时空可组合性的编程范式

A Programming Paradigm for Spatiotemporal Composability

Yifan Shi · Wei Zhang · Tianyi Cui

中文翻译版 · 2026

时空可组合性的编程范式

Yifan Shi1,2, Wei Zhang1, Tianyi Cui2
1Peking University 2DeepSeek-AI

摘要

从插件系统到自演化代理框架，现代软件越来越需要动态组合，但其形式化基础仍未充分发展。我们识别了该问题的两个正交维度：时间可组合性，即在移除组件时完全恢复其副作用的能力；空间可组合性，即声明和响应式管理组件间依赖关系的能力。我们通过将经典的效应和协效应概念提升到运行时机制来解决这两个维度的问题。具体而言，我们形式化了可恢复效应，其中每个上下文转变都携带一个运行时跟踪的逆变换。我们形式化了响应式协效应，其中上下文的每个变化都会通知组件，从而

它的余效应应规范。我们将效应上下文和余效应应上下文统一为一个单一的上下文类型，这构成了一种编程范式。在此之后，我们将这些机制结合到组件的概念中，并给出一个动态组合的演算，其元理论将单个组件的时空可组合性扩展到交错组件的整个系统。我们在 Cordis 中实现了这些思想，Cordis 是一个时空可组合性元框架，提供了带有效应跟踪和余效应应解析的核心库，以及带有配置协调和热模块替换的声明式组件加载器。

目录

2

1. 引言

组合——将复杂系统从更简单的部分组装起来——是软件工程的一个基础原则 [1]。传统上，组合是静态的：函数调用、模块导入和类继承在编译时就已确定，并在整个执行过程中保持不变。然而，现代软件越来越需要动态组合，其中组件可以在运行时被加载、卸载和重新配置。插件架构 [2]
和自演化智能体都需要能够安全地动态添加和移除功能的系统，但当前实践依赖的是粗粒度机制 [3]，只能通过重启来重新配置，从而丢弃运行时状态。尽管动态组合在实际中的重要性日益增加，但与静态组合富有的形式化框架相比，其理论基础仍然发展不足。

1.1. 可组合性的维度

为了描述动态组合的要求，我们在对组合的代数方面进行充分研究的基础上，识别出两个正交的维度：

- 时间可组合性 (Temporal composability) 关注时间维度：在移除组件时，组件对共享环境所做的修改必须完全且安全地撤销。这要求跟踪组件执行的每一次资源分配、事件注册和状态变更，并保证在移除时能够有序地回收这些资源。
- 空间可组合性 (Spatial composability) 关注空间维度：组件必须能够以结构化且可验证的方式声明、发现并解决彼此之间的依赖关系。这要求管理依赖关系拓扑，并在依赖关系发生变化时协调组件的生命周期。

在静态环境中，时间上的可组合性归结为词法作用域（例如，RAII [4]，括号模式
[5]），空间上的可组合性归结为模块导入解析 [6]。在动态环境中，当组件在运行时到达和离开时，这两个维度都变得显著更难：时间上的可组合性必须处理生命周期长、状态性效应，其作用域不是词汇上限定的；空间上的可组合性必须处理在执行过程中出现、消失或改变身份的依赖关系。

1.2. 动机示例

1.2.1. 插件系统

插件系统是动态组合的典型实例。我们使用 Visual Studio Code (VSCode)
这一最广泛使用的可扩展集成开发环境之一作为代表性示例。
时间限制。VSCode 在一个称为扩展的共享进程中运行所有扩展

主机。虽然扩展可以动态安装，但该主机不提供在运行时卸载单个扩展代码的机制。一旦扩展的 activate 函数执行完毕，禁用或卸载它都需要重启整个主机，这会影响所有已加载的扩展。纯声明式扩展，如主题、键绑定和代码片段，不包含任何

代码可以自由移除。然而，在按安装量排名的前100个扩展中，有87个包含可执行代码，因此在移除时需要进行这样的重启。虽然VSCode提供了一个停用钩子，但它仅作为在主进程终止期间的优雅关闭回调，因此不支持实时移除。此外，该钩子将效应释放与效应创建（在激活时）分开，违反了关注点局部性，使得完整清理难以验证。
空间限制。VSCode确实提供了extensionDependencies来声明扩展之间的依赖关系，但使用很少：在按安装量排名的前100个扩展中，只有7个在非内置扩展上声明了extensionDependencies。这种稀缺反映了扩展API的形态，该API暴露了固定的表层扩展点，例如com -

命令、视图和语言特性。扩展通过这些点向宿主提供贡献，而不是相互依赖，所以扩展之间的依赖关系很少出现。此外，VS Code 的扩展间交互机制没有提供结构化的契约：它通过 vscode.extensions.getExtension(...).exports
将一个扩展的功能暴露给其他扩展，但返回值是无类型的（默认是
any），因此依赖方无法依赖已检查的接口。简而言之，VSCode
引导扩展使用一组固定的宿主提供的扩展点，并且不提供安全、结构化的方法让它们相互依赖。
这两个限制并非 VSCode 独有；它们在插件系统中普遍存在 [2, 7]，仅在程度上有所不同。
1.2.2. 自演化智能体工具
现代 AI 代理依赖于运行时代理工具 [8–10]。这些系统可能会组合多样化的

工具套件[11]和执行环境，管理权限和沙箱，维护会话状态和持久性，提供上下文管理和内存系统[12]，协调子代理和多代理工作流[13]，并向用户和自动化公开接口。未来的运行环境可能会生成并部署自身组件的修改，同时持续处理请求。模型合成的可重用工具为组件级自我修改[14]提供了更窄的前导。每一次这样的修改本身就是动态组合的一个实例。
由于这些修改是连续发生的且人类监督有限或不存在，动态可组合性变得必不可少。没有时间可组合性，每一次自我修改都迫使进行完整重启，从而丢弃所有进程本地累积的状态；在这种情况下

频繁发生时，累计的不可用性会变得显著，而飞行中的任务会反复被中断；更糟糕的是，错误的自我修改可能会禁用恢复所需的进程。没有空间可组合性时，每个模块必须自己检测并适应依赖模块的变化——这些变化可能是出现、消失或身份改变——而且只能通过临时手段进行；更糟糕的是，简单的代码替换策略可能会悄悄破坏依赖模块，或引入只有在重新加载时才显现的循环依赖。

1.2.3. 粗粒度解决方法

动态可组合性之所以受到有限的正式关注，其中一个原因是操作系统和容器编排器已经提供了粗粒度的替代方案。操作系统以进程为粒度提供时间上的可组合性；容器编排器

1数据来自 2026 年 6 月 9 日的 Visual Studio Code 市场。
5

[3] 在服务粒度上产生空间可组合性。实际上，大多数软件通过依赖这些粗粒度机制来容忍缺乏细粒度可组合性的情况：表现异常的模块通过重启进程来处理，服务依赖关系则由容器编排器管理。
然而，这种变通方法带来了巨大的成本。从时间上讲，每次重启都会丢弃所有进程本地的累积状态（例如缓存、连接、部分计算），并且重建这些状态需要数秒到数分钟[15]；在此期间保持可用性需要冗余副本，这会导致资源开销，以弥补单个组件无法恢复的缺陷。从空间上讲，容器级编排无法表达共享地址空间的组件之间的依赖关系，并且会为本可以}

是本地函数调用。两种机制都在进程和容器的边界处操作，然而现代系统越来越倾向于在更细的层次上进行组合。这种粒度不匹配要求一种组合抽象，以在组件自身的相同层次上管理效应和依赖关系。
1.3. 贡献
动态可组合性的两个维度分别涉及计算如何修改其环境以及如何依赖其环境。这两个方向就是效应系统 [16, 17]
和共同效应系统 [18, 19] 所形式化的：效应提供了关于环境修改推理的形式词汇，共同效应用于关于环境需求的推理。然而，现有的形式化仅将推理限制在词法固定作用域的编译时分析，而不扩展到组件动态出现和消失的场景。

运行时。通过将效应提升到可恢复的运行时模型，将协效提升到反应式依赖解析机制，我们获得了动态可组合性的统一形式基础，这一基础与语言无关，并适用于任何需要动态组合的软件架构。我们的贡献如下：
1. 我们形式化了可恢复的效应（第3.1节）：每一个上下文变换都携带显式的逆操作，运行时进行跟踪，并且跟踪与恢复都保持组合性，因此在组件移除时上下文得以恢复。这确立了局部时间可组合性。
2. 我们形式化了反应式协效（第3.2节）：组件声明它所需的协效作为规范，每次上下文的变化都会根据该规范通知组件，标记为激活、停用或中性。这确立了局部空间

可组合性。
3. 我们将效应上下文和协效应上下文统一为单一的上下文类型（第3.3节），其中协效应上的观测等价为效应提供了独立性，构成时空可组合性的编程范式。
4. 我们给出了动态组合的演算（第4节），它将两种机制结合为组件的概念，并赋予其生命周期操作语义。其元理论将时空可组合性从单个组件扩展到整个交错组件系统。
5. 我们在Cordis中实现了这些思想（第5节），这是一个时空可组合性元框架，提供了一个核心库，实现具有效应追踪和协效应解析的形式模型，并提供具有配置调和和热模块替换的声明式组件加载器。
6

2. 预备知识

本节简要概述效应系统和共效系统——这两个构成我们工作的理论支柱。我们假设读者熟悉基本类型理论和范畴理论；这里的目标是固定符号并介绍第3节将会作为运行时机制操作的关键抽象。

2.1. 效应

在简单类型lambda演算（STLC）[20, 21] 中，一个类型判断 **Γ** ⊢ ： 表示在上下文 **Γ** 下，项 的类型是 。效应系统通过细化类型来描述一个计算可能产生的副作用，从而生成如下形式的判断:

Γ ⊢ : effect (1)

这里，结果类型被标注为效应代数的一个元素，用以描述计算可能产生哪些副作用，从而能够对有状态计算进行组合推理。这种方法起源于 Lucassen 和 Gifford [22]，他们引入了一种

区分类型、效应和区域的种类化类型系统，以发现并行程序中的调度约束。

单子效应。Moggi [16] 首次通过单子在范畴上建模计算效应；Wadler [23] 在 Haskell 中推广了这一方法。范畴上的一个单子 (,,) 将有副作用的计算封装为类型为 () 的值，其中 ： → () 将纯值提升，：(()) → () 对嵌套计算进行顺序处理。经典实例包括 Maybe 单子（用于部分计算）、State 单子（用于可变状态）以及 IO 单子（用于外部交互）。

代数效应。Plotkin 和 Power [17, 24] 证明了代数操作确定单子，建立了一个框架，使效应接口与其实现解耦。效应签名 **Σ** 声明了一组操作（例如，get : () → ，put : → ()）。

用于状态）；程序可以自由地调用操作，而不必承诺特定的解释。Plotkin 和 Pretnar [25] 随后引入了效应处理器，它通过提供续延语义来解释操作：

handle with { op(,) ⋯ } (2)

处理器接收操作参数 和有界续延 ，它可以调用零次、一次或多次，从而在统一框架中实现异常、协程和非确定性 [26]。诸如 Koka [27, 28]、Eff [29] 和 OCaml 5 [30] 等语言采用了代数效应，并在设计上做出了不同的权衡。

2.2. 余效应应

与效应相对的，余效应应系统 [18, 31] 丰富了上下文而不是类型，产生如下形式的判定：

Γ coeffect ： (3)

这里，上下文被注释为余效应应代数的元素，用于描述⋯

计算需要来自其环境的内容，例如可访问的资源、需要持有的权限，

7

或依赖的服务。虽然 effect 模型描述了程序对世界的影响，coeffect 则模型世界对程序的约束。

Comonadic coeffects。使用 comonad 来构建依赖上下文的计算的想法最早由 Uustalu 和 Vene [32] 提出，他们提出了对称 (半)幺半群 comonad 作为 Moggi 的 monadic 效应框架的对偶，捕捉了诸如数据流和属性评估的概念。Petricek 等人 [18] 在此基础上提出了 coeffect 作为上下文依赖的统一静态分析。一个 comonad (,,) 捕捉了依赖上下文的计算：：() → 从上下文中提取当前值，：() → (()) 为嵌套访问复制上下文。环境 comonad () = × 模型依赖于固定环境 ；流 comonad () = → 模型依赖于时间数据。

分级协作效应。为了更细粒度的跟踪，分级协作效应系统使用一个预排序的半环 = (,⩽,+,×,0,1) 作为协作效应代数 [33]，这一学科后来被 Gaboardi 等人 [19] 与分级效应统一。 的元素注释每个变量绑定以量化其使用情况：0 表示未使用，1 表示线性使用， 表示有界使用，∞ 表示无限制使用。半环运算按顺序 (×) 和并行 (+) 组合协作效应，使得在统一的代数框架 [37] 中能够实现精确的资源跟踪、敏感性分析 [34] 和信息流控制 [35, 36]。

2.3. 与动态可组合性的关系

效应和协作效应系统沿两个互补方向组织对计算的推理：效应描述计算如何修改其环境，而协作效应

描述它如何依赖于其环境。这两个方向对应于第1节中识别的动态可组合性的两个维度：

- 时间可组合性要求组件对共享环境的修改在卸载时是可逆的。相关的影响是有状态的，它们会持久地改变环境；撤销这种改变需要该改变有逆操作。
- 空间可组合性要求组件之间的依赖关系被声明并以响应式方式管理。这些依赖关系正是协作用（coeffects）所捕捉的内容，管理它们相当于将每个依赖与环境所提供的资源进行匹配。

然而，经典的效应（effect）和协作用（coeffect）系统是静态工具：效应在词法固定的作用域内被跟踪，并由编译期处理程序处理；协作用注释

会在执行前确定的上下文中进行验证。相比之下，动态组合需要对在运行时到达和离开的组件，在不断变化的上下文中保持这些保证。没有固定的词法作用域可以界定部署后加载的插件；没有编译时上下文可以预见运行时配置中出现的依赖关系。这促使我们转变视角：与其通过更多注解扩展静态类型系统，我们不如将效应和协效应的概念结构具体化，以便运行时可以直接操作它们，从而动态地建立这些系统在静态时提供的保证。

8

3. 可逆效应与反应性余效应应

本节将第2节中引入的效应和余效应应的概念提升到运行时机制，构建动态组合理论。核心思想是将携带效应和余效应应的类型环境转化为上下文类型，即在运行时可操作的类型，将上下文具象化为一等公民。对于效应类型，我们将其建模为带有逆变换的上下文变换，从而实现局部时间组合性。对于余效应应上下文，我们将其建模为携带依赖信息的类型，从而实现局部空间组合性。随后对余效应应的观测等价性为效应提供了独立性。既携带效应又携带余效应应的统一上下文本身就构成了一种编程范式。

3.1. 可逆效应

时间可组合性是指在运行时加载和卸载组件的能力，使得在卸载后，共享环境能够恢复到合成前的状态。这要求组件对环境的每一次修改都必须是可追踪和可恢复的。因此，我们将一个效应建模为类型为 $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$ 的函数：将其应用于当前上下文，它会产生修改后的上下文以及一个明确的逆操作。提供该逆操作可以使效应被还原，并将其返回运行时环境可以使效应可追踪。我们称这种效应为可回退效应：通过在执行过程中追踪并组合这些逆操作，完整的环境恢复就成为一种结构性保证。

3.1.1. 效应上下文

给定任意非纯函数 _impure：→ ，我们将其转换为纯形式 ：$\Gamma\times \to \Gamma\times$

，其中 Γ 是上下文，所有可能的副作用都可以表示为 Γ 上的变换。对于任何固定输入 ：，由此诱导的映射 pr1((,)) 捕捉了 的副作用，而与返回值无关。因此，Γ 上的效应存在于 $\Gamma \to \Gamma$ 的变换单子中，运算为复合 ，其中每个单子公理都可以直接对应为效应的性质：

- 闭合性：两个效应的顺序组合仍然是一个效应；
- 结合性：复合效应不依赖于括号的方式；
- 单位元：$\mathrm{id}\Gamma$，即 Γ 上的恒等函数，作为复合运算的单位元。

为了对可以撤销的效应建模，我们将每个变换 与另一个可以撤销 的变换 配对，并称 为 的左逆，在本文中简称为逆。

撤销是单向的：所谓的逆是针对 定义的，而不是 。变换对自身带有一种乘法关系：

定义 1. 将上下文变换对的扭曲组合定义为

(1,1) (2,2) (1 2, 2 1) (4)

就 本身而言，左操作数在右操作数之后作用，而逆元的累积顺序相反。它使得 $(\Gamma \to \Gamma) \times (\Gamma \to \Gamma)$ 成为一个幺半群，单位元为 $(\mathrm{id}\Gamma, \mathrm{id}\Gamma)$，是变换幺半群与其对偶的乘积，我们称之为 Γ 上的扭曲组合幺半群 Γ。

为了在上下文本身中跟踪效应，我们引入以下定义：

9

定义 2. 给定一个上下文 Γ，定义其效应上下文为：
$\Gamma \quad \Gamma \times (\Gamma \to \Gamma)$ (5)
它可以理解为一个二元组 $(,)$，其中：
- $:\Gamma$ 表示当前上下文状态；
- $:\Gamma \to \Gamma$ 表示累加器，是到目前为止执行的效应的逆的组合函数，

以及将上下文恢复到其初始状态的函数。
特别地，初始效应上下文可以表示为 $(0, \mathrm{id}_\Gamma)$。
我们也写作 $2_\Gamma = \Gamma \times (\Gamma \to \Gamma)$，以此类推构成塔结构。
由于存在累加器 ，对 Γ 执行的所有效应都可以被跟踪和恢复。下面我们给出用于跟踪和恢复的具体构造。
定义 3. 定义上下文函数对的变换 track_Γ ：
$\mathrm{track}_\Gamma : (\Gamma \to \Gamma) \times (\Gamma \to \Gamma) \to \Gamma \to \Gamma$
$\mathrm{track}_\Gamma = (,) \ (,) \ ((), \ )$ (6)
该变换将一个前向函数 及其候选逆函数 转换为

效应上下文 Γ 的一个变换。将 $\mathrm{track}_\Gamma(,)$ 应用于状态 $(,)$ 会通过 变换 ，并将逆映射 作用于
，从而在上下文中跟踪 的效应。

定理 4. 对于每个 $(,) \in (\Gamma \to \Gamma) \times (\Gamma \to \Gamma)$，以下图交换，即:
$\mathrm{pr1} \ \mathrm{track}_\Gamma(,) = \mathrm{pr1}$ (7)

证明。对于所有 $(,) \in \Gamma$:
$(\mathrm{pr1} \ \mathrm{track}_\Gamma(,))(,) = \mathrm{pr1}((), \ )$
$= ()$
$= (\mathrm{pr1})(,)$ □

定理 5. track_Γ 是从 Γ 到 $\Gamma \to \Gamma$ 的幺半群同态。即:
1. $\mathrm{track}_\Gamma(\mathrm{id}_\Gamma, \mathrm{id}_\Gamma) = \mathrm{id}_\Gamma$;
2. 对于所有 $(1, 1), (2, 2) \in \Gamma$ ，
$\mathrm{track}_\Gamma((1, 1) \ (2, 2)) = \mathrm{track}_\Gamma(1, 1) \ \mathrm{track}_\Gamma(2, 2)$ (8)

证明。
1. 单位映射映射到单位，因为 $\mathrm{track}_\Gamma(\mathrm{id}_\Gamma, \mathrm{id}_\Gamma)(,) = (, \ \mathrm{id}_\Gamma) = (,)$。
2. 对于乘法，取任意 $(,) \in \Gamma$:

(track Γ (1,1) track Γ (2,2))(,)= track Γ (1,1)(2(), 2) = (1(2()), 2 1) =
track Γ (1 2, 2 1)(,) □ 定义 6. 在 Γ 上定义变换 recover Γ ： recover Γ : Γ → Γ recover Γ =
(,) ((),id Γ) (9) 该变换将恢复函数 应用于当前状态 并将 重置为恒等。下图说明了在对 Γ
应用一系列效应 track(1,1),⋯,track(,) 后，recover 如何将上下文恢复到初始状态： 1 ′ 1 ′
track track track recover Γ Γ Γ Γ Γ Γ Γ Γ 图中显示，跟踪的效应随后由 recover
作用，将初始效应上下文带回自身。每一步跟踪所保持的是恢复本身的结果，无论其来自何种状态： 定理 7. 对于每个
(,) ∈ Γ ，以及每一对 (,)，满足 (())= ,

recover Γ (track Γ (,)(,))= recover Γ (,) (10)
证明。
recover Γ (track Γ (,)(,))= recover Γ ((),)
= ((((())),id Γ)
= ((),id Γ)= recover Γ (,) □
一系列的配对不需要单独的论证。令 (1,1),⋯,(,) 按顺序从 (,) 应用，并写作 0 = 以及 =
(−1)。根据定理 5，复合 track Γ (,) ⋯ track Γ (1,1) 是扭曲复合 (⋯ 1, 1 ⋯
) 的 track Γ ，如果每个 都满足 ()= −1，则 (1 ⋯)()= 0 =
。因此，该配对满足定理 7 在 处的假设，并且一次应用定理得到
recover Γ ((track Γ (,) ⋯ track Γ (1,1))(,))= recover Γ (,) (11)
取 (,)= (0,id Γ)，恢复将通过这种方式到达的每个状态带回 (0,id Γ)。 一个满足 = id Γ
的配对在每个状态下都满足假设。
恢复通过数量 () 读取状态，我们称 ()= 0 为正确性

Γ 中状态的不变量。
11

3.1.2. 可逆效应函数

上一节的跟踪/恢复模型先验地假设逆函数已知：track Γ (,) 在看到任何上下文状态之前就固定了 ，因此同一个 必须适用于效应作用的每一个状态。然而在实际操作中，每个效应的逆函数并非先验已知：它必须由调用者在效应应用点提供。此外，recover 是全有全无的：它不能选择性地撤销某一个效应而保留其他效应。为了解决这两个问题，我们在输入端和输出端都增强了模型：

1. 在输入端，我们不仅转换 Γ ，还返回一个逆函数，这样逆函数就在效应应用的地方提供：$\Gamma \to \Gamma \times (\Gamma \to \Gamma)$，即 $\Gamma \to \Gamma$ ；
2. 在输出端，我们不仅转换 Γ ，还返回一个逆函数

在它的一侧，以便一个效应可以被撤销而其他效应保持不变：$\Gamma \to \Gamma \times (\Gamma \to \Gamma)$，即 $\Gamma \to 2\Gamma$ 。

这种增强保持了输入和输出之间的结构一致性，因此我们仍然可以定义相应的理论来保持轨迹的数学性质。

得到的类型是效应函数 Γ 及其有目击的细化 Γ ： 定义 8. 定义效应函数 Γ 和有目击的效应函数 Γ 为： Γ $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$ Γ $(: \Gamma \to \Gamma \times (\Gamma \to \Gamma)) \times ((: \Gamma) \to ((: \Gamma) \times (: \Gamma \to \Gamma) \times ((,) = () \to$ () =))) (12) 其中 () 产生一对 (,) 表示： • ：Γ 是新的上下文； • ：$\Gamma \to \Gamma$ 是当前效应的逆函数。

Γ 的一个元素根据状态选择其逆，并且约束 () = 确保在应用效应的地方可以撤销它，而在其他地方没有限制。 单个

与 = idΓ 在每个状态下同时满足约束，并通过以下方式诱导出 Γ 的一个元素：

(,) ((),)，定理 11 显示这是一个同态映射。该约束可以

通过下列交换图进行可视化，确保逆运算 确实在应用 的状态下逆转变换：

pr1

pr2

Γ Γ

Γ

由于效应函数 Γ 不再是上下文上的自同态，它们无法直接组合。因此我们定义了

一种新的效应组合运算：

定义 9. 给定函数，$\in$ Γ ，定义它们的效应组合 为：

12

: Γ → Γ
=
(,)= ()
(,)= ()
(,)
(13)
定理 10. 效应组合将 Γ 的幺半群结构传递给 Γ 。也就是说，
1. (Γ ,) 是一个幺半群，其单位元为 Γ (,id Γ);
2. 赋值 (,) ((),) 是从 Γ 到 Γ 的幺半群同态。
证明.
1. 结合律和单位律从 的逐分量性质直接得出。
2. 写作 = ((),); 则 (1 2)()= (1(2()), 2 1)，这正是 (1,1) (2,2)
的像，并且 (id Γ ,id Γ) 映射到 Γ 。□
定理 11. 见证保持在效应组合下的存活，并且每个状态都有统一的逆见证。也就是说，
1. Γ 是 Γ 的子幺半群；
2. 定理 10 的同态将每个满足 = id Γ 的对映射到 Γ 。
证明.
1. 单位元属于 Γ ，因为 id Γ ()= 。对于封闭性，取 , ∈ Γ 和任意 ∈ Γ ，且

令 (,) = ()，(,) = ()，使得 ( )() = (,)。则 () = 且 () = ，因此 ( )()
= () = 。
2. = id Γ 表示在每个 上有 (()) = ，因此这样的配对在每个状态上都可以被观察到。□
正如 track 将 Γ 上的一对变换提升到 Γ ，我们定义 effect 将 Γ 提升到 Γ ：
定义 12. 定义效应函数变换 effect Γ 如下：
effect Γ : Γ → Γ → 2 Γ
effect Γ = (,) 令 (,) = () 在
((,), track Γ (, pr1))
(14)
由于 effect Γ () 本身属于 Γ ，其返回值在定义 8
的意义上是一种逆。该逆本身是通过交换效应的两个方向得到的一对的
track。普通的跟踪规则再次适用：撤销该效应本身就是一种效应，通过 转换状态，而撤销它的方法是再次执行该效应。

这就是 pr1 所做的。因此，其逆操作正好组合到传递给它的累加器上，正如 track 所规定的。我们现在可以为 effect
证明类似于 track 的性质。定理 13. effect 保持 运算。也就是说，, ∈ Γ : effect Γ () effect Γ ()=
effect Γ ( ) (15) 证明。取任意 (,) ∈ Γ ，并设 (,)= ()，(,)= ()，则 ( )()= (,
) 且 pr1 ( )= (pr1) (pr1)。然后

(effect Γ () effect Γ ())(,) = ((,), track Γ (,pr1) track Γ (,pr1))
= ((,), track Γ (, (pr1) (pr1)))
= effect Γ ( )(,)
其中第一步在 (,) 和 (,) 展开定义 12，第二步是定理 5，第三步折叠定义 12。□
两层之间的关系如下图所示。其上三角形是 的见证条件，根据定义 8，而下三角形是关于 ′ 是否以与 相同的方式被见证的问题。

,

,

,

pr1
pr1
pr2
pr2
effect
Γ Γ
Γ Γ
²Γ
在这些层之间，投影 pr1 将每个提升的映射与其提升的映射关联起来，就像定理 4 中 track Γ 所做的那样。
定理 14. 设 ∈ Γ，写作 pr1 ，并设 ′ effect Γ ()，其前向映射为 ′ pr1 ′ 。则
1. pr1 ′ = pr1;

2. 对于每个 (,) ∈ Γ，所构造的提升逆 ′ pr2(′ (,)) 和逆 pr2(()) 满足 pr1 ′ = pr1。 证明。 1. 根据定义 12，′ (,)= ((),)，其状态为 ()= (pr1)(,)。 2. 这是定理 4 应用于 ′ = track Γ (,)。 □ 下方三角形是否闭合可通过计算提升逆返回的值来确定： 定理 15. 设 ∈ _Γ 并记 pr1 。固定 (,) ∈ Γ，令 (,)= ()，并记 (Δ,′) 为 effect Γ () 在 (,) 的值。则 ′ (Δ)= (,) (16) 状态得以精确恢复。累加器也被恢复，即等价于 effect Γ () ∈ _Γ，当且仅当 = id Γ；并且在每种情况下 ( )()= ()，因此正确性不变量得以保持。 证明. 根据定义 12，Δ = (,) 并且 ′ = track Γ (,)，于是 ′ (Δ)= ((),)= (,) 14

使用 ()= 。属于
Γ 需要此等式在每个输入处都等于 (,)；取 = id Γ
将累加器的等式变为 = id Γ，而该条件反过来也会对每个 给出累加器的等式。最后 ( )()=
(())= ()。□
因此，只有当在 处见证的逆元使 在每个状态下都可以恢复时，下三角形才会闭合，因此 effect Γ 不会将
Γ 映射到
Γ。每种情况下成立的是在 上的一致性：recover Γ (′ (Δ))= recover Γ (,)，这就是定理 7
对累加器假设的全部内容，因此恢复操作不会影响恢复目标。
按照应用的相反顺序恢复效应不需要进一步操作，因为每个逆元都会遇到它自身应用所产生的状态：
定理 16. 令 1,⋯, ∈
Γ 按顺序从 (0,id Γ) 应用并按相反顺序恢复。则

1. 每次回退都会恢复其应用所运行的上下文状态；
2. 每个中间状态都满足可靠性不变式。

证明。每一步都是一次应用或一次回退。一次应用将 (,) 转换为 (,)，其中 () = ，因此它通过定理 7
保持 ()，其假设恰好是
Γ 的证明。
按照相反的顺序进行回退将每个逆操作交给自己所产生的状态，因此根据定理
15，该回退能精确恢复前一个状态并同样保持 ()；两个结论都不依赖于逆操作接收的累积器。□
3.1.3. 效应的独立性
在其应用所产生的状态下回退效应正是定理 16 所涵盖的；在任何其他状态下回退效应是本小节所讨论的。后者有两种情况会发生。逆操作可能在后续效应仍然存在时执行，这就是撤销某个效应所碰到的情况。

来自正在运行的系统的组件的作用是；而一个序列可能会交错几个组件的效应，每个组件都保留自己的逆操作，因此一个组件的逆操作会被另一个组件的应用所分隔。在两种情况下，当一个逆操作遇到被外来效应移动的状态时，它是否仍然可以恢复它原本设计要恢复的状态是一个换位问题：必须换位的是一个效应可以执行的每个变换与另一个效应可以执行的每个变换，包括前向映射和生成的逆操作。单个累加器无法解决这两种情况，
是一个复合体，它以一定顺序一次性运行它所包含的每个逆操作。

定义 17. 对于一个效应函数 ∈ Γ，变换幺半群 () 是由 的前向映射及 生成的每个逆操作所生成的 $\Gamma \to \Gamma$
的子幺半群，而生成元

() 的元素是该生成集合的元素：
() {pr1 } ∪ {pr2(()) | ∈ Γ} (17)
由一对 (,) ∈ Γ 诱导的效应以 和 为其生成元，其生成的逆变换在每个状态下都是 。
引理18. 交换性在生成元上已经解决，并且 不会扩展任何变换半群。也就是说，
1. 如果 (1) 的每个生成元与 (2) 的每个生成元都交换，则 (1) 的每个元素与 (2) 的每个元素都交换；
2. (1 2) (1) ∪ (2)。

证明。
1. 与(2)的每个生成元交换的映射在 **Γ** → **Γ** 中形成一个子幺半群，因为id **Γ** 在其中，并且如果和′ 满足条件，则 ′ 也满足条件。根据假设，该子幺半群包含(1)的生成元，因此包含(1)。固定 ∈ (1)，与交换的映射同样形成一个包含(2)生成元的子幺半群，因此也包含(2)。
2. 根据定义9，1 2的前向映射是(pr1 1) (pr1 2)，它在任意状态下产生的逆是 ，其中由2生成，由1生成。因此，(1 2)的每个生成元都是两个生成元的复合。□
定义19. 当满足以下条件时，作用函数1, 2 ∈ **Γ** 是独立的：
1. 一个的每个变换与另一个的每个变换都交换，
∈ (1)，∈ (2)． = (18)
2. 两者的变换都不干扰对方生成的逆，

∈ (2)，∈ **Γ** . pr2(1(())) = pr2(1()) (19) 并且1和2互换时同样成立。 当和′ 对每个≠ ′ 相互独立时，族() ∈ 称为成对独立。
一个族可以重复一个效应函数，使一个效应函数独立于自身即意味着()可交换。
对于由对(1,1)和(2,2)引起的效应，子句(1)根据引理18(1)为四对(1,2; 1,2; 1,2; 1,2)的可换性，子句(2)直接成立，一个引起的效应在每个状态都生成一个逆。 在下的可交换性是不同的性质。 1 2 = 2 1 相当于两种顺序的复合前向映射与彼此以及两种顺序的复合逆映射与彼此的对应，每个逆映射进入其自身应用所产生状态的复合；而独立性则涉及每个变换的关系。

将一个效应映射到另一个变换的每个变换，即一个正向映射与一个外部逆映射配对
包括在内。
在独立性下，逆变量可以在后续效应移动的状态运行，以及它
撤回 它自身的贡献，没有其他：
定理20。让e1，.....，en ∈ E
Γ 是两对独立的，并且从 γ0 开始按顺序应用。写
Fi Pr1 Ei，设 **Δ**i Fi（**Δ**i−1），其中 δ0 γ0，令 Gi Pr2（Ei（**Δ**i−1））为反 ei，得到
在适用的范围内。固定 j 并写作 δ′
对于序列的态，i （fi → fj+1） (δj−1)
EJ省略了，因此δ'
j = δj−1。那么对于每个带有j的u≤u ≤n，
1. δu = fj（δ′
u） 和 gj（δu）= δ′
;
2. 每个 ei i> j 在 δ′ 处得到
i−1 与 δi−1 处得到的逆 gi 相同。
证据。
1. 第一个方程是关于 u 的归纳法。在 u = j 时，读数为 δj = fj（δj−1），即
δj的定义。对于归纳步骤，δu+1 = fu+1（δu）= fu+1（fj（δ′

))= (+1(′
))= (′
+1)，中间的等式是定义19中条款(1)对+1和的应用，它们是该族的不同效应，因为+1> 。对于第二个等式，条款(1)将通过在之后应用的前向映射提出，使得的见证在其所在的唯一状态下被使用：
()= (⋯ +1)()= (⋯ +1)(((−1)))= ′

最后的等式基于((−1))= −1，这是定义8要求在−1处的见证。

2. 根据 (1)，状态 −1 为 (′ −1)，且 ∈ ()，因此定义 19 中关于 和 的条款 (2) 给出
pr2(((′ −1))) = pr2((′ −1))。□
条款 (1) 定位了逆操作到达的状态：它是同一序列在未应用该效应时会到达的状态，无论之后应用了哪些效应。条款 (2) 定位了其他逆操作保持的状态，两者结合可以将定理重新应用到较短的序列上：
推论 21. 设 1,⋯, ∈ _Γ 彼此独立，并按顺序从 0 应用，且令 1,⋯, 如上所述。在 处按 {1,⋯,} 的任意排列顺序应用这 n 个逆操作将到达 0。
证明. 对 作向下归纳。设排列以 开始。根据定理 20(1)，在 处应用 会到达 ′ ，即省略后序列到达的状态，并根据定理 20(2)

剩余效应应的逆就是手中得到的
。该序列是成对独立的，作为一个子族，所以归纳假设可应用于它和排列的其余部分；空序列达到 0。□
LIFO 顺序就是这样一种排列，而定理 16
在其中不需要任何假设就适用。独立性带来的好处是所有其他顺序，并且它可以与交错多个组件的序列一起使用，正如第 4.4.2 节将其应用到整个系统的轨迹中。
这些构造共同构成可逆效应：*中的每个效应函数 Γ 清楚地提供了自己的逆，效应在效应上下文 Γ 上跟踪这些逆，而 运算在组合它们的同时保持可逆性。它们提供的是局部时间可组合性，所谓局部是指所保证的只是单个组件的效应所独立产生的。我们认为这一点

成为以下标准：对于组件应用的每一个效应函数序列，累加器恢复其开始时的上下文（定理 7），而恢复序列则将每个逆操作置于其自身应用运行的状态（定理 16）。加载组件就是应用这样的序列并在 中累积其逆操作；卸载组件就是应用 。该标准遗漏了两点，而这两点在多个组件同时运作时都会出现：以累加器施加的顺序之外进行恢复，以及交错其他效应的序列。独立性提供了它们（推论 21），这是对效应的条件，而不是对构造的属性，第 3.3.2 节是确定满足该条件的规范所在，第 4.4.2 节则是读取整个系统轨迹所提供保证的地方。

当独立性失败时，顺序必须在其他地方执行：在一个组件内部通过累加器来执行，累加器会以后进先出（LIFO）的顺序回退所有效应（第4.3.2节），而在组件之间则通过声明的共效性来执行，共效性会将一个激活相对于另一个进行排序（第4.3.1节）。

3.2. 响应式共效性
空间可组合性是组件声明彼此依赖的能力，以及系统在运行时解析、提供并撤回这些依赖的能力。这要求每当共享上下文发生变化时，都必须重新评估依赖关系的满足情况，以便当组件的依赖关系变得可用时激活组件，而当依赖关系被撤回时使其停用。因此，我们将组件的依赖关系建模为一种规范，并将对上下文的每一次更改根据该规范分类为激活、停用或中性。

根据规范进行分类就是检测满意度变化的方式；对该分类作出反应就是驱动激活和停用的方式。我们称这种余效应应为反应性：通过

将上下文更改进行分类，并从中驱动激活与停用，正确的协效应顺序就成为一种结构保证。

3.2.1. 协效应上下文
传统的控制反转（IoC）容器 [38] 通常将依赖建模为简单的键值映射。本节将 IoC
形式化为一个协效应上下文，该上下文与可逆效应协同作用，为动态组合提供数学基础。
定义 22. 给定类型族 : → Type，定义协效应上下文为依赖偏函数类型：
Σ (:) (20)
其中 ：Σ 是一个有限偏函数，它将每个 ∈ dom() 映射到类型 的值。
我们记作：
• () 表示函数应用（当且仅当 ∈ dom() 时定义）；
• [ ] 表示在 处绑定值 的表，并在其他位置与 一致；
• 表示限制（当且仅当 ∈ dom() 时定义）；
• ∈ dom() 表示成员关系。

使用类型族 确保每个依赖键 都与特定的值类型
相关联，为依赖访问提供静态类型安全。扩展和限制带有前置条件，由下面的操作强加：依赖不能被重复提供（扩展时 dom()）也不能在不存在时撤销（限制时 ∈ dom()）。违反前置条件会被视为错误并且不会产生任何转换，因此描述实际发生转换的效应代数适用于这些操作而不变。想要将失败内化的读者可以将下面每个 Σ Σ 阅读为 Σ → (Σ) 并在 单子中组合（第 2.1
节），代价是将每个恒等映射替换为操作域上的部分恒等映射。基于此上下文结构，我们定义两个核心操作：

定义 23. Σ 上的获取和设置操作定义如下：
get : (:) → Σ
get = ()
set : (:)× → Σ Σ× (Σ Σ)
set = (,) ([ ], ′ .′)
(21)
其中 get() 的前提条件为 ∈ dom()，而 set(,) 的前提条件为 dom()。
值得注意的是，set(,) 的类型是
Σ，精确来说是在余效应应上下文中的一个效应函数。因此我们可以直接应用第 3.1 节的效应机制：effectΣ 提供依赖注册的自动跟踪和恢复。这就是反应性余效应应与可回退效应之间的协同作用：余效应应操作是效应，而效应是可回退的。
get 提供给组件的是一个值，而组件可以用该值做的事情取决于该键的余效应应提供的内容。因此，一个键不仅仅承载一个值类型：

定义 24. 键 处的协效应是一个三元组 (, ,)，其中 是定义 22 中的值类型， 是在
上的等价关系，用于比较键 处的值（见第 3.3.2 节），而 是一组协效应操作，即键
处绑定的值为组件提供的操作。

持有它。操作 ∈ 具有参数类型 和结果类型 ，并且只作用于值：
: → × ( )× (22)
其前两个组成部分形成在 上的效应函数，如定义 8 所要求，
其第三个是结果。每个操作都必须尊重 ：在
相关的值上，它要么在两者上都定义，要么在两者上都不定义；在定义的地方，它产生 相关的后继，逆操作同样将
相关的值映射到 相关的值，并且产生相等的结果。操作通过其提升作用于余效应应上下文
Σ()() (,,)= ()(()) ([ ], ′ .′ [(′ ())],) (23)
当 ∈ dom() 时定义，其前两个组成部分是在 Σ 上的效应函数。
对 上的 操作进行类型化即将其限制于 的绑定：提升读取并

写入绑定并保持其他所有键不变，因此不需要额外条件来说明这一点。当隔离生效时，它所到达的绑定就是该领域解析到的绑定（定义 28），两个共享同一领域的键共享一个绑定。一个行为依赖于另一个键的操作将该键的值读入其参数
，并且下一小节的反应性规则会在读取该值的组件运行期间保持该值固定（定理 63）。
3.2.2. 规范与通知
前述定义描述了如何注册和访问各个依赖。然而，访问缺失的依赖会导致运行时失败。因此，组件应在其声明的所有依赖项都存在后才激活，而不是选择性地访问它们。

神秘地，在缺少某个组件时会失败。这引出了两个问题：组件声明的依赖是否被共同满足，以及当该状态变化时系统应如何响应。协效上下文 Σ 具有一种自然的观察结构，使得这两个问题都易于处理：对于任何协效规范 ，定义满足谓词：
∈ . ∈ dom() (24)
该谓词是可判定的（因为 dom() 是有限的）。由于对 的所有变更都通过效应函数进行（其逆函数可以恢复以前的域）
，满足性的变化可以在每个效应边界被检测到。这是反应性代数基础：效应系统保证每一次协效变化都被观察到。
定义 25. 协效规范为：
Σ () (25)
表示组件从环境声明的依赖集合。

使该规范具有反应性的原因在于它如何分类状态转换。任何将 转换为 ′ 的效应，都可以根据规范 ∈ Σ
判断其满足状态是否被改变来进行分类：
定义 26. 给定一个余效应应规范 以及状态 ,′ ∈ Σ，定义：
19

notify(,′)
{
激活如果　且′
失效如果　且′
否则为中性
}
(26)
这是定义良好的，因为　是可判定的，并且所有状态转换都由效应函数介导。反应式不变量为：激活转换触发组件效应的执行（带完全效应跟踪），而失效转换通过应用累加器触发恢复。这些转换的精确操作语义取决于它们与控制流的交互，在第4节中有详细展开。
set 和 notify 一起提供的是局部空间可组合性，与之前的“局部”含义相同，保证是单独读取一个组件的协效应。我们将其视为以下准则：组件仅在满足其规范的状态下激活。

因此它永远不会读取不存在的绑定，并且对上下文的每一次更改都根据该规范进行分类，因此在满足条件丧失发生时就会被检测到，并驱动取消激活。从以上定义可以立即得出两半部分，满足条件是在组件将要激活时检查的前提条件，而通知是在每次转换时定义的。该标准涵盖了协效应顺序的一个方向，而不是另一个方向。如果组件提供一个键，而组件声明 ∈ ，那么只有在激活并提供之后才能激活，因为 要求属于dom()。反之则不成立：卸载会将从dom()中移除，因此破坏了的满足条件，但单凭通知无法在自身拆卸需要的整个期间内保持可读，也无法延迟的恢复直到完成。

已经完成。在执行解除激活后下达提款指令是对其他组件的条件，而不是对执行操作的组件的条件，因此它属于担保的全局形式，第 4.3.1 节提供了实现所需的机制。
3.2.3. 隔离与拦截
基本共效上下文 Σ
模型化了一个扁平的依赖表。然而在实际操作中，系统可能需要为不同组件将不同的值绑定到相同的逻辑依赖。 本节通过两种机制扩展共效上下文：共效隔离（相同的键在不同上下文中解析方式不同）和共效拦截（对依赖访问的跨切行为）。
实现。两种机制在作用对象上与 get 和 set
不同。提供操作会写入每个组件都读取的共享表，因此它是对该表的一个效应，并携带一个

逆向以撤回它。隔离和拦截则调整在一个上下文下组件的键的解析方式，而保持表本身不变。将一个操作类型化为效应固定了其指称，即一个与逆向配对的继承状态，但不固定其实现方式，实现方式决定了如何执行该逆向。
定义 27. 一个上下文上的效应函数允许两种实现方式：
• 就地实现会改变上下文并返回一个非平凡的逆向；继承状态与输入别名相同，恢复时运行该逆向以撤销变更。
• 派生实现保持输入不变，并返回一个从其派生的新上下文，其逆向为恒等；恢复时丢弃派生的上下文。一个从另一个派生的上下文正是定义 32 的递归结构所承载的那部分。

在纯函数式环境中，两者是一致的，而命令式宿主可以在每次操作中选择任意一种；第5.1.2节实现了两者。隔离和拦截给出了派生实现——20

完全否定：每个产生一个新的上下文，其自身的表不同于继承的表，因此每个在下面都被标记为从上下文到上下文的映射，而不是作为效应函数。共享表中的内容没有变化，因此没有可追踪的逆，也没有 Definition 12 可以提升的内容，并且恢复时会丢弃派生上下文及其携带的调整。在派生表上进行赋值会覆盖继承表在该键上的内容，这就是为什么这两种操作都不带前置条件。

协效应隔离。通过引入隔离领域，协效应隔离允许相同的依赖在不同的上下文中绑定到不同的值。这在多租户系统、测试环境和组件沙箱中有广泛应用。

定义 28. 将带隔离的协效应上下文定义为：
Σiso ( )× ((:)) (27)

它可以表示为一对 (,)，其中：
• : 是隔离域表，为每个隔离的键分配一个域标识符；一个
dom() 外的键解析为其自身的域，因此我们写作 ()= ( ) ；
• : (:) 是依赖表，它是从域标识符到类型化值的部分依赖函数。
这种两层映射结构将逻辑层与存储层解耦，使依赖访问具有上下文感知。当访问一个键 时，系统首先解析 ()
来获取域标识符 ，然后访问 () 获取实际值。
定义 29. 对 Σiso 的 get、set 和 isolate 操作如下：
get : (:) → Σiso ()
get = (,) (())
set : (:)× () → Σiso Σiso × (Σiso Σiso)
set = (,) (,) ((,[()]), (′ ,′).(′ ,′ ′ ()))
isolate : × → Σiso → Σiso
isolate = (,) (,) ([ ],)
(28)

where get 和 set 承载定义 23 的前置条件，这些前置条件沿着 传递，即 () ∈ dom() 和 () dom()。由
isolate(,) 推导的上下文将领域 分配给
并且继承依赖表保持不变，因此已经隔离的键会被重新分配而不是被拒绝。
共效隔离机制本质上实现了一个运行时临时多态系统。通过隔离领域标识符，同一个依赖键可以在不同的上下文中解析为完全不同的值，并且这种多态性可以在运行时动态调整。与传统的依赖注入相比，共效隔离提供了更细粒度的控制，使特定组件可以定制化隔离；set 保持为效应函数（_Σiso），因此继承可回退性，而 isolate
不需要回退性，而是推导出一个上下文而不是写入共享表。

系数拦截。第二种机制，系数拦截，将跨切元数据附加到依赖访问上，在不修改依赖值的情况下添加行为。该元数据可以是上下文携带的，也可以是组件声明的，因此我们扩展了系数上下文和系数规范：21

定义 30. 定义带有拦截的协效上下文和规范为：
Σinter ((:)→)× ((:) (→))
inter (:)
(29)
上下文 Σinter 是一对 (,)： 是安装在上下文自身上的上下文携带元数据，默认为空 ()； 将每个键
映射到从元数据 到值 的提供函数。规范 ∈ inter 携带组件声明的元数据，为每个键分配其元数据
()，其中 dom() 用作依赖集合。每个键都用一个幺半群 (, ⊕ ,) 装备其元数据：合并 ⊕
是结合的，其单位元为 （空元数据）。
定义 31. Σinter 上的 get、set 和 intercept 操作为：
get : (:)× → Σinter
get = (,) (,) ()(⊕ ())
set : (:)× (→) → Σinter Σinter × (Σinter Σinter)
set = (,) (,) ((,[ ]), (’ ,’).(’ ,’))

拦截 : (:)× → Σinter → Σinter
拦截 = (,) (,) ([() ⊕],)
(30)
其中 get 和 set 承载了定义 23 中提供者表的前置条件，即 ∈ dom() 和 dom()。intercept(,)
推导的上下文将 合并到在 处继承的元数据上，并保持提供者表不变。
当带有规范 的组件访问键 时，系统会计算 ()(() ⊕ ())：组件声明的元数据与上下文携带的元数据 合
并，然后将提供者函数应用于结果。此合并遵循每个键自身的语义（例如标量字段会被覆盖，集合值字段会取并集），并偏向右侧，因此 ()
具有优先权，可以覆盖组件的声明，从而允许封闭的上下文在不修改该组件的情况下约束组件如何使用共同效应（例如第 6.3 节）。

3.3. 上下文范式
第3.1节和第3.2节各自作用于一个上下文，第一个作为效应的载体，第二个作为余效应应的载体，但未明确单个同时承载两者的上下文的形式。本节为这种统一提供了一个具体构造，从余效应应中组装出一个观察等价，补充第3.1.3节留下的效应独立性，并论证所得的上下文类型本身构成一种编程范式。
3.3.1. 统一上下文
对于一个上下文 Γ，效应上下文 Γ （第3.1节）提供了一个更高层次的抽象，承载前一级上下文以及该层的累加器（定义 2）。使该结构递归化并将其与余效应应上下文 Σ 结合可得到如下类型：
定义32. 上下文类型 Γ∞ 定义为：
Γ∞ Γ . Γ× (Γ → Γ)× Σ (31)

三个投影是：22

- **Γ**：当前上下文状态（递归）；
- **Γ → Γ**：累加器，用于恢复该层的效应；
- **Σ**：承载依赖信息的共效上下文。

根据此定义，效应将 **Γ**∞ 映射到自身，将 -塔统一为单一的自相似类型。共效上下文 **Σ**
在结构上是整合的：依赖操作（设置、获取）作用于 **Σ**，累加器跟踪它们的反转。由于支持 **Σ** 的类型族
不受约束，系统需要在各组件之间共享的任何状态都可以编码为具有适当值类型的依赖——**Σ**
包含了所有共享的可变状态，而不仅仅是组件间的依赖。组件与其环境的每一次交互都通过这个单一实体进行。
分层组合。**Γ**∞ 的递归结构支持分层控制：

父上下文聚合多个子级效应，形成一个树状控制结构，在保持模块化的同时实现统一的跨级管理。效应转换实现了字面意义上的“插件”比喻：

- 加载组件对应于执行其效应（插入）；
- 卸载组件对应于恢复其效应（拔出，不影响其他正在运行的组件）；
- 不同层级的组件可以独立加载和卸载；父上下文聚合并管理其所有子组件的效应，从而实现任意嵌套组合。

3.3.2. 观察性等价

第3.1节的恢复保证断言状态的相等（定理7），这是一种理想化，因为物理状态无法恢复到原来的样子。例如，空闲

将一个块释放给分配器而不恢复 malloc 之前堆的布局；并且生成的名字不会被丢弃它的逆操作恢复，因为下次创建会生成一个新的名字[39]。因此，第3节中的等式应当被理解为在等价 下成立，我们将 定义为观察等价：当没有观察者能够
区分两个状态时，它们是相关的。比较行为而非表示是程序等价性的一条既定途径[40]，且这种比较得出的关系取决于观察者所能使用的内容[41]。上下文观察者所获得的是其所携带的协效应（coeffects），每个协效应都有其自身的等价（定义24），因此上下文的关系是由它们组合而成。组合它是本小节的任务，且

通过对其进行商操作是获得第 3.1.3 节要求的独立性的方式。
定义 33. 当两个共效上下文将相同的键绑定到相关值时，它们是相关的，而当一个上下文的两个状态的共效投影满足以下条件时，它们也是相关的：
′ dom() = dom(′) ∧ ∈ dom(). () ′ ()
′ ′

(32)
记 为 的共效投影（定义 32）。
状态中没有键绑定的部分因此被遗忘，而遗忘它正是让定理 7 至少可以在
下读取的原因：上述示例的堆布局和生成名称不在关系范围内，除非某个键绑定它们。第 3.2.2 节对
的需求是随之自然得出的，而不是假设的。相关状态具有相同的域，因此它们在 23 上一致。

满足谓词 并且在定义 26 中的分类 notify 上，而反应性是 Σ/ 的一个属性。将该关系称为观察性的，是对每个 的一个声明，即它不会区分比 的操作所能分辨的更多。一个值的观察者执行这些操作并读取它们的结果。
定义 34. 设 承载定义 24 意义下的一组操作 ，并对每个参与参数 ： 的效应函数 ()，记 ()
为其变换幺半群（定义 17）。对 的测试是对幺半群 ()，∈ 的生成元的有限字母序列，每个字母应用到前面字母留下的值；其结果是操作的前向映射产生的结果，如果前置条件失败则未定义。值 ,′ ： 是无法区分的，记作 ≈

′ ，当上的每个测试要么在两者上都有定义，要么在两者上都没有定义，并且在两者上产生相同的结果时。 引理 35. 不可区分性是操作所遵循的最粗关系。也就是说， 1. 的每个操作都遵循≈，符合定义 24 的意义； 2. 的每个操作遵循的每个等价关系都包含在≈中。
因此，每个可允许的选择都包含在≈中，且≈本身是可允许的。 证明。 1. 设≈′ ，并设∈ 作用于一个参数。在测试前加一个字母仍然是测试，所以前向映射所达到的值是不可区分的，任何由不可区分参数得到结果的逆映射值也是不可区分的；一个字母的测试在两者上都定义或都不定义，并且结果相等。 2. 设是这样一个等价关系且′ 。测试的每个字母都是前向映射或由

运算的逆，以及 respect 在任一方向上携带 ，使所达到的值保持关联，并且每个字母的结果都相等。因此，每个测试在 和′ 上都一致。□
将 替换为 = 并不单独足够，因为一个效应函数不仅返回状态，还返回逆，而两个被 识别的状态必须产生识别的逆。
定义 36. 当
,′ ∈ Γ. ′ () (′) (33)
时，映射 ：Γ → Γ 尊重 。
当两个映射在每个状态上都一致时，它们是相关的；当 Γ 中的两个对在两个分量上都一致时，它们也是相关的：
∈ Γ. () ()
(,) (′ , ′) ′ ∧ ′ (34)
尊重 的映射是可以下降到 Γ/ 的映射，而由 关联的两个映射是可以下降到同一映射的两个映射。一个效应函数需要两者：第一保证其计算的状态在商集上确定，第二保证它返回的逆也确定。

定义 37。阅读定义 8 中直到 ：当 遵循 作为映射 Γ → Γ 时， ∈ Γ 位于 Γ ，并且记 (,) = ()，对于所有 ∈ Γ 有：
1. () ；
24

2. 遵守 。
将 视为 Γ 上的等式即可恢复定义 8。
引理 38. 按定义 37 读取 _Γ，第三节 3.1 中断言的每一个状态等式在将 = 替换为 后仍然成立，并且从 (0, id_Γ)
可达的每个状态的累加器都遵守 。
证明. 累加器是逆映射的组合，每个逆映射根据定义 37(2) 都遵守 ，并且遵守 的映射组合也遵守 ，基例是
id_Γ。随后第三节 3.1 的证明保持不变，“遵守”表示关系通过逆映射传递：由 2(2) 1 和 1(1) ，可得
(1 2)(2) ，这就是每次逆映射组合所采取的步骤，而定理 7 的正确性不变量通过该步骤表示为 ()
0。□
定义 19 所要求的交换性也可通过同一引理在 的意义下理解，且其读取方式是

那种方式才使其变得可实现：两个操作可能会留下 可识别的值，但仍然算作可交换。对于两个操作，它要求的比效应
函数它们的提升所引起的更多，一个操作也产生一个结果。

定义 39. 当操作 和 ′ 的提升在每对参数上作为效应函数（定义
19）是独立的，并且彼此的变换不会干扰对方产生的结果时，称它们是独立的：
：, ∈ (′ Σ), ∈ Σ. pr3(Σ)(())) = pr3(Σ)()) (35)
并且在交换 和 ′ 的情况下也成立，写作 (Σ) 表示所有参数上提升 Σ() 的变换幺半群，正如定义 34 中写作
() 表示操作本身的变换幺半群一样。当 是可交换的，当且仅当
中的任意两个操作是独立的，同时一个操作也被认为是与自身独立的。

在不同的键上，该条件完全成立。
定理 40. 不同键上的操作是独立的。
证明. 设 位于 ，′ 位于 ′ 且 ≠ ′ 。根据定义 24，(Σ) 的每个生成元都是形式为 [
(())] 的映射，其中 是 上的映射，要么是前向映射的提升，要么是已产生逆映射的提升，对于 ′ 在 ′
上同理。这两个映射是可交换的，每个映射仅读取和写入一个键，且两个键不同，并且引理 18(1)
将生成元间的交换性推广到这两个单子上。对于第二个条件，Σ 在 上产生的结果（无论是逆还是结果）都由 ()
决定，而 (′ Σ) 的每个生成元都保持其不变。□
如果一个键的值是独立添加和删除条目的表，则该键是可交换的，路由注册或事件监听器注册是代表性例子：两个注册在

任一顺序都会留下一个回答每个测试相同的表，而任一注册都可以在另一注册保持的情况下撤销。一个值为有序链的键则不行，因为在另一个中间件之前插入的中间件会看到不同的请求，并且无法在不干扰另一者的情况下撤销任一顺序。开头示例的分配器根据其接口发布的内容进行划分。当它分发的句柄通过键的任何操作都不比较时，
可能将两个堆关联到句柄重新命名之后，这也是 CompCert 将程序及其翻译的内存状态关联的方式
[42]，并且分配是可交换的；当地址作为输出通过相等性比较时，没有允许的
会使两种分配顺序一致，因此该键不可交换。

组件执行的操作是一系列操作，其中每个操作可能依赖于之前操作的结果，而具有这种形状的效应函数就是下面定理所讨论的内容。

定义 41. 协效应介导的效应函数形成最小集合 Σ Σ，该集合包含单位元 Σ 并在以下操作下封闭：对于一个键
，一个操作 ∈ ，一个参数 ：，以及一族成员 () ∈
的成员，
(,,) = Σ()() (,) = () (,) (36)

再次是一个成员。每个阶段执行一个操作，并通过结果选择后续操作，因此参数可能依赖于已经获得的结果。出现在一个成员中的操作就是其各阶段执行的操作，涵盖每一种结果的选择。

定理 42. 设 1, 2 ∈ Σ，且两者操作发生的每个键均满足可交换性 -

tative（定义 39）。那么 1 和 2 是独立的（定义 19）。

证明。通过对定义 41 的构造进行归纳，() 位于由
中出现的运算生成的生成子单元中：单位生成平凡单元，而一个阶段是 Σ() 与某个成员的 -复合，Lemma 18(2)
适用于此。
因此对于定义 19 的条款 (1)，根据 Lemma 18(1)，1 中出现的运算生成器与 2
中出现的运算生成器交换就足够了。当两个运算位于不同的键时，这是定理
40；当它们位于同一键时，该键同时承载两种运算，并且根据假设是可交换的。
对于条款 (2)，取 ∈ (2)，它是 2 中出现的运算生成器的复合，然后对 1
的构造进行归纳。单位在每个状态下产生 id Σ。在一个阶段，设 (,,) =

Σ()() 和 (,)= ()，因此该阶段在 上产生 。对操作的独立性，
每次应用于 的一个生成元，会再次在 () 上产生 和 ，因此选择相同的延续 ，并且条款 (1)
将其运行的状态置于 ()，根据归纳假设再次得到 。因此该阶段在 () 上产生 。□
组件与其环境之间的每次交互都通过上下文进行，并且类型族
没有约束，因此系统可以在自身的键上绑定跨组件共享的每个位置（第 3.3.1
节）。组件的效应函数随后是沿着协效应投影的协效应中介函数的提升，而独立性传递到该提升，其变换仅移动投影。第 3.1.3 节留下的假设是

以这种方式满足后，整个组件系统的时间可组合性也随之而来。分解所划分的是计算中可交换的部分与对顺序敏感的部分。可交换的部分由效应承载：组件以其任务要求的任何顺序执行它们，并且推论 21 以系统方便的任何顺序恢复它们，组件之间互不限制。对顺序敏感的部分由协效应承载，因为操作不可交换的键是其顺序必须从效应之外强加的键，而有两个地方可以施加它。在一个组件内部，累加器强加它，以 LIFO 顺序恢复所有效应（定理
16）。在组件之间，声明的协效应强加它，一个组件提供另一个组件声明的内容。

以及满足声明之前的规定（第 3.2.2 节）。因此，可组合性是在组件层面上实现的，而不是单一效应的层面，这是第 4
节所处理的规模。
26

定理有两个值得指出的限制。将每个共享位置绑定到一个键是这种范式的规范，而不是该构造的属性，因此系统无法将某个位置具体化为协效应时，它就超出了第6.1节的范围，也超出了该定理的适用范围。而键的交换性是该键所发布接口的属性，因此满足它是提供该键的组件的责任，而不是使用它的组件的责任。

3.3.3. 语境范式的定位

编程范式在处理副作用的方式上有根本差异。有两个既定极点定义了整个范围：明确状态传递（函数式）。为了保持引用透明性，纯函数式语言将副作用建模为对状态的显式转换。状态单子 →

(,) [23] 在每次计算中穿插环境。这种方法提供了强大的组合性保证：效应在类型中可见，并且适合进行等式推理。然而，它带来了显著的人机工程成本：调用链中的每个函数都必须接受并返回状态参数，即使它只是简单地传递未更改的状态。随着效应维度的增加（日志、配置、I/O），单子堆叠或效应处理器样板代码会激增。
隐式变更（命令式/OOP）。主流的命令式语言允许组件修改共享状态并在调用点不明确声明的情况下访问依赖关系。在效应方面，一个代表性例子是 React 的 useEffect 钩子：它在组件的内部 fiber
上注册了持久副作用，但无论是效应目标还是注册对象都不会

tration机制作为一个显式参数出现——识别依赖于隐藏运行时状态中的调用顺序位置。在coeffect方面，Java的服务定位器模式（例如，Spring的ApplicationContext.getBean(...)）在运行时从一个进程范围的注册表中检索依赖项，每个调用位置都需要进行空值检查和类型转换；依赖关系是隐式的，并散布在整个代码库中。更一般而言，要理解f()如何修改或依赖系统，需要传递地阅读其实现。重构变得脆弱，因为移动或删除调用可能会悄然破坏远处的不变性。上下文范式结合了函数式方法的可追踪性和命令式方法的人机工程学。效应和coeffect都通过一个

显式上下文参数。因此，每个操作都是归因于其被调用的特定上下文，从而归因于该上下文所属的组件。除了结合两极的优势之外，上下文范式还允许开发者单独处理每个效应和依赖，并将它们自动组合到系统的行为中。对于可撤销效应，开发者提供每个原子操作的逆操作，而任何复合操作的逆操作通过组合得出（第3.1节），因此组件的拆除是由其加载过程推导而来的，而不是与之同时编写。对于响应式共效，组件只声明其所需的依赖，运行时自动解析并重新连接它们（第3.2节），在提供者加入时保持依赖一致连接。

被移除或替换。在两个方向上，本应依赖开发者自律的正确性成为了该范式的结构性特征。27

4. 动态组合演算
第3节仅在局部形式上建立了空间和时间的可组合性。将它们应用到整个系统需要将系统分解为组件，每个组件将协作用规格与已见效应函数配对，以便对共享环境的每次交互都可归因于其中之一。下面的各节为该分解提供了操作语义，并在全局形式上建立了空间和时间的可组合性。
第4.1节和第4.2节介绍了可以为生命周期制定规则的最小演算，该演算将每次转换视为原子、立即且绝对可靠；第4.3节放弃了这三个假设，对于每个方向上可能进行的转变仅使用一次原子性，引入运行时在转换开始与结束之间插入的控制流形式，并且

到达微积分的一个真实运行时实现；第4.4节建立了该微积分的元理论，即保持性、全局时间和空间可组合性、进展性以及汇合性。

4.1. 组件与纤程
本节固定规则作用的对象：组件；纤程，即承载自身生命周期状态的组件的实例化；以及注册表，它保存纤程承载的状态，并从中读取协效应上下文。
组件。组件给定为一个三元组，其协效应部分分为从环境中读取的内容和提供给环境的内容。
定义43. 一个在上下文 $\mathbf{\Gamma}$ 上的组件，同时承载效应和协效应（定义32）定义为：
$\mathbf{\Gamma}$ $\mathbf{\Gamma} \times \mathbf{\Gamma} \times$
$\mathbf{\Gamma}$ (37)
表示一个三元组 (,,)，其中：
• ：$\mathbf{\Gamma}$ 是定义25中的协效应规范，声明所需的依赖

来自环境；
• ：$\mathbf{\Gamma}$ () 是提供，声明组件可能提供的协效应键，并且其效应函数不会写入 之外的任何键；
• ： $\mathbf{\Gamma}$ 是定义 8 中的已见效应函数，定义组件激活时贡献的效应，以及撤销它们的逆函数。
这两个声明是一个接口的两个方向， 是组件从环境读取的内容， 是组件写入环境的内容，并且第 4.2
节不允许一个注册表中的两个纤程的提供相交。下标在整个 $\mathbf{\Gamma}$ 上取，协效应上下文是其投影之一（定义 32），因此定义 25 的 $\mathbf{\Sigma}$ 在这里写作 $\mathbf{\Gamma}$。
提供的互斥性是本章与第 3.2.3 节不同的地方。

定义28的隔离允许一个密钥通过一个领域表解析，以便两个纤程可以在不同领域中提供相同的密钥；携带领域的演算会将不相交性放宽为领域内的不相交性，并会根据声明该密钥的纤程的领域解析声明的密钥。我们这里不引入领域，而是改为在一个共享领域中读取每个密钥，这正是使上述不相交性成为正确条件并且每个密钥的提供者唯一（定义45）的原因。它限制的是组件的实例化次数：一个具有非空提供的组件一次只有一个纤程，因此下面的多个实例化是组件28的实例化。

不提供任何内容，这是只消费或注册其他组件的组件的常见情况。 在运行系统中实例化的组件会随时间被激活和停用，因此它具有生命周期状态，而转换是将其从一个生命周期状态移到另一个生命周期状态的操作：激活会执行
，在上下文中累积副作用，而停用则应用累加器以恢复上下文。在其最简单的形式中，生命周期是图 1 的两状态模型，第 4.2 节提供了其规则；第 4.3 节随着每个控制流特性的引入对其进行了完善。 L-重新加载 L-卸载 非活动 激活 图 1 |
基础组件生命周期 纤程。一个组件可能被实例化多次，每个实例化都携带其自身的生命周期状态。我们将这样的实例化称为纤程。纤程记录组件

生成它的过程、它实例化所依附的 fiber、它提供的余效应应，以及它在生命周期中的位置。
定义 44. 固定一组 的 fiber 名称。实例化组件 (,,) ∈ **Γ** 的 fiber 是一个元组 ,,,,,,，其中
- ：**Γ**，：**Γ**，且 ：**Γ** 是定义 43 中的余效应应规范、提供和效应函数；
- ： ∪ {} 是父节点，即该 fiber 所实例化的上级 fiber，或根标记 ；
- ：**Σ** 是该 fiber 自身的余效应应表（定义 22），在激活前为空，并在其效应运行时由效应写入；
- :{⊥,} 是退休标志，初始 fiber 为 ⊥，一旦 orchestration 已退休该 fiber，则为 ；
- ：**Θ Γ** 是生命周期状态，在第 4.2 节的双状态模型中为

Θ Γ | (,) (38)

其中 ：**Γ** → **Γ** 是累加器，：→ 是提交视图。

提交视图 在转换提交时，将每个 fiber 声明的键发送到提供它的 fiber 的名称。第 4.3
节用正在进行的转换所需的扩展替代 **Θ Γ**；定义 44 的其余部分对两者只给出一次，只是 在第 4.3
节每一层引入的更丰富效应类型中被读取。
注册表。一个状态根据名称保存其 fibers，同时读取该安排中 fiber 的身份以及第 3.2 节的余效应应上下文。
定义 45. 写作 **Γ** 表示 **Γ** 上的 fiber 集。状态 ∈ **Γ** 携带一个注册表

： **Γ** (39)

一个有限偏函数，其父指针形成一个以 为根的树，并且与 **Γ** 中没有 fiber 的 的名称的其他内容一起。我们写作
() 表示 ()，并在状态明确时通过下标 缩写 () 的字段，因此 ,,,,,, 是字段的

定义 44 和 , 是 所携带的累加器和提交视图；[′]、[⋯] 以及 分别是与
在一个字段、一个纤程以及一个纤程的存在上不同的状态。29

纤程的名称赋予其一种身份，即使自身发生变化也能保持这种身份：下面的每一条规则会重写一个纤程的生命周期状态，而不影响其他纤程，因此规则必须说明影响哪一个，并且有两个字段是指向纤程的，而不是描述它们，即父 和提交视图
。名称是原子：没有规则会计算名称，检查其结构，或者通过除了相等之外的任何方式关联两个名称，引入一个纤程只是选取一个尚未使用的名称。这是动态创建本地名称的规范
[39]，在这里用于纤程身份。每个拥有表的纤程意味着协作用上下文是派生的而非存储的：它是活跃纤程共同提供的内容。
{ | ∈ dom(), = (−,−)} (40)
这个并集定义良好，因为纤程只写入其声明的键，dom() 。

并且各个纤程的规定是互不相交的（定义 43），因此每个 ∈ dom()
都位于恰好一个活跃纤程的表中，我们将其名称写作 provider() ∈ ，并称其为
的提供者。因此，每个键只有一个可能的提供者，由规定决定，而不是由状态决定。没有规则直接写入
：一个纤程的规定是其自身效应函数执行的集合操作，这些操作落在 中，因此已经是状态 返回的一部分，并
且它们会随累加器再次离开。只有效应的共效部分以这种方式被记录，因为只有共效部分是其他纤程声明的对象；在
中改变其他地方状态的效应由
像其他效应一样被追踪，但没有任何纤程可以在规格中命名它们，因此它们不构成任何排序约束。第 3.2.2
节的满足关系随后不变地适用，其中 简写为

。当某个 fiber 已安装某个密钥时，该密钥正好位于 dom()
中，其提供的是它可能安装的密钥，而不是它已拥有的密钥，因此
已经要求每个声明的密钥都必须有一个提供者。仅在 fibers 上取并集，正是允许一个 fiber
在没有撤回任何内容之前停止提供的原因，第四章第4.3.1节将其转化为排序规则。

4.2 基础演算
本节给出了图1所示的双状态生命周期的演算，仅此而已：每个 fiber 对比的目标，以及移动它的五条规则。
目标视图。规则将每个 fiber 与一个目标进行比较，即它是否应该运行，以及针对其依赖项的哪种解析。目标并不仅是 fiber
本身的属性，因为 fiber 声明的密钥是针对整个状态解析的，所以它是该状态上的一个谓词。

定义 46. 在 下， 的目标视图将每个已声明的键映射到其提供者，因此它是一个从 到 的全映射，当
根本不应该运行时，它为 ⊥：
target() { ⊥ 如果 ∨ ¬( )
(∈) provider() 否则 (41)
当每个纤程都已到达其目标视图时，状态为静止：
quiet() ∈ dom(). { target() = ⊥ 如果 = 非活跃
target() = 如果 = 活跃(−,) (42)
30

目标只对应两件事，而不对应其他任何事：通过 的退休，以及通过 和 provider
的共效分辨，每个声明的键都在定义 43 的唯一共享领域 上读取。
定义 44 的提交视图与目标视图类型相同，生命周期通过比较它们来驱动： 是被激活的分辨 ，target() 是它应该
运行的分辨，每条规则根据它们的一致或差异触发。记录一个提供者而不是值使比较可用，因为否则提供相等值的不同
fiber 会被判为相等。组件读取的值是通过视图访问的，因为提供者的表中保存了该值，且实现会在 fiber.committed
中保存映射，并在 fiber.target 中保存其哈希（第 5.1.3 节）。

规则。基础演算认为每个转换都是原子性的、立即的且绝对可靠：激活在一步中应用其效应函数，停用在一步中应用累加器
，并且两者都能成功执行。第4.3节放弃了这三条假设。

五条规则生成两种关系。一个编排规则，前缀为O-，写作
，是编排者可以执行的动作；其前提说明动作何时是合法的，而不是何时发生。一个生命周期规则，前缀为L-，写作
，是系统在其前提成立时自动采取的步骤。一系列步骤混合这两种规则，下面的 表示仅生命周期步骤。

dom() ∈ dom() ∪ {} (,,) ∈ Γ ∈ dom(). ∩ =
[,,,,,⊥,]
O-插入 (O-Insert)

∈ dom()
[ ]
O-退役 (O-Retire)

= = . ≠

O-移除 (O-Remove)

插入和退役是唯一的外部输入：协调者请求一个 fiber 存在或停止存在，从不直接设置其生命周期状态。O-Retire 对 fiber
的状态无条件，因为退役只是一个请求，而生命周期规则负责执行它。出于相同的原因，退役与移除是分开的：仍然处于
状态的已退役 fiber 必须先被停用，否则提前移除会丢弃累加器并导致泄漏。前提 . ≠
通过在父节点之前移除子节点保持树的良好结构。O-Insert
的最后一个前提是单源规则强制执行的地方：一个键只有一个可能的提供者，因为协调者不允许第二个组件声明它。

= = target()≠ ⊥ ()= (,)
[(,)]

L-Reload
= (,) target()≠ ()=
[ ]

L-卸载
L-重载在逆操作的同时安装已提交的视图；L-卸载应用逆操作并丢弃已提交的视图。两者都由相同的比较驱动：当纤程没有
已提交的视图且其目标视图不是 ⊥ 时触发 L-重载，当它持有的已提交视图不是其目标视图时触发 L-卸载。这是第 3.2 节的
反应纪律，可以从目标读取，该目标对退休和余效应应都有响应：每当目标视图变化时，无论是哪一个引起的变化，都会启
动一个转换。
实例化。组件可以在安装其效应的同时实例化另一个组件，这就像插件主机在加载自身插件时加载它自己的插件一样。到目
前为止的规则留给了

仅注册到编排规则，因此这样的实例化无处发生。一个原语为它提供了一个位置。
定义 47. 的一次应用，或其迭代之一（在第 4.3.2 节适用的情况下），可以注册一个组件 (,,) ∈
Γ。它不使用状态映射，而是将该组件的 O-Insert 与 = 一起使用，并且其逆操作是已注册纤程的
O-Retire。该规则获取名称，但须满足 O-Insert 的新鲜性前提，然后将其交给效应函数。
逆操作是撤销而不是移除，其原因是逆操作必须在其被调用的任何地方都能应用。O-Remove
带有前提，因此由它构建的逆操作可能失败：一个其子仍为
的父节点无法运行其累加器，并且没有规则会移动该子节点，因为定义 46 并未读取纤程树。O-Retire 唯一的前提是 ∈
dom()。

前提。在注册时留下的条目在其所在状态下已被注销，(⊥)，并且包含一个空表，它是引理57的残余条目
：它仅在控制字段中与纤程的缺失不同，且没有规则区分两者。注销一个子节点会设置，因此将其目标视图置为⊥，之后
普通规则会将其恢复到。父节点不会被迫等待，因为O-注销是无条件的，所以无论子节点是否已离开，L-
卸载都适用于父节点。孙节点是一次达到一层，由子节点自身的累加器注销子节点注册的内容。定理66涵盖了这种级联以及
第4.3.1节沿着余效应应施加的级联。
约束。手头有唯一的例外后，效应函数所遵循的规范可以

被给予。它界定了应用程序写入的内容，以便应用它的规则能够考虑到所有其他变化，以及应用程序读取的内容，以便一个
纤程(fiber)看到它所声明的余效应应而不会看到注册表中的更多内容。界定写入是令第4.4节能够将表1视为写入内容完整清
单的原因。
定义48. 当对于每个 ∈ **Γ** 且 ∈ dom()，写入 = ()时，映射 ：**Γ** → **Γ** 被限制在：
1. （写入）dom()= dom()，对于每个 ∈ dom()且 ≠ ，有()=
()，并且()与()仅在方面不同；
2. （读取）两个状态在、在每个 ∈ dom()上的限制|，以及状态中没有纤程表名通过带到该状态的部
分上都一致，则映射它们到相同三个方面也一致。
当每次应用它及其每次迭代都受限于时，效应函数被称为限制在。

在适用第 4.3.2 节的情况下，要么注册一个组件（定义 47），要么其状态映射 pr1 及其生成的逆映射都被限制在
。每个 fiber 的效应函数都必须限制在该 fiber 内。 注册操作在其获取的名称上写入 O-Insert
条目，而不写入其他内容；其生成的逆操作 O-Retire 则在该名称的
上写入，而不写入其他内容。因此，无论哪种应用，都不会写入已存在 fiber 的控制字段，除了该
，而且不会读取任何内容。 条款 (2)
解释了为什么组件可以读取它声明的值：这些值位于其提供者的表中，因此一个不读取任何表但只读取
的效应函数无法使用其自身的协因素。它不能读取的是 之外的表或任何控制字段，这保证了组件不会基于其未声明的
fiber 的生命周期状态进行分支。

规则是非确定性的：多个纤程可能持有与其目标视图不同的已提交视图，并且该关系对它们之间的顺序没有任何承诺。它们
也是反应性的 32

只是，因为没有规则提到调度器；步骤是任何规则应用的序列，所以在所有此类序列上证明的定理对运行时可能采用的每种调度策略都成立。

4.3. 进行中的转换

本节在四种情境下扩展基本演算。第一种满足第3.2节的某些要求，而第4.2节无法表达，即在其依赖项可能占据的时间间隔上扩散的失活；另外三种情况放弃了转换是原子、立即和万无一失的理想化假设，而这些都不是实时运行时中的转换。被放弃的是整个转换作为一步，而不是一步是应用单条规则的一次操作，这四种情况具有一个共同的结构性后果，这里只说明一次：一个不是一步的转换在进行过程中需要一个状态来占据，对于它可能运行的每个方向各需要一个状态。

定义 49. 本节的生命周期状态通过如下方式替换 $\Theta\,\Gamma$：

$\Theta\,\Gamma$ 非活跃(ζ) | 重新加载(i,g,ω) | 活跃(g,ω) | 卸载(g,ω,ζ) (43)

其中 i : iter Γ 是剩余的效应迭代器（见下文定义 51），g : $\Gamma \to \Gamma$ 是迄今构建的累加器，ω : d →
是已提交的视图，ζ : {⊥} ∪ Ξ
是由卸载携带的结果，表示其停用的目标，由非活跃携带的结果表示其达到的目标，该结果要么是 ⊥，要么是第 4.3.4 节提供的错误集合 Ξ 中的一个错误。
当协程处于三个携带累加器和已提交视图的状态之一时，则认为已安装；当它携带错误结果时，则认为失败：
installed(γ) θ ≠ 非活跃(-), failed(γ) ξ ∈ Ξ. θ = 非活跃(ξ) (44)
当 ω(k) = m 时，已安装的协程 将 解析为 。定义 46 的静默状态在更宽的状态空间中被读取为：
quiet(γ) ∈ dom(Fγ).

ζ ≠ ⊥ ∨ 如果 θn = 非活跃（ζ）则 targetn（γ）= ⊥
如果 θn = 主动（−，ωn），则 targetn（γ）= ωn
⊥否则
(45)

第4.1节的定义适用于该状态空间，需固定两种读法。首先，
第4.2节的非活跃在O-Insert结尾中被解读为Inactive（⊥），在O-Insert中则称为Inactive（−）
在O-Remove的前提下。其次，σγ 仍然将有源纤程的表合并，因此
其跃迁在任一方向进行时，通过其所保持的ω读取其余效应应
且不提供任何自己的作品;其转换已经写出的键因此尚未出现
一个被依赖者可能会激活的对象。在两态演算中，这个区别是空的，每一个
光纤已安装，已激活。
图2描绘了这些状态形成的生命周期，下面的四个子节提供了
边缘的规则。
33

L-开始 L-结束
L-离开 L-卸载
L-迭代
L-转向
L-提升 非活跃 活跃
重新加载中
卸载中
图 2 | 生命周期及进行中的转换；两个转换状态已标出
4.3.1. 撤回
第 3.2 节要求依赖项需在其依赖之前激活，且依赖项只能在其依赖已停用后撤回其提供。前半部分在基础演算中已经成立：激活需要 ，因此声明 的纤程不能在某个纤程主动提供 之前激活。后半部分才是实质性内容，它必须提供的
不仅仅是状态变化的顺序。当组件因其提供者消失而被拆除时，它在运行自身的拆除代码，而这些代码可能需要正在被撤回的协作用；关闭连接池通常意味着移交连接

回到提供给它们的任何内容。下半部分必须传达的是，消费者在自身停用期间仍然可以读取，并且提供者撤销只能在此之后生效。基础演算根本无法实现这一点：它的L-Unload会同时移除提供内容并运行反操作，消费者的解构无法占据它们之间的间隔。该层将此步骤分为两部分，并通过以下条件保护后半部分。
定义50。当某个其他已安装的纤程解析某个键到纤程时，纤程在上被依赖：
relied() ∈ dom(), ∈ . ≠ ∧ installed() ∧ ()= (46)
= (,) target() ≠
[(,, ⊥)]
L-Leave
= (,,) ¬ relied() ()=
[()]
L-Unload
L-Leave记录了停用的决定，但并不立即执行，从而阻止了纤程……

在提供其副效应的同时保持其自身的已提交视图以及其他所有人的视图不变。L-Unload
应用累加器，丢弃已提交的视图，并将纤程 保留在它所携带的结果上；在第 4.3.4
节提供另一种情况之前，该结果为
⊥。它是演算中唯一应用累加器的规则。顺序的两个部分随后由形式的不同部分承载：可见性部分由已提交视图承载，而 L-Unload 将其作为最后一个动作丢弃；顺序部分由前提 ¬ relied() 承载，我们称之为守卫，该守卫保持
的撤回，直到解析它为 的每个消费者都完成。定理 63 确立了这两点。 34

该保护是按绑定而不是按纤程施加的：relied() 测试某些已提交视图是否命名为 ，因此声明不含
的任何键的纤程不会成为障碍，而已在其他域解析了 的键的纤程也不会（见第 3.2.3 节）。在第 4.2
节的单源纪律下，每绑定的理解与更粗略的测试 ≠，∈. installed() ∧ ∈
相一致，即一个键在那有一个可能的提供者。这类保护通常会导致死锁。使其避免死锁的是 以及
仅在 纤程上取并集：一旦 L-Leave 标记了 ，它的表格就会从 中移除，因此没有目标视图可以再命名
，且每个已提交到 的消费者本身也正在退出。定理 66
将其转化为保护总是被释放的论断。该保护沿协效而不是沿纤程树对失活进行排序：父节点可能

在其子对象仍在加载时运行其逆操作，因为 relied 仅涉及已提交的视图。因此，父对象和子对象的顺序比定理 63
对提供者及其消费者的排序更弱，并且当父对象和子对象的效应在环境状态中相遇时，它们受定义 60 的独立性假设约束。

4.3.2. 迭代

一次激活可能会依次执行多个效应，而停用必须恢复这些效应。我们使用效应迭代器来建模这种激活，每次迭代都会产生修改后的上下文、一个逆操作和一个连续操作：

定义 51. 定义效应迭代器 iter_**Γ** 和已见证效应迭代器 iter*_**Γ** 为以下递归类型：

iter_**Γ** . **Γ** → **Γ** × (**Γ** → **Γ**) × ()
iter*_**Γ** . (: **Γ** → **Γ** × (**Γ** → **Γ**) × ()) × ((: **Γ**) → ((, ,) = () ()
))
(47)

其中 () 产生一个三元组 (,,)，表示：

- 是新的上下文；
- 是当前效应的逆函数；
- 表示继续操作：

表示迭代终止；
() 提供下一次迭代。
见证在定义 33 的 处读取，正如定义 37 读取 的内容。
Γ：一个 ∈ iter
Γ 在 iter
Γ 中，当且仅当 遵守 且它产生的每个 都遵守 并且满足上述条款时。三元组按组件比较， 仅与
比较，而 () 与 (′) 比较，当且仅当 ′ 。
迭代器上的 是满足这些条款的最大关系。将 取为 **Γ** 上的相等关系，则可恢复严格读取。
效应迭代器转换 effectiter
Γ 将 effect**Γ** 扩展到迭代器结构通过递归调用：
定义 52. 定义效应迭代器转换 effectiter
Γ 如下：
35

effectiter
Γ : iter
$\Gamma \to \Gamma \to 2\Gamma$
effectiter
Γ = (,)
(,,)= ()
= track Γ (,pr1)

| ((,),)
| (′) (,)= effectiter
Γ (′)(,)
(,)
(48)
在每次迭代中，逆映射 会以应用顺序组合到 上，因此累加器 1 ⋯ 在应用时自然以后进先出 (LIFO)
的顺序恢复效应。因为 effectiter Γ 的结果落在与 effect Γ 相同的 $\Gamma \to 2\Gamma$
上，迭代器本身就是一种效应，可以在任何可以使用效应的地方使用。组件的整体激活就是这样的一种使用，这正是

本节的其余部分形式化了这一点，而实现允许在每个变异点使用迭代器（第5.1.1节）。(iter) 延续使得在任意
两个连续迭代之间可用一个边界，此时上下文是迄今为止迭代所形成的，并且累加器只恢复这些内容且不多于这些。从这个意义上说，效应迭代器是一种具化的定界延续，这是主流语言通过 yield
操作符[43]暴露出来的结构，因此该模型可以直接映射到它们已经提供的生成器。在微积分中，从这里开始，定义44中的
在 iter Γ 下读取，将原子效应函数替换为迭代器会将基本 L-Reload
分裂为一个追踪经过的已开始状态，并为线程提供了从该状态中出去的第二种方式。

= (⊥) = target() ≠ ⊥
[(,id Γ ,)]
L-开始
= (,,) target() ≠ (,)= (,id Γ) ∨ ()= (,,−)
[(,,⊥)]
L-转移
= (,,) target()= ()= (,,(′))
[(′ , ,)]
L-迭代
= (,,) target()= ()= (,,)
[(,)]
L-完成

每一次迭代都将新生成的逆函数按 的形式组合到累加器中，遵循定义 52，因此累加器以后进先出顺序应用这些逆
函数。在任意两次连续迭代之间，如果系统的目标视图发生了变化，它可能会偏离该转换，应用到目前为止累积的逆函数以恢复上下文。L-Divert 像其他停用操作一样通过
路由，而不是在当前位置应用累加器，它在这里遇到的守卫是空的，是一个从未 的纤程，既不提供任何内容
，也不出现在任何已提交视图中。它的两个备选项中的第一个会中止纤程正在执行的迭代，而这只有在迭代边界才有可能，因此操作的粒度在于

偏移可能发生在迭代器上；第二种情况允许该迭代落地，而第4.3.3节是其所需的位置。
一个普通的效应函数 (Γ) 是退化情况，其中第一次迭代已经产生 。这样的转换仍然会经过
，并且 L-Divert 在那里仍然适用，但累加器是 id Γ
并且没有运行任何迭代，因此不会恢复任何内容，并且转换要么安装它的所有效应，要么不安装任何效应。

4.3.3. 异步性
到目前为止，各层允许环境在一次迭代与下一次迭代之间移动，并假设每次迭代本身瞬间完成，其启动和结束被视为一步。我们抽象地对非即时性建模：一次迭代会生成类型为 () 的值，其中
是一个不透明类型构造，其定义属性是在提交与解决之间，外部状态可能发生变化。
在此模型下，一次迭代在一个状态下启动，在另一个状态下结束，而纤程在飞行过程中是 的。该层增
加的是惯性：一旦启动，迭代就会结束，其结束不能被拒绝。在飞行过程中转动的目标视图因此不能通过中止迭代来响应，只能通过 L-Divert 的替代方法让它落地。

可用：迭代到达终点后，纤程随后会停用。因此，这一层不会增加任何规则，也不会有规则匹配的类型；在 **Γ** 的粒度上，惯性就是它的全部内容，它以对主机可以采取 L-Divert 哪种备选项的限制形式存在。 这个备选项是基础演算无法表达的。在基础演算中，目标视图已经改变的转换会在发现它的同一步骤中被撤销；而在这里，正在进行的迭代必须先完成，所以在其逆操作运行时，纤程需要有个位置可以驻留，而唯一合理的地方就是 持有迭代产生的逆操作。通过
路由则会让纤程在一步的时间内提供其余效应应，并迫使其依赖项去

对已经正在离开的组件进行激活。这是在实现中重载和卸载的相互链式操作。停用也可能通过复合组件而非规则直接链回激活。L-卸载对目标视图没有前提条件，因此在纤程停用期间目标视图发生的任何变化，累加器都会运行并且纤程变为非活动状态，从该状态L-开始可以立即启动新的转换。 4.3.4. 失败 到目前为止，每条规则都假设其运行的效应会成功，而运行时不能保证。组件安装的效应会超出跟踪它们的上下文，它们所影响的对象可能会拒绝：端口已绑定、文件不存在、对等方未响应。失败的转换仍必须保证纤程的效应被恢复，而不是搁浅。

设 **Ξ** 为一组错误，并改进定义 51 的效应迭代器，使得一次迭代可以抛出而不是产出三元组：
fail
Γ . **Γ** → (**Ξ**, **Γ**× (**Γ** → **Γ**)× ())
fail
Γ . (: **Γ** → (**Ξ**, **Γ**× (**Γ** → **Γ**)× ()))
× ((: **Γ**)→ ((,,)= () ()))
(49)
该见证仅约束 情形，对于模式不匹配的情况无效，因为抛出没有需要撤销的，且 由
携带，从此处在 fail
Γ 中读取。定义 52 的提升保持不变，只是将三元组的地方替换为传播的抛出，因此一个抛出

迭代器在任何有效应的地方都可以使用，就像普通迭代器一样。该层增加了一条规则，并利用定义49的第二个结果，O-Remove不需要扩展即可允许它。L-Iter、L-Finish和L-Divert的前提都是在它们匹配的三元组周围读取。raise是迭代
所执行的操作，因此该规则是从的退出。37

= 重新加载(,,) ()= 左()
[卸载(,,)]
左提升
左提升在记录之前恢复。纤程进入卸载，携带错误作为其结果，累积到失败迭代的累加器在此应用，并且纤程到达未激活( )状态而没有安装任何内容，其状态与中止的左分流只能在纤程携带的结果中产生的状态不同。像处理每个其他停用一样路由失败，使得每个结果只能通过左卸载可达，这就是定理59建立的唯一事实。左开始以未激活(⊥)作为前提，因此生命周期是

没有因错误结果而重新进入；这是结果的实质，它会保留一个其效应函数在运行状态下已经显示不可靠的
fiber，而不是在未改变的环境下重试它。失败的 fiber
也不会阻塞任何东西：它是，因此它不携带任何已提交的视图，也无法使依赖成立。
失败会记录在 fiber
上，而不是传播到其父级，因此一个转换失败的组件会让其兄弟组件继续运行，这是插件主机所期望的行为，也是结果按 fiber 而不是整个状态属性进行处理的原因。
4.4. 元理论
第 4.3 节提供了十条规则：第 4.2 节的三条编排规则；用于激活的 L-Begin、L-Iter 和
L-Finish；用于激活可能结束的两种方式的 L-Divert 和 L-Raise

早期；以及用于停用的 L-Leave 和 L-Unload。本节从全局形式中读取这些规则的两个可组合性维度，一个线程的保证在其他线程之间执行的任何操作下都成立，并添加了只有整个系统才能被要求的内容：它总是达到其目标所要求的配置，并且该配置是静态组装所产生的配置。以下每个属性都是步骤序列的属性，因此我们对步骤进行索引，并根据该索引读取状态的字段。两个约定将第 3.3.2 节引入本节。以下每个状态之间的等式都按照定义 33 的观察等价 进行读取，就像引理 38 读取第 3.1 节的等式一样，并且效应函数所遵守的见证条件是定义 37 给出的。

按照定义51对迭代器的读取，以及在下文≈处对注册迭代的读取。
定义53. 用为步骤编号，使得是前个步骤达到的状态，并写作
step () (50)
表示在采取的步骤：它应用的规则（十种之一），以及它应用该规则的名称 ∈ 。序列从0开始，且dom(0) =
，因此每个纤程都是通过O-Insert出现的，无论是协调器的还是迭代采取的（定义47）。的某个字段会带有上标作为索引，因此、、、和分别表示在时的生命周期状态、已提交视图、表、累加器和剩
余迭代器，同时和表示登记表和

本身的余效应应上下文，如定义 45 中所述的 和 。谓词以状态作为它们的参数，其他的一切作为下标，所以
installed、target、relied 和 quiet 是定义 46、定义 49 和定义 50 在 的谓词。
的一个事件是索引的最大区间 [,]，在此区间内 installed 成立。它在 处打开，其中 > 0 且 ¬
installed−1，即初始没有任何纤程安装的空 0；它在 处关闭，当 installed 且 not installed+1
时，最终事件不必执行此操作。第 4.3 节的每条规则都以形如 [⋯] 结束，其中前提从 计算出 ，并作为
保留它，如果没有计算任何东西，而括号则编辑命名字段的 38。

注册表。这两部分被分别命名，并且都是对整个 **Γ** 的映射。由作用于 的规则在 处采取的一步的状态映射为
Ψ
{
pr1 在 L-Iter、L-Finish 和降落 L-Divert 处
在 L-Unload 处
id**Γ** 在所有其他规则处
}
(51)
其中 和 是迭代器和由 携带的累加器，而编辑 edit : **Γ** → **Γ**
是方括号表示的函数，将它所命名的字段赋值为前提在 处计算出的值。因此，它们都是由 step 与
一起固定的，并且在每个状态下都有定义，这就是定理 61 和引理 71 能够在远离
的地方对它们进行求值的原因。每一步可分解为
+1 = edit(**Ψ**()) (52)
例如，在 L-Unload，edit 为 [()]，而在 O-Remove，它是移除 。

这也是为什么后半部分是编辑而不是赋值。字段沿着相同的缝隙划分：表，一旦创建的O-Insert将其设置为空，没有
edit会写入它，以及控制字段，，，，，和dom()，除了通过定义47中的原语之外，没有
Ψ会写入。 当两个状态在控制字段之外的所有内容都一致时，写作 ≈ 。
关系≈不是定义33中的，两者也没有相互细化，因为每者都会遗忘对方必须保留的内容。恢复精确性是关于效应的声明，
因此≈比较表和环境状态完全一致，只遗忘注册表关于哪个fiber安装了它们的记录。规则读取控制字段以决定其是否适用，
所以必须保留

它们，本节将其理解为定义33与登记簿域以及每个纤程的每个控制字段一致性的连结：
∧ dom()= dom() ∧ ， ∈ {,,,,,}. (()) (())(53)
函数类型的字段，如 和 中的 ，被比较时如定义36比较映射，迭代器则如定义51比较两个迭代器，而其他类型
的字段按照相等性比较。下面的结果在两种关系下都成立，每个状态的一半各使用一种关系，Lemma 55 为十条规则中的
部分建立一次性结论。
表1是第4.3节的十条规则以此类写入方式理解。累加器、已提交视图以及剩余迭代器都是
的组成部分，因此第三列记录了对其的写入。

它们也是如此，而第四列的迭代的逆产生 他们的名字，id**Γ** ，其中 L-Divert 中止该迭代。由迭代器构建的 **Ψ**
注册一个纤程（定义 47）时，该注册会携带 O-Insert 行在其获取的名字上的写入，而累加器退役一个的 L-Unload 则携带
O-Retire 行的写入。下面的每个情况分析都是对表格的查找，并且五次查找足够频繁以至于需要命名。39

规则
+1
Ψ 控制字段已编辑
O-插入 未定义的 (⊥) id Γ dom()
O-退休 无约束 未更改 id Γ
O-移除 (−) 未定义 id Γ dom()
L-开始 (⊥) (,id Γ ,) id Γ
L-迭代 (,,) (′ , ,) pr1
L-完成 (,,) (,) pr1
L-转向 (,,) (,,⊥) id Γ 或 pr1

L-举起 (,,) (,,) id Γ
L-离开 (,) (,,⊥) id Γ
L-卸载 (,,) ()
表1 | 规则作用于纤程 的写入情况，其中 step 表示在 上应用的规则。
引理54. 将表1 与定义48结合，对于每个步骤 和所有在 中存在的纤程 ,:
1. 仅当步骤 作用于 时，+1
≠
，写入位于 Ψ 内；
2. 的存在仅在步骤 = L-Begin() 时产生，仅在步骤 = L-Unload() 时消失，因此在 的某一阶段中
保持不变；
3. Ψ =

仅在 step = L-Unload() 时适用，且没有其他步骤将 应用到该状态；
4. ¬ installed
∧ installed+1
step = L-Begin()，且 installed
∧ ¬ installed+1
step = L-Unload();
5. 、、 和 随着 的进入而产生，并且不再被写入，而 是单调的，仅在 时写入，并且只由
O-Retire 写入。
证明。令 step 在 上应用 。根据定义 53，它可分解为 edit Ψ，其中 edit 写入表 1
第五列所列字段且不写其他字段，而 Ψ 是 id Γ 、 的某次迭代的应用，或累加器
，这是这些迭代所得逆的组合。

根据定义48，这三者中的每一个都被限制在，因此Ψ不会写入存在的纤程的任何字段，只有会写入，以及注册
添加的条目和它的逆写入的。因此，这两部分划分了写入，每个子句是该划分在一个字段上的读取。第二列和第三列的一
种读取使用了两次：是没有提交视图的生命周期状态，L-Begin是从其引出的规则，而L-Unload是进入它
的规则，而其他每一行都将其前提的不变地携带到结论中。
(1) 一个edit不写入任何表，第五列没有命名任何表，而Ψ不会写入一个存在的 其中 ≠ 。因此 只能在 =
时移动，并且只能在Ψ内移动。

(2) 是 的一个组成部分，这只由 edit 写入，并且只在 fiber 上该步骤作用，因此根据上文的理解， 在 的
L-Begin 时出现，并在 的 L-Unload 时消失。 的一个片段是 installed 成立的时间间隔，因此是
被定义的整个时期，所以没有规则落在其内部。
(3) 第四列，其中累加器仅在 L-Unload 出现：其他规则采用前向映射 pr1 或 id Γ ，而且没有 edit
对状态应用任何映射。
(4) installed 是 ≠ (−)，根据上述理解，L-Begin 和 L-Unload 是唯一前提和结论在 是否为
上不同的规则。作用于某个 ≠ 的步骤…

不写 ，而登记添加的条目位于 中不存在的名字上。
40

(5) 第五列的任何一行都不会命名 、、 或 ；这些只有在条目 O-Insert
添加时才会出现，其结论会写出这些值，就像 O-Insert 执行注册时一样。只有 O-Retire 会在 上写出
，无论是由协调者执行还是作为注册的反向（定义 47）；O-Insert 会在尚不存在的名称上设置 =
⊥，所以没有步骤会将 返回为 ⊥。□
另外三个查找说明了规则无法看到的内容。首先，它们只通过上述观察读取状态，因此整个演算可以降至 Γ/ 。
引理 55。（-不变性）假设如上所述 ′ 。那么第 4.3 节的规则如果在 上作用于 时适用，则它在 ′
上作用于 时也适用，并且两次应用所达到的状态仍然通过 相关。

证明。第 4.3 节的每个前提属于四种类型之一，并且每个前提都读取关系保持的组成部分。一个前提将 或
与模式匹配，O-Remove 的前提 . ≠ 读取控制字段。O-Insert 的前提 (,,) ∈ Γ 和 . ∩ =
读取 、 和 。提到 target 或 relied 的前提读取 、 内的已提交视图，以及 dom()，定义 45 从
和 dom() 计算该域，而定义 33 仅在它们的域一致时关联两个 coeffect 上下文。其余前提读取
dom()。没有前提以除 之外的方式读取值 ()，因此没有前提会分隔两个 相关的状态。

对于结论，+1 = edit(Ψ()) 根据定义 53。edit
分配的值是它匹配的前提组成部分，通过上文段落以及定义 51，在两个状态下相关，这一定义关联了迭代器在
-相关状态下产生的三元组。而 Ψ 遵循 ：它是 idΓ ，或者是 的迭代，根据定义 51 要求其遵循 ，或者是
内的累加器，它是各个逆运算的组合，根据同一定义各自遵循 。□
一个状态携带的名字可以通过两种观察方式读取，分别是 dom()
和控制字段的索引，以及绘制名字的规则，该规则绘制任何尚未使用的名字（定义 47）。因此，在
下读取下面的结果也需要在重命名下读取，这是第 4.1 节中所阐述的规范。

引理 56.（协变性。）设 ：→ 是一个双射，并且设 是携带登记表 −1 的状态，其中每个出现在
或 中的名称都被替换为其像。则 是一个状态，在 是良构的地方也良构，并且 step = ()
当且仅当 (()) 将 传递到 +1 时，也会将 传递到 +1。
证明。一个前提仅通过将名称与另一个名称比较来读取名称，无论是直接，如 O-Insert 的新鲜度 dom() 和
O-Remove 的 . ≠ ，还是通过名称表，目标 target 和依赖 relied 读取 和
。双射保持每个这种比较。规则写入的唯一名称是 O-Insert 设置的 和 L-Begin 设置的 ，两者都

根据其前提内容得出，因此写操作与 交换；一个效应函数根本不写任何名称，仅通过定义 47 的原语来获取名称，而定义
48 将其限制在该原语添加的条目上。良构性（定义 58）有四个条件，用于比较名称与名称。□
因此，一个序列及其重命名遵循相同的规则并以相同的顺序到达仅在 上不同的状态。两个序列在其登记所涉及的名称上
有所不同，但其余部分相同，因此被认为是相同的，下面的结果按照将其识别的重命名来解读。
41

第二个查找是，剥离掉除 名称之外的一切条目的条目对规则是不可见的，这正是使定义 47
可以退役一个其恢复状态为空的纤程，以及引理 72 可以移除已删除事件所做注册的原因。
引理 57.（残余条目。）当 = ，= (⊥)，= ，且没有 的 = 时，称 在
为残余条目；一个残余条目满足 ≈ 。如果 在 为残余条目，那么对每条规则和每个 ≠ ：
1. 在 上作用于 的规则在 上作用于 也适用，并且两者达到的状态仅在
条目上不同，该条目仍然是残余的；
2. 反之，在 上作用于 的规则在 上也适用，除非它是抽取名称 或声称 的键的 O-Insert。

证明。一个残余的 对于没有作用于 ≠ 的规则的任何观察均不提供前提。它不是活跃的，因此 不进入任何
，且 不提供任何密钥，使得 和 target 不受影响；installed 失败，因此 对 relied
不贡献任何析取；没有 ′ 命名 ，因此 O-Remove 的前提 ′ . ′ ≠ 不受影响；并且 、 和
仅被作用于 的规则读取。条款 (2)
的两个前提除外，移除放宽了这两个前提，一个缺失的名称是新的，一个缺失的提供满足所有其他条件。根据引理
54，没有作用于 ≠ 的规则会写入 的字段，因此该条目得以存活，并且根据定义 48，这一步的状态映射受限于
，因此 为空。□

简化生命周期状态以及与之匹配的规则，会产生成一个子演算，并非每个结果都能在这种简化下保留。第4.3.1节被省略是重要的情况，这是第4.3节开始的划分，从元理论的角度来读：其守护式（guard）建立了定义58的第（3）条和第（4）条，而定理63依赖于守护式所创造的区间，所以没有它，这三者都会失效。其他三个小节所添加的内容可以在不扰乱以下结果的情况下简化掉，它们仅仅向定义49固定的单一状态空间添加规则。

4.4.1. 保持性

定义45固定了寄存器的形状，并且在下文的结果能够增加值之前，需要用规则去检查它。本小节识别了规则所保持的不变量，其内容是

第一个子句是那个形状，其余的是那些结果所假设的内容。
定义 58. 当且仅当对于所有 , ∈ dom() 和所有 ∈ ，注册表 是良构的：
1. ∈ dom() ∪ {} ;
2. ≠ ∩ = ;
3. installed() 在 上是全函数且取值在 dom() 中；
4. installed() ∧ ∈ ∧ ()= installed()。
子句 (1) 是定义 45 的树，一次读取一条边，同时保持游标指向注册表中的父节点。该定义还要求的无环性不需要子句，因为指针所指向的纤程在命名它的纤程之前已被注册。
定理 59.（保持性）如果 是良构的，那么无论步骤 应用哪条规则， +1 也都是良构的。每个子句在 +1
都可以从 的所有四个子句中建立。

证明。令步骤 作用于 。
(1) 根据表 1，只有 O-插入 和 O-删除 会写入 或 dom()。O-插入 以 ∈ dom( ) ∪ {}
作为前提，这是它添加的纤程的条款，并且它保持其他所有 不变 42

在扩大 dom() 时。O-Remove 对所有 满足 ≠ ，因此没有存活的 命名它移除的 fiber。

(2) O-Insert 的最后一个前提是 . ∩ = ，这是它添加的 fiber 的条款，并且根据表 1，没有其他规则写入
或扩大 dom()。下面使用两个推论：
dom() 根据定义 43，因此不同的表是互不相交的，并且 是一个函数；并且 ∈ ∩ ′ 强制 =
′ ，所以 至多有一个可能的提供者。

(3) 根据引理 54(2)，唯一写入 的规则是 L-Begin，其前提 = target ≠ ⊥ 使其在 上是全域的，并取值于
dom( )，命名目标提供者。根据表 1，唯一缩小 dom() 的规则是 O-Remove，其前提

= (−) 给出 ¬ installed ，因此根据条款 (4) 在 时，没有任何 拥有 ()= 对于 ∈
，同时 installed ；并且 本身也不携带任何 。
(4) 根据引理 54(2) 和 (4)，该条款在 +1 时只能在某些 installed 已被移除、某些 已被写入，或者某些 所命名的
fiber 已经离开 dom() 的情况下失效。最后一种情况是 O-Remove，其被移除的 fiber 没有安装，因此根据
时的条款 (4)，该 fiber 不被任何已安装 的 命名。第一种情况是 的 L-Unload，其前提 ¬ relied 表示
≠ , ∈ . installed ()≠
并且它不会为 ≠ 写任何 ，并且保持 ¬ installed+1 ，因此该条款同样对 成立。第二种情况是 的
L-Begin，写入 target

，其值是的键的提供者，因此在时处于状态；该步骤不会改变其他纤程的，所以它们在+1时
也被安装。□
L-Unload上的守卫承载了条款（3）和（4）。O-Remove的前提 . ≠ 仅涉及父指针；防止已提交视图引用被移
除纤程的是守卫，它早些步骤、出于不同原因被施加。由于故障也通过传递，因此该论证不必重复用于
错误结果。由此产生两个基础演算不具备的结论。O-Remove释放的名称可能被O-Insert重新使用，因为没有过时的已提交
视图可以引用它；并且一个纤程可能

在它变为不活跃（）时立即移除，而不进行单独检查是否有人依赖它。
4.4.2. 时序可组合性
局部时序可组合性使用一个累加器恢复一个效应序列（第3.1.3节）。注册表为每个纤程保存一个累加器，纤程交错执行：在
时刻将逆操作组合到上与时刻运行之间，其他纤程已经改变了状态。是否仍然撤销它被构建来撤销的操作，
是全局形式保证所断言的，并且它所依赖的条件是介入步骤与可交换。
定义60. 对于 ∈ iter_Γ ，令reach()为包含并在续步下封闭的最小迭代器集合，并通过在迭代器上取定义17中的
变换幺半群来读取。

对于它的生成器，每个迭代器在 reach() 中的正向映射以及生成的逆映射：
reach() { | ∈ ∧ ′ ∈ , ∈ Γ . ′ ()= (−,−,(″)) ″ ∈ }
() {pr1 ′ | ′ ∈ reach()} ∪ {pr2(′ ()) | ′ ∈ reach(), ∈ Γ}
(54)
在适用第 4.3.4 节的三元组周围读取 ，并写 len() 为在继续序下 reach() 的链 的 | |
的上确界。当两个迭代器，在定义 19
的意义下独立时，它们就是独立的，这一解释结合这些变换幺半群，并且一个迭代的 yield 是它的逆映射及其继续内容。

∈ (), ∈ ()。
′ ∈ reach(), ∈ (), ∈ **Γ**。pr2,3(′ (())) pr2,3(′ ()) (55)
并且在中对称，按定义36将应用于映射，按定义51将应用于延续，以及在注册迭代（定义47）中按其所命名组件的一致性应用。当对于每个 ≠ ′ ，和′相互独立时，迭代器族() ∈ 被称为两两独立；当() ∈ 是两两独立时，步骤序列被称为两两独立，其中是序列曾持有的名称集合，每个纤程由协调器插入，且每个迭代注册一个纤程。
这种意义上的独立性是踪迹理论作为原始概念所采用的：可交换动作生成

在序列上生成一种等价关系，其中重新排列两个相邻的独立操作会保持终点[44]，而引理71则说明这些规则的重新排列。一族而不是集合能够保持一个组件的两个名称在作用域内：那么条件要求该组件的效应函数独立于自身，也就是说要求() 是可交换的。第一个条件是定理61使用的，而第二个是定理73额外需要的：重新排列两个纤程的步骤会在另一个纤程移动过的状态下评估迭代器，而仅交换映射本身并不能说明迭代器在该处产生相同的逆和相同的续延。检查第一个条件不需要超出迭代本身，因为引理18(1)将生成元的可交换性传递到它们生成的单子。

在这些条件下，定理7的单累加器不变性在交错操作下仍然成立，其形式赋予时间可组合性具体内容：运行一个逆操作只撤销该纤程的贡献，而不影响其他。\n定理61.（恢复精确性）假设步骤序列两两独立，设在处开始的一个事件，且 ⩾ 位于该事件中，并且设1 < ⋯ < 为[,)中不由操作的纤程的索引。那么\n_() ≈ (**Ψ**_ ⋯ **Ψ**_1)() (56)\n也就是说，在处应用的累加器，将会得到，除控制字段外，由这些相同步骤从产生的状态 。将右侧视为从未开始时到达的状态，还假设在[,)中没有纤程的寄存器执行步骤，因为纤程的寄存器

是不存在以获取它的那一个。 证明。对索引 进行归纳，范围为剧集中 +1 的索引。在 = 时，−1 的步骤是 L-Begin，根据定义 53 剧集的开启，因此 = id**Γ** 根据表 1，索引集合为空，命题为 ≈ 。每步使用两个事实。由于 edit 仅写控制字段，+1 ≈ **Ψ**()，并且由于 () 中的每个映射除了注册添加的字段外不写控制字段，根据定义 48 结合定义 47，每个这样的映射会将 ≈-相等状态映射到 ≈-相等状态。 令步骤 作用于 。由于剧集在 和 +1 时处于开启状态，引理 54(4) 排除了 的 L-Begin 和 L-Unload，并且 O-Insert 和 O-Remove 读取了一个被 installed 否认的 ，剩下

两种情况。当规则是 L-Iter、L-Finish 或一个着陆 L-Divert 时，表 1 给出 **Ψ** = pr1

以及 +1
=
，对于迭代产生的逆 。定义 51 的见证条件为 (**Ψ**())= ，直到 ≈，这是迭代登记一个纤程（引理 57）的地方，并且
根据上述方程携带 ≈，所以
44

+1
(+1)≈ (
)(Ψ())=
()
其中规则是 L-Leave、L-Raise、取消的 L-Divert 或 的 O-Retire，表 1 给出
Ψ = id Γ 且 +1
=
，因此相同的方程在 = id Γ 时也成立。无论哪种方式，归纳假设都会保持索引集合不变，这就是定理 7
每次一步的计算方法。
让步骤 在 ≠ 上操作。那么根据表 1 +1
=
，并且 Ψ ∈ ()，或者 Ψ = id Γ ，如果规则是编排规则，则独立性给出

(+1)≈
(Ψ())= Ψ(
())
这就是附加 Ψ 后的归纳假设。□
推论 62.（终端恢复）设步骤序列两两独立，并且

一个在 b 打开并在 u 关闭的 的事件，无论 得出什么结果。然后，按照定理 61 所述，设 1 < ⋯ < ，
+1 ≈ (Ψ ⋯ Ψ1)() (57)
O-Remove 移除的纤程也不会留下任何东西，其前提仅允许 = (−)。
证明。根据引理 54(4)，步骤 u 是 的 L-Unload，其 Ψ 根据引理 54(3) 是
，因此 +1 ≈
()，而定理 61 适用。无论是陈述还是 ≈ 都未提及 ，根据表 1，
这是 L-Divert 和 L-Raise 导致状态不同的唯一区域。□
根据上述结果，组件之间假设成对独立，第 3.3.2 节则证明了这一点：
每个组件执行的每个效应都是一个密钥的操作，而

每个键都是可交换的，由这些操作构建的任意两个效应函数都是独立的（定理
42）。将该结果从效应函数推广到迭代器并不需要新的内容，协效应介导的效应函数（定义
41）已经根据每个阶段产生的结果选择接下来的操作，而这正是迭代器在其续体中承载的内容。第 3.2 节的协效应操作属于根本不需要假设的情况：组件贡献的映射是集合操作及相应限制的复合，每当它们涉及不相交的键时，这两个操作可交换，而定义 58 的第 (2) 条款使不同纤程的规定相互独立。
4.4.3. 空间可组合性
局部空间可组合性要求组件遵循其自身规范，仅在...时激活

在其依赖项被提供的地方开始，并对每一个上下文变化进行分类（第3.2.2节）。全局形式增加了对其他纤程进行量化的内容：提供者只有在每个解析它的依赖项都已停用之后才会撤回一个绑定，并且过渡安装其效应的解析在其下不会改变。两条协作用性质分别提供这两点，它们是一起证明的，是一个不变量的两半，即引理54(2)所确立的一个事件中的不变性。排序定理说明在事件中处于活动（Active）然后卸载（Unloading）阶段时这种不变性所带来的作用，而一致性定理说明在安装其效应的阶段这种不变性所带来的作用。定理63。（排序。）只有在其依赖项被提供的地方，纤程才开始一个过渡：
45

step = L-Begin()　(58)
设进一步的 [′ ,′] 是 的一个区间，且 ′
()= 对某些 ≠ 和 ∈ 成立，设 [,] 是包含 ′ 的 的区间，并且让 在 [′ ,′] 上取值。则
1.
()= ;
2. < ′ ，如果 [,] 关闭，则 ′ < ;
3. ∈ dom(
) 且
()= ′
()。
证明。第一个命题是 L-Begin 的前提 target
≠ ⊥，根据定义 46 得到 。
(1) 是引理 54(2)。
对于 (2)，在 ′ − 1 的 L-Begin 写入 ′
= target′ −1
，其值是提供者，因此 ′
= (−,−); 在 − 1 的 L-Begin 保留

= (−,−,−)，因此 ≠ ′ ，因此 < ′ ，两个剧集均由定义 53开始。设 [,] 闭合并假设
⩽ ′ 。则 ∈ [′ ,′]，因此 installed，并且根据 (1)，()= ；即 relied，而 处的 L-Unload
否认了这一点。因此 ′ < 。对于 (3)， 是 ′ 处 的提供者，因此 ∈ dom(′)。 的任何 L-Unload
都不在 [′ ,′] 内：在 [,] 闭合的位置，它发生在 > ′ （根据 (2)），而在未闭合的位置，引理 54(4) 表明
完全没有 L-Unload。由于 ′ = (−,−)，因此表 1 使得 L-Leave 成为 在 [′ ,′]
内可以执行的唯一规则，并且其 Ψ 是 id$_{\Gamma}$；根据引理 54(1)， 在该区间内保持不变。□

如果过渡分布在多个步骤中，否则可能会安装针对已在其下发生变化的分辨率计算的效应，而有两个前提防止了这种情况。L-Iter 和 L-Finish 携带 target()= ，因此过渡仅在其已提交的视图仍然是其目标视图时才会继续，而 L-Divert
携带其否定，因此目标视图的任何变化都会使纤程离开过渡。L-Raise 完全不依赖于目标视图，因为 raise 是迭代执行的操作，而不是环境请求的操作，并且无论如何都会退出过渡。变化的两个方向没有被区分：一个依赖消失的组件和一个依赖被替换的组件通过相同的路径离开，因为变为 ⊥ 的目标视图和变为其他纤程的目标视图对于 来说同样不等价。

惯性是阻止这成为有关每一步的保证的原因。一个已经在飞行中的迭代，当目标视图切换时，仍然会按照 L-Divert 着陆，而该着陆会安装一个针对不再成立的分辨率计算的效应。因此，规则提供的是一个析取式，而第二个分支是使第一个分支安全的原因。

定理 64。（分辨率一致性。）设 处开放的 的一个片段 [,] 在 时
= 。则 在该片段的初始区间 [,] 上是 (−,−,−)，并且过渡的每一次迭代都在同一个分辨率
下运行：
∈ [,]。如果 step ∈ {L-Iter(),L-Finish()} target
= (59)

当纤程离开该区间，使得 < 时，以下情况中恰有一种成立：
1. step = L-Finish() 且 +1

= (−,);
2. step ∈ {L-Divert(),L-Raise()}，并且在某个 > 时，该情节结束，并且 +1 ≈ (Ψ ⋯
Ψ1)()，如推论 62 所述。
证明。位于 −1 的 L-Begin 写入 ，根据表 1，它是进入该生命周期状态的唯一规则；其前提 =
(⊥)，并且根据引理 54(4)，任何第二次应用它都会发生在情节之外。因此 占据了
[,] 这个 [,] 的初始区间，并且不会再次进入。
46

第一个结论是前提 target()= ′ ，表 1 给出了 L-Iter 和 L-Finish，并且根据引理 54(2) 有 ′ = 。
对于二分法，step 是一个其前提具有 = (−,−,−) 而结论没有的规则，其中表 1 提供了
L-Finish、L-Divert 和 L-Raise；第一个落入 (−,)，另外两个落入 (−,,−)，由引理 54(4)
使 L-Unload 成为唯一出口，且推论 62 提供了方程。一个降落 L-Divert 的迭代贡献是
自身的，因此属于累加器撤回的映射之一。当 = 时，序列以仍在运行的转换结束，第一个结论就是所断言的全部。 □
4.4.4. 进展

延迟提供者撤回直到其从属项消失的守卫，仅在最终释放时才会证明定理 63。一个注册表纤程上的关系承载了该论证。
定义 65. 注册表名称上的优先关系为
∩ ≠ (60)
因此 可以提供 声明的密钥。它只读取 和 ，根据引理
54(5)，它们随着纤程条目的创建而存在，并且不会再次被写入。
定理 66 和定理 73 的建立基于 是无环的假设，这是一个假设，而不是定义所给出的内容，
适用于声明自己提供的密钥的组件。 排序的是两个纤程的激活，而不是它们的

生命周期： 表示 必须在 之前变为活跃状态，而提供者比其消费者存活的性质是定理
63(2)，这是关于受保护演算的定理。一个纤程的目标视图既对应于创建它的纤程，也对应于它的提供者。创建者写入的是
，通过定义 47 的原语，并且根据引理 54(5) 是单调的。因此，创建者在整个子纤程的存在期间最多可以改变一次其
子纤程的目标视图。进展性是某条规则适用的断言，因此它是在主机必须提供的规则上进行表述的：L-Begin、L-Leave、L-
Unload、着陆规则 L-Iter、L-Finish 和 L-Raise，以及 L-Divert。它在任何地方都不依赖于 L-Divert 的中止选项，因此受第
4.3.3 节惯性约束的主机也适用。

定理 66.（进展性）假设 无环，对于每个 有 len()⩽ ，并且定义 60 中名称集合
有限；且每一步都应用生命周期规则。记 () 为作用于 的步骤数，并且
() |{ : target
≠ target+1
}| (61)
为其目标视图发生变化的次数。那么
1.（无死锁）¬ quiet 意味着某条生命周期规则在 上可应用；
2.（终止性）()⩽ (+ 4)(()+ 1)，且 () 和 ∑() 都是有限的。
因此，每一个最大生命周期步骤序列最终都会进入静止状态。
证明。无死锁。设 ¬ quiet，则存在某个纤程 不满足定义 49 中静止状态的任何一条条款。根据表 1
对四种类型的对应情况，可以得出：
•
= (⊥) 且 target

≠ ⊥：L-开始适用；
47

•
= 重新加载(−,−,) 并且目标
= : L-Iter、L-Finish 和 L-Raise 中的哪一个 由
() 选择，就应用哪一个；
•
= 重新加载(−,−,) 并且目标
≠ : 如果
() 引发，则应用 L-Raise，否则应用 L-Divert，执行该迭代而不是中止它；
•
= 激活(−,) 并且目标
≠ : 应用 L-Leave。
不要让任何纤程属于这些类型，留下一些 0 使得
0 = 卸载(−,−,−)。按如下方式构造 0,1,⋯ : 给定 属于 卸载 的情况，要么 ¬ relied
，则应用 L-Unload

到 并且构造停止，或者存在 +1 ≠ 且存在 ，使得已安装的 +1 和 +1 () =
。在后一种情况下 ∈ +1 ∩ dom( ) +1 ∩ ，第二个成员资格是定理 63(3) 在
+1 所处的 情节中，因而 +1。此外 target +1 ≠ +1 : 一个卸载纤程位于定义
的并集之外，因此在 处 未提供或由除 之外的纤程提供。如果 +1 在活动 () 或重新加载
() 中，它将属于被排除的四种类型之一，所以它属于卸载 () 并且构造

继续。 是-递增的，因此根据无环性是不同的，并且 dom( ) 是有限的，所以构造终止。 终止。两个命题对
() 有界。 (A) 在目标 在 上保持不变的最大区间内，最多有 + 4 步作用于 。 阅读表 1 中 列，从
(−,) 并且 ≠ 时，纤程采取一个 L-离开 和一个 L-卸载，然后，如果 ≠ ⊥，采取一个 L-开始
和最多 len()⩽ 次着陆，加上第二个 L-卸载，其中最后一次着陆是 L-提升；从与 ≠ 对应的
出发，它采取 L-改道 代替 L-离开，而从任何其他状态则采用该序列的后缀。该区间内不会出现更多的
L-改道 或 L-离开，L-开始 写入的 即目标 = 本身。

并且在 (−,) 时，在 (⊥) 时 = ⊥，以及在 ()
时完全没有规则适用。
(B) 如果 target ≠ target+1 且步骤 作用于 ，那么要么 ，要么步骤 写入 。根据定义
46，target 的值是 以及 键提供者的表的函数；提供者满足 ∈ dom()∩ ，因此
，并且根据引理 54(1)，表仅在作用于自身纤程的步骤中改变。无环性在第一种情况下给出 ≠ ，而引理 54(5)
的单调性允许每个纤程的某一 出现第二种情况。
根据 (A) 间隔计数将 () 约束为 ()⩽ (+ 4)(()+ 1)，并且根据 (B) 每一轮

目标要么消耗一个严格在之下的纤程步骤，要么是所提供的唯一一次轮转，因此 () ⩽ 1 + ∑_{}
()。由于是无环的并且是有限的，递归关系 () (+ 4)(2 + ∑_{} ())
是良成立的，并定义了，使得 () ⩽ ()；因此 () 是有限的，并且 ∑_ () ⩽ ∑_ ()。
根据 (1) 无法扩展的序列是静止的。□
假设是有限的，而不是从中推导出来的，并且关于组件的一个条件可以保证它。有宿主持有的组件是有限数量的程序，在
任何操作之前就给定，因此如果没有组件能够以任何方式间接注册一个组件的纤程，而该组件又注册了自己的纤程，那么注
册形成一个有界深度的树，且 len() ⩽ 对其进行界定。

分支。假设排除了一个会无限注册自身实例的组件。48

目标记录提供的 fiber 而不是布尔值，并且在第 4.2
节的单源原则下，这两者驱动相同的转换，对于某个键，该键只有一个可能的提供者。视图获得的是上述结果的词汇，第 63 定理和第 64 定理都涉及针对 fiber 激活的解析，这也是这些结果能够在第 3.2.3 节的作用域解析下存活的原因，在该解析下，一个键在不同的领域会解析到不同的提供者，而这些提供不再强制视图。实现保留了这种作用域，并将视图保持在 fiber.committed（第 5.1.3 节）。

4.4.5. 会合性

到目前为止的结果关于单个 fiber。描述整个系统的特性是其动态历史不留痕迹：无论激活和停用的顺序如何……

一个运行系统经历的变化，它静止时的状态就是相同的插入和注销操作，如果每个最终处于活动状态的组件按依赖顺序加载一次，并且从未卸载过，将产生的状态。生命周期关系是会聚的，它收敛的正常形式就是静态组装的形式。这对于动态组合来说，是变更传播为增量计算建立的从零重新评估一致性的类比[45]。
该论点仅涉及本身。编排步骤是输入，两个序列给出不同的输入会落在不同的位置，这没有什么特别的原因；问题在于生命周期规则能否产生分歧，这些规则在下一个纤程步骤以及一个重新加载纤程退出时是非确定性的。

首先需要三个引理。第一个引理确定了哪些纤程最终会处于活跃状态，而不参考任何步骤序列，这使得它成为输入的函数，而不是调度的函数。
定义 67。当一个纤程没有退休、注册它的纤程被支持，并且它声明的每个键都由一个被支持的纤程提供时，该纤程在 上被支持。dom() 上的支持关系是这两条子句所读出的两个关系的并集，

$$\vee \quad = \quad (62)$$

并且当它是良基的（引理 68）时，我们用 表示支持集合，即在 上被支持的纤程：

$$\in \quad \neg \wedge (= \vee \quad \in) \wedge \quad \in . \in . \in \quad (63)$$

其中 = 标记了协调器插入的纤程，而在其他情况下 是激活登记为
的纤程。条款不读取任何字段，只有 、、、。两部分都将一个纤程与其正下方的纤程相关联，是父节点而不是祖先，并且是直接提供者而不是传递提供者，因为条款就是这样写的；当下面的结果需要顺序时，它们取传递闭包，其最小元素、最大元素和线性化就是 的最小元素、最大元素和线性化。条款引用了 本身，因此定义是沿
的递归，而使其具有解的正是以下内容。 引理 68。（支撑是良基的。）设 无环，且 通过一系列步到达。则
是良基的，并且 是定义 67 的唯一解，仅是 、 和 的函数。 49

证明。按每个名称被注册步骤的索引对 dom() 的名称进行排序，定义 53 通过从空注册表开始序列提供了该索引。
的父半部分在该索引中下降：一个 O-插入有 ∈ dom() 作为前提，因此父指针指向一个较早注册的纤程，迭代它可
以在有限步骤内遍历一个名称的整个祖先。因此，一个循环必须使用 ，由于 是无环的，它必须混合两者，这需要某个
声明一个键，而这个键信息可能由 自身子树的纤程提供。这样一个纤程是由 或
的某个后代的激活注册的，因此在 的 L-Begin 之后的某一步注册；该 L-Begin 有
作为前提，因此提供该键的纤程在此之前已经是活跃的，而且条款 (2)

根据定义58，没有第二个可能的提供者给出这个键。因此，那个本可以闭合循环的光纤从未被注册，并且该边不存在于
dom() 中。一个良基递归有一个解，并且子句仅读取四个字段。□
最后一个子句读取 ，即组件可能提供的键，而目标读取 dom()，即其光纤已安装的键，并且定义43仅通过
dom() 关联两者。因此，支持集通常会对 光纤进行过度近似，而闭合差距的条件如下。
定义69. 当组件 (,,) 的一个激活完成后安装了 中的每个键时，该组件在其提供上是完全的，因此在实例化它的每个
光纤上都有 dom()= 。

像独立性（定义 60）一样，这是仅对组件的条件，不涉及生命周期状态，也不涉及步骤，而独立性已经限定了它可能失败
的范围：如果一个组件仅在另一个组件效应能够触及的上下文状态下安装一个键，它的前向映射将不会与该组件的映射交换
，因此一个纤程安装的键是由其组件决定的，而不是由调度决定的。完全性所增加的是，这个固定集合是
的全部，而不是其真子集。

引理 70.（静止时的支持。）设 为无环关系，设 quiet()，且 中没有纤程失败，并且
的每个组件在其提供上是完全的（定义 69）。那么支持集是活跃纤程的集合：
= {: = 活跃(−,−)} (64)

证明。设 ′ 为右侧。不存在失败的纤程时，根据定义 49 的静默性，仅有 (⊥) 和
作为状态，并且有
∈ ′ target() ≠ ⊥
根据定义 46，当 ¬ 且每个 ∈ 都在 dom() 中时，右边成立，而且
dom() = ∈ ′ 根据定义 69。中间子句是目标不再携带的内容，而注册提供了它：只有当 ≠
的纤程被 的激活所注册时，才会被注册，并且如果 ′ ，那么 不是
，因此其累加器已运行并根据定义 47 使 退役，从而得到 。因此 ′ 满足定义 67
的各个条款，而引理 68 给出它们的唯一解，所以
= ′ 。□

引理 71。（换位。）设这些步骤两两独立，且 结构良好，并且设步骤 和 +1 作用于不同的纤程 和 。
1. 如果两者都应用激活规则，即 L-Begin、L-Iter 或 L-Finish，并且步骤 +1 在 上可应用，那么步骤 在步骤 +1 从
产生的状态上可应用，并且两种顺序将达到相同的 +2。50

2. 如果步骤 在 处应用一个激活规则，步骤 +1 在 处应用一个编排规则，并且步骤 没有注册
，那么两者同样成立。 证明。对于 (1)，根据表 1， 的步骤写入了 ，并且在 Ψ ∈ () 内，表写入了
和效应部分。因此它不影响 和 ，并且根据定义 60 的第二个条件，也不改变由
产生的逆和继续部分，因此只需检查步骤 +1 中提到 target 的前提。它的终止部分无法发生，因为没有激活规则写入
。它的解决部分也不能移动：步骤 +1 在 处可应用，将每个 ∈ 放入 dom()，且定义 58 的条款 (2)
使提供该 的纤程成为

唯一能够做到的，因此 ，并且没有 的写入会作用到 的键。同样的论证在另一个方向上也使步骤
可应用。最后 Ψ ∈ () 且 Ψ+1 ∈ () 根据定义 60 的第一个条件可交换，并且这两个编辑都写入不同 fiber
的控制字段，因此复合操作无论顺序如何都相同。
对于 (2) ，根据表 1，编排步骤有 Ψ+1 = id Γ，因此两个状态映射直接可交换，并且它的 edit+1 仅在 上写入
或 dom()，而激活步骤既不读取也不写入：后者的前提读取 、、 和 target，而对一个新的 进行
O-Insert 不会移动任何 target，新 fiber 不提供任何内容，而 O-Retire 或 O-Remove 的 则保持

在其所在的位置， 在一种情况下是非活跃的，在另一种情况下在其表中不受影响。因此步骤
保持适用。反过来，编排步骤的每个前提要么在 处读取，而步骤 不写入，要么是 O-Insert
的两个前提之一，一个较小的寄存器仅放宽了它，因此其在 +1 的适用性也保证了其在 的适用性；这里步骤
不注册 正是保持 在 中存在的原因，O-Retire 和 O-Remove 需要它。□
引理 72.（删除。）设步骤序列两两独立，设每个组件对其提供是完全的（定义 69），设其达到一个静止的
，此时没有纤程失败，设 [,] 是 的一个关闭的事件段，且没有任何与 的 的事件段关闭时在

序列，并且在 [,] 期间没有纤程 注册发生事件。用 表示这些注册所使用的名称。然后删除在 [,] 中作用于
的步骤，以及每一个作用于 名称的步骤，留下的步骤序列将达到一个在 之外与 ≈ 相等、并且在 之外
相等的状态。
证明。被删除的步骤会将状态保持在它们找到的状态。设 1 < ⋯ < 为 [,] 中作用于除 之外的纤程的步骤。推论
62 读作
+1 ≈ (Ψ ⋯ Ψ1)()
其右侧即为 [,] 中幸存步骤自身产生的效应，−1 ≈ 以及它们对除
之外的纤程控制字段的编辑，删除操作未涉及这些字段。根据表 1

被删除的步只写字段，Lemma 54(4)将其恢复为在处的(⊥)，没有纤程失败，并且在−1时也
保持如此。一个不变量携带后缀。写作′表示存活步骤在对应于的点达到的状态。我们声称，对于每个 > ，
≈ ′ ，的每个名称在处都是痕迹性存在并在′中不存在，并且两个状态在外的每个名称的每个字段上都一
致。当 = + 1时，这就是上文段落和Definition 47的结合，这使得每个由在时运行的累加器退役的名称为
(⊥)并持有一个空表，根据假设的纤程没有任何事件。归纳步骤是Lemma 57(1)在的每个名称上应用的结果。

依次进行：在 外部作用的一步在两个状态下具有相同的前提，再次到达近似相等的状态，并使 的条目保持残余。在
的名称上作用的一步是被删除的那些步骤之一，而引理 57(2) 解释了为什么它必须被删除而不是保留，O-退休或 O-移除
51

对于一个没有纤程可作用的缺席名称；通过（1）同样这样的一步不会将任何领域移出，因此舍弃它保持了不变量。因此最终状态是≈-相等的，并且在之外相等。没有剩余的步骤会丢失前提。一个作用于 ∪
{}的步骤仅通过target()或relied()读取。第一个依赖于，当声明了提供的一个键，因此
，并且当注册了，这使得 ∈
。在第一种情况下，根据假设的事件不会关闭，因此在时它是开放的，其中quiet给出 = target_，并且引理70将其值置于活动纤程之中，而不在其中；由于一个键最多只有一个可能的提供者，没有提供任何键。

在的L-Begin处的也不例外。第二个只通过的值读取，删除该步骤只能使依赖变为错误，这放松了对L-Unload的保护，而不是阻止它。这样的步骤读取的名字也被不变式覆盖。成对独立性是效应函数的一个性质，因此删除步骤会保留它。□
定理73。（合流性。）设一序列步骤到达一个静止的，在该状态下没有纤程失败，设这些步骤两两独立并且每个组件在其提供上是全的（定义69），设如定义67所示。那么
1.（规范形式。）是通过一系列按照原顺序执行相同编排步骤，从0到达的，最多涉及约减撤回的条目名称。

那些在每个生命周期步骤之前由协调器插入的 fiber，以及在注册该 fiber 后的步骤中每一个其余的 fiber，它针对其操作的 fiber 起作用，并且对于 的一个线性化顺序 的枚举 1,…,，它以该顺序取每个 的一次事件。
2.（汇合性）从 0 出发的任意两个这样的序列，若采取相同的协调步骤，在按引理 56 中的重命名之后，会达到通过 和 ≈ 相关的状态。
证明。(1) 序列中的事件分两类：在 时关闭的事件和仍然打开的事件，根据 quiet 和引理 70，它们分别是 每个 fiber 的一次事件。关闭的事件先进行，按其数量归纳。在每个阶段，选择一个关闭事件的 fiber ，在仍有关闭事件的 fiber 中它是 -最大的；根据引理 68，它必然存在。

以及 的有限性。引理 72 的三个假设随后被满足。没有任何 满足 拥有闭合的历程，这是由最大性决定的。而在 [,] 期间没有注册的纤程 也没有历程：这样的纤程会被在 时运行的累加器退役（定义 47），并且根据引理 54(5) 保持退役，因此它的目标视图是 ⊥，引理 70 将其置于 之外，因此它在 时没有打开的历程；且 通过其父指针将其与 关联，因此根据最大性它也没有闭合的历程。该引理移除了历程，以及它注册的名称的步骤，让 在那些名称完成之前保持原位。度量下降了一个单位，因此没有剩余的闭合历程。位于 之外的纤程不执行生命周期步骤。根据引理 70 和 quiet，它在 时也没有打开的历程。

现在没有任何结束步骤，所以它完全没有事件，并且在整个过程中都是(⊥)；L-Begin是那里唯一适用的规则，应用它会开启一个事件。接下来是编排步骤。由编排器插入的某个纤程的编排步骤通过引理71(2)向前移动一个位置，越过另一纤程的生命周期步骤，该引理适用，因为的一个纤程步骤不会注册这样的名称：注册会生成新名称，而这里的名称是原始序列中的O-Insert引入的。对于同一纤程的生命周期步骤，没有可以交换的内容，的O-Insert已经在每个步骤之前，而的O-Retire或O-Remove仅在之外应用，自身没有生命周期步骤。依次将每个步骤移到最前面可保持它们的相对顺序。某个激活的纤程上的编排步骤

注册的不能去前面，其前提要求存在光纤，所以它停留在注册放置它的位置；根据上面的段落，它在之外起作用，因此根据引理71的同一条款，与它和注册之间的所有事物交换。 52

通过对 | | 的归纳，剧集被排序并且连续。设 1 是 中 -最小的元素。则 1 = 且 1 = ，因为定义
67 将 1 的键的提供者和在 中注册 1 的 fiber 放入其中，而 将它们都置于 1 之下。因此 target1
不读取另一个 fiber 的任何字段，并且，没有剩余的编排步骤去写 1，也没有位于 1 之下的 fiber
去退休它，是恒定的。作用于 1 的每一步都是激活步骤，没有剧集关闭，其剩余前提读取 1 和 1，根据表 1 只有
1 写入；因此每一步在每个更早的状态都是可应用的，并且引理 71 将它向前移动一个位置而不移动终点。其他 fiber 在
1 步骤之前的步骤数在每次应用时减少一个，因此 1 的剧集

变成一个初始连续块。该论点在块之后的后缀上对 {1}重复，其中1在整个过程中处于活跃状态且不再采取进一步的
步骤，因此它也贡献了一个常数目标。这种枚举通过构造线性化了。
对于(2)，两个序列通过(1)都化简为一个规范序列，而两个化简在重命名前覆盖相同的。定义67中读取了、、和
，其中后三者在一个纤程的条目中只写一次（引理54(5)），所以必须看到的是相同的名字出现并携带相同的、和，
并且相同的名字被撤回。假设两个序列共享插入操作。它们也共享注册：一个

的 fiber 在每一次迭代中都会登记迭代器所命名的组件，而定义 60 的第二个条件在交错中保持固定，因此在 -fiber
下的注册树是该 fiber 组件的函数；这些注册所使用的名字不共享，正是在这里应用了引理 56，通过双射匹配两个树。一个
retire（退休/注销）要么是一个 orchestration 步骤，被共享，要么是累加器 O-Retire
执行的步骤，它注销的恰好是同一次激活注册的名字。两个线性化
的枚举之间只因不可比较事件的交换而不同，而引理 71
再次表明端点不会因此改变，因此两个规范序列是一致的。随着定理 66
的终结，生命周期关系因此具有唯一的标准形式。□

失败在该说明中被排除，因为它是真正的分歧来源，而微积分不应被理解为否认它：一个步骤是否提升取决于它运行的状态
，因此一个调度可能会使某个纤程失败，而另一个调度则完成它，此时两个静止状态在该纤程的生命周期状态上有所不同。
根据推论62，它们在其他任何方面都没有差异，该推论将失败纤程对状态的贡献视为零。
在第4.2节的基础微积分中，同样的定理成立，并且证明除了去掉一个条款之外不需要其他替代。L-Unload在此没有任何保
护，因此引理72的最后一段是空的；该引理的其余部分只依赖于quiet，而基础微积分对此保持不变。
该定理就是允许像对待静态程序一样推理Cordis应用程序的依据。

整体组装。一个添加组件、移除组件、替换提供者并恢复替换的协调器，保证能够达到通过最初写下最终组合状态所能得到
的状态，而组件作者在推理哪些副作用在作用域内时，可以仅仅考虑静止状态。它也界定了这种保证：它讲的是状态，而不
是系统在此过程中产生的输出，这正是第6.1节在边界内跟踪的获取与跨越边界的发射之间所作的区分。53

5. 实现与案例研究
本节介绍 Cordis，它将第 3 节的形式模型实现为实用的编程抽象。Cordis
是一个时空可组合性的元框架：不同于针对特定领域的应用框架（例如，网页路由、ORM、UI
渲染），它不规定具体场景；其唯一职责是提供通用的动态组合语义。该实现分为三层：（1）核心库（第 5.1
节）直接实现效应和协效应系统；（2）组件加载器（第 5.2
节）在核心功能基础上扩展了配置调和和热模块替换；以及（3）应用框架，如 Koishi（第 5.3
节），在前两层之上构建特定领域的功能。
5.1. 核心库

表 2 总结了理论构造与其运行时对应项之间的对应关系。具体来说，我们在本节中始终使用下面介绍的运行时名称，将理论符号保留用于形式对应。我们还使用 @@name 表示框架内部的符号键，因此 ctx[@@store]
中的括号表示对上下文中不透明槽的符号键访问，而不是对字符串键映射的索引。

理论（第3节，第4节） 实现
Γ∞ ctx，一等上下文
∈ Γ 上下文树及运行系统触碰到的所有内容
Γ, iter
Γ 效应回调 返回 / 产生逆操作
effectΓ () ctx.effect(callback)
Σ, Σiso, Σinter ctx[@@store], ctx[@@isolate], ctx[@@intercept]
get(), set(,) ctx.get(key), ctx.set(key, value)
isolate(,) ctx.isolate(key, realm)
intercept(,) ctx.intercept(key, metadata)
,,,,,, fiber，Γ中组件的实例化
dom() 通过ctx.registry枚举
：fiber.uid
：Γ fiber.inject
：Γ 组件提供的内容
：
Γ fiber.apply
：fiber.parent.fiber.uid，拥有它所实例化的上下文的fiber

派生实现（定义27） fiber.ctx，fiber运行的子上下文
（定义44） fiber.state，生命周期状态，其LOADING为，FAILED为()
recover，累加器 fiber.dispose，累加器
（定义44） fiber.committed，已提交视图
provider() 一个 Impl，其提供者 fiber 为 ACTIVE
target(,) fiber.target，通过刷新重新计算（算法5），其中 ⊥ 为 INACTIVE
，惯性（第4.3.3节） fiber.inertia，正在进行的过渡句柄
O-Insert，O-Retire（定义47） ctx.use 及其回调的逆操作（算法4）
O-Remove 从运行时中移除的fiber，并清空 uid
L-Begin，L-Iter，L-Finish 执行的迭代循环（算法1）

L-在迭代边界上使守卫失败 (算法 1)，或重新加载链入卸载
L-离开刷新，将纤程标记为卸载中 (第 10 行)
L-卸载卸载及其惯性链 (算法 5)
在 L-卸载卸载上守卫，等待已通知的依赖 (第 25 行)
L-引发记录在纤程上的错误，其目标设置为 ⊥
表 2 | 理论与实现的对应关系
本节剩余部分自下而上构建核心库。第 5.1.1 节实现可回退效应，这是通过其修改上下文的唯一原语；
第 5.1.2 节实现其上的反应型余效应应；第 5.1.3 节将两者组合到组件生命周期中；
第 5.1.4 节公开基于它们构建的上下文级操作。
55

5.1.1. 效应跟踪
本节实现可回退效应（第3.1节）。Cordis中的每一次上下文变更都通过单一原语 ctx.effect
进行：协作用提供、组件实例化以及每一个其他上下文变更操作都归结为一次 ctx.effect
调用，因此通过上下文执行的任何操作都会被自动跟踪，并在组件卸载时恢复。
从操作上讲，ctx.effect 是 effectiter Γ（定义52）的实现：它接受类型为 iter Γ 的回调，并将其提升到 iter
Γ，从而产生一个释放闭包，当调用时可以恢复该效应。Cordis 通过这一操作同时接受 Γ 和 iter
Γ（特设多态）；我们选择迭代器形式作为代表，因为普通效应函数是一个退化的迭代器，它只返回

单个逆操作。该操作没有检查的是 Γ 所携带的证据：回调提供一个逆操作，而该逆操作能恢复其伴随效应是组件作者的义务，而不是运行时验证的属性。定理 61 是演算调用它的地方，6.1 节则界定了该义务。

算法 1 展示了 ctx.effect 的构造。我们用 $\triangleleft$ 表示在 之后运行 的清理操作，用 id
表示无操作；因此，每次将新的逆操作添加到前端会产生后进先出（LIFO）恢复。

算法 1 效应追踪
1 异步函数 execute(callback, guard)
2 iter ← callback()
3 inverse ← id
4 当 guard() 为真时
5 (value, done) ← await iter.next()
6 如果 value 存在，则 inverse ← value $\triangleleft$ inverse
7 如果 done 为真，则中断
8 返回 inverse
9 函数 effect(ctx, callback)
10 armed ← true

11 任务 ← 执行(callback, () $\triangleleft$ armed)
12 异步函数 dispose()
13 如果未武装则返回
14 armed ← false
15 recover ← 等待任务
16 recover()
17 ctx.dispose ← dispose $\triangleleft$ ctx.dispose
18 返回 dispose
引擎 execute 以效应迭代器 (iter Γ, 定义 51) 的形式驱动回调，并将每步生成的逆操作折叠为单一复合操作。在每一步之前，它会检查调用者提供的保护；一旦保护触发，迭代停止，当前累积的逆操作将保留。这是第 4.3.2
节中的步骤边界中断：(iter) 的延续由迭代器的 done 标志和保护共同实现。
ctx.effect 是 execute 的一个轻量包装，增加了两点。首先，自我释放：保护报告 armed 标志，返回的 dispose 将 armed 置为 false，同时

中止任何正在进行的迭代，并使恢复最多触发一次。触发两次会在没有应用效应产生的状态下应用一个反向操作，在那里没有任何东西可以将其恢复到原状态。其次，父级组合：dispose 被添加到封闭上下文的 accu 前面 - 56

累积的逆 ctx.dispose，因此子效应的逆本身是对父级的效应，这就是 2Γ 的递归结构。组件级别（第 5.1.3 节）复用了相同的 execute，同时添加了一个守卫，用于测试 fiber.target 的稳定性，而不是 armed。

5.1.2. 共效操作

本节实现了反应性共效（第 3.2 节）。所有共效操作作用于每个上下文携带的三个符号键槽：

• @@store：值存储 ：(:)，从领域符号到类型值；
• @@isolate：领域表 ：Map(,)，从共效键到领域符号；
• @@intercept：拦截表 ：(:)→ ，为每个键分配其元数据。

前两个组合成两层解析 → ()→ (()) : ctx.get(key)

(算法 2) 从 @@isolate 读取领域符号 ()，然后从 @@store 读取绑定值 (())。
间接引用允许隔离将一个键重定向到独立绑定，而 @@intercept
仅在访问绑定时被咨询，调整其使用方式而不是它解析出的内容。我们将这些操作分为两部分实现：(1)
提供与通知，用于安装或撤销绑定并将变化传播给依赖者；(2) 隔离与拦截，用于重塑键的解析方式。 提供与通知。因为 set(,) 的类型是 Σ（第 3.1 节），系数提供是一次 ctx.effect 调用，并继承其自动跟踪与恢复。算法 2 实现了 ctx.set(key, value)，即具体的 set(,)：回调将在存储中将一个值绑定到

领域符号 ()，返回的释放函数会将其移除。安装和移除操作都会调用 notify，以将更改传播给依赖组件。
算法 2 共作用操作

```
1 函数 get(ctx, key)
2 realm ← ctx[@@isolate][key]  ()
3 返回 ctx[@@store][realm]  (())
4 函数 set(ctx, key, value)
5 函数 callback()
6 realm ← ctx[@@isolate][key]  ()
7 ctx[@@store][realm] ← value  [() ]
8 notify(ctx, [key])
9 返回函数()
10 删除 ctx[@@store][realm]   ()
11 notify(ctx, [key])
12 返回 ctx.effect(callback)
```

算法 3 通过测试每个活动 fiber 的 fiber.inject 中是否出现了更改的键且解析到相同的
realm，从而将每个绑定更改传播给依赖对象；如果是，它就调用

刷新（第5.1.3节）以根据新状态重新评估该 fiber，并返回其重新评估的
fibers，以便调用者可以等待它们。这是定义26的反应性分类：翻转满足条件的变化会激活或停用
fiber，而刷新操作的幂等性使中性变化无害。第5.1.3节中讨论了这种重新评估与各种控制流的交互。

算法 3 反应式通知
1 函数 notify(ctx, keys)
2 受影响 ←
3 对于 all_fibers 中的 fiber 执行
4 对于 keys 中的 key 执行
5 如果 key ∈ fiber.inject 且 fiber.ctx[@@isolate][key] = ctx[@@isolate][key] 则
6 刷新(fiber)
7 受影响 ← 受影响 ∪ {fiber}
8 中断
9 返回 受影响

一个绑定仅在安装它的 fiber 处于 ACTIVE
状态时才被视为对依赖方可用，因此刷新会针对处于活动状态的提供者解析每个声明的 key，而不仅仅是针对存储。
这是定义 46 中提供关系的体现，并且它使得依赖方能在撤销发生前一步看到变化：已经进入 UNLOADING
的提供者已经停止提供，因此其依赖方会重新计算一个未满足的目标视图，并在其绑定仍然存在的情况下开始自身的拆解。

隔离与拦截。这两个操作在结构上做的是同样的事情：每个操作都会派生出一个子上下文，用于调整一个继承的表的键，而不影响父上下文，因此恢复是隐式的：只需丢弃子上下文，无需运行显式的逆操作。ctx.isolate(key, realm) 用 realm
或默认生成的新符号覆盖域映射
（实现了隔离，定义 29），因此两个为同一键分配不同符号的上下文会形成独立绑定。ctx.intercept(key, metadata)
将元数据合并到拦截表
中（实现了拦截，定义 31）：根据该定义，新的元数据会与上下文当前对应键的已有内容合并，并优先于已有内容。
5.1.3. 组件生命周期

组件通过 ctx.use 被实例化为一个 fiber。本节将 fiber（在第 5.1 节中引入）赋予操作意义，即第 4.3.3
节中的惯性状态机。以下算法由两个字段驱动：fiber.parent，即形成组件层级的 fiber.ctx 的父上下文（$\mathbf{\Gamma}\infty$
的递归结构，第 3.3.1 节），以及 fiber.inertia，对飞行中的异步转换的句柄（如果空闲则为 null）。
算法 4 显示了组件实例化。组件将协作用规范 component.inject () 与效应函数 component.apply
配对；实例化将组件的配置绑定到 fiber.apply（第 9
行），即配置应用后的效应函数（），然后生命周期执行它。回调函数（第 2 行）是在父 fiber 中跟踪的效应：当

执行时，它通过调用 refresh（算法 5）来启动子组件的生命周期；恢复时，它将子组件的目标强制为 ⊥
并触发卸载。这是定义 47 的注册原语，其中 callback 作为其 O-Insert，闭包 callback 返回作为其
O-Retire：一个实例化是父组件的普通跟踪效应，因此卸载父组件会级联到其子组件。
算法 4 组件实例化
1 函数 use(ctx, component, config)
58

2 函数 callback()
3 refresh(fiber)
4 返回函数()
5 fiber.target ← ⊥
6 卸载(fiber)
7 fiber ← Fiber(parent: ctx, inject: component.inject)
8 fiber.ctx ← ctx[fiber fiber]
9 fiber.apply ← () component.apply(fiber.ctx, config)
10 ctx.effect(callback)
11 返回 fiber
算法 5 实现了第 4.3.3 节的惯性状态机，其中 reload 和 unload 是惯性的：一旦进入，转换会运行完成，然后系统才响应目标状态的变化。它使用了两个对共效存储的辅助查找：resolve(inject) 返回声明键当前解析到的绑定，provided(fiber) 返回该 Fiber 安装的键。refresh 函数从共效存储重新计算 fiber.target，并且如果 fiber 尚未处于转换中，则启动 reload 或 unload 任务2。

reload 函数记录当前目标并执行组件的 effect 函数 apply。
完成后，它会检查目标是否仍匹配：如果匹配，则 fiber 进入 ACTIVE；如果不匹配（无论新目标是 ⊥ 还是不同的提供者集合），则它会链入 unload。
对称地，unload 会按 LIFO 顺序恢复所有跟踪的 effect，然后要么进入 INACTIVE，要么链入 reload。
这种相互递归实现了惯性属性：一旦过渡开始，它会在任何新过渡开始前完成。
算法 5 组件生命周期
1 function refresh(fiber)
2 target ← target(,)
3 if target = fiber.target then return
4 fiber.target ← target
5 if fiber.inertia then return
6 if target ≠ ⊥ then
7 fiber.state ← LOADING
8 fiber.inertia ← create_task(reload(fiber))
9 else

10 fiber.state ← UNLOADING 在任何反向操作计划之前停用
11 fiber.inertia ← create_task(unload(fiber))
12 async function reload(fiber)
13 target0 ← fiber.target
14 fiber.committed ← resolve(fiber.inject) 提交视图
15 recover ← await execute(fiber.apply, () fiber.target = target0)
16 fiber.dispose ← recover fiber.dispose
17 if fiber.target = target0 then
18 fiber.state ← ACTIVE
2create_task 会调度一个异步函数以并发运行，并返回一个句柄（存储在 fiber.inertia 中）。
我们显式地写出来以保证语言无关性：使用急切调度（例如 TypeScript 的 Promise），调用是隐式的，返回的 Promise 就是句柄；而使用懒惰调度（例如 Python 协程、Rust futures），宿主必须生成任务，它才能继续执行。
59

19 通知(fiber.ctx, provided(fiber))
20 fiber.inertia ← null
21 否则
22 fiber.state ← 卸载中
23 fiber.inertia ← 创建任务(unload(fiber))
24 异步函数 unload(fiber)
25 等待所有(notify(fiber.ctx, provided(fiber)).map(f f.await())) 清空依赖者
26 等待 fiber.dispose()
27 fiber.dispose ← id
28 fiber.committed ← ⊥
29 如果 fiber.target = ⊥ 则
30 fiber.state ← 非活动
31 fiber.inertia ← null
32 否则
33 fiber.state ← 加载中
34 fiber.inertia ← 创建任务(reload(fiber))
fiber.target 通过将每个声明的键解析到当前的协效存储并将提供它的 fiber 的 uid 组合来计算，所以它是 target(,)
的摘要（定义 46）。
通过其提供者而不是其值来识别绑定，这就是单次比较的原因

对于记录的目标足够：UID
是新生成的且从不重复使用，因此被替换的提供者不会被误认为是它所替换的那个，即使两者提供的值相等。由于
notify（第 5.1.2 节）在每次协效变化时都会重新计算目标，当其声明的键之一被不同的 fiber 提供时，fiber 会精确地重新加载。因此，原地覆盖自身绑定的提供者不会被观察到；希望其替换被传播的组件会先撤销绑定，然后重新安装。该算法在两个互补的级别上运行。在转换级别，reload 和 unload
在完成时检查目标，从而实现跨转换的惯性链。在每次转换的迭代级别内，效应执行（算法 1）在每次

迭代边界，使得在单次转换中可以进行部分回滚。这两个机制分别对应于第4.3.3节中的转换间链式操作和定理64所依赖的转换内陈旧检查。三行承载了定理63的协效应排序，而它们各自的位置决定了排序的成立。reload
在第14行提交已解析的视图，而 unload 仅在每个逆操作运行完之后才丢弃它，因此一个 fiber
在被加载期间，包括它自身的拆解操作，都读取相同的绑定。refresh 在第10行将 fiber 标记为
UNLOADING，然后创建转换任务，这就是 L-Leave 步骤：fiber
停止提供服务，而依赖项在其任何逆操作调度之前会基于此重新计算。unload 接着在第25行等待每个已通知的

依赖于达到 INACTIVE，这是 L-卸载上的保护；只有当其声明的键解析到与提供者相同的领域符号时，通知才会承认一个依赖项，这是保护的需求在运行时的形式，要求依赖项看到来自此纤程的键，而不仅仅是声明它。等待位于整个恢复过程之前，而不是在被等待的某个逆变中，因为 fiber.dispose
同步启动一个纤程的效应，而如果等待放在其中一个效应中，其余的操作将无法排序。终止遵循定理 66：一个纤程只会等待那些已经无法满足的依赖项，而自己也是提供者的依赖项则以相同的方式等待自己的依赖项，因此提供者图是按需遍历的，而不是提前分析。

5.1.4. 上下文访问
第 5.1.2 节的共效操作形成了一个反射 API：共效通过 ctx.set(key, value) 写入，通过 ctx.get(key)
读取，均以名称为键。Cordis 在这个反射 API
之上增加了第二种、更本地化的上下文扩展和使用方式：属性访问。组件可以通过 ctx[key]
作为属性访问共效，就像它是上下文的本地结构一样，而不是通过方法调用。在 TypeScript 中，Cordis 使用一个 Proxy
来实现，每次属性访问都由其 get 捕获器进行处理。算法 6 展示了上下文如何在第 5.1.2 节的原始 get
之上解析对共效的访问。
算法 6 通过 Proxy 进行的上下文访问
1 函数 resolve(ctx, key)
2 fiber ← ctx.fiber
3 重复
4 如果 key ∈ fiber.committed 则返回 fiber.committed[key]
5 如果 key ∈ fiber.inject 则抛出 INACTIVE_ACCESS

6 如果 fiber = root 则抛出 UNDECLARED_ACCESS
7 fiber ← fiber.parent.fiber
算法 6 从访问上下文向上遍历 fiber 链：在第一个其已提交视图绑定了 key 的 fiber
时，访问被授权并返回该绑定；如果遍历到一个声明了 key 但尚未提交的 fiber，则该 fiber
未加载，访问失败；如果到达根节点仍未声明任何内容，则访问被拒绝，视为未声明。这就是代理与裸 ctx.get
不同之处：ctx.get(key) 是对存储的查找，返回绑定的值或空值，且永不失败，而代理则是针对访问 fiber
自己的视图进行解析，并在使用时强制执行 coeffect 规范 。读取视图而不是存储也是定理 63 的基础，因为它

是使一个组件仍能读取依赖的原因，即使该组件的拆卸是由该依赖消失触发的。
这种拒绝是访问点执行的运行时检查。因为组件的协程规范
是静态声明的，因此原则上可以在编译时检测到相同的违规，通过在执行前将每个 ctx[key] 与声明的 对应起来；第 6.4
节讨论了宿主语言的类型级依赖声明和编译时元编程如何恰当地实现这种调节。
5.2. 组件加载器
核心库为组件开发者提供了用于动态组合的命令式原语，例如 ctx.effect、ctx.use 和
ctx.set。对于应用协调者来说，另一个关注点出现，他们将预先存在的组件组装成运行系统并调整

在其生命周期内的组成。组件加载器通过引入声明式配置层来解决这一问题：协调器将所需的组成指定为持久化数据结构，
而加载器将对该规范的更改转换为相应的命令式纤程操作。61

5.2.1. 声明式配置
第4节将运行中的系统分解为纤程，每个纤程都是一个组件的实例化。
实例化所需的一切都可以声明，因此协调器可以将整个系统描述为一个声明式配置：这是一个持久化记录，加载器将其实现为纤程并与之保持一致。
条目。配置由条目组成。每个条目指定一个纤程并管理它，绑定是双向运行的：加载器会通过调整纤程来响应条目字段的变化，而修改自身配置或禁用自身的组件会将变化写回其条目。
定义74. 一个条目声明一个单独的纤程，记录：
• id — 一个稳定的标识符，当其组的子列表变化时用作调和键；
• url — 要实例化的组件模块的URL；

• isolate — 一个应用于条目上下文的隔离注解；
• intercept — 一个应用于条目上下文的拦截注解；
• config — 绑定到组件以形成其效应函数应用的配置；
• disabled — 条目是否在管理上被关闭。
一个条目可以作为忠实的规范，因为支持纤程的正是条目记录的内容。定义 67 的支持集合包含 、、 和 ，且别无其他，而条目提供了全部四项：disabled 提供 ，树中的条目父节点提供 ，url 选择声明 和 的组件。支持集合未读取的字段是纤程的运行时状态，实例化也不需要这些，且引理 70 确定了支持

与每个组件安装它声明的每个键（定义49和定义69）相关的静止状态的活跃纤程集合。这些条目形成了一个配置树，是系统加载内容的权威记录。一个条目可以是映射到单个纤程的叶节点，或者它的组件可能反过来加载更多组件，使该条目成为一个分支节点。Cordis 提供了用于这种分组和嵌套加载的组件：@cordisjs/group
以子条目列表作为其配置并将其作为子组加载，@cordisjs/include 加载外部配置文件（YAML 或 JSON）并将其条目嫁接为嵌套子树。两者都是基于定义47（算法4）的注册原语的普通组件，因此嵌套树保持在演算之内，以下结果对其同样适用。

对账。当一个条目的记录发生变化时，加载器会进行增量对账，而不是完全拆除 Fiber 并重新构建。以这种方式进行对账是合理的，原因由元理论提供。
• 定理 73 将静止状态仅视为最终配置的函数：无论加载器在途中执行了哪些实例化和注销操作，也无论顺序如何，系统最终静止在从零开始加载最终配置时会达到的状态。最终加载哪些组件，仅根据声明来判断，只要每个组件安装了它声明的所有键（定义 69）；一个在某些配置下声明并仅安装了某键的组件，仍然可以被加载器对账，但所加载组件的集合也对应于那些配置。

• 定理66证明了系统确实会静止，因此一旦发布了其实例化和注销，调和过程就完成了。
• 推论62将离开的纤程对状态的贡献置为零，因此重建一个条目会撤销其纤程安装的内容，并保持其周围的纤程状态不变。
62

• 定理63允许条目一起实例化，不需要协调器安排加载顺序：一个声明的键尚未提供的 fiber 会在其 L-Begin
处等待，而其提供者先离开的 fiber 会在它之前被停用。因此，依赖关系约束的是 fiber
激活的时机，而不是其模块被获取和评估的时机，因此加载器可以并发加载模块，而启动大型配置时主要耗费时间。
在条目声明的 fiber 之上，加载器会根据条目的哪些字段发生了变化来调度，并对每个字段应用最小干扰操作。
• id, url — 重新构建条目，因为其标识或其组件已更改；
• isolate — 重新分配条目的 realms（算法7）；
• intercept — 就地更新，因为拦截元数据在读取时会被参考，无需重新加载；

• config — 传递给组件，由组件决定如何应用新的负载，通常通过与之前的负载进行差异比较，并且仅在发生实质性变化时重新加载。特别地，@cordisjs/group 条目的 config 是其子条目的列表，因此它会根据子条目的 id 作为键差异来应用更新，创建、删除或更新每个子条目；由于更新存活的子条目会重新进入同一字段分发，组协调和条目更新会沿着树递归进行；
• disabled — 设置时卸载 fiber，清除时重新加载。
托管域。核心中的隔离在一个键（第 5.1.2 节）处派生了一个覆盖域表
的子上下文，这在上下文树保持静止时足够。条目可以在运行时在组之间移动，因此加载器管理自己的域，而 isolate

字段在每个键之间选择两种作用域规则。值为 true 表示请求一个本地域，该域对条目是私有的，并由条目携带的 ID 标记，无论条目移到哪里都会随身带着；字符串表示请求一个全局域，由每个使用该字符串命名的条目共享，因此移动这样的条目会改变它与哪些条目共享绑定，而不是它属于哪个域。一旦没有条目命名该域，该域就会被丢弃。重新分配条目的域会确定哪些键改变了域，条目本身是否是已改变键的提供者，以及需要通知哪些依赖项。中间的问题是最难的，因为一个域符号可能被几个纤程共享，其中只有一个是提供者。加载器通过分隔符回答这个问题：每个键对应一个符号
，每个上下文在其下存储自己的标签。分隔符写入上下文中，并由其后代继承，因此

条目的标签和提供者的标签完全一致，当两者在一个孤立作用域 内派生时，这就是在
处的绑定是条目自身且必须随其移动的情况。
算法 7 孤立领域重新分配
1 函数 patch_isolation(entry, ′)
2 ← entry.ctx[@@isolate]
3 store ← entry.ctx[@@store]
4 Δ ← {| ()≠ ′ ()} 领域发生变化的键
5 对于 属于 Δ do
6 entry.ctx[] ← 新标签
7 diff[] ← ((), ′ (), entry.ctx[], store[()].fiber.ctx[])
8 entry.ctx[@@isolate] ← ′
9 reload(entry.fiber)
10 对于 属于 Δ do
63

11 (1, 2, 1, 2) ← diff[]
12 如果 1 = 2 且 store[1] 存在且 store[2] 不存在 绑定是该条目的自身
13 store[2] ← store[1]
14 删除 store[1]
15 函数 affected(fiber,)
16 (1, 2, 1, 2) ← diff[]
17 返回 fiber.ctx[@@isolate][] ∈ {1, 2} 且 (fiber.ctx[] = 1) ≠ (2 = 1)
18 notify(entry.ctx, Δ, affected) 替代算法 3 的领域测试
该测试启用分隔符的一个属性。
下的标签写入条目的上下文，并被从它派生的每个上下文继承，而且在每次重新赋值时都会重新绘制，因此对于上下文 ′ 有:
′ [] = 1 ′ 是从条目的上下文派生的 (65)
对该条件写成 own(′)，其中 2 = 1 是提供者的实例。

重新分配将满足 own 的上下文从 1 移动到 2，并将其他上下文保持在原位，通过上述循环，它仅在提供者满足 own 时将绑定移动到 2。依赖方在其位于 的自身域中看到绑定，而该绑定所在的域正是它的位置。当 own 在依赖方和提供者上达成一致时，要么双方都移动，要么都不移动，因此依赖方之后看到绑定的情况与之前完全相同。当 own 将它们区分开时，一方移动而另一方保持原位，因此依赖方会获得或失去绑定。不平等性就是这种分离，而成员资格测试会移除在任一域中都无法解析 的依赖方，这是迁移操作无法覆盖的部分。

5.2.2. 热模块替换

热模块替换 (HMR) 在模块级别应用可回退效应模式：当

源文件发生变化时，通常在开发过程中，系统会在不中断进程的情况下就地替换受影响的模块。由于一个 fiber 已经绑定了其所有组件的副作用和协作用，作为组件的模块可以仅通过 fiber 操作进行替换：处理掉旧的 fiber 会回收组件安装的所有内容，而从重新加载的模块实例化的新 fiber 会重新安装它。因此，HMR 不需要开发者标注的接受边界，这与 Webpack [46] 或 Vite [47] HMR 不同。 @cordisjs/hmr 组件提供了 HMR 引擎，它分三个阶段操作。 阶段 1：模块分类。引擎接收两个输入：存储集（自上次重载以来内容已更改的文件 URL 集合）和外部模块集（不能被...

被热替换，而是触发完整重启）。为 url 直接导入的模块编写
get_imports(url)，它会对更改的依赖子图进行分类，标记每个模块为接受或拒绝：
算法 8 模块分类
1 函数 classify(stashed, externals)
2 accepted ← stashed
3 declined ← externals
4 pending ←
5 对 stashed 中的 url 进行处理
64

6 待处理 ← 待处理 ∪ (get_imports(url) (已接受 ∪ 已拒绝))
7 重复
8 进度 ← 假
9 对于待处理中的每个 url 执行
10 如果 get_imports(url) ∩ 已接受 ≠ 则
11 已接受 ← 已接受 ∪ {url}
12 待处理 ← 待处理 {url}
13 进度 ← 真
14 否则如果 get_imports(url) 已拒绝 则
15 已拒绝 ← 已拒绝 ∪ {url}
16 待处理 ← 待处理 {url}
17 进度 ← 真
18 否则
19 待处理 ← 待处理 ∪ (get_imports(url) (已接受 ∪ 已拒绝))
20 直到 进度 为假
21 已拒绝 ← 已拒绝 ∪ 待处理
22 返回 (已接受, 已拒绝)
基于缓存文件的导入初始化，这个不动点算法在模块的某个导入被接受时接受该模块，在其所有导入被拒绝时拒绝该模块；任何未决定的模块，如果陷入导入循环，默认为拒绝。

阶段 2：陈旧条目检测。
使用已接受和已拒绝条目，之后引擎将组件条目过滤为陈旧条目，即其依赖树到达已更改的模块的条目。它使用 get_dependencies 遍历每个条目的树，该函数在尊重拒绝边界的情况下收集模块的传递导入：
算法 9 陈旧条目检测
1 函数 get_dependencies(root, declined)
2 deps ←
3 函数 traverse(url)
4 如果 url ∈ deps 或 url ∈ declined 则返回
5 deps ← deps ∪ {url}
6 对于 get_imports(url) 中的每个 child 执行 traverse(child)
7 traverse(root)
8 返回 deps
9 函数 detect(entries, accepted, declined)
10 stale_entries ←
11 对于 entries 中的每个 entry 执行
12 tree ← get_dependencies(entry.url, declined)
13 如果 tree ∩ accepted ≠ 则
14 accepted ← accepted ∪ tree
15 stale_entries ← stale_entries ∪ {entry}
16 返回 stale_entries

当一个条目的树与已接受的树相交时，该条目就显得过时；然后该树被合并到已接受中，因此沿途的每个过时模块在下一阶段都会被作废。
65

第3阶段：事务性重载。最终，引擎会重新加载陈旧的条目。它会使已接受模块的缓存失效，并备份每个被移除的模块以便回滚，然后通过其 URL 重新导入每个陈旧条目的组件模块，并替换为新的 fiber：

算法10 事务性模块重载

```
1 function reload(ctx, accepted, stale_entries)
2 backup ← invalidate_caches(accepted)
3 try
4 for entry in stale_entries do
5 entry.fiber.dispose()
6 entry.fiber ← ctx.use(import(entry.url), entry.config)
7 catch error
8 restore_caches(backup)
9 for entry in stale_entries do
10 entry.fiber.dispose()
11 entry.fiber ← ctx.use(backup[entry.url], entry.config)
12 throw error
```

事务性保证确保系统永远不会进入半重载状态：如果任何模块导入失败（例如，由于语法错误），缓存将被恢复，每个

过时的条目从备份[entry.url]中重建，之前刚刚恢复缓存的组件会撤销已经进行的交换。

5.3. 案例研究：Koishi

Koishi 是一个基于 Cordis 4 构建的开源聊天机器人应用框架。在四年的开发过程中，它累计了超过 4000 个社区贡献的插件，涵盖从即时通讯 (IM) 适配器和数据库驱动，到管理控制台和终端用户功能。其规模和多样性使其成为在生产环境中验证 Cordis 动态可组合性的代表性案例。

元框架的表达能力和通用性。Koishi 作为服务器端机器人运行，其每个功能都是基于第 5.1 节中的上下文原语实现的插件；Koishi 本身只提供聊天机器人领域的词汇。相同的模型会在完全

不同的运行时：Koishi 的网页控制台是第二个独立的 Cordis 应用程序，其插件组成了浏览器及其用户界面的原语，而不是服务器的原语。以上不同的设置确立了第 3 节模型的两个属性。(1) 它具有表达性：其原语足以承载一个完整的生成系统，主机框架只提供领域词汇。(2) 它具有通用性：它固定了效应和余效应应的组合方式，同时将它们的意义留给每个应用程序，因此不预设特定的领域或特定的运行时。无需认知负荷的时间组合性。第 1.2.1 节中调研的插件系统无法在不重启扩展的情况下卸载单个扩展的效应。在 Node.js 上，这意味着必须清除 ES 模块和 CommonJS 模块系统的缓存，因为一个

通过 ES 加载器导入的模块可以出现在两者中。

4Koishi 目前使用 Cordis v3。本文介绍了 Cordis v4，它改进了效应和协效应语义，并重新设计了加载器；核心组合模型在两个版本中共享。

5Koishi 使用插件一词来表示本文形式化为组件的概念。

66

主机。Koishi 例行执行此操作：协调器从控制台禁用一个插件，其效应随之被撤销；在开发过程中，HMR 引擎在保存时重新应用已编辑的插件，同时保留系统其他部分的缓存状态和实时连接。Cordis 不仅使这种移除成为可能，而且使插件作者轻而易举地完成。由于通过上下文执行的效应会被跟踪，并且其逆效应会自动组合（第3.1节），即使是经验不足的作者也能为插件的上下文介导效应获得有序清理，而无需编写卸载路径。这实现了第1.2.1节所指出的关注点局部性：原本依赖每个作者勤勉完成的正确性，现在通过抽象一次性解决。

在开放生态系统中的空间可组合性。与第1.2.1节的插件系统相比，在那些系统中插件间几乎不存在依赖关系，Koishi的生态系统展示了真实的依赖拓扑：IM适配器提供对每个消息平台的访问，数据库驱动提供持久化存储，而功能插件将这些声明为余效应应并访问它们。在运行时重新配置一个提供者，例如切换存储后端或重新连接适配器，仅会重新激活其解析依赖发生变化的依赖者（第3.2节）；一个依赖不可用的插件将在依赖出现之前保持非激活状态，不会出错。案例研究证实了这一组合在独立创作的代码中也成立：一个插件及其依赖通常由不同作者编写，而...

仅在连接它们的余效应应上进行协调，因此反应性余效应应保持了开放生态中独立贡献者的程序集的一致性。
有效性威胁。本研究中的证据来自单一生态系统和单一主机语言，因此无法区分该范式的优点与其 TypeScript 实现或 Koishi 特定领域的优点，而且这是观察性研究，而非针对另一种架构的控制比较。因此，案例研究所建立的是一个存在与采用的结果，而非量化结果；对该抽象的开销以及其对开发者生产力的影响与基线进行测量仍然是未来的工作。
6. 讨论
前面章节中提出的形式模型和实现介绍了一种

用于动态组合的编程范式。本节考察了该范式如何扩展到更广泛的工程问题，并讨论了设计中的矛盾与未解决的问题。
6.1. 系统边界
第 3.1 节中的每个效应都具有逆效应，而该逆效应的具体内容由系统边界决定。该边界将系统运行的环境分为两部分。
(1) 当系统能够独占地修改某个位置并能够恢复修改前的状态时，该位置位于内部，因此对其操作会被记录在 Γ 中，并且可以在以后恢复。
(2) 当任何一种能力失败时，该位置位于外部，因此对其操作表现为 id_Γ，因此既不被跟踪也无法恢复。本节讨论了该边界的属性及其对恢复的影响。

来自余效应应的边界。余效应应通过具体化外部位置来移动边界：它将对该位置的每次访问限制在它提供的一组操作之内，每个操作都可以提供一个逆操作，因此那些作为 id_Γ 的操作会被记录在 Γ 中并被恢复。67

因此，边界是按位置而不是按媒介划定的，因为上述两种能力都是位置的属性，而具象化会改变访问位置的方式，同时保持其媒介不变。例如，当系统单独写入时，一个内存区域属于内部，当其他进程也写入时属于外部；当只有系统可以访问时，一个文件属于内部，例如位于私有路径下的临时文件，而当它是其他程序可以读取或写入的路径时，则属于外部。移动边界本身就是一种权衡，即环境是否为某个位置提供可恢复语义以及在每次访问时提供这些语义的成本。我们将在第6.7节讨论这种协同设计。
获取与发出。访问边界之外的操作通常

分两个阶段进行。(1) 在获取阶段，操作获得访问权限并在边界内安装一个记录：open 安装一个 close
移除的描述符，malloc 预留一个 free 释放的块，fork 启动一个 kill 终止的子进程。该记录本身是重新体现位置的副效应的一部分，例如，它在它维护的映射中的一个条目，而安装该条目是一个可逆的效应。该记录同时也是数据可以离开通道。(2) 在发送阶段，操作通过该通道推送数据，就像写入交给文件的字节或发送放上线的数据报一样，推送行为充当 id Γ ，将数据留在其他方可以读取和写入的位置。因此，这两个阶段位于边界的两侧：获取阶段保持在边界内部，而发送阶段则跨到外部。

扣留与补偿。一个必须从一次发射中恢复的系统有两种方式可供选择。一种是扣留发射，直到产生它的状态确定会持续存在，这就是回滚恢复的输出提交问题 [48]。另一种是补偿
[49]：一种将状态恢复到应用程序提供的等效状态的动作，这种等效比定义 33 的
更粗，例如删除已创建的文件或退还已收取的费用。这样的动作按与逆操作相同的后进先出（LIFO）顺序组合，因此第 3.1 节的组合方式也适用于它们。元理论则不适用：定义 60 的交换性是针对 证明的，需要针对更粗的等效重新建立。
6.2. 服务多路复用
动态组件平台如 OSGi [50] 围绕服务组织组合：单位

提供者在接口下发布的功能，消费者通过绑定使用它。Cordis 的协因模型呼应了这一概念，一个服务对应于键背后的接口。提供服务的组件是其提供者，而注入服务的组件是其消费者。单个服务可以由多个提供者实现，这种多重性可以通过两种形式实现。(1) 独占绑定：多个实现共享一个接口，但一次最多绑定一个实现；协调器选择绑定哪个实现，切换实现需要卸载一个提供者并加载另一个提供者，从而暂时扰动每个消费者的依赖。(2)
服务代理：一个充当接口入口的中央服务，由支持提供者和

消费者，使得多个提供者可以共存，并且代理会在它们之间分发每个请求。与独占绑定相比，代理吸收了这种干扰：更新一个后端提供者时，代理保持不变，因此消费者不会看到它们的依赖发生变化，也不会触发重新加载。
服务代理具备三项功能：负载均衡、滚动更新和跨进程调用。
负载均衡。当多个提供者共存时，代理会根据可配置策略（例如轮询、最少负载、延迟加权）在它们之间分配请求

由消费者明确指定的目标。由于提供者是普通组件，它们可以被添加或移除以上下调整容量；每个提供者通过可回退效应向代理注册，因此卸载它会还原注册并自动将其从代理的路由集合中移除。滚动更新。在运行时升级服务实现可以简化为受控的提供者转换 [51, 52]。为了执行转换，新提供者会以附加线程的形式加载并注册到代理；一旦它变为 ACTIVE，流量会逐步从旧提供者转移到新提供者（例如，通过调整选择权重），而旧提供者在不再承载正在进行的请求后会被卸载。这种提供者转换将传统上属于基础设施级操作（例如，容器编排，蓝-

绿色部署）到应用程序级组合模式中。
跨进程调用。服务中介也可以应用于跨进程边界 [53]。每个进程托管其自身的 Cordis
上下文和本地提供者；一个协调组件将它们连接起来，将每个视为远程提供者。跨进程服务访问通过 RPC 机制进行调解，该机制保持接口不变，使分布对消费者透明。需要注意的是，跨进程调用会产生延迟，并可能在执行过程中失败，因此同步暴露它会阻塞调用者。因此，旨在跨进程暴露的接口必须根据异步契约进行设计。
6.3. 访问控制和沙箱
鉴于一个由独立组件组装的应用程序，保护应用程序调用

用于两种互补机制：（1）限制组件可以访问的依赖项；（2）将不受信任的代码与宿主环境隔离。Cordis
通过依赖声明和拦截支持第一种机制；第二种机制则需要外部沙箱。
基于能力的访问控制。依赖访问机制（第5.1.4节）已经构成了一种对代理中介属性的访问控制形式：组件只能访问其声明的依赖项；未声明的访问将引发错误。这在结构上类似于基于能力的安全[54–56]，其中权限由持有引用授予，而非由环境权限授予。inject 声明充当能力请求，而上下文代理充当能力中介。由于这些请求

如果静态声明，一个组件所需的通过代理中介的完整能力集合在其运行前就已知，使得协调器可以在加载时审查和批准这些能力，而不是在访问发生时才发现它们。
这种中介通过拦截机制推广到精细粒度的策略。访问控制元数据可以由上下文携带或由组件声明（定义 30），在调用依赖项时，提供者会参考这些元数据来决定请求是否被允许。例如，文件系统依赖项可以携带声明组件可以读取或写入哪些路径的元数据，提供者会根据元数据检查每次调用。由于这种拦截存在于上下文中，而非任一方的代码中，协调器可以调整它，以在不修改提供者的情况下限制任何组件对依赖项的访问。

例如，授予社区组件只读数据库访问权限，而核心组件保持完全访问权限。此外，由于拦截只影响依赖项的调用方式，而不影响其是否被满足，它可以在运行时安装、重新配置或移除，而不会触发任何重新加载或扰动依赖关系图。

对不受信任的组件进行沙箱化。当一个组件的代码不可信时，仅靠语言级别的访问控制是不够的，因为具有访问主机运行时权限的恶意组件可以直接访问底层对象，使此类检查失效。沙箱化需要一个超出语言级手段范围的执行边界，例如软件故障隔离 [57]、独立的语言运行时、沙箱化进程或虚拟化容器
[58]。无论机制如何，不受信任的组件都在自己的沙箱化上下文中运行，并通过桥接访问主机提供的依赖项，这推广了第 6.2 节的跨进程调用：相同的透明性论点使这种桥接访问与本地注入无法区分。在主机端，该桥接是一个普通的纤程，其能力可以通过上述访问控制进行削弱。

6.4. 语言独立性与选择
尽管 Cordis 是用 TypeScript 实现的，但上下文范式与语言无关：时空可组合性仅由其两个可组合性维度定义，因此可以在任何在这两个维度上满足特定要求的语言中实现。我们将依次分析每个维度的这些要求。
时间可组合性。在最基本的层面上，时间可组合性要求闭包：可回退效应将一个动作与其逆操作配对，并且该逆操作必须作为一个值捕获，同时捕获它恢复的状态，以便在拆解时可以重放。除此之外，一个组件的代码及其加载时产生的副作用必须能够在运行时引入和撤销。
一种语言如何满足这个第二个要求取决于它的执行模型。在

在托管运行时中，这表现为一个程序化的模块注册表，其中加载的模块一旦没有引用，就可以从注册表中驱逐并进行垃圾回收；例如，Node.js
就提供了这样的注册表。原生代码没有模块注册表，因此引入和撤销表现为显式的动态链接和解除链接（例如，Unix 上的 dlopen/dlclose，Windows 上的
LoadLibrary/FreeLibrary）[59]，即将目标代码加载到运行中的进程中，然后再分离它。WebAssembly
根据其嵌入环境选择一种方式：在托管嵌入环境下（例如 JavaScript
主机），模块实例由主机的收集器回收，或者在原生嵌入环境下（例如
Wasmtime）被释放。在这些机制中，可回退效应模型将加载视为对上下文的一个效应，伴随

逆操作用于撤销模块引入的符号、类型或处理器的注册。
空间可组合性。空间可组合性要求组件能够声明其依赖项，并且运行时能够提供和注入这些依赖项。这可以归结为依赖注入（DI）问题 [38]，该问题在不同语言中以两种不同层面表现：依赖项的类型方式以及对其访问的调解方式。
在类型层面上，语言应提供一种方式让开发者表达类型良好的依赖项访问。消费者通过从上下文中读取其键来获取共效，因此上下文类型（第 3.2.1 节）必须记录每个键的共效。类型类（Haskell）[60] 和特征（Rust）[61]
通过允许提供者从其自身扩展上下文类型来实现这一点。

通过实例或 impl [62] 使用模块。TypeScript 的模块扩展 [63] 同样允许提供者模块将声明合并到上下文类型中。在运行时级别，依赖访问必须动态调节：键背后的余效应应应可能会随着提供者的加载和卸载而变化，并且在不同环境中可能会有不同的解析方式。6 CommonJS 通过 require.cache 暴露模块缓存；ES 模块没有公共的清除
API，尽管模块仍然可以通过引擎内部接口进行管理。70

上下文。因此，语言需要一种方式透明地干预访问，同时保持使用者的代码不变，例如通过 JavaScript 的 Proxy 对象 [64] 或 Python 的描述符协议 (__get__) [65]。如果缺少这样的原语，运行时反射 [66, 67]
可以动态地调节访问，但代价是类型安全性和开发者体验的下降。
在这两个层面上，元编程设施同时提供类型和调节。注解 [68]
和装饰器将元数据附加到声明上，处理器将其展开为调节访问的访问器；编译时元编程（例如 Rust 的过程宏、Scala 宏 [69]、Zig 的 comptime）为每个依赖项生成类型化的声明及这样的访问器，从而不需要通用的拦截原语。
6.5. 相互依赖与组件粒度

在反应性余效应应模型中，依赖循环只会使相关组件永久处于非活动状态：假设有两个组件和，如果需要提供的一个键，而又需要提供的一个键，那么两者的满足谓词都永远不会为真。与并发系统中的死锁不同，死锁依赖于调度并必须在其发生时检测，而这种情况仅从依赖声明就可以预测，因此运行时在加载组件时可以报告它。实际上，大多数表面上的相互依赖可以分解为更细粒度的组件，从而消除循环。考虑两个组件：一个服务器（提供网络接口）和一个访问控制器（执行授权策略）。这两个组件进行双向交互：访问控制器调节到达服务器的请求，并且

服务器暴露一个端点用于修改访问控制策略。单体设计会使每个组件依赖于其他组件。然而，这两种交互方向在逻辑上是独立的关注点。将它们分解可以得到四个组件：server-core、access-control-core、request-mediation（依赖两个核心以对传入请求实施访问控制）、以及policy-management（依赖两个核心通过服务器暴露策略修改）。通过这种方法，循环被消除，因为没有核心依赖于另一个核心；只有集成组件依赖于两个核心。
这种分解原则上总是可行的，因为每个双向交互都可以分解为独立的单向绑定，但它会增加组件的数量：在一般情况下，给定 个相互交互的组件，组件数量为

集成组件的数量可能会随着的增加呈二次增长，因为每对相互作用的组件可能需要为每个交互方向提供不同的组件。这不会影响正确性或运行时性能（组件是轻量级的），而更细的粒度可能是有益的：用户可以仅加载他们需要的特定集成绑定，从而有效提高系统的可组合性。然而，这可能会影响开发者体验：更多的组件意味着需要更多的配置、更多的命名，以及更多的认知负担来理解依赖关系图。
缓解这种粒度成本是工程问题而非理论问题。实际策略包括软件包打包（即将相关的细粒度组件分组为一个可安装单元）、基于约定的连接（即自动连接组件-

其名称或类型与某种模式匹配的组件），以及脚手架工具（即从声明性规范生成样板集成组件）。这些策略在保持无环模型形式保证的同时，将编写负担减轻到更接近单体情况的程度。

6.6. 依赖类型和版本控制
在形式模型中，依赖链接完全由键身份建立：提供键 的组件满足任何在其依赖集合中声明 的组件。类型族 确
保单个编译单元内的类型级一致性，但当组件独立开发和构建时，这一保证就会失效，而这种情况在组件生态系统中很常见
。这种失效导致两个不同的问题。
接口漂移。提供者可能会修改与
关联的接口（添加字段、更改方法签名、改变行为契约）在版本之间，而使用早期接口编译的消费者仍然声明相同的键
。依赖在协效层面上（ ∈ dom()）是满足的，但运行时值不再

符合消费者的期望，导致类型错误、方法未找到的失败，或行为静默偏差 [70]。
键冲突。两个独立开发的提供者可能使用相同的键名 来表示完全不相关的接口。由于仅通过键身份建立链接，期望一个
提供者接口的消费者会在没有任何兼容性检查的情况下接受另一个提供者的值。与接口漂移不同，在接口漂移中，提供者和
消费者至少共享一个共同的血统，键冲突则涉及期望类型和实际类型之间完全没有关系，使得由此产生的失败不可预测且难
以诊断。
这两个问题都指向同一个差距：余效应应模型只提供名义上的链接（通过键名），而没有版本化或结构化的链接（通过接口
兼容性） [71]。我们讨论

三种弥合差距的方法，从最依赖基础设施到最与语言无关。 关键命名空间。将键空间从 扩展到 × ，其中 标识定
义接口的包，通过构造消除键冲突：独立开发的接口即使具有相同的局部名称，也会占据不同的键。这是最直接的解决方案
，但也是耦合度最高的：它将包命名空间嵌入到正式模型中，使系统依赖于外部包注册表来确定键的身份。
对等依赖。较轻的耦合方式是通过主语言包管理器声明版本约束 [72]。这是 Cordis
当前采用的方法。组件依赖在语义上是对等依赖：组件不会打包其依赖

cies 内部管理，但期望运行时上下文提供它们。支持 peer 依赖的包管理器（例如
npm）可以强制版本兼容性：如果提供某个 key 的包的版本超出了消费者声明的 peer 范围，则不兼容性会在安装时被捕捉
，而不是在运行时出现错误。然而，这种方法有两个限制：（1）它依赖于提供者严格遵守语义版本控制，这是一种无法强
制执行的约定；（2）包管理器通常会将每个依赖项解析为单一版本，这会阻止在一个应用程序中加载同一包的多个版本组
件。 结构兼容性。一个完全与语言无关的方法将用兼容性谓词替换成员检查 ∈ dom()，该谓词验证提供者的实际

接口在结构上包含了消费者的期望。这类似于结构子类型[73]：如果提供的接口是所需接口的子类型，那么提供者就满足消
费者。挑战在于以语言无关的方式定义这个谓词：对于记录类型（宽度子类型），结构兼容性是直接的，但对于行为契约（
例如，前置/后置条件[74]、效应规范[22]）就变得复杂，并且一旦参数化多态引入有界量化[75]，则变得不可判定。72

这三种方法解决了问题的不同方面。设计一个统一的依赖模型，将这些方法结合起来，同时保留余效应应模型的动态组合保证，仍然是一个未解决的问题。

6.7. 与语言和操作系统的协同设计

第6.4节确定了主机语言为上下文范式必须提供的最小内容。本节讨论相反的问题，即与该范式协同设计的语言或操作系统可以超出该最小要求提供哪些功能。

与语言的协同设计。围绕上下文范式设计的语言可以在两个方面优于库：它给予上下文的语义，以及它给予效应和余效应应的原语。

这样的语言可以在保持上下文语义的同时，使上下文再次成为隐式的。

第3.3节。命令式语言已经对每条语句在隐式上下文中运行，并且该单一上下文既不跟踪效应也不解析协效应。上下文范式则区分多个上下文，其中一个操作要么修改它运行的上下文，要么从中派生另一个上下文（定义27）。原位实现会修改环境上下文，就像命令式语言那样。派生实现则引入一个单独的上下文，语言必须为此提供一个构造。使上下文隐式既带来便利性，也带来安全性好处。(1) 在库实现中，每个涉及效应或协效应的函数都以普通参数或接收者形式接收上下文，如第5.1节所示。当语言隐式提供上下文时，函数就不再需要

拿去吧。(2) 每个上下文都有其自身的生命周期状态和已提交视图（第4.1节）。库实现将上下文作为普通变量传递，因此组件可能会通过闭包或全局变量错误地访问另一个组件的上下文。它在那里安装的一个副作用随后会泄露出自身的生命周期，它在那里读取的一个因果效应则会逃逸其依赖规范。将上下文设置为隐式可以解决这两个问题。

这样的语言还可以让它的编译器知道副作用和因果效应。(1) 对于副作用，副作用迭代器（定义51）在每一步都分配一个闭包来保存逆效应以及它恢复的状态。通过执行副作用的语法，编译器可以为整个迭代生成一个单一的状态机，并在其帧中保存那些逆效应。(2) 对于因果效应，

协作用规格可以被纳入类型系统，有两个好处。首先，依赖循环会在编译时报告，而不是留到运行时（第6.5节）。其次，依赖可以通过其类型的结构而不是仅通过键的身份进行比较，就像行类型所做的那样[28]，这是第6.6节中结构兼容性的类型级支持。
与操作系统的协同设计。第1.2.3节观察到动态可组合性的粗粒度替代方案，其中操作系统在进程粒度上提供时间上的可组合性，而其上的容器协调器在服务粒度上提供空间上的可组合性。与该范式协同设计的操作系统将支持细粒度组合，通过使协作用规格成为组件声明的一部分。

它能够触及的全部范围，并通过提供自身资源作为余效应应应来实现。这样的操作系统可以提供第6.3节提到的沙箱机制，而该机制依赖于语言之外的机制。它通过将一个组件限制在其声明的依赖项上来实现，当组件被加载时提供这些依赖项，并且不允许从内部访问其他内容，就像 WebAssembly 模块在实例化时从其嵌入者处获取导入一样 [76]。它 73

还可以提供第 3.2.3 节中提到的共效隔离和拦截，作为其自身的能力，为每个组件以不同方式绑定一个键，并中介它所提供的访问。这样的操作系统也可以将其自身的资源提供为共效。在边界之外的资源被设为可回退状态，其中运行时会记录每次获取的资源与作出获取的组件（第 6.1 节）的对应关系，并且每个运行时都会保留自己的记录。一个将资源作为共效提供的操作系统只需要保留一次记录，因为它是发放资源的一方，可以将其归因于请求的组件。内存和文件描述符是最直接的候选对象，为了恢复而对其进行追踪已经在内核接口中实现过 [77, 78]。此外，操作系统可以

可逆性 6.1 节所述的一些操作只能被保留或补偿。一个以事务方式写入持久存储的系统可以回滚它
[79]，而一个基于写时复制或不可变存储的系统则可以通过移动指针到达早期状态 [80, 81]。
7. 相关工作
动态可组合性与几个已建立的研究领域相交。我们调查最相关的研究方向，并将我们的贡献与它们区分开来。
7.1. 效应和余效应应应系统
第 2 节回顾了效应和余效应应应作为我们工作理论基础的支柱。我们首先介绍现在在工业实践中常见的单子效应系统，然后调查三条扩展效应和余效应应应的研究线路，这些方向与 Cordis
相关：将代数效应重新解释为能力、赋予效应可逆语义以及统一效应和

在单一分级纪律下的协效应。
单子效应系统。一类库将效应编码在现有通用编程语言的类型系统中，将它们表示为运行时执行的单子值。Scala 中的 ZIO [82] 将计算建模为 ZIO[R,E,A]，TypeScript 中的 Effect-TS [83] 则为
Effect<A,E,R>，这是一个通用类型，其参数描述其结果、类型化错误以及上下文必须提供的服务；fp-ts 库 [84] 则通过基于 Reader 的单子转换器编码相同的错误和需求通道。这些系统与 Cordis
有两个区别。首先，跟踪是通过单子嵌入获得的：程序只有在写入效应类型内部时才能取得，而 Cordis
则将效应作为对普通宿主代码的覆盖来跟踪。其次，需求通过解释来清除，即通过安装的服务来履行。

为其操作提供资源，而当这种服务被撤回时，其操作所执行的内容仍然保留在原位；而 Cordis
则将每个效应与一个逆效应配对，并随着提供者的来去重新解决需求（第 3.1 节，第 3.2 节）。
代数效应作为能力。代数效应（第 2.1 节）使效应操作对类型系统可见。与我们工作最接近的扩展是 Brachthäuser 等人的 Effekt 语言，它将效应类型重新解释为能力 [85, 86]：效应类型表达的是计算从其上下文中所需的内容，而不是它可能产生的副作用。这个视角，像我们的工作一样，将上下文视为能力的中介。Cordis 和 Effekt 在两个方面有所不同。 (1)
目的上，代数效应使效应可见以实现模块化解释，从而提供了一个

操作许多处理程序语义，而 Cordis 使它们可见以便于跟踪和回溯，将每个上下文转换与其逆操作配对。(2)
在设置中，Effekt 在类型级别静态地约束效应，默认使用基于作用域的推理，其中能力是

二等和局限于它们的词法作用域，并通过装箱恢复一等使用，这种方式通过在类型中跟踪捕获的能力来解除该限制；而Cordis则在运行时规范副作用，旨在组件移除时实现完全的资源回收；第6.7节讨论了在这种意义上将上下文设为二等的语言会提供什么。可逆副作用语义。一条平行线为副作用赋予可逆语义，而不是解释性语义。Heunen等人[87]通过将Hughes的箭头适配为匕首箭头和逆箭头，在可逆设置中建模副作用，捕捉诸如序列化和可变存储等操作可逆的副作用。这是与我们的可回退副作用最接近的形式描述：两者都将每个副作用与撤销它的手段配对，而不是直接处理它。

通过一个处理器。两者的区别在于可逆性所在的位置，以及它们需要多少可逆性。Heunen 等人在一个指称的、范畴设置中工作，在那里可逆性是一种全局属性，由构造保证，因为每个计算都是可逆的，并且其逆是双向的，从范畴结构中可以恢复。Cordis 在运行时跟踪逆，并且对逆的要求较少：不是整个计算必须可逆，而是每个原子效应必须允许一个单向逆，由调用者在应用点提供，而不是从中派生，由此任何复合操作的逆通过组合得到（第 3.1 节）。

作为统一效应和协效应的分级类型。Orchard
等［88］提出了分级模态类型作为一个总括概念，涵盖了效应推理（通过分级单子）和协——

效应推理（通过分级余模态实现），在 Granule
语言中实现，展示了一个类型系统可以同时跟踪计算的行为和所需条件；最近的工作将余效应应扩展到类似 Java
的命令式语言 [89, 90] 以及按值调用-推送调用（call-by-push-value） [91]。所有这些都在类型层面上操作：效应和余效应应是静态注解，在编译时对词法固定的作用域进行检查。我们的贡献与此分析是正交的：我们将相同的两种概念提升到运行时机制，这使得 Cordis 可以处理动态组合。随着已加载组件集合的发展，时间回退和空间依赖会被重新解析，而不是在固定程序文本上一次性确定。

7.2. 编程范式

第 3.3.3 节将上下文范式确立为一种调节效应和余效应应的规范。

通过明确的上下文。两个已建立的范式值得明确比较：一个共享我们的术语，另一个共享我们处理横切关注点的方式。上下文导向编程（COP）。COP [92, 93] 为语言提供了层——部分方法和类定义，它们根据执行上下文在运行时被激活和停用，从而行为可以自适应，而基础代码无需命名其上下文依赖 [94]。COP 与 Cordis
都将上下文视为一等、可在运行时修改的实体，并且动态激活和停用行为，但这种相似性只是表面上的。在 COP 中，“上下文”指的是周围的执行情况（例如，位置、用户、模式），而激活会在动态作用域范围内改变方法分派；一个层既不跟踪

它所引起的副作用也不会被撤销，并且激活不受依赖满足的控制。在 Cordis 中，环境是 $\mathbf{\Gamma}\infty$
实体，用于调解副作用和余效应应应：激活会运行组件可撤销的副作用，并由响应式余效应应应的满足驱动（第 3.2
节），而停用会完全撤销它们。COP 变化的是运行何种行为；Cordis
组合并撤销组件安装的副作用和依赖。它们的区别在于权衡。COP
将激活折叠进宿主语言的方法分发中，从而获得动态作用域的层扩展，但代价是语言特定性；而 Cordis
作为一种语言无关的覆盖层，通过共享上下文对激活进行反应性解决。因此，Cordis 可以仅将其表达为余效应应应 75

COP的全球性、价值驱动的片段：在实现之间进行上下文依赖的选择，但不是动态作用域的激活。 面向方面的编程。AOP [95, 96] 将横切关注点模块化为一个方面：一个在基程序中选择连接点的切入点，以及在每个连接点编织的通知。Cordis 解决了相同的上下文行为问题，否则这些行为会分散在各个组件中，但它的方面类比是一个余效应应：一个多个组件声明依赖的共享媒介点，从而可以在不修改任何组件的情况下重塑横切行为。这两种范式在两个方面有所不同。(1)
声明性与无感知性：AOP
的切入点是无感知并被量化的，匹配任意其代码未意识到被通知的连接点，而Cordis将横切行为限制在每个

组件声明的，因此它的作用范围正好就是声明的表面。这产生了确定性和可追踪性：应用协调器可以在配置层检查并管理横跨组件的内容，而无需读取或分析其源代码，而 AOP 关注点只能通过量化其上的切面来理解。(2) 生命周期集成：Cordis 中的横切变更由组件的效应携带，在组件卸载时恢复，并反应性地传播到其依赖者，因此它是在动态组合模型中的一次移动；动态 AOP 系统 [97, 98]
也可以在运行时进行织入和拆解，但作为独立操作，它既不绑定于组件的生命周期，也不触发被通知代码的重新解析。

7.3 时间可组合性
时间可组合性涉及在运行程序中替换或移除组件

同时恢复它安装的效应。以往的方法根据它们如何处理即将离开的组件的状态和效应进行分类：将状态传递到后续版本、通过开发者编写的清理来恢复效应、在预先固定的范围内自动逆转效应，或者通过在接口上进行插入，从运行时累积的记录中回收资源。
有状态的前向迁移。一类广泛的系统通过在运行程序中跨版本携带组件状态来替换组件，从而实现无停机操作。所有系统都遵循相同的时间纪律：组件只有在达到安全、无交互的点时才能被替换。Kramer 和 Magee 将这一标准确定为静止状态 [51]，Vandewoude 等人随后将其放宽为影响较小的平静状态 [52]；我们的滚动更新模式（第 6.2 节）

通过在卸载提供者之前清空正在进行的请求来强制执行它。动态软件更新 (DSU)
然后通过手写的转换函数将状态向前迁移：Hicks 等人的 C 通用 DSU [99]、Stoyle
等人通过共自由性分析实现的类型安全更新点 [100] 以及 Hayden 等人的 Kitsune [101] 都将旧版本数据映射到新版本表示，继承堆对象、打开的文件和连接，同时重新初始化未迁移的部分。相同的方法也延伸到持久状态：Overeem 等人 [102] 通过手写的升级操作在保持系统可用的情况下，将运行中的事件存储数据在不同的模式版本之间转换。Erlang/OTP [15] 在进程级别采取相同立场，通过 code_change/3 迁移状态并恢复

通过重新启动受监督的进程而不是恢复其效应来修复错误；JavaScript 的热模块替换（例如，webpack [46]、Vite [47]）在模块级别也执行同样的操作，通过 module.hot 或 import.meta.hot API 在重新加载时传递状态。与 Cordis 的模块替换（第 5.2 节）相比，这些方法更优雅地迁移内存中的状态：Cordis
会撤销旧组件的跟踪效应，并从干净的状态重新应用新组件，因此组件自身的内存状态无法保留。

除非放置在生命周期更长的依赖中，否则会重新加载，并且在可还原效应之上进行 DSU
风格的前向迁移是未来的工作。尽管如此，Cordis 的方法在两个方面更为通用：它不需要 DSU 和 HMR
所需的手写迁移函数，并且支持完全卸载组件并回收其资源，而不仅仅是就地更新组件。
开发者编写的恢复。第二类通过开发者手写的清理或补偿逻辑来恢复组件的效应。插件生命周期约定（例如 OSGi
[50]、Eclipse 的扩展点、IntelliJ 和 VSCode）将清理工作委托给开发者编写的卸载回调；命令模式 [103]
将操作与用于撤销/重做栈的撤销方法封装在一起；Saga 模型 [49] 架构了一个长期运行的事务

每个步骤都配有一个补偿操作；代数效应处理器可以附加在拆除时运行的终结器 [104]；事件溯源 [105] 则通过附加补偿事件而不是执行逆操作来撤销状态。在所有这些方法中，逆操作都是一种非强制性的职责，与操作解耦，因此被遗忘的逆操作会悄悄泄露资源（如第 1.2.1 节的实证所述）。React 的 useEffect 钩子 [106] 最接近于在结构上将效应与其逆操作配对，它返回一个在每次重新执行前和卸载时由运行时调用的清理函数。其不足之处在于可组合性：钩子只能在组件或另一个钩子的顶层调用，不能在条件语句、循环或嵌套函数中调用，且其效应主体既不接受异步函数也不接受迭代器。因此，效应无法被组装

来自其他效应或与控制流交错，不留下任何可以从中得出复合逆的东西。Cordis 效应没有这种限制：它们是可以自由组合并可能异步运行的普通操作，并且只需要为每个原子效应手工编写逆，从而可以通过组合得出任何复合的逆，因此组装现有效应根本不需要编写任何逆。每个效应与其逆的这种结构配对使得系统的完整恢复成为系统的不变量，而不是开发者的纪律问题。
静态作用域反转。第三类效应通过构造自动反转效应，但将反转限制在预先固定的作用域内。软件事务内存 [107,
108]，源自硬件事务内存 [109]，记录读/写日志，以便

一组内存操作要么提交，要么中止，将内存回滚到事务前的状态。可逆计算，从Landauer和Bennett的热力学分析[110, 111]到可逆语言如Janus[112]，更进一步，使整个计算的每一步在全局上都是可逆的。可逆过程演算将回溯内置于语义本身：RCCS[113]在每个进程旁携带一个内存，并允许在过去导致的状态因果等价时回退一个步骤，而Phillips和Ulidowski[114]为CCS、ACP和CSP一致地推导出可逆操作符，同时保留它们的正向操作语义。他们的因果一致性准则是Cordis恢复遵循顺序的并发对应物，一个累加器在最后应用组件自身的逆操作。

先进先出顺序以及第 4.3.1 节的保护机制延迟提供者的撤回，直到其消费者已停用（定理
63）。然而，可及性由语义固定，每个执行的操作仍然可以撤销，而 Cordis
组件为每个原子效应提供一个逆操作，其累加器将上下文恢复到其组合开始的位置。线性类型 [115]、RAII [4] 和 Rust
的所有权系统 [61] 将资源的释放与词法作用域绑定。它们各自静态地固定了撤销的范围和可及性；相比之下，Cordis 并不提前固定这样的范围：它在组件生命周期内撤销任意上下文操作，并将词法资源管理视为补充，适用于单一组件中的局部资源。

介入式回收。第四类回收方法是当组件自身没有提供逆操作时，由另一个家庭记录组件获取的资源，在运行时控制的接口上进行回收。Nooks [77] 对每个跨越 Linux 内核及其可加载扩展的调用进行封装，使得扩展触及的内核对象都通过对象跟踪器，对象跟踪器的记录告诉恢复管理器在扩展失败时释放哪些资源；影子驱动 [78]
从另一侧获取同样的调用，记录决定驱动状态的请求和配置，以便可以恢复重启后的实例。Akeso [116]
则通过编译器插桩获取记录，将内核执行划分为可嵌套的恢复域，记录其状态变化和跨线程依赖，并

将一个出错的请求与依赖它的每个域一起回滚。因此，回收是基于运行时维护的记录，而不是开发者记得编写的清理操作，这使得这一系列成为最接近可回滚效应的系统级先例。它在词汇和影响范围上与 Cordis 不同。平台固定了可以记录的内容，无论是作为每种内核对象类型的发布代码、每个驱动类的一个影子，还是每个受监控分配器的逆操作，因此一个组件可能只持有平台已经知道如何释放的资源；而 Cordis 组件则会引入自己的效应，并为每个原子操作提供一个逆操作（第 3.1 节）。回收同样受限于提交请求或同一扩展的重启，而 Cordis 会在组件的整个生命周期内进行回滚。

并将删除传播到其依赖项，这些依赖项则依次释放它们自己的影响（第3.2节）。
7.4. 空间可组合性
空间可组合性涉及组件如何声明和绑定对其他组件的依赖。之前的机制根据绑定对变化的响应方式进行区分：在初始化时对依赖进行连接、对整个组件的可用性作出响应，或者以单个值的粒度传播变化。
初始化时的依赖连接。两种已建立的机制在初始化时将组件连接在一起。依赖注入框架 [38]（例如 Spring [117]、Guice、Angular、Inversify）在初始化时将依赖注入到组件中，而UI框架上下文（例如 Vue.js 的 provide/inject 和 React 的 Context API）将其传递给组件

树。一些支持动态作用域（例如，Spring 的 prototype/request 作用域，Angular 的层级注入器），但都不进行响应式的重新解析：当运行时替换或移除提供者时，现有的依赖项既不会被停用，也不会被重新初始化，而且没有一个提供我们组件状态机所提供的生命周期管理。Cordis 的响应式共效（第 3.2 节）提供了这一功能：当满足谓词发生变化时，通知机制会触发生命周期转换。
可用性响应式组件模型。最接近我们响应式共效的先例是对服务可用性作出响应。OSGi 的声明式服务和 iPOJO [118, 119] 允许组件声明提供和所需服务，运行时会在服务出现和消失时自动激活和停用它们；iPOJO 的 Gravity 项目 [119] 明确针对

自主运行时适应不断变化的服务可用性，其提供/需求模型直接预示了 Cordis 的 ctx.provide/ctx.get 模式。R-OSGi [53] 通过 RPC 将相同的抽象透明地扩展到分布式环境，将网络故障映射为服务撤回事件，这一模式在第 6.2 节作为 Cordis 模型的扩展进行了讨论。所有这些系统都通过一个停用回调进行恢复，但该回调在两个方面存在限制。首先，回调是手写的，因此资源安全依赖于开发者的自律，一旦遗忘就会静默地泄漏。其次，回调是同步的：如果拆卸需要与正在离开的依赖进行异步交换，框架不提供等待该交换的协议，从而迫使

针对可能已经陈旧的引用的阻塞等待。Cordis
的反应性副作用弥补了两个缺口：停用会恢复依赖项累积的效应，而其惯性卸载状态（第 4.3.3
节）在对进一步更改采取行动之前完成异步拆卸。

值级反应性。函数响应式编程（FRP）[120] 及其现代形式（例如 SolidJS 中的信号 [121, 122]、Vue 的响应系统、Angular Signals）在值级粒度上传播变化：当信号发生变化时，派生计算会同步重新评估或在调度器下重新评估 [123]。Cordis 的反应性副作用在组件级粒度上起作用，增加了值级传播无法建模的异步生命周期语义。同样的粒度差异也适用于一致性：传播在

一个轮次，按照依赖图确定的顺序，使得 FRP 可以要求没有派生计算同时读取更新的输入和过时的输入，这就是无瞬变性 [124]，而 Cordis
没有轮次的对应概念，协调操作一次到达一个，并且只保证没有单个转换跨越其余效应应的两个解析（定理
64）。这两者是互补而非竞争的：一个 Cordis 余效应应本身可以携带反应值，并且组件只在实际使用的部分上进行更新，将组件级反应性细化为跨越两个层次的更精细的反应性余效应应。
8. 结论
我们提出了一个动态可组合性的形式基础，通过将效应和余效应应的经典概念提升到运行时机制。可回退效应解决了局部时间上的问题

可组合性：每个上下文变换都带有运行时跟踪的逆变换，而跟踪和恢复都保持可组合性，因此在组件移除时上下文可以恢复。反应式协效应解决了局部空间可组合性：每当上下文发生变化时，组件都会根据其协效应规范收到通知，每次变化被分类为激活、取消激活或中性，协效应隔离会影响声明键的解析方式，而协效应拦截会影响绑定的使用方式。我们将效应上下文和协效应上下文统一为单一上下文类型，其中对协效应的观测等价性为效应提供独立性，构成了一种时间空间可组合性的编程范式。将这些机制结合成组件的概念，则得到

一种动态组合的演算，其元理论将时空可组合性从单个组件扩展到一套交错组件的整个系统。我们将这一范式实现为Cordis元框架，核心库提供效应追踪和共效解决，以及带有配置协调和热模块替换的声明式组件加载器。Koishi案例研究在一个拥有超过4000个社区插件的生产系统中验证了Cordis的设计。

超越人工策划的插件生态系统，未来验证的一个有说服力的方向是自演化智能体挂载（第1.2.2节），在这种情况下，AI代理能够持续生成并替换其自身的挂载组件，且几乎无需人工监督。在这种环境下应用Cordis，将验证在快速

组件更换，以及在频繁拓扑变化下依赖协调的空间保证。这种验证将展示该范式作为可恢复、协调和连续自我演化的基础，在代理工具和其他自主系统中的适用性。

References

[1] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” Communications of the ACM, vol. 15, no. 12, pp. 1053–1058, 1972, doi: 10.1145/361598.361623.

[2] D. Birsan, “On Plug-ins and Extensible Architectures,” ACM Queue, vol. 3, no. 2, pp. 40–46, 2005, doi: 10.1145/1053331.1053345.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” Communications of the ACM, vol. 59, no. 5, pp. 50–57, 2016, doi: 10.1145/2890784.

[4] B. Stroustrup, The Design and Evolution of C++. Addison-Wesley, 1994.

[5] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy, “ Asynchronous Exceptions in Haskell,” in Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, in PLDI '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 274–285. doi: 10.1145/378795.378858.

[6] L. Cardelli, “Program Fragments, Linking, and Modularization,” in Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997), ACM Press, 1997, pp. 266–277. doi: 10.1145/263699.263735.

[7] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley, 2002.

[8] R. Lopopolo, “Harness Engineering: Leveraging Codex in an Agent-First World.” [Online]. Available: https://openai.com/index/harness-engineering/

[9] Anthropic, “Harness Design for Long-Running Application Development.” [Online]. Available: https://www.anthropic.com/engineering/harness-design-long-running-apps

[10] L. Wang et al., “ A Survey on Large Language Model Based Autonomous Agents,” Frontiers of Computer Science, vol. 18, no. 6, p. 186345, 2024, doi: 10.1007/s11704-024-40231-1.

[11] Y. Qin et al., “Tool Learning with Foundation Models,” ACM Computing Surveys, 2025, doi: 10.1145/3704435.

[12] C. Packer, V . Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, “MemGPT: Towards LLMs as Operating Systems,” CoRR, vol. abs/2310.08560, 2023.

[13] T. Guo et al., “Large Language Model Based Multi-Agents: A Survey of Progress and Challenges,” in Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, in IJCAI 2024. 2024, pp. 8048–8057. doi: 10.24963/ijcai.2024/890.

[14] T. Cai, X. Wang, T. Ma, X. Chen, and D. Zhou, “Large Language Models as Tool Makers,” in Proceedings of the Twelfth International Conference on Learning Representations , in ICLR 2024. 2024. [Online]. Available: https://openreview.net/forum?id=qV83K9d5WB

[15] J. Armstrong, “Making Reliable Distributed Systems in the Presence of Software Errors,” Doctoral dissertation, 2003. [Online]. Available: https://erlang.org/download/armstrong_thesis_2003.pdf

[16] E. Moggi, “Notions of computation and monads,” Information and Computation, vol. 93, no. 1, pp. 55–92, 1991, doi: 10.1016/0890-5401(91)90052-4.

80

[17] G. Plotkin and J. Power, “ Adequacy for Algebraic Effects,” in Foundations of Software Science and Computation Structures, F. Honsell and M. Miculan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24.

[18] T. Petricek, D. Orchard, and A. Mycroft, “Coeffects: unified static analysis of context-dependence,” in Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II, in ICALP'13. Riga, Latvia: Springer-Verlag, 2013, pp. 385–397. doi: 10.1007/978-3-642-39212-2_35.

[19] M. Gaboardi, S.-ya Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, “Combining effects and coeffects via grading,” in Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, in ICFP 2016. Nara, Japan: Association for Com - puting Machinery, 2016, pp. 476–489. doi: 10.1145/2951913.2951939.

[20] A. Church, “ A Formulation of the Simple Theory of Types,” The Journal of Symbolic Logic, vol. 5, no. 2, pp. 56–68, 1940, doi: 10.2307/2266170.

[21] B. C. Pierce, Types and Programming Languages. MIT Press, 2002.

[22] J. M. Lucassen and D. K. Gifford, “Polymorphic Effect Systems,” in Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. doi: 10.1145/73560.73564.

[23] P . Wadler, “Monads for functional programming,” in Program Design Calculi , M. Broy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264.

[24] G. Plotkin and J. Power, “Notions of Computation Determine Monads,” in Foundations of Software Science and Computation Structures , Berlin, Heidelberg: Springer Berlin Heidel - berg, 2002, pp. 342–356. doi: 10.1007/3-540-45931-6_24.

[25] G. Plotkin and M. Pretnar, “Handlers of Algebraic Effects,” in Programming Languages and Systems (ESOP) , Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. doi: 10.1007/978-3-642-00590-9_7.

[26] M. Pretnar, “ An Introduction to Algebraic Effects and Handlers. Invited tutorial paper,” Electron. Notes Theor. Comput. Sci. , vol. 319, no. C, pp. 19–35, Dec. 2015, doi: 10.1016/j.entcs.2015.12.003.

[27] D. Leijen, “Koka: Programming with Row Polymorphic Effect Types,” Electronic Proceed- ings in Theoretical Computer Science , vol. 153, pp. 100–126, Jun. 2014, doi: 10.4204/eptcs.153.8.

[28] D. Leijen, “Type directed compilation of row-typed algebraic effects,” in Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages , in POPL

'17. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. doi: 10.1145/3009837.3009872.

[29] A. Bauer and M. Pretnar, “Programming with algebraic effects and handlers,” Journal of Logical and Algebraic Methods in Programming, vol. 84, no. 1, pp. 108–123, Jan. 2015, doi: 10.1016/j.jlamp.2014.02.001.

[30] K. Sivaramakrishnan et al., “Retrofitting parallelism onto OCaml,” Proc. ACM Program. Lang., vol. 4, no. ICFP , Aug. 2020, doi: 10.1145/3408995.

81

[31] T. Petricek, D. Orchard, and A. Mycroft, “Coeffects: a calculus of context-dependent computation,” in Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, in ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 123–135. doi: 10.1145/2628136.2628160.

[32] T. Uustalu and V . Vene, “Comonadic Notions of Computation,” Electronic Notes in Theoretical Computer Science, vol. 203, no. 5, pp. 263–284, 2008, doi: 10.1016/j.entcs.2008.05.029.

[33] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic, “ A Core Quantitative Coeffect Calculus,” in Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410 , Berlin, Heidelberg: Springer-Verlag, 2014, pp. 351–370. doi: 10.1007/978-3-642-54833-8_19.

[34] J. Reed and B. C. Pierce, “Distance makes the types grow stronger: a calculus for differential privacy,” SIGPLAN Not. , vol. 45, no. 9, pp. 157–168, Sep. 2010, doi: 10.1145/1932681.1863568.

[35] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, “ A core calculus of dependency,” in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 147–160. doi: 10.1145/292540.292555.

[36] D. E. Denning, “ A lattice model of secure information flow,” Commun. ACM, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.

[37] U. Dal Lago and F. Gavazzo, “ A relational theory of effects and coeffects,” Proc. ACM Program. Lang., vol. 6, no. POPL, Jan. 2022, doi: 10.1145/3498692.

[38] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern.” [Online]. Available: https://martinfowler.com/articles/injection.html

[39] A. M. Pitts and I. D. B. Stark, “Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's New?,” in Mathematical Foundations of Computer Science 1993 (MFCS 1993) , in Lecture Notes in Computer Science, vol. 711. Springer, 1993, pp. 122–141. doi: 10.1007/3-540-57182-5_8.

[40] G. D. Plotkin, “LCF Considered as a Programming Language,” Theoretical Computer Science, vol. 5, no. 3, pp. 223–255, 1977, doi: 10.1016/0304-3975(77)90044-5.

[41] D. R. Ghica, K. Muroya, and T. Waugh Ambridge, “ A Robust Graph-Based Approach to Observational Equivalence,” Logical Methods in Computer Science , vol. 21, no. 2, p. 8:1–8:95, 2025, doi: 10.46298/LMCS-21(2:8)2025.

[42] X. Leroy and S. Blazy, “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations,” Journal of Automated Reasoning, vol. 41, no. 1, pp.

1–31, 2008, doi: 10.1007/s10817-008-9099-0.

[43] R. P . James and A. Sabry, “Yield: Mainstream Delimited Continuations,” in First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011) , 2011, pp. 20–32. [Online]. Available: https://homes.luddy.indiana.edu/sabry/files/yield.pdf

[44] A. W. Mazurkiewicz, “Trace Theory,” in Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II , in Lecture Notes in Computer Science, vol. 255. Springer, 1986, pp. 279–324. doi: 10.1007/3-540-17906-2_30.

82

[45] U. A. Acar, G. E. Blelloch, and R. Harper, “ Adaptive functional programming,” ACM Transactions on Programming Languages and Systems , vol. 28, no. 6, pp. 990–1034, 2006, doi: 10.1145/1186632.1186634.

[46] webpack, “Hot Module Replacement.” [Online]. Available: https://webpack.js.org/api/hot-module-replacement/

[47] Vite, “HMR API.” [Online]. Available: https://vite.dev/guide/api-hmr

[48] E. N. (M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “ A Survey of Rollback-Recovery Protocols in Message-Passing Systems,” ACM Computing Surveys, vol. 34, no. 3, pp. 375–408, 2002, doi: 10.1145/568522.568525.

[49] H. Garcia-Molina and K. Salem, “Sagas,” in Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, in SIGMOD '87. 1987, pp. 249–259. doi: 10.1145/38713.38742.

[50] OSGi Alliance, OSGi Core Release 8. OSGi Alliance, 2020. [Online]. Available: https://docs.osgi.org/specification/osgi.core/8.0.0/

[51] J. Kramer and J. Magee, “The Evolving Philosophers Problem: Dynamic Change Man - agement,” IEEE Transactions on Software Engineering, vol. 16, no. 11, pp. 1293–1306, 1990, doi: 10.1109/32.60317.

[52] Y. Vandewoude, P . Ebraert, Y. Berbers, and T. D'Hondt, “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates,” IEEE Transactions on Software Engineering, vol. 33, no. 12, pp. 856–868, 2007, doi: 10.1109/tse.2007.70733.

[53] J. S. Rellermeyer, G. Alonso, and T. Roscoe, “R-OSGi: Distributed Applications Through Software Modularization,” in Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference , in Middleware '07. 2007, pp. 1–20. doi: 10.1007/978-3-540-76778-7_1.

[54] J. B. Dennis and E. C. Van Horn, “Programming Semantics for Multiprogrammed Computations,” Communications of the ACM , vol. 9, no. 3, pp. 143–155, 1966, doi: 10.1145/365230.365252.

[55] M. S. Miller, K.-P . Yee, and J. Shapiro, “Capability Myths Demolished,” technical report SRL2003–2, 2003. [Online]. Available: http://zesty.ca/capmyths/usenix.pdf

[56] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, “Capsicum: Practical Capa- bilities for UNIX,” in Proceedings of the 19th USENIX Security Symposium, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf

[57] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient Software-Based Fault

Isolation,” in Proceedings of the 14th ACM Symposium on Operating Systems Principles , in SOSP '93. 1993, pp. 203–216. doi: 10.1145/168619.168635.
[58] A. Barth, A. P . Felt, P . Saxena, and A. Boodman, “Protecting Browsers from Extension Vulnerabilities,” in Proceedings of the 17th Annual Network and Distributed System Security Symposium, in NDSS '10. 2010. [Online]. Available: https://www.ndss-symposium.org/ndss2010/protecting-browsers-extension-vulnerabilities/
[59] W. W. Ho and R. A. Olsson, “ An Approach to Genuine Dynamic Linking,” Software: Practice and Experience, vol. 21, no. 4, pp. 375–390, 1991, doi: 10.1002/SPE.4380210404.
83

[60] P . Wadler and S. Blott, “How to Make Ad-hoc Polymorphism Less Ad Hoc,” in Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '89. 1989, pp. 60–76. doi: 10.1145/75277.75283.

[61] N. D. Matsakis and F. S. K. II, “The Rust Language and Type System,” in ACM SIGPLAN ML Family Workshop, Gothenburg, Sweden, Sep. 2014.

[62] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller, “Modular Type Classes,” in Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '07. 2007, pp. 63–70. doi: 10.1145/1190216.1190229.

[63] Microsoft, “Declaration Merging.” [Online]. Available: https://www.typescriptlang.org/docs/handbook/declaration-merging.html

[64] T. Van Cutsem and M. S. Miller, “Proxies: Design Principles for Robust Object-oriented Intercession APIs,” in Proceedings of the 6th Symposium on Dynamic Languages, in DLS '10. 2010, pp. 59–72. doi: 10.1145/1869631.1869638.

[65] R. Hettinger, “Descriptor HowTo Guide.” [Online]. Available: https://docs.python.org/3/howto/descriptor.html

[66] P . Maes, “Concepts and Experiments in Computational Reflection,” in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) , 1987, pp. 147–155. doi: 10.1145/38765.38821.

[67] G. Bracha and D. M. Ungar, “Mirrors: design principles for meta-level facilities of object-oriented programming languages,” in Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOP - SLA), 2004, pp. 331–344. doi: 10.1145/1028976.1029004.

[68] R. Rouvoy and P . Merle, “Leveraging component-based software engineering with Fraclet,” Annals of Telecommunications , vol. 64, no. 1–2, pp. 65–79, 2009, doi: 10.1007/s12243-008-0072-z.

[69] E. Burmako, “Scala Macros: Let Our Powers Combine!,” in Proceedings of the 4th Workshop on Scala, in SCALA@ECOOP '13. 2013, p. 3:1–3:10. doi: 10.1145/2489837.2489840.

[70] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic Versioning and Impact of Breaking Changes in the Maven Repository,” Journal of Systems and Software, vol. 129, pp. 140–158, 2017, doi: 10.1016/j.jss.2016.04.008.

[71] P . Lam, J. Dietrich, and D. J. Pearce, “Putting the Semantics into Semantic Versioning,” in Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software , in Onward! '20. 2020, pp. 157–179. doi: 10.1145/3426428.3426922.

[72] P . Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, “Dependency Solving: A Separate Concern in Component Evolution Management,” Journal of Systems and Software, vol. 85, no. 10, pp. 2228–2240, 2012, doi: 10.1016/j.jss.2012.02.018.

[73] L. Cardelli, “Structural Subtyping and the Notion of Power Type,” in Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '88. 1988, pp. 70–79. doi: 10.1145/73560.73566.

[74] B. Meyer, “ Applying "Design by Contract",” Computer, vol. 25, no. 10, pp. 40–51, 1992, doi: 10.1109/2.161279.

84

[75] B. C. Pierce, “Bounded Quantification is Undecidable,” Information and Computation, vol. 112, no. 1, pp. 131–165, 1994, doi: 10.1006/inco.1994.1055.

[76] A. Haas et al. , “Bringing the web up to speed with WebAssembly,” in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM, 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[77] M. M. Swift, B. N. Bershad, and H. M. Levy, “Improving the reliability of commodity operating systems,” in Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP), ACM, 2003, pp. 207–222. doi: 10.1145/945445.945466.

[78] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, “Recovering device drivers,” ACM Transactions on Computer Systems, vol. 24, no. 4, pp. 333–360, 2006, doi: 10.1145/1189256.1189257.

[79] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, “Operating System Transactions,” in Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), ACM, 2009, pp. 161–176. doi: 10.1145/1629575.1629591.

[80] O. Kiselyov and C.-chieh Shan, “Delimited Continuations in Operating Systems,” in Modeling and Using Context (CONTEXT 2007) , in Lecture Notes in Computer Science, vol. 4635. Springer, 2007, pp. 291–302. doi: 10.1007/978-3-540-74255-5_22.

[81] E. Dolstra and A. Löh, “NixOS: a purely functional Linux distribution,” in Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP), ACM, 2008, pp. 367–378. doi: 10.1145/1411204.1411255.

[82] ZIO, “ZIO: Type-safe, composable asynchronous and concurrent programming for Scala.” [Online]. Available: https://zio.dev/

[83] Effect, “Effect: A TypeScript library for building robust applications.” [Online]. Avail - able: https://effect.website/

[84] G. Canti, “fp-ts: Functional programming in TypeScript.” [Online]. Available: https:// github.com/gcanti/fp-ts

[85] J. I. Brachthäuser, P . Schuster, and K. Ostermann, “Effects as capabilities: effect handlers and lightweight effect polymorphism,” Proc. ACM Program. Lang., vol. 4, no. OOPSLA, 2020, doi: 10.1145/3428194.

[86] J. I. Brachthäuser, P . Schuster, E. Lee, and A. Boruch-Gruszecki, “Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back,” Proc. ACM Program. Lang., vol. 6, no. OOPSLA1, 2022, doi: 10.1145/3527320.

[87] C. Heunen, R. Kaarsgaard, and M. Karvonen, “Reversible Effects as Inverse Arrows,” in Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming

Semantics (MFPS XXXIV), in Electronic Notes in Theoretical Computer Science, vol. 341. 2018, pp. 179–199. doi: 10.1016/j.entcs.2018.11.009.

[88] D. Orchard, V .-B. Liepelt, and H. Eades III, “Quantitative program reasoning with graded modal types,” Proc. ACM Program. Lang. , vol. 3, no. ICFP , 2019, doi: 10.1145/3341714.

[89] R. Bianchini, F. Dagnino, P . Giannini, E. Zucca, and M. Servetto, “Coeffects for sharing and mutation,” Proc. ACM Program. Lang. , vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563319.

85

[90] R. Bianchini, F. Dagnino, P . Giannini, and E. Zucca, “ A Java-like calculus with heterogeneous coeffects,” Theoretical Computer Science, vol. 971, p. 114063, 2023, doi: https://doi.org/10.1016/j.tcs.2023.114063.

[91] C. Torczon, E. Suárez Acevedo, S. Agrawal, J. Velez-Ginorio, and S. Weirich, “Effects and Coeffects in Call-by-Push-Value,” Proc. ACM Program. Lang., vol. 8, no. OOPSLA2, Oct. 2024, doi: 10.1145/3689750.

[92] R. Hirschfeld, P . Costanza, and O. Nierstrasz, “Context-oriented Programming,” Journal of Object Technology, vol. 7, no. 3, pp. 125–151, 2008, doi: 10.5381/jot.2008.7.3.a4.

[93] P . Costanza and R. Hirschfeld, “Language constructs for context-oriented programming: an overview of ContextL,” in Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05), ACM, 2005, pp. 1–10. doi: 10.1145/1146841.1146842.

[94] G. Salvaneschi, C. Ghezzi, and M. Pradella, “Context-oriented programming: A software engineering perspective,” Journal of Systems and Software , vol. 85, no. 8, pp. 1801–1817, 2012, doi: 10.1016/j.jss.2012.03.024.

[95] G. Kiczales et al. , “ Aspect-Oriented Programming,” in ECOOP'97 — Object-Oriented Programming, 11th European Conference, in Lecture Notes in Computer Science, vol. 1241. Springer, 1997, pp. 220–242. doi: 10.1007/BFb0053381.

[96] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “ An Overview of AspectJ,” in ECOOP 2001 — Object-Oriented Programming, 15th European Conference, in Lecture Notes in Computer Science, vol. 2072. Springer, 2001, pp. 327–353. doi: 10.1007/3-540-45337-7_18.

[97] A. Popovici, T. Gross, and G. Alonso, “Dynamic Weaving for Aspect-Oriented Programming,” in Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), ACM, 2002, pp. 141–147. doi: 10.1145/508386.508404.

[98] J. Bonér, “What Are the Key Issues for Commercial AOP Use: How Does AspectWerkz Address Them?,” in Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), ACM, 2004, pp. 5–6. doi: 10.1145/976270.976273.

[99] M. Hicks, J. T. Moore, and S. Nettles, “Dynamic Software Updating,” in Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, in PLDI '01. 2001, pp. 13–23. doi: 10.1145/378795.378798.

[100] G. Stoyle, M. Hicks, G. Bierman, P . Sewell, and I. Neamtiu, “Mutatis Mutandis: Safe and Predictable Dynamic Software Updating,” in Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in POPL '05. 2005, pp. 183–194. doi: 10.1145/1040305.1040321.

[101] C. M. Hayden, K. Saur, E. K. Smith, and M. Hicks, “Kitsune: Efficient, General-Purpose Dynamic Software Updating for C,” ACM Trans. Program. Lang. Syst., vol. 36, no. 4, 2014, doi: 10.1145/2629460.

[102] M. Overeem, M. Spoor, and S. Jansen, “The Dark Side of Event Sourcing: Managing Data Conversion,” in IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, in SANER '17. 2017, pp. 193–204. doi: 10.1109/SANER.2017.7884621.

[103] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1994.

86

[104] D. Leijen, “ Algebraic Effect Handlers with Resources and Deep Finalization,” technical report MSR-TR-2018-10, Apr. 2018. [Online]. Available: https://www.microsoft.com/en-us/research/publication/algebraic-effect-handlers-resources-deep-finalization/

[105] M. Fowler, “Event Sourcing.” 2005.

[106] J. Lee, J. Ahn, and K. Yi, “React-tRace: A Semantics for Understanding React Hooks,” Proc. ACM Program. Lang., vol. 9, no. OOPSLA2, pp. 471–498, 2025, doi: 10.1145/3763067.

[107] N. Shavit and D. Touitou, “Software Transactional Memory,” in Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, in PODC '95. 1995, pp. 204–213. doi: 10.1145/224964.224987.

[108] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, “Composable Memory Transac - tions,” in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, in PPoPP '05. 2005, pp. 48–60. doi: 10.1145/1065944.1065952.

[109] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in Proceedings of the 20th Annual International Symposium on Computer Architecture, in ISCA '93. 1993, pp. 289–300. doi: 10.1145/165123.165164.

[110] R. Landauer, “Irreversibility and Heat Generation in the Computing Process,” IBM Journal of Research and Development, vol. 5, no. 3, pp. 183–191, 1961, doi: 10.1147/rd.53.0183.

[111] C. H. Bennett, “Logical Reversibility of Computation,” IBM Journal of Research and Development, vol. 17, no. 6, pp. 525–532, 1973, doi: 10.1147/rd.176.0525.

[112] T. Yokoyama and R. Glück, “ A Reversible Programming Language and its Invertible Self-Interpreter,” in Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evalua - tion and Semantics-Based Program Manipulation , in PEPM '07. 2007, pp. 144–153. doi: 10.1145/1244381.1244404.

[113] V . Danos and J. Krivine, “Reversible Communicating Systems,” in CONCUR 2004 — Concurrency Theory, 15th International Conference, in Lecture Notes in Computer Science, vol. 3170. Springer, 2004, pp. 292–307. doi: 10.1007/978-3-540-28644-8_19.

[114] I. Phillips and I. Ulidowski, “Reversing Algebraic Process Calculi,” in Foundations of Software Science and Computation Structures, 9th International Conference (FOSSACS 2006), in Lecture Notes in Computer Science, vol. 3921. Springer, 2006, pp. 246–260. doi: 10.1007/11690634_17.

[115] P . Wadler, “Linear Types Can Change the World!,” in Programming Concepts and Methods: Proceedings of the IFIP Working Group 2.2/2.3 Working Conference , North-Holland, 1990, pp. 561–581. [Online]. Available: https://homepages.inf.ed.ac.uk/wadler/papers/linear/linear.ps

[116] A. Lenharth, V . S. Adve, and S. T. King, “Recovery domains: an organizing principle for recoverable operating systems,” in Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2009, pp. 49–60. doi: 10.1145/1508244.1508251.

[117] C. Walls, Spring in Action , 6th ed. Manning Publications, 2022. [Online]. Available: https://www.manning.com/books/spring-in-action-sixth-edition

87

[118] C. Escoffier, R. S. Hall, and P . Lalanda, “iPOJO: an Extensible Service-Oriented Component Framework,” in IEEE International Conference on Services Computing, 2007, pp. 474–481. doi: 10.1109/SCC.2007.74.

[119] H. Cervantes and R. S. Hall, “ Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model,” in Proceedings of the 26th International Conference on Software Engineering, in ICSE '04. 2004, pp. 614–623. doi: 10.1109/ICSE.2004.1317483.

[120] C. Elliott and P . Hudak, “Functional Reactive Animation,” in Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, in ICFP '97. 1997, pp. 263–273. doi: 10.1145/258948.258973.

[121] G. H. Cooper and S. Krishnamurthi, “Embedding Dynamic Dataflow in a Call-by-Value Language,” in Programming Languages and Systems (ESOP 2006) , in Lecture Notes in Computer Science, vol. 3924. Springer, 2006, pp. 294–308. doi: 10.1007/11693024_20.

[122] I. Maier and M. Odersky, “Deprecating the Observer Pattern with Scala.React,” technical report EPFL-REPORT-176887, 2012. [Online]. Available: https://infoscience.epfl.ch/record/176887

[123] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, W. De Meuter, and others, “ A Survey on Reactive Programming,” ACM Comput. Surv. , vol. 45, no. 4, 2013, doi: 10.1145/2501654.2501666.

[124] A. Margara and G. Salvaneschi, “On the Semantics of Distributed Reactive Programming: The Cost of Consistency,” IEEE Trans. Software Eng., vol. 44, no. 7, pp. 689–711, 2018, doi: 10.1109/TSE.2018.2833109.