# 综述论文翻译

## 摘要

软件故障定位是识别程序中故障位置的行为，被广泛认为是程序调试中最繁琐，耗时且昂贵但同样重要的活动之一。 由于当今软件规模越来越大，越来越复杂，发生故障时手动定位故障正在迅速变得不可行，因此，强烈要求软件开发人员能够以最少的人为干预指导程序中的故障位置。 这种需求反过来又促进了广泛的故障定位技术的建议和发展，每种故障定位技术都旨在简化故障定位过程，并通过以独特方式攻击问题使故障定位过程更加有效。在本文中，我们编录并提供了这些技术的全面概述，并讨论了与整个软件故障本地化相关的关键问题和疑虑。

## 1.介绍

软件是我们今天生活的基础，随着其使用和采用的不断增加，其影响力几乎无处不在。事实上，目前，软件不仅仅适用于医疗，航空和核能等行业中的许多安全和关键安全系统，而且也是至关重要的。毫不奇怪，这种趋势一直伴随着（如果不能仅仅因为）软件的规模和复杂性的急剧增加。不幸的是，这也导致了更多的**软件错误**，这往往导致执行失败，造成巨大的损失[260,275,365]。此外，安全关键型系统中的软件故障不仅会造成财务损失，而且会造成生命损失，这是令人担忧的前景[368]。美国国家标准与技术研究院（NIST）2006年的一份报告[304]指出，软件错误估计会使美国经济每年损失595亿美元（占GDP的0.6％）;从那时起，成本无疑增长了。修复或响应这些错误的成本超过一半传递给软件用户，而软件开发人员和供应商则吸收其余部分。

即使由于错误行为或故障的某些其他表现而发现软件故障，找到并修复它们也是完全不同的事情。故障定位主要关注前者，即识别故障的位置，在历史上一直是一项人工任务，由于其大型化和复杂性，已被认为是耗时且乏味且昂贵的[347]如今的大型软件系统。此外，手动故障定位在很大程度上**依赖于软件开发人员的经验**，判断和直觉来识别可能存在错误的代码并确定优先级。这些限制引起了人们对开发可部分或全部自动化软件故障本地化同时减少人力投入的技术的兴趣。虽然一些技术是相似的并且有些非常不同（就数据消耗的类型，程序组件的重点，比较效率和效率等而言），但它们都试图从独特的角度来攻击故障定位问题，并且通常相对于彼此提供优点和缺点。随着许多技术已经存在，其他技术不断被提出，以及从理论和实践两方面都取得了进展，重要的是要编目和概述当前最先进的故障定位技术，以便提供为那些已经在这个地区的人们提供了一个全面的资源，并且他们有兴趣为它做出贡献。

为了提供涵盖自1970年代后期以来与软件故障定位相关的大多数出版物的完整调查，我们创建了一个出版库，其中包括**从1977年到2014年11月发表的331篇论文**。我们还搜索了硕士和博士学位。 这些问题与软件故障定位密切相关，如表1所列。

\*

表一

我们库2中的所有论文都按年份排序，结果如图1所示。如图所示，2001年以后出版物数量迅速增长，这表明越来越多的研究人员开始致力于软件故障领域 过去十年的本地化。

图二

另外，根据我们的知识库，图2给出了与软件故障定位相关的出版物数量，这些出版物出现在高质量和领先的期刊和会议上，重点放在软件工程—IEEE Transactions on Software Engineering (TSE)，ACM Transactions on Software Engineering and Methodology(TOSEM)， International Conference on Software Engineering (ICSE)，ACM软件工程基础国际研讨会和ACM自动化软件工程国际会议 - 从2001年到2014年11月。这一趋势再次支持软件故障定位不仅仅是一个重要而且也是一个热门研究 主题，并在过去十年中一直在高质量的软件工程期刊和会议上进行过深入的讨论。

因此，关于各种技术的文献丰富，旨在促进故障定位并使其更为有效.3尽管这些技术具有类似的目标，但它们可能彼此非常不同，并且通常源于源于几个不同的学科。尽管我们的目标是尽可能全面地涵盖尽可能多的故障定位技术，但不论广度还是深度，都不能涵盖所有这些文章。在这次调查中，我们主要关注的是定位**Bohrbugs**的技术[139]。那些用于诊断Mandelbugs [139]，如性能错误，内存泄漏，软件膨胀和安全漏洞不包括在范围内。此外，由于空间限制，我们将技术分组到适当的类别进行集体讨论，并强调最重要的特征，并将这些技术的其他细节留给他们分别发表的论文。针对特定应用领域的技术尤其如此，例如并发错误和电子表格的故障定位。对于这些，我们提供了一个帮助读者理解的综述。

以下术语在本文中重复出现，因此为了方便起见，我们根据[37]中提供的分类法为其提供定义：

 •failure:服务偏离正确行为时。

 •error:是系统中可能导致故障的一种情况。

 •fault:是导致错误的根本原因，也称为bug。

本文的其余部分按以下方式组织：我们首先描述第2节中的传统和直观的故障定位技术，第3节转向更先进和复杂的技术。第4节中列出了一些热门的主题节目已用于不同的案例研究并讨论这些计划多年来如何演变。**第5节描述了评估故障定位技术的有效性的不同评估指标，随后在第6章中讨论了故障定位工具。最后，分别在第7节和第8节中给出了关键方面和结论。**

## 2. 传统和直观的故障定位技术

本节介绍传统和直观的故障定位技术，包括**程序日志记录，断言，断点和分析。**

2.1程序记录

用于产生程序记录的语句（例如打印）通常以特别的方式插入到代码中以监视变量值和其他程序状态信息[105]。 当检测到异常程序行为时，开发人员根据保存的日志文件或打印的运行时信息检查程序日志，以诊断潜在的故障原因。

2.2断言

断言是在程序正确运行期间添加到程序中的约束条件。 开发人员在程序代码中将这些断言指定为条件语句，如果它们评估为false，则会终止执行。 因此，它们可用于在运行时检测错误的程序行为。 有关使用断言进行程序调试的更多细节可以在[309,310]中找到。

2.3断点

断点用于在执行到达指定点时暂停程序，并允许用户检查当前状态。 触发断点后，用户可以修改变量的值或继续执行以观察错误的发展。 数据断点可以配置为在指定表达式的值发生变化时触发，例如变量值的组合。 只有满足用户指定的谓词时，条件断点才会暂停执行。 早期的研究（例如[80,155]）使用这种方法来帮助开发人员在符号调试器的控制下执行程序时定位错误。 更高级的调试工具，如GNU GDB [121]和Microsoft Visual Studio Debugger [255]也采用了相同的方法。

2.4分析

性能分析是指标（如执行速度和内存使用率）的运行时分析，通常针对程序优化。 但是，它也可以用于调试活动，如下所示：

  •检测不同功能的意外执行频率（例如，[43]）;

  •识别内存泄漏或意外执行不良的代码（例如，[150]）;

  •检查懒惰评估的副作用（例如，[313]）。

使用分析程序调试的工具包括GNU的gprof [120]和Eclipse插件TPTP [108]。4

## 3.先进的故障定位技术

随着当今软件系统的规模和规模的扩大，传统的故障定位技术在隔离故障的根源方面效果不佳。因此，最近出现了许多先进的故障定位技术，它们使用因果关系[215,288]，它与哲学理论有关，目的是表征事件/原因（本例中的程序错误）和现象/影响之间的关系（在我们的例子中执行失败）。有不同的因果关系模型[288]，如基于反事实的，基于概率的或基于统计的，以及因果演算模型。其中，概率因果模型是故障定位中应用最广泛的一种，用于识别造成执行失败的可疑代码。

在本次调查中，我们将故障定位技术分为8类，包括基于切片的，基于频谱的，基于统计的，基于程序状态的，基于机器学习的，基于数据挖掘的，基于模型的技术和其他技术。已经有许多研究评估特定断层定位技术的有效性。然而，他们都没有提供关于所有这些技术的全面讨论。

### 3.1。切片技术

程序切片是**一种通过删除不相关的部分来将程序抽象为简化形式的技术，从而使得所产生的切片在某些规范方面仍将与原始程序相同**。自从Weiser在1979年首次提出静态切片以来，已经发表了数百篇关于这个主题的论文[52,344,394] [361]。

静态切片[360]的一个重要应用**是在程序员找到程序中的错误时减少搜索域**。这是基于这样的想法，即如果测试用例由于语句中的变量值不正确而失败，则应该在与该变量语句对相关联的静态片中找到缺陷，从而允许我们将搜索限制在切片中比看整个节目。 Lyle和Weiser通过构建一个程序模块（作为两组静态切片的集合差异）来扩展上述方法，以**进一步减少可能出现故障位置的搜索域**[235]。尽管基于静态切片的技术已经过实验评估并被证实对故障定位有用[207]，但一个问题是，**处理指针变量可能导致数据流分析效率低下，因为大量的数据事实是通过指针变量的解引用引入的需要存储。**用于识别程序访问的各种存储器位置之间的等价关系的等价分析用于提高存在指针变量时数据流分析的效率[220]。两个等价的内存位置在一个过程中共享相同的数据事实集合。因此，数据流分析只需要计算代表性存储位置的信息，其他同等位置的数据流可以从代表位置获得。静态分片也可应用于二进制可执行文件[192]和类型检查程序[343]中的故障定位。

**静态分片**的**一个缺点**是给定语句上给定变量的分片包含所有可能影响该语句中该变量值的可执行语句。因此，它可能会产生一些骰子，但不应包含某些陈述。这是因为我们无法通过静态分析来预测某些运行时间值。为了从骰子（以及切片）中排除这些额外的语句，我们需要使用**动态切片**[20,202]而不是静态切片，因为前者可以识别影响在特定位置观察到的特定值的语句，而是可能会影响到后者的价值。研究报告，其在程序调试中使用动态切片概念。

在[379]中，Wotawa将动态切片与基于模型的诊断相结合，以实现更有效的**故障定位**。针对程序使用给定的测试套件，收集发现的错误变量的动态切片。构造了击中集，其中至少包含来自每个动态切片的一条语句。声明错误的概率是根据涵盖该声明的击中数计算的。

Zhang等人[407]提出了**多点动态切片技术**，该技术与三种技术的切片相交：反向动态切片（BwS），前向动态切片（FwS）和双向动态切片（BiS）。 BwS捕获任何影响故障变量输出值的执行语句，而FwS则基于失败和成功的测试用例之间的最小输入差异来计算，从而隔离触发故障的输入部分。BiS在执行失败的测试用例时翻转某些谓词的值，以便程序生成正确的输出。

Qian等人[297]提出了一种**面向情景的程序切片技术**。用户指定的方案被识别为额外的分片参数，并且与特定计算相关的所有程序部分都位于给定的执行方案下。实现面向场景的切片技术有三个关键步骤：场景输入，场景相关代码的识别，以及最终收集面向场景的切片。

**基于动态切片的技术的一个限制**是它们不能捕获执行**遗漏错误**，这可能会导致程序中某些关键语句的执行被忽略，从而导致失败[411]。吉莫西等人。 [142]解决方法提出使用**相关切片**来定位负责执行遗漏错误的错误语句。鉴于执行失败，相关切片首先以与经典动态切片相同的方式构造动态依赖图。然后用动态依赖关系图增强动态依赖关系图，并通过在增强动态关系图上进行不正确输出的传递闭包来计算相关切片。但是，程序语句之间的**错误依赖关系可能会被包括在内以产生过大的相关切片**。为了解决这个问题，Zhang等人[411]引入了隐式依赖性的概念，其中依赖关系可以通过谓词切换获得。 Weeratunge等人使用了类似的想法。 [358]在并发程序中识别遗漏错误的根本原因，其中**使用了双重切片**，**动态切片和跟踪差异的组合**。

静态和动态分片的另一种方法是使用**基于数据流测试的执行分片来定位程序错误[**21]，其中针对给定测试用例的执行片包含由该测试执行的一组语句。选择执行静态分片切片的原因在于，静态切片着重于查找可能对任何输入的感兴趣变量有影响的语句，而不是由特定输入执行的语句。这意味着静态片不会使用任何显示错误的输入值，并且在调试中违反了一个非常重要的概念，这意味着程序员在失败的测试案例下而不是在通用测试案例下分析程序行为。尽管已经提出了不同的算法[51,204,408,409]来解决这些问题，但收集动态切片可能会消耗过多的时间和文件空间。相反，如果我们从测试的执行中收集代码覆盖数据，则为给定的测试用例构造执行片相对容易。不同的基于执行片的调试工具已经在实践中开发和使用，如Telcordia的χSuds（原Bellcore）[22,427]和Avaya的eXVantage [372]。 Agrawal等人[21]通过检查一个失败和一个成功测试的执行模块来定位程序错误，将执行片应用于错误定位。 Jones等人[186,187]和Wong等人。 [375]根据以下观察结果使用多个成功和失败的测试来扩展该研究：

 •执行一段代码的更成功的测试，代码包含错误的可能性就越小。

 •对于执行一段代码的给定错误，测试失败越多，它就越有可能包含此错误。

我们使用以下示例来**演示静态，动态和执行切片之间的差异**。 使用表2第2列中的代码作为参考。 假设它在s7有一个错误。 输出变量product的静态片包含所有可能影响产品s1，s2，s4，s5，s7，s8，s10和s13的值的语句，如第三列所示。 产品的动态切片仅包含确实影响产品相对于给定测试用例的语句，其中包括s1，s2，s5，s7和s13（如第四列所示），当a = 2时。 相对于给定测试用例的执行片包含此测试执行的所有语句。 因此，测试用例a = 2的执行片由s1，s2，s3，s4，s5，s6，s7，s12，s13组成，如表2第五列所示。

表2

上述基于切片的技术的一个问题是该错误可能不在切片中。即使骰子存在缺陷，仍然可能有太多的代码需要检查。为了克服这个问题，文献[373]提出了一种基于块间数据依赖性的增强和细化方法。前者包括搜索域中用于检查的附加代码，其基于其与当前正在检查的代码之间的块间数据依赖性，而后者使用额外的成功测试的执行片段从搜索域中排除较少的可疑代码。另外，切片是有问题的，因为它们总是冗长且难以理解。在[20dui5]中，提出了使用屏障的概念，为较小的程序切片提供了一种过滤方法，并且具有更好的可理解性。 [330]的作者提出了精细切片，以便仅查找有助于计算值并将值复制到特定变量的生成器语句。解释为什么生产者语句影响特定变量值的语句从薄片中排除。

### 3.2。基于频谱的节目技术

在第3节开始讨论之后，我们想强调许多基于频谱的技术受基于概率和统计的因果关系模型的启发。有了这个理解，我们现在解释这些技术的细节。

程序谱从某些角度详述程序的执行信息，例如条件分支的执行信息或无循环程序内路径[149]。它可以用来跟踪程序行为[305]。 Collofello和Cousins [79]早期的研究表明，这种光谱可以用于软件故障定位。当执行失败时，可以使用这些信息来识别导致故障的可疑代码。代码覆盖率或可执行声明命中频谱（ESHS）指示执行期间已覆盖测试中的程序的哪些部分。有了这些信息，就有可能确定哪些组件参与了故障，缩小了搜索导致执行失败的故障组件的可能性。

#### A.符号

P程序

NCF包含声明的失败测试用例数量

不包括语句的失败测试用例的NUF数量

NCS涵盖声明的成功测试案例数量

NUS不包括声明总数的成功测试用例数量

NC数量的测试用例，涵盖声明总数

NU数量不包括语句总数的测试用例数量

NS总数成功的测试用例

NF第i个测试用例失败的测试用例数

#### B.技术

早期研究[19,201,203,341]仅使用失败的测试案例进行基于频谱的故障定位，尽管随后这种方法被认为无效[21,185,366]。后来的研究使用成功和失败的测试案例并强调它们之间的对比来获得更好的结果。设置联合和设置交集在[303]中提出。 set union着重于由失败的测试执行的源代码，但没有通过任何成功的测试。这样的代码比其他代码更可疑。 set交集不包括由所有成功的测试执行但不由失败的测试执行的代码。 Renieris和Reiss [303]提出了另一种基于ESHS的技术，即最近邻居，该技术将失败的测试与成功的测试进行对比，测试的失败与测试失败最为相似。如果差异集中存在错误，则位于该错误集中。对于差异集中未包含的错误，该过程继续：首先构造程序依赖关系图，然后逐步包含并检查图中的相邻未检查节点，直至检查图中的所有节点。**最近邻的想法**与刘易斯的反事实推理类似[216]，它声称，对于两个事件A和B，A引起B（在世界w中）当且仅当在所有可能世界中与w最相似的世界，A不会发生，B也不会发生。反事实推理的理论在其他研究中也有发现[137,179,400]。

**直观地说**，一个语句的执行模式越接近所有测试用例的失败模式，语句越可能出错，因此语句看起来越可疑。出于同样的原因，声明的执行模式越接近失败模式，声明似乎就越不可疑。可以使用基于相似性系数的度量来量化这种亲密度，亲密程度可以解释为陈述的可疑性。

基于**ESHS**的基于相似系数的流行技术是**Tarantula** [186]，它使用覆盖率和执行结果（成功或失败）来计算每条语句的可疑程度为（NCF / NF）/（NCF / NF + NCS / NS ）。对西门子套件[185]的一项研究表明，**塔兰图**拉在识别出第一个错误陈述之前检查的代码更少，与其他设备相比，如集合联合，交集，最近邻和原因转换等，使其成为更有效的故障定位技术[77] ]。根据Tarantula计算出的可疑性，像[186,187]这样的研究使用不同的颜色（从红色到黄色到绿色）来提供每个程序语句在执行测试套件时参与的可视化映射。执行语句的测试用例越多，分配给语句的颜色越亮（更红）。在[94]中，Debroy等人进一步修改狼蛛技术。由相同数量的失败测试用例执行的语句被分组在一起，然后按降序排列失败测试用例的数量。使用狼蛛，报表按每个组内的可疑性排列。

为了便于讨论，我们再次使用表2中的代码。 假设我们有两个成功的测试用例（a = 0和a = 1）和一个失败的测试用例（a = 2）。 例如，可以使用上面讨论的狼蛛技术来计算每条语句的可疑值。 结果如表3所示。

表三

表3中的第三到第五列代表三个测试用例的语句覆盖率。 具有“●”的条目表示语句由相应的测试用例覆盖，而空条目表示语句不是。 第六和第七列给出了每个陈述的NCF和NCS的值。 根据狼蛛的定义，每一条语句的可疑值计算并显示在第八栏中。 每条语句的排名在最右边的列中给出。 正如我们可以看到的那样，错误陈述s7的排名最高。

近年来，还提出了其他技术，它们在**故障定位方面的性能**与塔兰图拉一致，甚至超过塔兰图拉。 基于Ochiai相似系数的技术[11]通常被认为比狼蛛更有效，其公式如下：

Och

Ochiai和最近邻模型有两个主要区别：1）最近邻模型利用单个失败的测试用例，而Ochiai使用多个失败的测试用例; 2）最近邻模型仅选择最接近的成功测试用例 类似于失败的测试用例，而Ochiai包含所有成功的测试用例。 Ochiai2 [267]是Ochiai的延伸，其公式如下：

Och2

在[267]中，Naish等人 提出两种技术O和OP（定义如下）。 技术O专为具有单个错误的程序而设计，而OP更适用于具有多个错误的程序。 他们的实验数据表明，O和OP比单一bug程序的Tarantula，Ochiai和Ochiai2更有效。 另一方面，Le et al。 [210]通过表明，Ochiai可以比O和OP更有效地处理单个bug的程序。

Suspect

表4列出了三十一种基于相似性系数的技术，以及它们的代数形式，这些形式已经用于不同的研究中[75,364,371]。另外一些使用类似方法的技术可以在[230]中找到。像Zoltar [170]和DEPUTO [9]这样的工具可用于计算与选定技术有关的可疑性。

实证研究还表明，[366,367,369,370,371]中提出的技术通常比狼蛛更有效。特别是在DStar [371]的情况下，针对表4中列出的所有基于相似性系数的技术以及狼蛛，Ochiai，Ochiai2，Crosstab [369]，H3b和H3c [366]以及RBF [367] - 表明DStar在大多数情况下胜过所有比较技术。

在最近的研究中经常讨论不同基于频谱的故障定位技术之间的比较[12,210,267,371]。然而，没有任何技术声称在任何情况下都可以超越所有其他技术。换句话说，Yoo等人的研究[397]支持不存在最佳的基于频谱的技术。

下面列出了一些基于程序频谱的故障定位技术的其他示例。

•程序不变量频谱（PIHS）：该频谱记录了程序不变量的覆盖率[107]，这是在程序执行过程中保持不变的程序属性。基于PIHS的技术尝试在失败的程序执行过程中发现违反程序属性的情况，以查找错误。潜在不变量[294]，也称为可能不变量[317]，是观察到的程序属性，在一些成功的执行集中被保留，但与不变量不同，不一定适用于所有可能的执行。应用这些技术的主要障碍是如何自动识别故障定位所需的必要程序属性。为了解决这个问题，现有的基于PIHS的技术通常将不变的成功执行范围作为程序属性。在研究[27]中，Alipour和Groce通过添加执行特征（例如不变量的块执行计数）来提出扩展的不变量。他们声称扩展不变量有助于故障定位。

•谓词计数谱（PRC）基于：PRCS记录谓词如何执行，并可用于跟踪可能出错的程序行为。这些技术通常被标记为统计调试技术，因为PRCS信息使用统计方法进行分析。这类故障定位技术包括LILBIT05（222）、Sover（223）等，请参阅第3.3节以获得更多细节。[266 ]的作者建议使用PRC可以实现比ESHS更好的故障定位效果。

·•方法调用序列命中频谱（MCSHS）：收集程序执行过程中涉及的方法调用序列的相关信息。在一项研究中，Dallmeier等人[84]从Java程序收集执行数据，并通过识别和分析方法调用序列来演示故障定位。考虑传入方法调用（如何使用对象）和传出调用（如何实现）。在另一项研究中，刘等人。 [225]从收集的程序执行数据构建软件行为图，包括函数之间的调用和转换关系。他们定义了一个框架来挖掘基于行为图的闭合频繁图，并使用它们来训练能帮助识别可疑函数的分类器。

•基于时间谱：时间谱[396]记录每个方法执行成功或失败的执行时间。观察到的行为模型使用从成功执行中收集的时间谱创建。在失败的执行过程中对这些模型的偏差进行识别并将其列为潜在的失败原因。

其他程序谱如表5 [149]中的程序谱也可用于识别程序中的可疑代码。

#### C.问题和关切

关于基于频谱的故障定位的各种问题和担忧也被深入地识别和研究。一个问题是，大多数基于频谱的技术不能校准失败和成功测试的贡献。在[385]中，所有的陈述分为可疑和不可疑的组。可疑组包含至少由一个失败的测试用例执行的语句，而不可信组包含剩余的语句。风险仅针对可疑报表进行计算，而不可信报表仅分配最低值。但是，成功的测试用例也可能包含错误。在[366]中，Wong等人重点讨论每个额外失败或成功的测试案例如何帮助定位程序错误。他们描述了对于一段代码来说，执行它的第一个失败的测试用例在计算其可疑性时的贡献大于或等于执行它的第二个失败的测试用例的贡献，该测试用例又大于或等于等于执行它的第三个失败的测试用例，等等。这个原则也适用于成功的测试用例提供的贡献。此外，执行语句的所有成功测试用例的总贡献应小于执行它的所有失败测试的总贡献。认识到故障定位往往是通过比较与失败的测试案例相关的信息和成功的测试案例来进行的，Wong等人[373]和郭等人。 [140]为了更有效的故障定位，试图回答选择哪个成功的测试案例进行比较的问题。根据基于控制流的差异度量，选择执行顺序与失败测试用例最相似的成功测试用例可以最小化故障的搜索域。

对于大多数基于频谱的技术，如果语句表现出相同的执行模式，那么分配给这些语句的可疑分数将完全相同的可能性很高。具有相同可疑性的语句会导致排名中的联系。为了打破这些关系，除了陈述覆盖之外，还可以利用与陈述执行频率有关的信息[7,213]。在[393]中，Xu等人评估不同的打破策略，包括基于语句顺序的策略，基于置信度的策略和基于数据依赖的策略。第7.6节将进一步讨论打破僵局的方法。另一个问题是，几乎所有基于频谱的技术都假定存在测试预测，这限制了它们的实际适用性。因此，谢等人。提出了基于变质关系和切片的整合的故障定位技术，其中程序执行切片被变形切片取代;一个单独的测试用例由一个变质测试组代替;并将测试用例的成功/失败结果替换为变形测试组的违规/非违规结果。 [71]的作者也使用变形关系和符号测试来进行程序调试。

然而，所有这些技术都强烈依赖于从程序规范中导出的变质关系。正确识别这种关系不仅困难，而且在实践中也很耗时。

赵等人。 [421,422]假定仅使用单独的覆盖信息可能不会揭示执行路径。因此，他们首先使用程序控制流图来分析程序执行，然后将失败执行的分布映射到不同的控制流。他们使用错误倾向性来限定每个块如何对失败和无错误置信度做出贡献，通过比较同一个失败执行路径上块的分布来量化每个块无错误的可能性。

仪表开销是另一个问题，它在故障本地化过程中引入了相当大的成本，特别是在资源受限的环境中。为了缓解这个问题，Perez等人[291]通过使用更粗糙的工具来减少这种开销，提出了创建动态代码覆盖。该技术首先分析程序的大型组件（例如，程序包或类）的覆盖率跟踪，然后逐渐增加可能出现故障的组件的检测粒度，直到达到语句级别。

### 3.3。基于统计的技术

[222]中介绍了一种统计调试技术（Liblit05），该技术可以在特定点上用仪器化谓词隔离程序中的错误。对于每个谓词P，Liblit05首先计算P为真的概率表示失败，失败（P）以及P的执行意味着失败的概率Context（P）。具有故障（P） - 上下文（P）≤0的谓词将被丢弃。剩下的谓词根据它们的重要性分数来区分优先级，这些分数表示谓词和程序错误之间的关系。应首先检查具有较高分数的谓词。奇林比等人。 [74]建议用路径配置文件替换谓词可以提高Liblit05的有效性。路径配置文件在执行过程中收集，并通过反馈报告汇总到多个测试用例的执行中。计算每条路径的重要性分数，并选择最前面的结果作为潜在根本原因。

在[223]中，刘等人。提出SOBER技术来排列可疑谓词。谓词P可以被评估为真实的多于

n

（）

一次执行一个测试用例。计算π（P）=

，在每次执行n t n f +时P被评估为真的概率

（）（）

测试用例，其中n（t）是P评估为真的次数，n（f）是P评估为假的次数。如果

π（P）在失败执行中的分布与成功执行中的分布显着不同，则P与错误有关。胡等人。 [168]使用类似的启发式来排列所有谓词。另外，他们应用非参数假设检验来确定成功和失败测试用例谓词频谱之间的差异程度。这项新的改进已经经过实践评估是有效的[416,420]。

[369]的研究提出了一个**交叉列表**（又名交叉表）分析技术来计算陈述的**可疑性**。为每个语句构建一个交叉表，其中包含两个垂直类别（覆盖/未覆盖）和两个水平类别（成功执行/失败执行）。假设检验用于提供执行结果与每个语句覆盖范围之间的依赖关系/独立性的参考。每个声明的确切可疑性取决于其报道与执行结果之间的关联程度。

Crosstab，SOBER和Liblit05之间的主要区别在于**交叉表**通常可用于对可疑程序元素（即语句，谓词，函数/方法等）进行排序，而最后两种只对故障定位的可疑谓词排序。对于Liblit05和SOBER，将前k个谓词的相应语句作为查找错误的初始集合。正如Jones和Harrold在[185]中所建议的那样，Liblit05没有提供量化所有语句排名的方法。定义了谓词的顺序，但该方法没有详细说明如何排序与谓词之外的任何错误相关的语句。对于SOBER，如果该错误不在最初的语句集合中，则必须通过在相应的程序依赖关系图上执行广度优先搜索来包含额外的语句，这可能会耗费大量时间。然而，交叉表不需要这样的搜索，因为该程序的所有陈述都是基于他们的可疑性进行排名的。 [369]报告的结果表明，交叉表几乎总是比Liblit05和SOBER更有效地定位西门子套件中的错误。

### 3.4。基于程序状态的技术

程序状态由程序执行期间特定点处的变量及其值组成，这可以作为查找程序错误的良好指示器。在软件故障定位中使用程序状态的一种方法是通过相对调试[4]，其中可以通过将内部状态与程序的“参考”版本的运行时间比较来定位开发版本中的故障。另一种方法是修改某些变量的值以确定哪一个会导致错误的程序执行。泽勒等人。通过对比[426]中描述的成功测试和失败测试执行之间的程序状态，通过它们的存储器图形，提出了一种技术，delta调试[400,401]。通过用成功的测试中的值替换失败测试中相同点的相应值并重复执行程序，测试变量的可疑性。除非观察到相同的故障，否则该变量不再被视为可疑。请注意，[400,401]中讨论的简化故障诱导输入的思想与其他技术是正交的，因为它显着减少了原始执行长度。 Delta工具[86]已广泛用于工业自动化调试。在[77]中，Cleve和Zeller将delta调试扩展到原因转换技术，以确定故障原因从一个变量到另一个变量的位置和时间。提出了一种名为cts的算法来快速定位程序执行中的原因转换。报告了基于组合测试的类似研究[273,274]，其将输入参数分成错误可能和健康可能，并确定最小的导致失效的参数组合。

但是，原因转换技术是一种成本相对较高的方法;在程序执行过程中可能会存在数千个状态，并且每个匹配点处的delta调试都需要额外的测试执行来缩小原因。另一个问题是，确定的位置可能不是错误所在的地方。古普塔等人。 [141]引入失败诱导排除的概念作为原因转换技术的延伸来克服这个问题。首先，delta调试用于识别导致失败的输入和输出变量。然后为这些变量构建动态切片。在输入变量的正向切片和输出变量的反向切片的交集处的代码被认为是可疑的。

Sumner等人通过将其与更精确的执行对齐技术相结合，进一步提高Delta调试的稳健性，精度和效率[338,339,389]。然而，delta调试仍然有三个局限性：它不能处理部分状态替换的混淆，不能找到执行遗漏错误，效率较差。为了解决这些限制，Sumner和Zhang [340]提出了一种原因推断模型，即比较因果关系，以提供一种系统的技术来解释执行失败与成功执行之间的区别。

由Zhang等人提出的谓词转换[405]是另一种基于程序状态的故障定位技术，其中程序状态被改变以强制改变执行失败时执行的分支。如果切换，可以使程序成功执行的谓词被标记为关键谓词。该技术首先发现变量中的第一个错误值。不同的搜索策略，例如最后执行的首次切换（LEFS）排序和基于优先级（PRIOR）排序，可以帮助确定关键谓词的下一个候选者。 Wang和Roychoudhury [352]提出了一种类似的技术，分析失败测试的执行路径，并改变该路径中分支的结果以产生成功执行。结果已更改的分支报表记录为错误。谓词切换的不足之处在于，程序状态的交替从来没有受到程序依赖性分析的指导，尽管故障通过程序依赖链内在地传播。 [217]中的研究扩展了谓词切换技术，并通过基于依赖性分析在失败执行中选择一个追踪点子集来减少程序状态的搜索空间。

杰弗里等人。 [172]提出了一种基于价值剖面的故障定位技术，以帮助开发人员进行软件调试。该方法涉及计算有趣的值映射对（IVMP），它显示了如何改变特定程序语句中使用的值，以便失败的测试用例能够产生正确的输出。从可用测试套件中的所有测试用例的执行中获取的分析信息中选择替代的值集合。不同的替代值集用于在每个失败的测试用例的每个语句实例中执行值替换。使用这些IVMP，然后可以根据为该语句识别出至少一个IVMP的失败执行次数对每个语句进行排名。在[417]中，Zhang等人声称一个声明中的一个错误可能会在它显示失败之前传播一系列受感染的程序状态。

另外，即使每次失败的执行都执行特定的语句，但该语句不一定是失败的根本原因。因此，他们使用边缘配置文件来表示程序执行情况，并评估通过每条边传播的受感染程序状态的可疑程度。通过将基本块与边缘相关联，生成可疑排名以定位程序错误。

### 3.5。基于机器学习的技术

机器学习是通过经验改进的计算机算法的研究。机器学习技术具有自适应性和鲁棒性，并且可以基于数据生成模型，但人员交互有限。这导致它们在诸如生物信息学，自然语言处理，密码学，计算机视觉等许多学科中的使用。在故障定位的情况下，手头的问题可以被识别为试图学习或推断基于故障的位置输入数据，如语句覆盖率和每个测试用例的执行结果（成功或失败）。

Wong等人[374]提出了一种基于反向传播（BP）神经网络的故障定位技术，BP神经网络是实践中最流行的神经网络模型之一[112]。 BP神经网络结构简单，使用计算机程序很容易实现。此外，BP神经网络有能力逼近复杂的非线性函数[154]。收集每个测试用例的覆盖率数据和相应的执行结果，并将它们一起用于训练BP神经网络，以便网络可以了解它们之间的关系。然后，将一套虚拟测试用例的覆盖范围输入到训练的BP网络中，每个虚拟测试用例只覆盖程序中的一个语句，并且可以将输出视为包含错误的每个语句的可能性。 Ascari等人[36]将基于BP的技术[374]扩展到面向对象的程序。由于已知BP神经网络遭受诸如瘫痪和局部最小值等问题，Wong等人[367]提出了另一种基于RBF（径向基函数）网络的方法，它们不易受这些问题影响，并且学习速度更快[211,357]。 RBF网络使用类似于BP网络的方法进行训练。一旦训练完成，关于每个虚拟测试案例的覆盖范围的输出被认为是相应陈述的可疑性。这种方法有三种新颖之处：1）用于在修改的RBF神经网络形式中表示测试用例，覆盖信息和执行结果的方法，2）用于同时估计隐藏的神经元数量及其接受场中心的创新算法， 3）基于加权比特比较的距离（而不是欧几里得距离）来测量两个测试用例的覆盖范围之间的距离。

在[57] Briand等人。使用C4.5决策树算法来构造将测试用例划分为不同分区的规则，这样在相同分区中失败的测试用例很可能因相同的原因故障而失败。潜在的前提是，根据测试用例的输入和输出（类别划分），可以识别测试用例的不同失败条件。决策树中的每条路径都代表了一条建模不同故障条件的规则，可能来源于不同的故障，并导致明显的故障概率预测。使用每个分区中失败和成功的测试用例的陈述覆盖率来使用与塔兰图拉[185]类似的启发式来对陈述进行排名以形成排名。然后合并这些个人排名以形成最终陈述排名，可以检查这些排名以定位错误。

3.6。