\#}_{\text{loop}}$。显然，有

$$M^{\#}_{\text{loop}}=M^{\#}\sqcup^{\#}[\![C]\!]^{\#}_{\mathcal{P}}\circ F^{\#}_B(M^{\#}_{\text{loop}})$$

如果我们定义函数 $G:X\mapsto M^{\#}\sqcup^{\#}[\![C]\!]^{\#}_{\mathcal{P}}\circ F^{\#}_B(X)$，那么 $M^{\#}_{\text{loop}}$ 就是函数 G 的不动点，记作 $\text{lfp}G$。

因此，我们可以把循环的计算表示为

$$[\![\text{while}(B)\{C\}]\!]^{\#}_{\mathcal{P}}(M^{\#})=F^{\#}_{\neg B}(\text{lfp }G)$$

3.1.3 基于抽象解释的程序分析

根据 Rice 定理，程序的任何非平凡语义性质都是不可判定的。因此，我们没有有效的精确算法。给定性质 P，我们基于 3.1.2 节的抽象解释理论，构造了一个分析程序 Analyzer_P。

如果 Analyzer_P 断言为真，即程序在抽象域上满足性质 P，Analyzer_P 表示的抽象行为是程序真实行为的上近似，那么子集必然也具有性质 P。如果程序并非所有抽象行为都满足性质 P，但由于采用了过近似，所以我们很难断定是否是因为过近似引入的并不存在的程序行为导致了误报。至于如何消除误报，不在这里讨论。

根据 3.1.2 节的抽象解释理论，Analyzer_P 对程序行为的近似是否精确，主要取决于程序的真实行为。另外，抽象分析是否可靠还需要根据具体域上的计算来做比较。

Analyzer_P 的实现可分为下面三个步骤：

- 首先，确定能够描述程序行为的具体语义，最好是可靠的形式化描述。一般来说，程序的语义要能够表征性质 P。在第 2 章中，我们介绍了常用的操作语义、指称语义和公理语义。但是，语义的具体表示形式，主要是由性质 P 来决定的。
- 其次，选择合适的抽象方式，定义一些分析中用到的逻辑谓词。一种比较理想的方式是，基于收集程序具体行为构造具体域，来找到最优近似，从而得到抽象域。我们可以只考虑如何来表示单个变量的值域，而不管变量之间的关系，但为了提高表

示的精度，抽象域要能够表示变量间的关系，如八边形或多面体等抽象域。一般来说，逻辑谓词的构造和抽象域的选择，也是由性质 P 来决定的。

- 最后，根据抽象域的表示，实现抽象域上的计算（包括最小上界、Widening 和比较等）及程序设计语言中的计算。实际上，就是为目标语言写一个基于特定抽象域的解释器，并求出每个程序点上的抽象状态。

在实现时，抽象域的构造不太可能一蹴而就，通常需要根据分析的效果进行调整。如果分析精度太低，那么需要选择表示能力更强的抽象域，而这会引入更高的计算代价。如果计算时间太长，那么需要考虑降低抽象域的精度，或者选择更加适合性质 P 的抽象域。当然，也有工作研究如何系统化地构造抽象域[9]，但其方法还没有达到实用化的要求。一种更常见的做法是构造一个抽象解释器 Analyze(P, A)，把性质 P 和抽象域 A 作为参数，从而根据分析效果来选择抽象域和性质 P 的表示。

抽象解释器的构造可以参照所分析的程序设计语言语法的结构化语义表示，快速地递归实现。对于循环和递归函数，我们需要求它们的不动点。在迭代的过程中，每个程序点收敛到不动点的速度不一样，我们通常采用 WorkList 算法[10]或者更加鲁棒的 Chaotic 算法[3]来实现。

在实际应用中，待分析的目标程序比较复杂，通常是以项目的形式来进行分析的。一个项目有很多个源文件，每个源文件有很多个函数，一个函数会调用多个函数，因此构成了复杂的调用图。处理函数调用的方法有很多，按照我们对精度分析的需求，常用的方法如下。

- 把函数当成外部输入。碰到函数调用时，我们只需要关心被调用的函数对当前环境造成的影响，即哪些变量的值被修改了。基于抽象解释理论，我们只需要把变量更新为能够包含真实状态的抽象元素即可，如 $\top$。显然，这种方法会引入大量非真实的程序行为。
- 利用被调用函数的语义摘要。通常采用自底向上的方式来生成函数摘要，例如，用指称语义来表示被分析的函数，或者简单地用(输入, 输出)来表示。一般来说，生成函数摘要时，我们假定函数的输入都是 $\top$。在具体的调用点上，根据调用点状态和摘要来计算函数的输出。
- 在线分析被调用函数。根据调用点的状态，我们给被调用函数的参数赋值，然后开始具体分析。这种上下文敏感的分析方法，精度很高，但会导致大量的计算开销，很难扩展到调用链比较长的复杂项目中。

在实际应用中，分析精度和分析速度是两个互相矛盾的目标，有不少工作讨论如何实现平衡，还有很多工作探讨如何有效利用更多的资源来改进这两个目标。

3.1.4　案例分析

很多程序设计语言（如 C/C++、Java 和 Python 等）都提供了通过内存地址或指针来访问对象的方法。本节主要介绍程序中因为地址访问而引起的缺陷，如空指针解引用、数组越界、资源泄露、释放后引用等。因为变量地址是程序执行时的信息，所以静态分析工具无法

获取。如果地址指向的是复杂对象，如数组和数据结构，那么其就会给程序分析带来麻烦。

在图 3-8(a)所示的 C 语言程序中，数组 a 被声明为 char 类型的，指针 b 访问它时，把指向的对象当成了 int 类型的。因此，b[3]实际上访问的是以 a 为首地址，偏移为 12 的位置。这导致数组访问的位置超过了原本对象的范围，可能会引发“不可知”的错误。基于抽象解释的程序分析，在计算每个变量的抽象状态时，很难刻画这种数据类型的转换。

在图 3-8(b)所示的 C 语言程序中，语句 6 与语句 7 分别访问 a[6]与 a[-3]，如果只考虑对目标数组 a 的访问，那么这必然导致数组越界。但考察结构数组 arr 的内存，它们访问的还是 arr 数组内的元素。根据 C 语言中复杂数据结构的内存组织，可计算出语句 6 与语句 7 分别落在了 arr[1]与 arr[0]内。图 3-8(b)所示的程序并未访问数组外部的数据，这种写法虽然合法但并不安全。因此在访问复杂对象时，我们还需要考虑对象之间的从属关系。

```
1 void example1() {
2   char a[10];
3   int *b = (int *)a;
4   b[3] = 0;
5 }
```

(a) 示例 1

```
1 struct X {
2   int a[6];
3 };
4 void example2() {
5   X arr[3];
6   arr[0].a[6] = 1;
7   arr[1].a[-3] = 2;
8 }
```

(b) 示例 2

图 3-8　通过地址访问对象的示例

C/C++语言的内存访问最为灵活，可以建立数据对象的内存模型，描述对象的存储方式，试图刻画程序的数据访问。在基于抽象解释进行程序分析时，我们关心的是程序状态，即变量在某个程序点上的值。因此如果我们想要建模来表示地址类型数据访问的语义，需要解决下列问题：

- 首先，我们要解决&x 的语义表示。&x 的具体语义，即 x 的地址，在程序执行时完全可以确定。在静态分析时，虽然不知道变量的具体地址，但地址指向的对象是可以确定的。因此，我们完全可以用地址指向的对象来表示&x 的语义。
- 其次，我们要解决地址的有效范围问题，考察变量在内存中的分配。C 编译器在分配栈上变量存储空间时，通常按照声明顺序从低地址到高地址进行分配；Java 编译器从堆上分配空间，所以变量的地址没有固定的规则。一般而言，程序对象在内存空间中的地址并不遵循一致的规则，反而依赖编译器和具体的执行过程。但是，程序访问内存对象必须满足类型约束，体现出局部性。如果访问超出对象边界，访问语义的正确性就无法保证。因此，我们引入区域内存（Region Memory）的概念，任何对象 o 的有效地址范围都表示为[0, sizeof(o) – 1]，即对象从开始到结束的地址范围。
- 再次，我们要处理地址运算。程序访问的对象，其类型不是基本类型，就是基本类型的复合类型。所以，程序访问的对象都可以用类型来约束，而基本类型的长度都

是确定的。任何类型的变量的开始地址都是 0，因此我们可以递归地计算对象内成员变量的相对地址。

- 最后，我们要处理类型的转换。对象在第一次声明时的类型为其固有属性。类型转换是对内存对象的重解释。访问对象时，应根据新类型来计算地址，但还要服从声明类型的存储组织约束。

令 E 是地址类型的表达式，那么 E 指向一个特定的变量。基于内存模型，我们补全 ToyC 的抽象语义，如图 3-9 所示。

$$
\begin{aligned}
&[\![E]\!]^{\#} : X \to X \\
&\qquad [\![\&x]\!]^{\#}(M^{\#}) = x \\
&\qquad [\![\text{malloc}_i]\!]^{\#}(M^{\#}) = a_i \\
&\qquad [\![*E]\!]^{\#}(M^{\#}) = M^{\#}([\![E]\!]^{\#}(M^{\#})) \\
&\qquad [\![*E_1 = E_2]\!]^{\#}_{p}(M^{\#}) = M^{\#}[*[\![E_1]\!]^{\#}(M^{\#}) \mapsto [\![E_2]\!]^{\#}(M^{\#})]
\end{aligned}
$$

图 3-9　地址变量访问的抽象语义

在 ToyC 里，程序可以多次调用 malloc，我们根据调用点 i 来区分，表示为 malloc_i。由于 malloc 的作用是从堆上分配内存，所以我们用 a_i 来表示第 i 个匿名对象。

下面，我们用扩充的抽象解释以及地址变量语义的抽象表示，以别名分析为例介绍面向特定性质的基于抽象解释的静态分析技术，然后以空指针解引用、数组越界、资源泄露和释放后使用等缺陷为例介绍面向特定缺陷类型的静态分析技术。

1．别名分析

如果程序设计语言支持引用，一个对象就可以通过多个变量来访问，那么这些指向同一对象的变量就构成了别名（Alias）。C/C++、Java 和 Python 等很多语言中都有别名问题。别名会导致一个对象可以通过多种途径来修改，给收集和分析程序的行为制造了麻烦，影响了程序分析的精度。

在分析程序时，我们为变量构建内存模型，以此记录地址变量指向的对象。根据抽象解释的语义，间接赋值语句改变的是被指向对象的值。如果我们再引用该变量，那么拿到的就是修改后的值。在抽象解释的扩展框架下，我们并不需要特别计算特定内存区域的别名。

在分支和循环结构中，一个地址变量可以在不同的执行路径中指向不同的对象。在汇聚点上，执行 Join 算子后，一个地址变量就可能指向多个不同对象。通过地址变量间接赋值时，抽象解释要更新所有被指向对象的值。从程序的具体语义看，这种情况不可能发生，也就是说，一个特定的执行序列，地址变量只能指向一个对象。因此，利用抽象解释再结合路径敏感分析，可以提高地址变量的分析精度。

2．空指针解引用

我们先分析空指针访问的具体语义。程序访问地址变量时，如果变量指向合法的对象，

即某个变量或者通过 malloc 分配的匿名对象，我们就能够拿到被访问对象的值；如果变量没有指向的对象，访问时就会出现问题。

从程序的具体语义来看，我们只需要关心地址变量是否指向某个对象。因此，我们可以构建地址抽象域 $\text{Addr}=\{\bot,0,1,\top\}$，其中 $\bot$ 表示地址变量未被初始化，0 表示空指针，1 表示非空指针，$\top$ 则表示 0 和 1 两种情况。

在构建程序的内存模型时，如果我们用 Addr 来表示地址变量的抽象值，那么判别空指针解引用的规则就很直接了。首先，我们扫描程序，找到访问地址变量的程序点。如果是基于 LLVM IR 来实现的，那么读内存就是 load 指令，写内存就是 store 指令，而地址变量就是指令中的源或者目标。然后，我们检查地址变量的值，如果是 0，那么一定会发生空指针解引用；如果是 $\top$，则可能发生；如果是 $\bot$，则代表地址变量未初始化。

分析实际程序时，地址变量的赋值很多时候由多个分支条件来确定，如果分析不够精确，就会引入很多误报。因此，空指针解引用的精度取决于抽象分析时其他依赖变量的分析精度。

在图 3-10 所示的例子中，在构建内存模型时，地址变量 j 指向了地址变量 i。如果 i 指向空，那么程序执行语句 6，根据内存模型，**j 访问的是空对象。如果 i 不为空，那么执行语句 4，这时由 i 的取值来决定访问的对象。

```
1 int testl(int *i){
2   int **j = &i;
3   if (i){
4     return **j;
5   }
    else {
6     return **j+1;
7   }
8 }
```

图 3-10　地址变量赋值的例子

在图 3-11 所示的例子中，charBad 函数是否有空指针解引用与参数 i 和 j 的指向有关。在 case_bad 函数中，c 和 d 是别名，即指向了相同的对象。基于扩展的抽象解释分析，c 和 d 通过同一条指向链指向了变量 a。如果用内联分析，那么语句 2 更新*i 后，会把 i 和 j 指向空对象，**j 自然地触发了空指针访问。在这个例子里，分析是否准确主要取决于过程间分析的策略。如果采用基于摘要的方式，那么就要考虑 i 和 j 是别名的情况。

3．数组越界

数组越界是 C/C++语言编写的程序中经常出现的错误。由于数组常用来存储外部输入，因此数组越界又称为“缓冲区溢出”。被人熟知的蠕虫病毒和很多互联网攻击，就是利用了缓冲区溢出。

我们通过下标来访问数组中的元素，这涉及地址或下标值的计算。我们先来考虑数组访问的具体语义。程序计算出来的地址值，只要不是 0 或者负值，便被认为是合法的地址值。

但在实际执行中，该地址可能是无效的，因为违反了操作系统的内存管理机制。因此，从地址值是否合法来判断数组越界不够准确。

```
1  int *charBad(char **i, char **j) {
2    *i = NULL;
3    printf("%d", **j);
4  }
5  void case_bad(){
6    char a = 0;
7    char *b = &a;
8    char **c = &b;
9    char **d = c;
10   charBad(c, d);
11 }
```

图 3-11　参数为别名的例子

构建内存模型时，我们用对象的类型来约束其内存中的组织。以图 3-12 所示的程序为例，对象 X 是一个有 10 个整数的数组，arr 是一个有 4 个 X 类型数据的数组。分析这段代码时，我们可以得到如图 3-13 所示的 bad 函数的内存模型，这里用 i32 表示 int 类型，从而直观地表示类型的长度。程序中的对象，可以分为全局、栈和堆等不同类型，标志是为了表示对象所处的位置。因为 arr 是一个复杂的数据结构，变量之间有从属关系，所以根据 bad 函数的内存模型，有 p.a[39]⊏p.a⊏p⊏arr。在计算缓冲区的访问地址时，如果下标指向的是缓冲区的元素，那么下标在后续的计算中必须被内存模型约束。

```
1  struct X {
2    int a[10];
3  }
4  void bad() {
5    X arr[4];
6    auto &p = arr[0];
7    p.a[39] = 0;
8    auto &q = arr[1];
9    q.a[-11] = 0;
10 }
```

图 3-12　数组越界的例子

当对象逐级嵌套时，越界检查就是一个递归的过程：

- 步骤 1：获取正在访问的内存对象的约束，如果访问的内存对象满足边界约束，那么结束检查；
- 步骤 2：向上追溯它的上级对象，并检查该对象。如果没有上级对象发生越界访问，那么结束检查；否则，执行步骤 1。

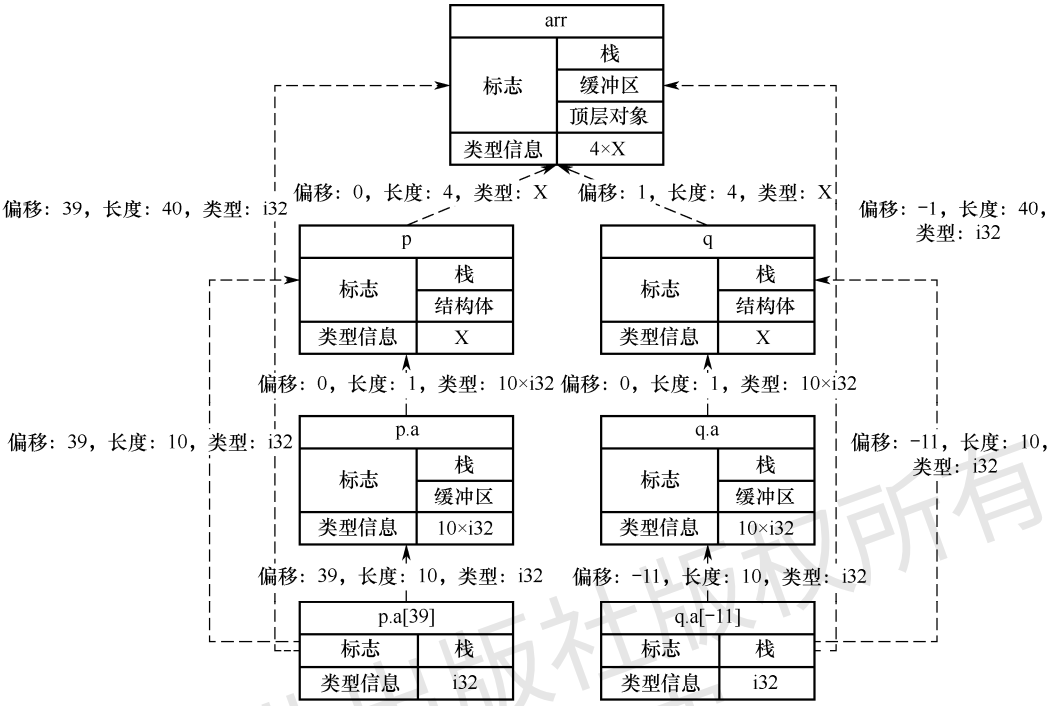


图 3-13　bad 函数的内存模型

图 3-14 给出了 bad 函数的检查过程。p.a[39]访问数组 p.a，类型和偏移的单位为 i32，偏移量 offsets 为 39，长度 length 为 10，有 offset≥length，即出现数组越界。p.a 指向的对象是 arr，计算 p.a[39]在 arr 中的偏移量，offset= 0×10 + 39 = 39，length=4×10 = 40，有 offset<length，即没有出现数组越界。同理，q.a[−11]在 arr 上访问时，offset = 1 × 10 + (−11) = −1，length = 4 × 10 = 40，有 offset < 0，即出现下溢。

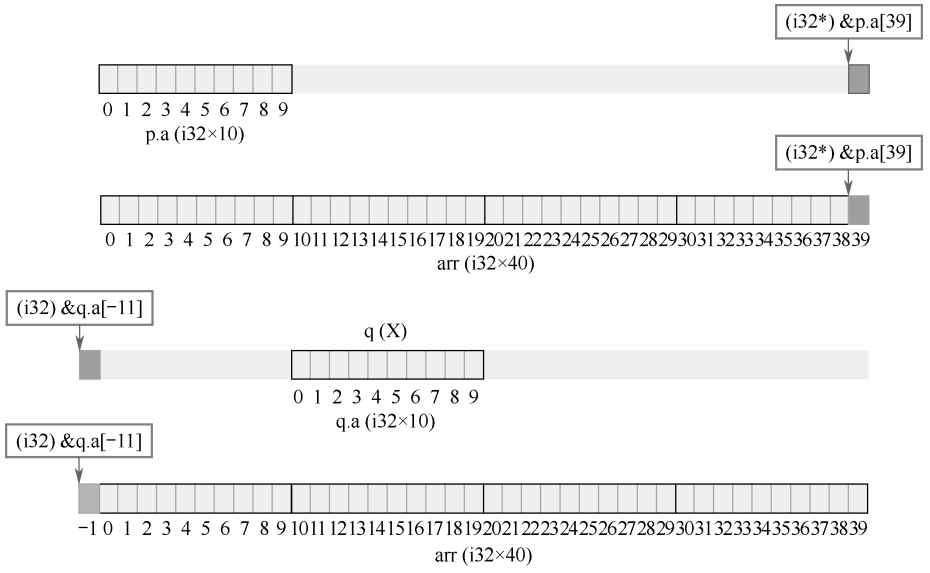


图 3-14　bad 函数的越界检查

4．资源泄露和释放后使用

程序执行时需要使用各种资源，如堆上的内存、文件和进程等。这些资源需要显式申请和释放，否则会导致资源耗尽，影响程序的执行。对于打开的文件和运行的进程等资源，系统需要分配内存记录其相关信息，实际上消耗的也是内存资源。系统会调用 close 和 wait 函数，目的是，请求操作系统释放对应的内存。

使用 C/C++语言编写应用程序时，开发人员需要管理堆上的内存对象。C 语言提供 malloc 和 free 函数分别用于申请和释放内存；C++语言提供 new 和 delete 函数管理内存。

程序中出现内存资源泄露的典型场景包括：

- 程序逻辑复杂，执行路径多，不同内存对象在不同路径上使用的策略各异；
- 内存对象在被调用函数中申请，通过返回值或参数传址的形式给调用者使用。

我们对内存对象 o 的生命周期建模，用布尔变量 o.state 来表示状态，1 表示活跃，0 表示消亡。显然，malloc/new 函数创建对象 o 并令 o.state=1，free/delete 函数销毁对象 o 并令 o.state=0。此外，程序必须用一个地址变量 p 来访问堆对象 o。显然，出现资源泄露的情况必须满足 o.state=1 且不存在 p 指向它。

别名让资源泄露的判别变得更复杂了。由于多个地址变量可以指向同一个内存对象 o，因此我们还需要跟踪地址变量是否活跃。我们使用集合抽象域描述一个地址变量可能指向的内存区域，例如，$\mathrm{ptr(p)}=\{o_1,o_2\}$ 表示 p 指向两个堆对象 o_1 和 o_2；用区间抽象域描述一个堆对象的引用计数，例如，$\mathrm{rc}(o_1)=[1,2]$ 表示 o_1 有 1 个或者 2 个引用。

图 3-15 给出了资源泄露的示例。

```
1  void intra_simple(){
2    FILE *f = fopen("text.txt", "r");
3    if(rand())
4      return; // leak
5    fclose(f);
6  }
```

图 3-15　资源泄露的示例

假设 fopen 函数打开的文件对象为 F，F 可以视为堆对象，那么执行语句 2，程序状态满足性质

$$\mathrm{ptr(f)}=\{F\}\wedge\mathrm{rc(F)}=[1,1]\wedge\mathrm{F.state}=1$$

执行语句 3 后，如果接下来执行语句 4，那么地址变量 f 被销毁，有

$$\mathrm{rc(\#F)}=[0,0]\wedge\mathrm{F.state}=1$$

如果执行语句 5，有

$$\mathrm{ptr(f)}=\{F\}\wedge\mathrm{rc(F)}=[1,1]\wedge\mathrm{F.state}=0$$

最后，地址变量 f 被销毁，有

$$\mathrm{rc(F)}=[0,0]\wedge\mathrm{F.state}=0$$

在函数的退出点，最终状态是上述状态的最小上界，即

$$rc(F) = [0,0] \wedge F.state = \top$$

此时，没有地址变量指向 F，并且 F.state 不为 0，故可能存在资源泄露。

在图 3-16 所示的程序中，我们也可以用抽象解释来计算内存对象的抽象值。use 函数让 x–>field 指向一个新的堆对象，但如果 x–>field 原本就指向一个堆对象，那么该操作可能会导致资源泄露。因此，资源泄露与否取决于 x–>field 指向堆对象的状态，即是否存在其他地址变量指向它。由于 use2 函数释放了 x–>field 指向的堆对象，该程序不会发生资源泄露。

```
1  struct node {
2    int *field;
3  }
4  void print(node *x){
5    printf("%d", *x->field);
6  }
7  void use(node *x,int n){
8    //if x->field holds some memory regions
9    //there may be a leakage
10   x->field =(int *)malloc(sizeof(int));
11   if(x->field == nullptr)
12     return;
13   *x->field = n;
14   print(x);
15 }
16
17 void use2(node *x,int n){
18   if(x->field != nullptr){
19     free(x->field);
20     x->field =(int *)malloc(sizeaf(int));
21   }
22   if(x->field == nullptr)
23     return;
24
25   *x->field =n;
26   print(x);
27 }
28
29 int main(){
30   node x, y;
31   y.field=(int *)malloc(sizeof(int));
32   use(&x, 10);   //ok
33   use(&y, 100);  //bad
34   use2(&x, 10);  //ok
35   use2(&y, 100); //ok
36   return 0;
37 }
```

图 3-16　内存的过程间使用示例

这个模型也可以用于检测释放后使用（Use-After-Free，UAF）问题。如果有 ptr(p)={o} 且 o.state=0，说明内存对象 o 已经被释放，解引用*p 访问的对象不确定，那么就会引起 UAF 问题。

在图 3-17 所示的程序中，执行语句 2 之后，假设 open 函数打开的文件对象为 f，那么执行语句 4，内存对象 f 被释放，有 $ptr(f)=\{F\} \wedge rc(F)=[1,1] \wedge F.state=0$；再执行语句 5，我们访问的对象有 $F.state=0$，说明 F 已经被释放，就出现了 UAF 问题。

```
1  void uaf(){
2    int f = open("text.txt", O_RDONLY,S_IRUSR);
3    if(rand())
4      close(f);
5    write(f, "Use after free!");
6  }
```

图 3-17　UAF 问题的示例

3.2　基于深度学习的缺陷分析

基于静态分析的缺陷分析需要人工地定义缺陷模式，代价一般较高。随着深度学习的发展，利用深度学习技术从大量的缺陷数据中自动学习缺陷代码的特征模式，为缺陷分析提供一种新的途径。基于深度学习的缺陷分析主要可以分为基于代码快照表示学习的缺陷检测和基于代码变更表示学习的缺陷检测两种技术。

3.2.1　基于代码快照表示学习的缺陷检测

基于代码快照表示学习的缺陷检测技术[11, 12, 13]是指在大量缺陷代码和缺陷修复代码的基础上，通过自动学习代码的语义和语法特征来有效地检测缺陷。现有的基于代码快照表示学习的缺陷检测技术主要存在两个问题。第一，除 Li 等人[12]和 Zhou 等人[13]提出的方法以外，其他方法都将源代码表示为 Token 序列或抽象语法树节点的平铺序列，这限制了深度学习模型学习具有数据依赖性和控制依赖性的代码语义和语法特征的能力，导致误报率高。第二，这些方法的代码快照表示学习仅仅将整个方法转换成一个向量表示，因此只适用于粗粒度的方法级缺陷检测，并不适用于细粒度的语句级缺陷检测，而且缺乏解释性。这两个问题使得基于代码快照表示学习的缺陷检测技术难以在实际中使用。

为了解决上述两个问题，Li 等人[14]提出了一种基于图的代码快照表示学习技术，在全面刻画代码中的数据依赖与控制依赖的基础上，实现语句级缺陷检测，如图 3-18 所示。

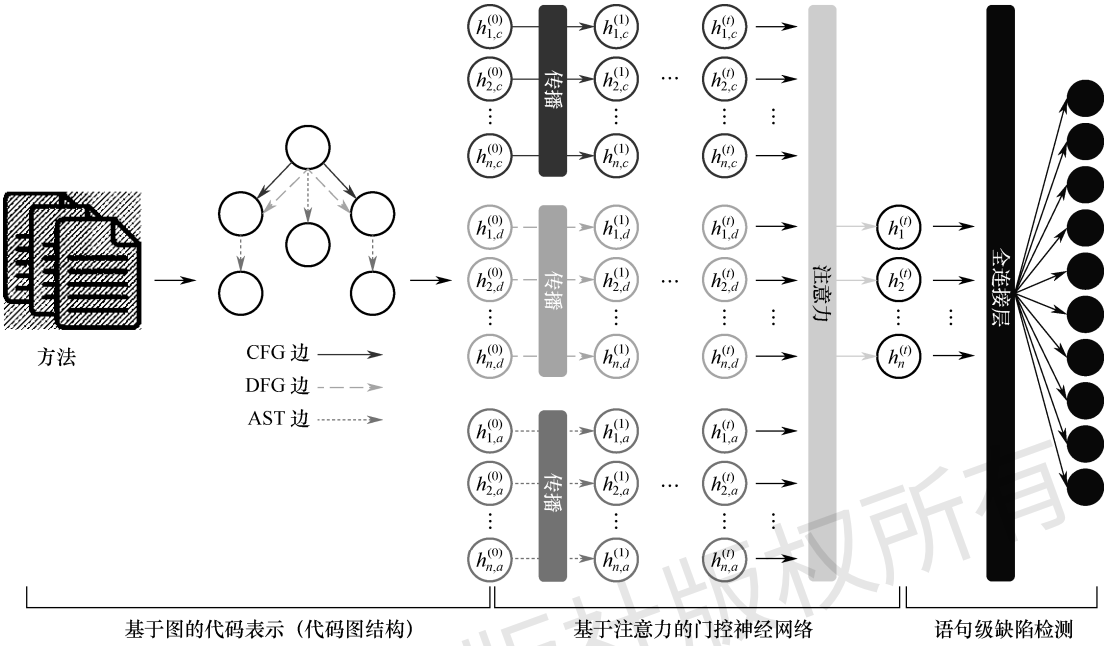


图 3-18　基于代码快照表示学习的语句级缺陷检测

具体而言，首先，该技术将一个方法转换成一个包含控制流图（CFG）、数据流图（DFG）和语句级别的抽象语法树（AST）信息的代码图结构，从而使语句之间的语义依赖性以及语法结构得到了全面的表示。同时，为了快速将方法转换成代码图结构，该技术实现了一种轻量级的分析方法，即在源代码级别上构造 CFG、DFG 信息且在语句级别上构建 AST 信息，无须像 Soot 或 Wala 工具那样需要完整的项目编译环境，便于分析大量的缺陷代码和缺陷修复代码数据。

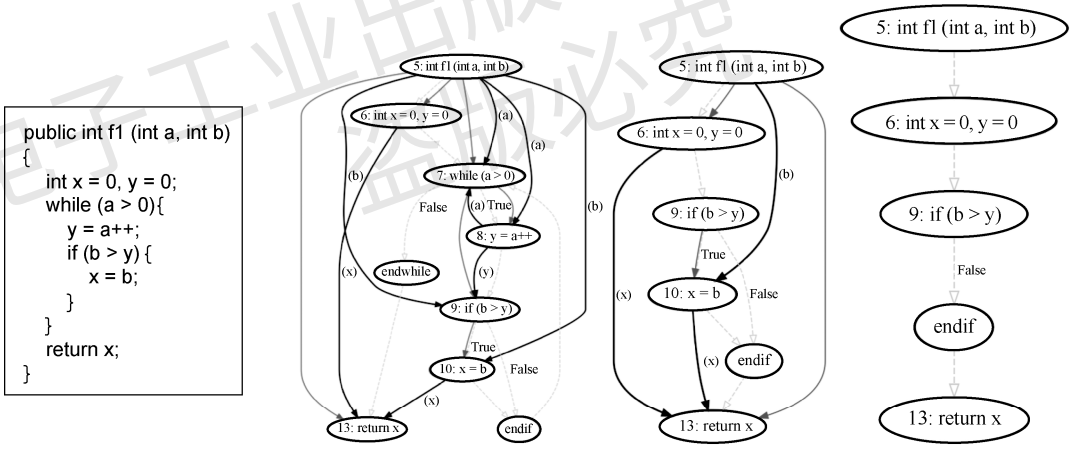
其次，该技术提出了一个基于注意力的门控神经网络方法，在构建的代码图结构的基础上，使用一个三层的基于注意力的门控神经网络来更新和学习其所包含的图节点的特征向量。该网络模型中的每一层又可以分为两个模块，第一个模块由 GGNN 模型中的 GRU（门控循环单元）构成，主要用来传播和聚集一个点和其周围邻居节点的表示信息。由于所构建的代码图结构有三种类型的边（即 CFG 边、DFG 边、AST 边），而且 GRU 是根据边的类型来聚集一个点的邻居节点信息从而生成该节点的向量表示的，因此一个节点会有三种向量表示。第二个模块采用的是 GAT 模型中的注意力单元，其会根据节点之间的权重来融合一个节点的三种向量表示，以此生成该节点的最终向量表示，同时为了增强模型的可解释性，还将对得到的权重进行了可视化操作。在经过三层神经网络的学习与更新后，图中每个节点的向量表示都蕴含了语法和语义信息。

最后，该技术进行语句级缺陷检测，采用单层神经网络作为分类器来预测图中每个节点是否有缺陷。此外，在模型进行融合操作的同时，会得到每个节点和相邻节点之间的融合权重，利用这些权重可以解释说明本次缺陷检测所得到的结果。

3.2.2　基于代码变更表示学习的缺陷检测

缺陷修复是一种代码变更的过程。然而，由于缺陷数据相对少，基于代码快照表示学习的缺陷检测技术难以理解缺陷代码和缺陷修复代码的特征，导致检测准确率不高。因此，一种更好的方法[15]是，首先利用海量的开源项目中的代码变更数据来实现代码变更预训练模型，然后在预训练模型的基础上，利用少量缺陷代码和缺陷修复代码的数据，通过模型微调的方法学习到其特征知识，从而提高检测的准确率。

具体而言，首先，该方法采用基于程序切片的代码变更表示，综合考虑变更代码和未变更代码之间的控制或数据依赖，如图 3-19 所示。针对一次代码提交中发生的代码的变更，该方法先对变更前（或变更后）的方法生成程序依赖图（PDG），再根据删除（或添加）的代码对变更前（或变更后）方法的 PDG 进行前向和后向的数据流、控制流切片，进而将变更前（或变更后）方法表示为一种由一系列程序切片、控制流通路、标记符号组成的自上而下的层次化结构。其中，控制流通路来自程序切片，代码语句来自控制流通路，而标记符号来自代码语句。这种层次化的表示方法能够更好地刻画代码变更中的语义信息。



(a) Code　　(b) PDG　　(c) Slice　　(d) Path

图 3-19　基于程序切片的代码变更表示示例

其次，在这种代码变更表示的基础上，采用稀疏 Transformer 模型实现代码变更的预训练。该模型中的稀疏模式由代码变更表示中的层次结构知识来决定，并使用位置编码来考虑代码语句间和代码标记符号间的排序关系。如图 3-20 所示，R1 代表程序切片包含了控制流通路，R2 代表控制流通路包含了代码语句，R3 代表变更的代码语句之间的重要性，R4 和 R5 代表与变更的代码语句具有数据依赖的语句，R6 和 R7 代表与变更的代码语句具有控制依赖的语句，R8 代表代码语句包含了代码标记符号。因此，稀疏矩阵刻画了代码变更表示中不同层次结构之间的关系，以便我们更好地学习到代码变更的语义。在训练时，该方法使

用了三个预训练任务来帮助预训练模型学习代码变更表示，即缺失节点预测、缺失边预测和代码变更翻译。缺失节点预测旨在更新代码变更表示中每个节点的表示，缺失边预测旨在将代码变更中不同层次结构之间的关联关系结合起来，而代码变更翻译旨在捕捉代码变更前到代码变更后的变更语义。

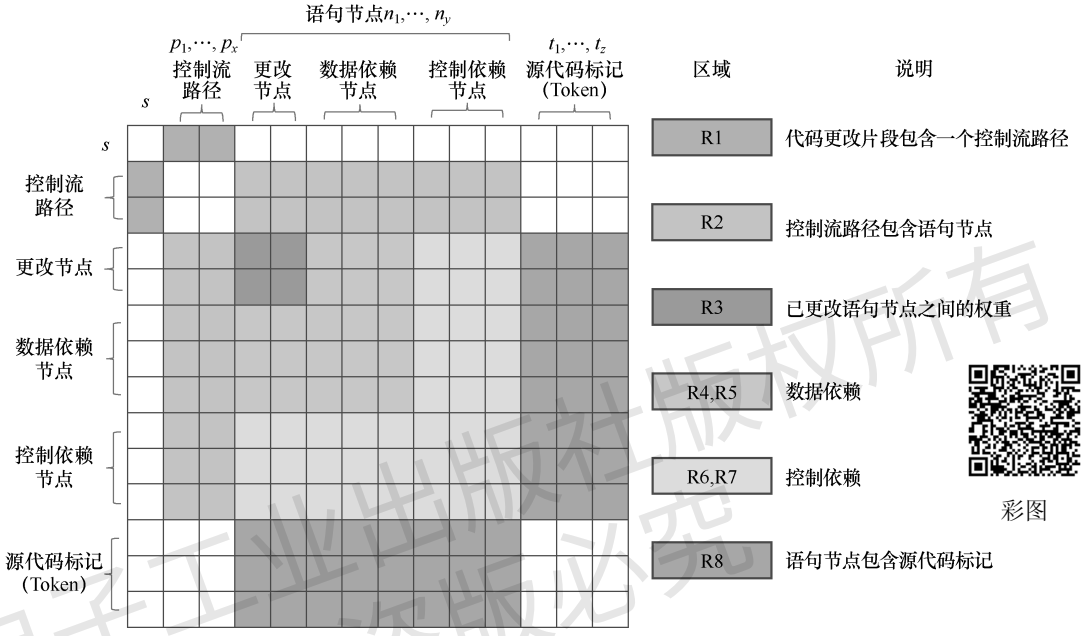


图 3-20　稀疏矩阵示例

最后，在代码变更预训练模型的基础上，采用模型微调的方式获得缺陷检测模型。该方法只需要少量的缺陷数据就可以完成缺陷代码和缺陷修复代码特征知识的学习。此外，由于该方法可以获得受缺陷影响的代码语句，而且代码变更表示学习方法细到了代码语句级别，因此，该方法可以实现代码行级的缺陷检测，比现有的方法级缺陷检测技术的粒度更细。

3.3　基于大模型的缺陷分析

随着大模型的发展，其较传统的深度学习模型具有更加突出的代码理解能力，因此大模型开始被应用于缺陷分析的各种场景，包括缺陷定位、缺陷检测、缺陷修复等。

3.3.1　基于大模型的缺陷定位

缺陷定位技术旨在确定测试所发现的缺陷在代码中的位置（如代码行）。现有的缺陷定位技术结合了静态和动态程序分析，以便计算代码行的缺陷可疑性。例如，基于频谱的缺陷

定位（SBFL）方法对失败/通过的测试用例的代码行覆盖数据进行统计分析，以计算代码行的可疑性。SBFL 仅依赖于测试覆盖率，因此在面向与数值相关的缺陷时不太适用；它还对底层测试套件的属性非常敏感，如覆盖率和失败/通过的测试用例的数量。基于变异的缺陷定位（MBFL）通过分析测试用例的行为来定位缺陷，其使用变异分析来评估特定代码行对测试结果的具体影响。尽管 MBFL 方法有效，但它是计算密集型的方法，开销较大。基于机器/深度学习的故障定位（MLFL）利用机器/深度学习方法来测试或执行与代码行的缺陷可能相关联的特征。MLFL 技术从各种信息中检测出有缺陷的代码行，包括现有 SBFL 和 MBFL 技术的可疑性得分、缺陷倾向特征（如代码复杂度指标）等。这些方法展示了机器/深度学习模型在完成缺陷定位任务方面的潜力。除了传统的机器/深度学习方法，大模型也开始被应用于缺陷定位任务。

LLMAO[16]是一个基于大模型的缺陷定位方法。它在从左到右的语言模型上训练轻量级的双向适配器。这些适配器只关注相对较少的参数，可以在小规模真实缺陷数据集上进行有效训练，而无须更新大模型的大量参数。LLMAO 不需要执行或分析测试用例，也不需要检查测试用例上的程序行为，所以相对来说是轻量级的。通过在特定的缺陷数据集上进行微调，LLMAO 可以应用到不同编程语言的缺陷定位场景中。具体来说，LLMAO 在大模型的每个训练样本中提取每个换行符所标记的最终隐藏状态作为该代码行的标记（即其中每个标记代表一行代码）。LLMAO 基于每个标记的丰富学习表示，通过训练更多 Transformer 层，使模型能够在上下文的代码行表示之间交换信息，从而为每行代码生成一个新的、具有双向信息的表示。因此，可以说 LLMAO 训练了一个双向适配器，该适配器由一定数量的 Transformer 层组成，并遵循标准的 Transformer 编码器架构。

3.3.2　基于大模型的缺陷检测

基于大模型的缺陷检测技术是指，利用大模型的代码理解能力识别与特定缺陷相关的敏感操作，并在此基础上通过模式匹配来发现潜在缺陷。本节将以资源泄露缺陷为例，介绍基于大模型的缺陷检测技术。

资源泄露缺陷是一类具有严重危害的缺陷类型。传统的资源泄露分析方法依赖于预定义的资源获取和资源释放的 API 知识，因此使用固定规则进行检测。然而，传统的资源泄露分析方法有两个局限。第一，该方法的可达性验证分析可能会导致误报。具体而言，静态分析中针对资源可达性的条件分支需要进行特殊判断和分析，尤其是在资源泄露检测时，控制流路径内的资源可达性分析会显著影响误报率。在实际项目中，资源的可达性判断需要处理的情况更加复杂。传统的资源泄露分析方法不能解决在复杂可达性条件下的资源可达性判断问题，这将导致在某些情况下，其会误报实际无法访问的资源。第二，该方法所覆盖的获取资源和释放资源的 API 种类有限。尽管目前已有的检测工具（如 FindBugs、Infer）被广泛使用，但它们通常只针对一些常见的资源操作类型的检查。换言之，一些特定领域、项目自定义、少见的资源泄露缺陷无法被这些工具检测出来。因此，系统中存在的资源泄露风险无法得到

全面地检测。相反地，对于大模型来说，其在预训练阶段接触过大量多样性代码和相关文档，大模型可以充当具有出色理解能力的广泛知识库，用来弥补传统方法依赖规则匹配的局限性。

InferROI[17]是一个基于大模型的资源泄露缺陷检测技术。与仅仅依靠匹配预定义的资源获取/资源释放 API 对空值进行检查不同，InferROI 利用大模型来结合资源管理知识和对代码上下文的理解，从而通过推断资源导向意图（包括资源获取、资源释放、资源可达性验证）来检测代码中的资源泄露缺陷。针对一个代码片段，InferROI 使用一个提示模板来指导大模型推断相关的资源获取、资源释放、资源可达性验证意图，从而消除对资源 API 配对的先验知识需求。通过汇总这些推断出的意图，InferROI 使用轻量级的静态分析算法从代码中提取控制流路径，从而实现资源泄露缺陷的检测。

InferROI 考虑了在单个控制流路径上的资源获取和释放，以便找到潜在的泄露缺陷。同时，InferROI 还考虑了资源在这些路径上的可达性对同一条件语句的其他分支的影响，从而减少了误报并提高了检测精度。

InferROI 的核心步骤是利用提示模板使大模型推断出当前代码片段中的资源导向意图。InferROI 参考了 OpenAI 的提示工程最佳实践，设计了包含三部分的提示，分别是：任务描述与指令、输出格式规范、代码占位符。

- 任务描述与指令介绍了任务背景并概述了需要由大模型执行的五个连续指令。前两个指令是引导指令，引导大模型以平滑的推理链从基本起点开始执行。在引导指令中，大模型被提示对涉及的对象执行类型推断，并确定哪些类型代表可泄露的资源。后三个指令要求大模型推断出三种类型的资源导向意图。在整个分析和推断过程中，大模型充分利用其丰富的背景知识来理解代码上下文并有效地执行指令。
- 输出格式规范旨在规范识别出意图的格式，以便后续解析大模型的输出。
- 代码占位符用来创建一个对大模型来说可作为特定提示的标记，从而提高生成代码的准确率。

3.3.3 基于大模型的缺陷修复

缺陷修复技术旨在自动地对包含缺陷的程序进行修改从而消除程序中的缺陷，提升开发人员的软件维护效率。缺陷修复通常分为以下三个阶段。

- **缺陷定位**：对包含缺陷的代码进行分析，找到可能出错的位置（如代码行）。
- **补丁生成**：在上述可能出错的位置上进行补丁生成，并通过程序归约（如测试用例）筛选符合条件的补丁（即似真补丁）。
- **补丁检查**：由于测试用例的充分性通常有限，通过测试用例验证的似真补丁也有可能是错误补丁，因此需要对上述似真补丁进行正确性检查从而得到正确补丁。

缺陷定位技术已经在 3.3.1 节进行了介绍，本节将介绍大模型在补丁生成及补丁检查上的应用。传统的补丁生成方法通常采用基于搜索、基于模板、基于约束等策略，对包含缺陷

的程序进行自动修改从而生成补丁。随着深度学习的快速发展，基于深度学习的补丁生成方法也受到了广泛关注。目前主流的基于深度学习的补丁生成方法将补丁生成问题转换为从缺陷代码到修复代码的机器翻译问题，从而在大量缺陷修复历史数据上训练出不同结构的深度学习模型。最近，大模型的出现又为补丁生成带来了新的技术途径。

基于大模型的补丁生成基础方法。由于大模型在大量开源代码上进行过预训练，因此即使在不进行进一步微调或者特殊处理的情况下，其生成补丁的正确性也都和在数据集上进行微调的小型模型相当，甚至更有竞争力。例如，在零样本的场景下，几种常见的基于大模型的补丁生成基础方法包括：仅给定上文让大模型做下文代码续写；让大模型重新生成整个函数；去掉出错行代码（但以注释形式在输入中给出），让大模型在上下文中间对出错行进行填空。这三种基础方法均取得了不错的补丁生成效果，甚至在有些案例上的效果超过了目前最好的微调小型模型。

AlphaRepair[18]是首个基于大模型的填空式补丁生成方法。AlphaRepair 的核心步骤是，对出错行进行基于不同规则的掩码操作，并通过让大模型补全掩码来生成补丁候选集合。具体而言，AlphaRepair 设计了三大类掩码策略，分别是：（1）完整掩码，即掩盖当前整行代码（缺陷代码行）、掩盖当前代码行的前一行代码、掩盖当前代码行的后一行代码。该策略的填空粒度最粗，通过让大模型直接重新编写整行代码来生成补丁；（2）部分掩码，即随机掩盖当前代码行中的部分代码 Token。该策略引入了随机性，使大模型能够考虑到各种不同的代码片段，从而增加了生成补丁的多样性；（3）模板掩码，即按照模板规则掩盖当前代码行中的部分代码，如掩盖函数调用中的函数名或者参数名、掩盖布尔表达式中的操作符或者变量名。该策略在保持一定随机性的基础上引导大模型按照一定的模板生成补丁，其对生成补丁的语法正确性有一定的保证。

这三种策略综合考虑了补丁生成的多样性和针对性，分别从不同角度引导大模型进行填空式的补丁生成。在补丁生成阶段，对于每个出错的候选代码位置，AlphaRepair 将依次应用上述的掩码策略生成多个候选补丁，并通过测试用例进行后续筛选和验证。AlphaRepair 的修复效果超过了目前最好的微调小型模型，本书写作时，它在缺陷修复数据集 Defects4J 上正确修复了 74 个缺陷（之前最好的补丁生成方法修复了该数据集上的 68 个缺陷）。

基于大模型微调的补丁生成方法。为了进一步增强大模型在补丁生成任务上的能力，各种面向补丁生成的大模型微调方法被相继提出。其中，最直观的微调方法是，在缺陷修复数据集上对大模型进行微调，即将缺陷代码和相关信息作为模型训练的输入，将修复代码作为模型训练的输出。通过微调，大模型将进一步学习如何根据包含缺陷的上下文和其他信息生成正确的补丁。

FitRepair[19]使用了以下两种不同的大模型微调策略。

第一种是知识增强的微调策略。FitRepair 使用原始缺陷项目的源代码构建了一个与预训练任务相近的数据集，通过掩盖部分（50%）代码标记，使得大模型能够学习该项目特定的代码标记和编程风格知识。具体而言，FitRepair 先从包含缺陷的项目代码库中提取函数源代码，并应用掩码区间预测目标（Masked Span Prediction）训练 CodeT5 模型。通常，掩码区

间预测目标只会遮盖原始代码标记中的一小部分（如 15%），但是有研究发现，通过增加掩码区间预测目标的预训练遮盖率，可以进一步提升大模型在下游任务上的性能，因此 FitRepair 在第一阶段采用了高达 50%的遮盖率，以便训练模型根据更少的上下文信息恢复更多被遮盖的代码片段，从而强制模型在有限的上下文信息中学习更多项目特定的知识。在这一阶段，FitRepair 利用知识增强微调来生成更多使用项目特定变量、方法调用和结构的代码。然而，无论是知识增强微调的 CodeT5 模型还是原始的 CodeT5 模型都存在相同的局限性，即其训练过程并非专门为修复而设计。原始的预训练和知识增强微调都使用掩码区间预测，该方法会遮盖多个不连续的代码标记范围。模型在训练期间的目标是，恢复所有被遮盖范围的原始标记。然而，填空式的补丁生成方法通常只会遮盖单个代码行或代码行的一部分，所以大模型只需要预测该单一范围的正确代码。为了缓解这个问题，FitRepair 提出了第二种策略。

第二种是面向修复的微调策略。FitRepair 使用原始缺陷项目来构建一个面向修复的数据集，数据集中的每个训练样本只遮盖了一个连续的代码序列，这使得经过微调的模型更加适合执行修复任务，即大模型只需要生成一个连续的代码序列。具体而言，FitRepair 先使用原始缺陷项目作为训练数据源。然后，在每个训练样本中，FitRepair 选择一行代码来进行遮盖，并使用单一标记遮盖这行代码。选定的代码行可被视为最终修复场景中的缺陷代码行。为了模拟基于模板的修复输入，FitRepair 随机选择一个修复模板（具体模板可以参考上文关于 AlphaRepair 的三大类掩码模板）。在这种微调策略中，FitRepair 的训练样本在设置上与模型执行的填空式自动程序修复任务非常相似。

上述两种微调策略的目标是，微调大模型，以便生成更多与项目特定、面向修复任务的补丁。这两种策略都使用了原始缺陷项目作为微调的数据集。然而，对于项目中特定的缺陷，其相关的修复代码元素（Fixing Ingredients）可能会因缺陷文件、缺陷位置和缺陷代码行的类型而大不相同。这些修复代码元素可能与当前缺陷位置相距较远，但大模型的输入上下文长度通常有限（如 CodeT5 的上下文窗口大小限制为 512 个 Token），因此无法将大规模的代码上下文全部作为大模型的输入。为此，FitRepair 提出一种新颖的提示策略，即相关标识符提示，旨在通过获取一系列与大模型以前未见过的相关/稀有的项目标识符来增强大模型生成正确补丁的能力。

具体而言，给定缺陷代码行信息，首先，FitRepair 提取包含缺陷的特定文件，并将其分割为各个代码行。由于修复缺陷所需的正确代码元素的很大一部分可以在同一文件中找到，因此 FitRepair 使用 Levenshtein 距离比率来衡量每行代码与缺陷代码行之间的相似度。假设有用的标识符可以从与缺陷代码行非常相似的行中获得，那么 FitRepair 根据字符串相似度分数从高到低的顺序对每行代码进行排名，就可以获取一份代码行的排名列表。FitRepair 进而从每行代码中提取标识符，获得一个带有排名的标识符列表。其次，FitRepair 进行过滤，先删除任何常见/简单的标识符，然后使用静态分析来删除在缺陷方法内无法访问的标识符。再次，FitRepair 为每个标识符提取有用的类型信息，以指示类型/返回类型以及它是否是方法调用或变量。最后，FitRepair 获得一个排名列表，其中包含来自同一文件中相似代码行的复杂标识符。根据排名列表，FitRepair 生成提示并指示大模型使用这些提取的标识符来生成

补丁，例如，一条“/* 在下一行使用 {} */”的提示，{}是具体的修复代码元素。这个提示会附加在被遮盖的标记之前，使模型可以直接在生成补丁中使用其所提供的这些标识符信息。

基于大模型的对话式补丁生成方法。经过指令微调的大模型在各种下游任务中具有良好的泛化能力，因此大模型被广泛应用在对话系统（如 ChatGPT）中。此外，大模型也展现了一定的自我批判和自我改进的能力，在对话上下文中可对问题的回答进行进一步精化。基于此，基于大模型的对话式补丁生成方法被提出。该方法以多轮次对话的形式，不断让大模型根据上一轮生成补丁的执行结果（测试用例执行情况）进一步生成质量更高的补丁。与单轮次的补丁生成方法相比，多轮次的补丁生成方法可以有效地利用补丁的动态执行信息作为反馈，从而进一步激发大模型生成高质量补丁的能力。这种对话式补丁生成方法是一种可拓展性比较高的方法，因此，更多相关的静态分析信息，甚至开发人员的反馈信息也可以加入迭代式过程中。

ChatRepair[20]是基于 ChatGPT 的对话式补丁生成方法，它整合了多维度的反馈信息从而迭代地向大模型提出查询并生成补丁。与现有直接利用缺陷代码的基于大模型的补丁生成技术不同，ChatRepair 还考虑了有价值的测试失败信息，以在生成补丁时进一步协助大模型。此外，与基于大模型的补丁生成技术连续从相同提示中采样不同，ChatRepair 跟踪对话历史并通过提示从以前失败和成功的修复尝试中学到更多知识。通过这种方式，ChatRepair 既可以避免以前的失败修复，又可以建立在以前的成功修复上，以实现更高效的补丁生成方法。

具体而言，ChatRepair 的输入包括初始提示、测试失败原始信息、测试用例集合、大模型（如 GPT-4）、可信的修复程序生成提示，以及最大对话长度和最大尝试次数两个超参数；其最终的输出包括一个可信的候选补丁列表以及 ChatGPT API 访问的总成本。最大尝试次数是 ChatRepair 的一个停止准则，如果其尝试修复一个缺陷所使用的最大尝试次数（对 ChatGPT 的查询次数）已经用完，那么修复过程就会停止，该限制是为了控制调用大模型的成本（考虑到大模型的性能不可控和随机）；而最大对话长度则进一步限制了生成补丁时涉及的历史最大量文本。ChatRepair 在缺陷修复数据集 Defects4J 上成功修复了 114 个缺陷，不仅超过了基本的 ChatGPT 的修复能力（80 个），而且超过了其他基于大模型的补丁生成方法。

基于大模型的补丁检查。尽管基于测试的补丁验证已被证明在实际系统中是可行的，但它存在测试过拟合问题，即通过所有测试的补丁不一定总是在语义上等同于相应的开发人员写的正确补丁。这是因为软件测试很难覆盖实际系统所有可能的程序行为。因此，在实际的缺陷修复过程中，仍然需要开发人员手动检查所生成的似真补丁，以找到最终正确的补丁。然而，鉴于潜在的似真补丁数量众多以及实际系统代码的复杂性，人工手动检查非常具有挑战性且会耗费大量时间。为减轻开发人员的负担，研究人员提出了多种技术来自动地检查补丁的正确性。传统的补丁正确性检查方法包括基于静态特性和规则的方法、基于动态信息分析的方法。随着深度学习的发展，研究人员也提出了数据驱动的补丁正确性检查方法。其中，基于大模型的补丁正确性检查方法展现出了较大的潜力和发展前景。目前，该方法主要利用

大模型对候选补丁进行代码向量表示或对候选补丁的生成概率进行计算，并基于此判定似真补丁和正确补丁。

Tian 等人[21]提出了基于嵌入表示（Embedding）的补丁正确性检查方法。该方法使用预训练语言模型 CodeBERT 对每个候选补丁进行向量表示，并在历史的正确和错误补丁数据集上训练出了一个二分类器。基于嵌入表示的补丁正确性检查方法是一个较为通用的框架，可以与不同的大模型相结合（利用不同大模型所返回的嵌入表示）。此外，可利用 AlphaRepair[18]为每个候选补丁生成一个准确的分数，以便更有效地对候选补丁进行排名。对于每个候选补丁，AlphaRepair 遮盖其中每个 Token，然后查询 CodeBERT 以获取该标记的条件概率。对于所有其他先前遮盖的 Token 位置应用相同的过程计算联合分数。联合分数可以理解为生成序列的条件概率（在给定前后上下文的情况下，根据 CodeBERT 生成该补丁的可能性是多少）。由于 AlphaRepair 考虑了多种掩码策略（完整掩码、部分掩码和模板掩码），因此在计算候选补丁联合分数时，AlphaRepair 会将上述联合分数除以待生成的 Token 数量，以便考虑 Token 数量差异为联合分数带来的偏差。

3.4 缺陷案例挖掘与分析

在企业开发实践中，软件缺陷案例的收集和整理是从历史中总结经验的重要途径。缺陷案例的挖掘与分析为企业开展案例化的缺陷管理、积累缺陷修复经验、防止类似缺陷的反复引入等提供了一种新的技术思路。广义上，“缺陷库”是系统化的企业缺陷表示、案例收集和管理机制，而并非局限于 Bugzilla 式的缺陷追踪系统。缺陷案例挖掘与分析不仅包括对缺陷本身的描述和状态跟踪，还包括与缺陷引入和修复相关的代码分析与整理。

3.4.1 缺陷库概述

企业级缺陷库是软件缺陷案例的集合，这些缺陷案例可能来自不同的项目、不同的开发阶段或不同的开发团队。这些缺陷案例通常包括缺陷描述、缺陷严重性、缺陷发现和修复时间、缺陷原因、缺陷相关的测试用例、缺陷修复代码块、缺陷引入代码块等信息。缺陷库的主要目的是提供一个丰富的、结构化的数据源，以便研究人员和开发人员能够从中挖掘有价值的信息，进而改进软件的质量和开发过程。例如，一个包含了缺陷引入代码块和缺陷修复代码块的缺陷库，可以用于挖掘缺陷模式及修复模式，以便在项目开发、发布阶段进行实时的缺陷检测和缺陷自动修复；一个包含测试用例的缺陷库，可以用于挖掘测试用例生成和回归测试选择的经验、知识，以便在测试阶段及时发现隐藏的缺陷；一个包含了缺陷发现和修复时间的缺陷库，可用于挖掘缺陷生命周期的管理知识，有助于管理者对相关项目策略的制定；一个包含了缺陷原因的缺陷库，对于理解和分析软件开发过程中的常见缺陷问题特别有

价值，通过分析缺陷原因，开发团队可以更深入地理解造成缺陷的根本原因，从而采取预防措施，减少未来项目中类似缺陷的发生。

缺陷库不仅是一个静态的数据集合，还是一个不断更新和演进的系统，随着新的缺陷被发现和修复，缺陷库会不断扩充和更新。大规模的缺陷库还常被用于机器学习和数据挖掘等领域，以训练模型识别缺陷模式，提高软件质量保证的自动化水平。

在学术界，研究人员通过收集开源项目的真实缺陷及其修复模式构建了多个缺陷库，如 Siemens Benchmark[22]、Corebench[23]、BugJS[24]、DbgBench[25]、Defects4J[26]等。这些缺陷库中的缺陷大多来自开源项目，而来自企业项目的缺陷相对较少，缺乏企业真实缺陷案例的代表性。此外，这些缺陷库需要人工参与构建，很大程度上限制了缺陷库的规模，导致我们难以采用数据驱动的方法挖掘缺陷发生和修复的深层规律。为了能够构建更大规模的缺陷库，提升缺陷挖掘的效率，业界主要采用两种缺陷案例收集方式。

- **基于提交消息分析的缺陷案例收集**。在代码库版本控制系统中，利用提交消息中的特定关键字（如“fix”“repair”）或模式（如以“BUG:”开头，关联缺陷追踪系统中的缺陷编号），从历史代码提交中获取与缺陷修复相关的代码提交（Bug-Fixing Commit，BFC），并将 BFC 中修改前和修改后的代码片段分别作为缺陷代码片段和修复代码片段。
- **基于自动化测试的缺陷案例收集**。通过提交消息中的特定关键字或模式过滤历史代码提交中的 BFC，然后通过执行 BFC 修改前后的代码来确定该 BFC 的真实性，即经过该 BFC 对代码的修改，是否使修改前失败的测试用例在修改后通过。由此，通过动态和静态分析技术定位代码中的缺陷代码段和修复代码段。

基于提交消息分析的缺陷案例收集是传统的缺陷库构建方法，该方法不仅无法确认 BFC 的真实性，而且 BFC 中可能存在大量的噪声数据（与修复无关的修改，如重构、新增功能等），导致基于该数据集进行数据驱动的下游任务模型在实践中表现不佳。因此，本书将主要介绍基于自动化测试的缺陷案例收集。

3.4.2　基于自动化测试的缺陷案例收集

基于自动化测试的缺陷案例收集的核心思想是，通过执行测试用例并根据测试结果来验证 BFC 的真实性。该方法将基于提交消息分析收集到的 BFC 作为潜在 BFC（Potential Bug-Fixing Commit，PBFC），即该代码提交可能修复了一个缺陷。为了验证修复缺陷的真实性，需要在 BFC 前后版本中执行与缺陷相关的测试用例。图 3-21 展示了基于自动化测试的缺陷案例收集的基本流程，主要包括以下五个步骤。

- **步骤 1：查找潜在 BFC**。在版本控制系统中，根据代码提交消息，通过关键字匹配、模式匹配和内容分析等方式，查找可能修复缺陷的提交，即潜在 BFC（PBFC）。
- **步骤 2：查找与修复缺陷相关的测试用例**。修复缺陷后，开发人员会创建新的测试用例来验证修复代码是否有效。因此，需要找到能验证缺陷被修复的测试用例。

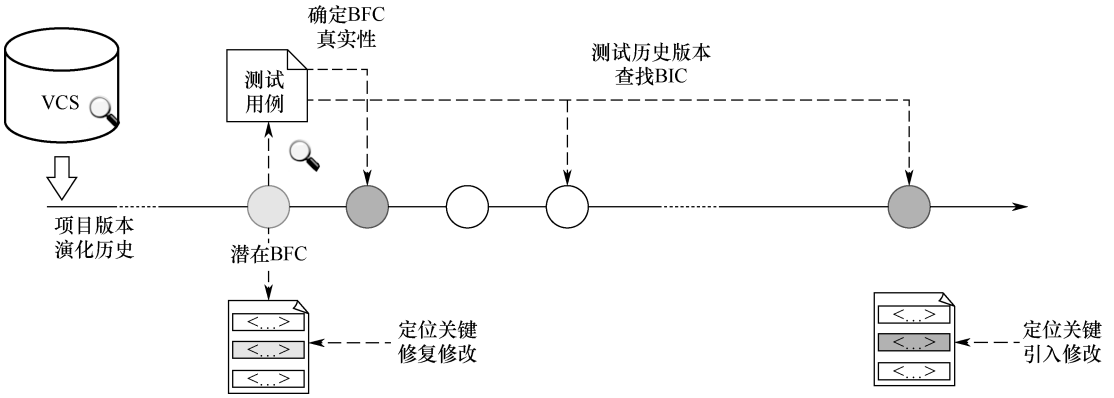


图 3-21　基于自动化测试的缺陷案例收集流程

- **步骤 3：确定 BFC 的真实性。**通常，根据测试用例在潜在 BFC 提交前后的测试结果，可以确定 BFC 的真实性。若在 BFC 提交之后，代码快照版本能够通过测试，但在 BFC 提交之前，代码快照版本不能通过测试，则认为该 BFC 是真实的。
- **步骤 4：寻找缺陷引入的提交**（Bug-Inducing Commit，BIC）。将测试用例向代码演化的早期版本迁移并执行测试用例。由于在缺陷修复前，代码是存在该缺陷的，因此，相应的测试用例都会失败。基于此，将该测试用例向更早的版本进行迁移，测试用例也应当失败，直到找到一个版本使得测试通过或者直到不存在被测试的代码为止。这个步骤相当于带着测试用例在项目版本演化历史中“旅行”，旅行的终点就是测试通过或被测代码不存在，从而也就证明了在该版本之后引入了该缺陷提交，即找到了 BIC。
- **步骤 5：定位关键修改。**由于一次代码提交中可能混杂着不同目的的代码修改，因此，在收集缺陷案例时，需要从该次提交的所有代码修改中定位出一个修复或引入缺陷的最小修改集合，以进一步用于缺陷相关知识的整理。在此步骤中，可以通过动态分析和静态分析结合来确定 BFC 和 BIC 中的关键修改，即缺陷案例的缺陷修复代码块和缺陷引入代码块。

上述步骤中的技术难点主要体现在以下三方面。

第一，测试用例搜索。为潜在 BFC 搜索测试用例的难点在于，如何确定测试用例与缺陷的相关性。首先，需要在代码中找到可能与缺陷相关的测试用例候选集。通常有四种不同方式来确定与缺陷相关的测试用例候选集[27]：

- 在潜在 BFC 中修改或者增加的测试用例。这些测试用例随着缺陷修改的代码一起提交，因此与缺陷通常具有较高的相关性。
- 在潜在 BFC 后 *n* 次代码提交中修改或增加的测试用例。这些测试用例可能是开发人员为复现和修复缺陷所补充的测试用例，因此也与缺陷有一定的相关性。
- 潜在 BFC 关联缺陷报告的文本中提及的提交所增加或者修改的测试用例。这些测试用例由于在缺陷描述中被明确提到，因此与缺陷有一定的相关性。
- 潜在 BFC 前后项目版本中都存在的测试用例。这些测试用例在缺陷修复前后都存在，因此也可能与缺陷存在联系。

这四种方式涉及的测试用例范围逐步扩大，会导致后续的测试成本也逐渐增加。在实际使用时，需要根据性能、准确性等实际情况，选择合适的测试用例候选集确定方式。

其次，在确定测试用例候选集后，可以通过执行测试用例来确定 BFC 的真实性。具体而言，对选定的测试用例，在潜在 BFC 前的版本和潜在 BFC 后的版本中分别执行测试用例，若在潜在 BFC 前的版本中测试失败但在潜在 BFC 后的版本中测试成功，则表明该测试用例确实验证了缺陷的修复动作，即确认了该 BFC 的真实性且测试用例与该缺陷相关。若没有满足上述条件的测试用例，则无法确认 BFC 的真实性，那么该潜在 BFC 应被舍弃。

第二，BIC 搜索。在确认了 BFC 及其所关联的测试用例后，为了更完整地记录该缺陷的引入情况，需要以 BFC 为起点，在更早的代码提交中搜索 BIC。该过程的难点是，如何降低搜索成本以及如何在项目版本演化历史中向更早的版本迁移测试用例。

一方面，为了降低搜索成本，需要尽快找到测试成功和测试失败的“分界线”。以代码版本管理工具 Git 为例，它提供了一种基于测试反馈的二分查找法 git-bisect，其能够自动以二分法选择历史提交。如果历史提交中的一个版本测试失败，则朝更远的历史继续进行二分查找；若测试通过，则朝更近的历史继续进行二分查找，直到找到一个版本测试成功且其紧接的下一个版本测试失败为止。在这种情况下，可以确定这次提交为真实的 BIC 并意味着缺陷由回归查找的结果引入。

在实际应用中，可能发生某个版本无法编译的情况。对此，现有研究[28]考虑到不可编译版本的连续性，使用指数振荡跳跃方法确定项目版本演化历史上无法编译、无法测试的版本区间边界，并根据边界进一步调整搜索范围。图 3-22 展示了该方法的一个示例。在该示例中，二分查找法发现在历史提交版本 V 上无法编译，则分别向更远（“−”）和更近（“+”）的方向以 2 的指数为步长跳跃，寻找一个可以编译的历史提交版本，并确定无法编译版本的区域边界。找到后，该方法将往反方向继续跳跃以缩小边界范围。最终在项目版本演化历史上不断来回振荡，确定最小的无法编译区间范围。该方法将演化历史分割为多个可编译区间，并在这些可编译区间上继续查找。

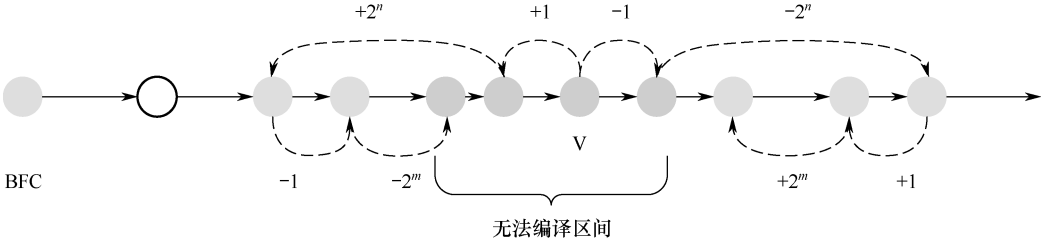


图 3-22　指数振荡跳跃方法示例

该方法最终会返回一个测试失败版本，而在这之前的项目版本，都无法编译。这种情况意味着缺陷可能由某个首次添加的功能特性引入，即 BIC 同时也是特性引入提交（Feature-Introduction Commit，FIC）。为了判定 BIC 是否与 FIC 相同，该方法进一步使用基于 TF-IDF 的信息检索变体技术及静态分析技术，分析 BFC 中被测试的核心代码，并追溯其

提交前是否存在，以做最终判定。

另一方面，为了实现向更早的版本迁移测试用例，需要相应的测试用例迁移技术。该技术主要包括两个核心步骤：一是确定需要迁移的测试用例所依赖的代码元素；二是调整迁移后的代码，以适应历史项目版本的代码上下文。在确定需要迁移的测试用例所依赖的代码元素时，首先在 BFC 后的项目版本中为测试用例构建函数调用图，并记录调用函数的集合，执行测试用例，并记录测试用例所覆盖的代码元素集合，再求交集。然后，使用代码差异分析技术（如 CLDiff[29]）将交集中的代码元素与历史上一个给定的待测试版本 Vt 进行差异分析，标记相对 Vt 增加或者修改的代码元素。最后，将修复修改外的代码元素增量迁移到 Vt 并尝试编译。在迁移过程中，如果发生编译失败，那么需要使用基于规则的代码转换技术，将迁移代码转换为适配历史项目版本的上下文（如 JDK、API 版本）的代码。例如，对于 Java 代码而言，不同的 JDK 版本可能造成无法编译的问题，因此需要对相应 JDK 版本不支持的语法结构进行改写，如将 JDK 8 中 Lambda 表达式中的迭代写法转换为 JDK 7 中的普通 For 循环。

第三，定位关键修改（步骤 5）。定位关键修改需要排除 BFC 和 BIC 中与缺陷修复或引入无关的修改，并保证 BFC 和 BIC 依然可以修复和引入缺陷，其通常包含以下过程：

- 使用静态分析技术（如 RefactoringMiner[30]）检测并排除 BFC 和 BIC 中的代码重构修改。
- 排除不会被测试覆盖的代码修改。
- 检测剩余代码修改中的依赖关系，对修改的代码根据依赖关系进行分组。
- 对剩余代码修改使用 Delta-Debugging[31]技术，定位最小化的缺陷引入和修复修改。
- 对无法使用 Delta-Debugging 技术的案例（BIC=FIC 的案例），在 BIC 后的项目版本中使用数据流、控制流分析技术定位直接导致测试失败的代码修改。

3.4.3 缺陷案例分类与利用

缺陷案例收集的目的是进一步对历史上产生的缺陷及其修复进行分析，进而从历史上总结缺陷引入的提交并吸取缺陷修复的经验。为此，需要对缺陷库中的缺陷案例进行整理，尝试挖掘缺陷模式及修复模式，从而帮助我们提升缺陷检测和修复代码的能力。缺陷模式和修复模式挖掘通常包含以下三个步骤：构建缺陷-修复对（Bug-Fix Pair，BFP）、度量 BFP 相似度及聚类、模式识别及提取。

构建缺陷-修复对。BFP 由缺陷库中每个缺陷案例的缺陷代码块及修复代码块组成。由于一般的缺陷库通常不具备 BIC，所以常用的方法是将缺陷案例中 BFC 前后的代码分别作为缺陷代码块和修复代码块。对于具备 BIC 的缺陷库，可以将 BIC 后的代码作为缺陷代码块。使用 BIC 构建的 BFP 通常具有更高的准确性；而仅使用 BFC 构建的 BFP 比较容易获得，但只适用于简单的缺陷模式，若缺陷引入与缺陷修复之间存在一定的时间差，则这类 BFP 将无法被构建。

度量 BFP 相似度及聚类。现有方法通常通过以下特征来度量 BFP 相似度：（1）代码文本相似度，即直接考虑 BFP 代码文本之间的相似度；（2）语法结构相似度，如 BFP 代码的抽象语法树、数据流图之间的相似度；（3）API 使用序列相似度，即 BFP 中的 API 调用序列之间的相似度。这种 BFP 之间的相似度通常使用 Jaccard 相似系数、Levenshtein 距离等指标来度量。根据相似度度量结果，采用 *k*-means、层次聚类、DBSCAN 等聚类算法对相似的 BFP 进行聚类，从而识别出同类的缺陷及其修复案例。

模式识别及提取。针对聚类后的每个簇，通常需要结合静态分析和字符串匹配算法来挖掘类簇中具有共性的代码模式。例如，在阿里的实践[32]中，首先使用静态分析技术将代码文本序列分别标记为“符号”（Symbols）和“参数”（Parameters）。其中，“符号”表示与上下文无关的代码文本序列，而“参数”表示与特定上下文相关的代码文本序列（如变量名）。然后，使用递归最长公共子串（Recursive Longest Common Substring，RLCS）算法，检测 BFP 之间共同的代码序列，以此作为缺陷-修复模式。图 3-23 中展示了缺陷-修复模式提取的示例。其中，Patch1 和 Patch2 为同一簇中待提取模式的两个修复代码片段。sb、builder、token 等变量名被标记为“参数”，其他代码文本则被标记为“符号”。首先使用 RLCS 算法（步骤①）检测参数以外两段代码的最长公共子串；然后通过保留两段代码的公共部分（步骤②）将“参数”替换为可根据代码上下文选择的“参数”标记，便完成了缺陷修复模式的提取。

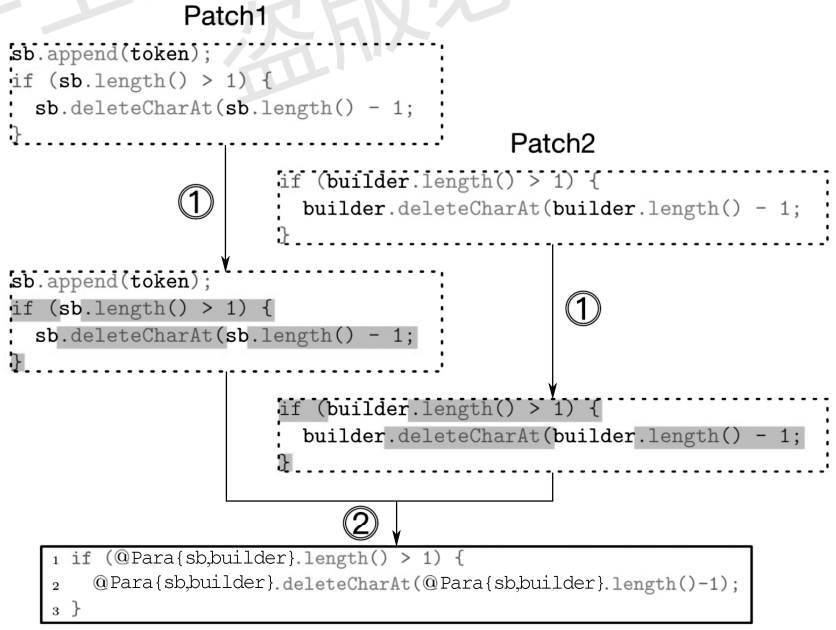


图 3-23　缺陷-修复模式提取的示例

缺陷模式可以通过同样的方式来挖掘。当有开发人员提交代码时，就可以使用缺陷模式库检测提交代码中是否存在缺陷，并根据缺陷模式推荐关联的修复模式。

3.5 小结

在软件项目中，不可避免地存在着缺陷。因此，我们需要了解缺陷分析的相关技术，包括基于静态分析的缺陷分析、基于深度学习的缺陷分析、基于大模型的缺陷分析，从而准确地定位、检测和修复软件项目中的缺陷。此外，随着缺陷库的有效构建，我们需要了解基于缺陷库的缺陷案例挖掘与分析，以帮助企业开展案例化缺陷管理，积累缺陷–修复经验，防止类似缺陷的反复引入。

参 考 文 献

[1] COUSOT P, COUSOT R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints[C]//Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1977: 238-252.

[2] COUSOT P, HALBWACHS N. Automatic discovery of linear restraints among variables of a program[C]//Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1978: 84-96.

[3] COUSOT P. Principles of Abstract Interpretation[M]. MIT Press, 2021.

[4] 张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. 软件学报, 2018, 30(1): 80-109.

[5] GIACOBAZZI R, RANZATO F. History of abstract interpretation[J]. IEEE Annals of the History of Computing, 2021, 44(2): 33-43.

[6] 陈立前, 范广生, 尹帮虎, 等. 抽象解释及其应用研究进展[J]. 计算机研究与发展, 2023, 60(2): 227-247.

[7] SCOTT D S, STRACHEY C. Toward a mathematical semantics for computer languages[M]. Oxford: Oxford University Computing Laboratory, Programming Research Group, 1971.

[8] STOLTENBERG-HANSEN V, LINDSTRÖM I, GRIFFOR E R. Mathematical theory of domains[M]. Cambridge University Press, 1994.

[9] BRUNI R, GIACOBAZZI R, GORI R, et al. Abstract extensionality: on the properties of incomplete abstract interpretations[J]. Proceedings of the ACM on Programming Languages, 2019, 4(POPL): 1-28.

[10] NIELSON F, NIELSON H R, HANKIN C. Principles of program analysis[M]. Springer, 2010.

[11] PRADEL M, SEN K. Deepbugs: A learning approach to name-based bug detection[J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA): 1-25.

[12] LI Y, WANG S, NGUYEN T N, et al. Improving bug detection via context-based code representation learning and attention-based neural networks[J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-30.

[13] ZHOU Y, LIU S, SIOW J, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks[C]//Proceedings of the 33rd International Conference on Neural Information Processing Systems. 2019: 10197-10207.

[14] LI R, CHEN B, ZHANG F, et al. Detecting runtime exceptions by deep code representation learning with attention-based graph neural networks[C]//Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. 2022: 373-384.

[15] ZHANG F, CHEN B, ZHAO Y, et al. Slice-Based Code Change Representation Learning[C]//Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. 2023: 319-330.

[16] YANG A Z H, MARTINS R, GOUES C L, et al. Large Language Models for Test-Free Fault Localization[J/OL]. arXiv preprint arXiv:2310.01726, 2023.

[17] WANG C, LIU J, PENG X, et al. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference[J/OL]. arXiv preprint arXiv:2311.04448, 2023.

[18] XIA C S, ZHANG L. Less training, more repairing please: revisiting automated program repair via zero-shot learning[C]//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022: 959-971.

[19] XIA C S, DING Y, ZHANG L. Revisiting the Plastic Surgery Hypothesis via Large Language Models[J/OL]. arXiv preprint arXiv:2303.10494, 2023.

[20] XIA C S, ZHANG L. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT[J/OL]. arXiv preprint arXiv:2304.00385, 2023.

[21] TIAN H, LIU K, KABORÉ A K, et al. Evaluating representation learning of code changes for predicting patch correctness in program repair[C]//Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 2020: 981-992.

[22] HUTCHINS M, FOSTER H, GORADIA T, et al. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria[C]//Proceedings of 16th International Conference on Software Engineering. IEEE, 1994: 191-200.

[23] BÖHME M, ROYCHOUDHURY A. Corebench: Studying complexity of regression errors[C]//Proceedings of the 2014 International Symposium on Software Testing and Analysis. 2014: 105-115.

[24] GYIMESI P, VANCSICS B, STOCCO A, et al. Bugsjs: a Benchmark of JavasCript Bugs[C]//Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification. 2019: 90-101.

[25] BÖHME M, SOREMEKUN E O, CHATTOPADHYAY S, et al. Where is the bug and how is it fixed? an experiment with practitioners[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 117-128.

[26] JUST R, JALALI D, ERNST M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]//Proceedings of the 2014 international Symposium on Software Testing and Analysis. 2014: 437-440.

[27] SONG X, WU Y, CAO J, et al. BugMiner: Automating Precise Bug Dataset Construction by Code Evolution History Mining[C]//Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering. 2023: 1919-1929.

[28] SONG X, LIN Y, NG S H, et al. RegMiner: towards constructing a large regression dataset from code evolution history[C]//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing

and Analysis. 2022: 314-326.

[29] HUANG K, CHEN B, PENG X, et al. Cldiff: Generating Concise Linked Code Differences[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018: 679-690.

[30] TSANTALIS N, MANSOURI M, ESHKEVARI L M, et al. Accurate and efficient refactoring detection in commit history[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 483-494.

[31] MISHERGHI G, SU Z. HDD: hierarchical delta debugging[C]//Proceedings of the 28th International Conference on Software Engineering. 2006: 142-151.

[32] ZHANG X, ZHU C, LI Y, et al. Precfix: Large-scale patch recommendation by mining defect-patch pairs[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. 2020: 41-50.