JoinActors：带连接模式的模块化 Actor 库

AymanHussein[a] 、hilipp Haller[b] 、annis Karras[a] 、ernán Melgratti[c] 、Alceste Scalas[a] 和 Emilio Tuosto[d]
a 丹麦技术大学，丹麦 b 瑞典皇家理工学院（KTH）
c 布宜诺斯艾利斯大学和 CONICET，阿根廷 d 格兰萨索科学研究所，意大利

摘要 连接模式 是消息传递应用中的一种高级编程结构。它们提供了一种直观、声明式的方法，用于指定并发与分布式组件如何协调，这种协调可能取决于消息组合所满足的复杂条件。连接模式启发了许多实现，但其中大多数并非以库的形式提供，而是领域特定语言，因此难以集成到现有生态系统中。此外，所有实现都附带预定义的匹配算法，而根据应用需求，这种算法未必是最优的。最近发布的库 JoinActors 解决了这些限制。它将连接模式集成到现成的 Scala 3 编程语言中，并且在设计上支持根据所用匹配算法进行模块化。

在本工作中，我们针对设计、开发和评估模块化连接模式匹配工具包这一问题展开研究。该工具包：(1) 可作为常规库使用，并在现有编程语言中提供对开发者友好的语法；(2) 采用可扩展设计，支持使用和比较不同的匹配算法。

我们分析了JoinActors如何实现上述目标 (1) 和 (2)。介绍JoinActors 的论文仅简要概述了其设计和实现（因为该论文的主要目标是形式化其新颖的公平匹配语义）。在本工作中，我们详细介绍并讨论了JoinActors的改进版本，重点关注其对元编程的使用（这使其能够提供类似于标准模式匹配的直观 API）以及其模块化设计。我们展示了这种设计如何支持集成多种具有不同优化的匹配算法，并通过涵盖不同工作负载的基准测试评估它们的性能。

我们展示了 Scala 3 元编程在现有语言中集成高级并发编程构造的一种复杂用法。此外，我们还讨论了相关见解和“经验
在优化连接模式匹配方面“学到的经验”，以及JoinActors的模块化设计如何为此提供便利——它使得我们能够系统地比较多种匹配算法实现。

我们采用最初介绍JoinActors的论文所使用的公平连接模式匹配 语义和基准测试套件。通过大量测试，我们确保新优化的匹配算法产生的匹配结果与原始JoinActors 库完全相同，同时实现显著更佳的性能。改进版 JoinActors 是本文的配套制品。

这项工作展示了连接模式在既有语言中实现分布式消息传递系统里的复杂协调模式时所具备的表达能力、有效性和易用性。它还展现出令人期待的性能，相较于以往工作有显著提升。除了实际应用前景之外，JoinActors的模块化设计还提供了一个研究平台，可用于探索和比较新的连接模式匹配算法，这些算法甚至可能基于截然不同的语义。

ACM CCS 2012

- **软件及其工程** → **领域特定语言**；**并发编程语言**；**分布式编程语言**；**并发编程结构**；

关键词 连接模式、Actor、协调、并发编程、编程模型、元编程

编程的艺术、科学与工程

提交时间 2025年10月2日

出版时间 2026年2月15日

DOI 10.22152/programming-journal.org/2026/11/4

© A. Hussein、P. Haller、I. Karras、H. Melgratti、A. Scalas、E. Tuosto 本作品采用“CC BY 4.0”许可协议授权，载于《编程的艺术、科学与工程》第 11 卷第 1 期，2026 年，第 4 篇文章；共 47 页。

清单 1 使用 JoinActors 库实现的支付服务。

```
类型 PaymentEvent = PaymentRequested | MerchantValidated | CustomerValidated

Actor[PaymentEvent |关闭, Unit类型]{
    接收 { self => { // 'self' 是 Actor 的“地址”，可用于发送消息
        情况PaymentRequested(id1)&:& MerchantValidated(id2) &:& CustomerValidated(id3)

            如果 (id1 == id2) && (id2 ==id3) =>
          核心服务！ PaymentSucceeded(id1)

          log(s"支付服务已处理支付请求 $id1")
          继续

        case 关闭() => 停止(())}
    }(matcherAlgorithm) // 选择要使用的连接模式匹配算法
}
```

1 引言

许多软件应用基于通过消息传递交换数据并进行协调的并发与分布式组件。此类交互可见于云微服务、边缘计算网格以及物联网（物联网）部署等场景。这些应用往往需要在收到特定一组消息、且针对其有效载荷的某些谓词得到满足之后，才执行特定任务。这可能需要开发非平凡的簿记逻辑来跟踪已观察到的消息，从而导致代码冗长且容易出错。

连接模式 提供了一种编程抽象，用于应对这些协调挑战。该思想最早在连接演算 [13],中提出，并将人们熟悉的模式匹配概念扩展到了消息传递场景：连接模式针对消息集合进行匹配，并在接收到所有必需的消息后“触发”。连接模式还可以包含守卫 （即针对消息负载的谓词），这些谓词必须满足，模式才能触发。通过在简洁的声明式规约中捕获协调逻辑，连接模式可以减少对手动状态管理的依赖，并提升分布式应用的清晰性与可靠性 [29, 第 4]节。

连接模式的一种近期实现是面向 Scala 3 的JoinActors 库 [17] 。它提供了一个受 Akka 和 Pekko[1] 等知名 Actor [1]框架启发的编程接口，不过JoinActors 允许 Actor 对消息组合（以连接模式表示）作出反应，而不是仅针对单条消息作出反应。

示例：使用 JoinActors 库实现的支付服务清单 1 展示了一个简化的移动支付后端服务。该服务以 Actor 的形式实现，接收邮箱中的消息，并通过连接模式匹配来处理这些消息。Actor 构造函数（从第 4 行开始）声明，该 Actor 接受**PaymentEvent**

[1]https://akka.io/、https://pekko.apache.org/ （访问日期：2026-02-12）。

（定义于第 2 行）或 关闭，[2] 并在终止时返回 Unit类型 的结果。Actor 的行为定义在 receive 代码块中（第 5–13 行），每当邮箱收到相应的消息组合时，就会触发两个连接模式（分别位于第 6–7 行和第 12 行）。

第 6–7 行的连接模式指定了两个条件：1. 邮箱中必须分别包含三条类型为 PaymentRequested、MerchantVali-dated 和 CustomerValidated 的消息。这些消息由参与支付工作流的其他代理发送，且可以按任意顺序送达；以及2. 对于这三条消息，守卫 （即第 7 行的 **if** 条件）必须计算为 **true** 。当它们的有效载荷（由变量 id1、id2 和id3 捕获）相等时，就满足此条件，这意味着它们指向同一个支付标识符。

每当满足这两个条件时，就会触发连接模式：也就是说，三条匹配的消息会从邮箱中移除，并执行 **=>**（第 8–10 行）后面的代码。此时，Actor 会向核心服务发送一条 PaymentSucceeded 消息（第 8 行），并记录该事件（第 9 行）。最后的继续（第 10 行）表示该 Actor 仍保持活动状态，可继续处理后续消息。

相反，第 12 行的连接模式会在接收到 关闭 类型的消息时触发。发生这种情况时，停止(()) 会使 Actor 停止执行并返回 Unit类型值 ()。

请注意，如果没有连接模式，支付服务就需要实现手动簿记逻辑，以检查新到达的消息在与之前接收的消息组合后是否满足守卫条件——因此，这些消息需要保存在内部状态中。当多个连接模式重叠并在相同消息上“竞争”时（如第 3.2 节的扩展示例所示），这会导致代码更加复杂且容易出错。

最后，在清单 1 的第 13 行，参数 匹配器算法 决定如何根据声明的连接模式匹配消息。这是 JoinActors 库设计中的关键要素：它允许根据应用程序的需求选择不同的匹配算法。

贡献：一个模块化、经过优化且可扩展的连接模式匹配库本文介绍了JoinActors 的修订扩展版本，相较于 [17]（其仅展示了该库的初步概念验证实现，并对其设计进行了高层次概述），该版本在模块化和性能方面均有显著提升。我们阐释了 JoinActors 如何利用 Scala 3 的元编程特性，将连接模式匹配嵌入现有语言，同时提供对开发者友好的 API。我们还利用JoinActors的模块化架构，实现并系统比较了多种经过优化的匹配算法，这些算法在 [17] 中实现了公平匹配语义；我们展示了这些优化如何显著影响不同工作负载下的匹配性能。

[2]在清单 1 中，符号 **|** 表示 Scala 3 中的类型联合。

我们在第 2 节介绍相关背景。在概述JoinActors（第 3 节）之后，我们详细介绍其设计与实现（第 4 节），重点讨论其模块化和元编程方面，以及经过优化的匹配算法；随后评估这些算法的性能（第 5 节）。我们在第 6 节讨论相关工作，并在第 7 节总结全文。

2 背景

本节介绍我们JoinActors库背后关键概念的背景信息：Actor 模型（第 2.1 节）、公平的连接模式匹配（第 2.2 节），以及 Scala 3 中的元编程（第 2.3 节）。

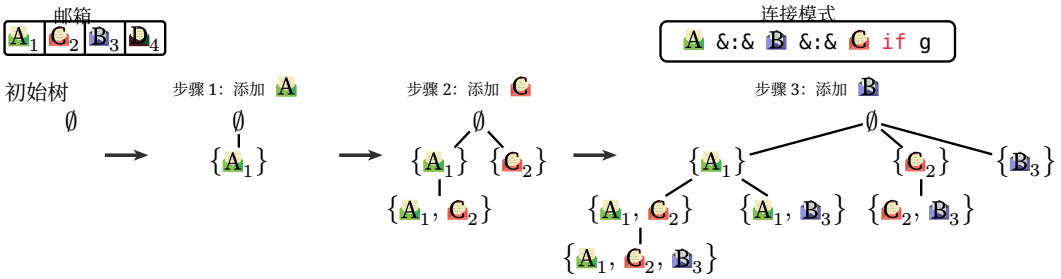
2.1 Actor 模型

JoinActors API 实现了流行的*Actor* 模型 [1, 18]的一种变体。在该模型中，计算围绕 Actor 组织 —— 这些独立实体封装私有状态并按顺序处理消息。Actor 仅通过异步消息传递进行交互，并使用非阻塞消息发送；收到消息后，Actor 可以更新其内部状态（例如改变自身处理后续消息的行为）、向其他 Actor 发送消息，以及生成新的 Actor。Actor 可以在消息中交换彼此的地址，从而能够在运行时动态重新配置 actor 网络。在 Akka/Pekko 等传统 actor 系统中，Actor 会对每条传入消息作出反应。相比之下，我们的JoinActors库允许 Actor 对其邮箱中的消息组合作出反应；此类组合以连接模式表示，如清单 1 所示。

2.2 公平连接模式匹配

文献中大多数连接模式的实现采用非确定性匹配语义：如果多个消息组合都能匹配某个连接模式，就会任意选取其中一个 [16, 29]。原则上，这意味着邮箱中的某些消息即使“可匹配”，也可能永远不会被选中。近期的工作 [17]提出了一种基于确定性 连接模式匹配语义公平合并模式匹配的连接模式匹配语义：当一个连接模式匹配多个消息组合时，优先匹配并消耗最早的消息——因此，可匹配消息不会无限期地留在邮箱中。 [17] 中介绍的JoinActors 早期版本通过两种算法实现了这种公平匹配语义：

无状态“暴力”匹配（暴力匹配器）。该算法会等待，直到邮箱中有足够的消息来“填满”连接模式。然后，它按类型筛选消息，枚举所有能够形成匹配的消息组合，并选出最公平的一组。该算法直接实现了形式化的公平匹配语义，但每当有新消息到达时，都会重新计算之前不匹配的消息组合，因此其效率会因工作负载而异。

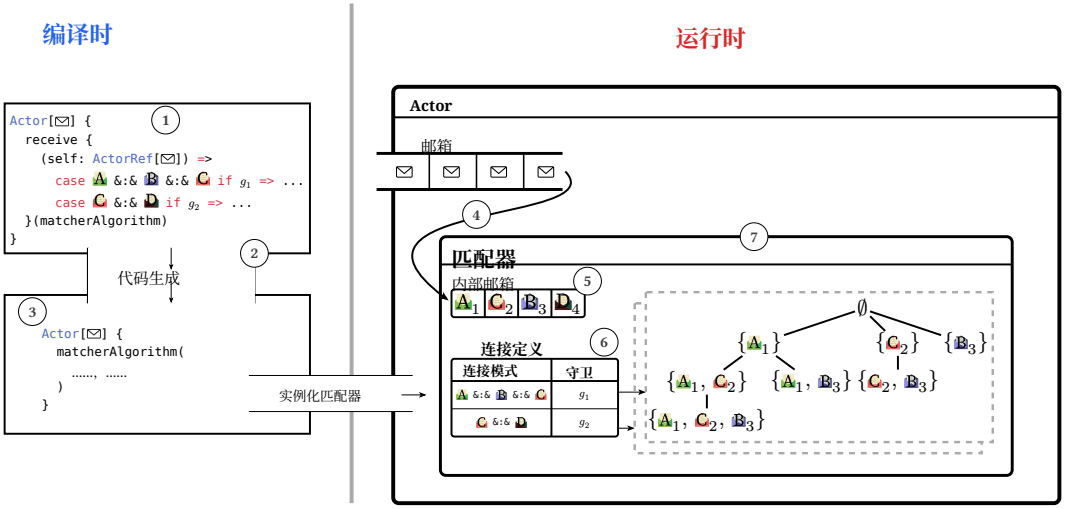


■ **图 1** 由**有状态树匹配器** 算法执行的增量分支过程，在 [17],示例 2.1 中有描述。

有状态匹配（**有状态树匹配器**）。该算法维护一种状态，即匹配树，用来追踪先前收到的哪些消息能够部分满足连接模式。新消息到达时，算法会检查匹配树，从而避免重新计算先前不匹配的组合。匹配树包含部分匹配，并会随着新消息到达而逐步构建（“分枝”）。从概念上说，这让人联想到 RETE 风格的 [11] 前向链式规则 AI 系统：连接模式类似于 RETE产生式，而到达邮箱的新消息则类似于添加到 RETE 工作记忆中的新数据；匹配树的节点表示逐步增强的测试，用于筛选消息并缓存中间结果，直到到达终端节点（此时一组消息完成了模式／产生式）。然而，与 RETE 风格的算法不同，分枝过程始终保持一个不变式：对匹配树进行深度优先遍历，总能得到最公平的匹配（如果存在）。每当有消息完成该模式时，都会对树进行剪枝，具体方式取决于守卫的计算结果是**真**还是**假**。该算法通常比无状态算法性能更好。下文示例 2.1 将介绍分枝过程。

示例 2.1（匹配树的分支）。图 1 展示了 [17] 中的 **有状态树匹配器** 算法如何逐步将 4 条传入消息（左上角的邮箱）与右上角的连接模式进行匹配。匹配树起初为空，随着新消息到达而逐步分支：例如，在步骤 2，树表明该模式可以仅由消息 **A**（邮箱中的位置 1）部分匹配，也可以仅由 **C**（位置 2）部分匹配，或者由 **A** 和 **C** 共同匹配。在步骤 3，深度优先遍历会到达一个叶节点，其中包含一个潜在的完整匹配，该匹配使用消息**A**、**C**和**B**。此潜在匹配用于计算守卫 g：如果结果为**true**，则匹配成功，这些消息将从邮箱中移除，所有从它们延伸出的子树也会被剪枝；否则，只剪除该叶节点，且不再对其进行检查，因为它代表一次失败的匹配。■

论文 [17] 还介绍了一个用于连接模式匹配的基准测试套件，我们在第 5 节的评估中采用了该套件。



■ **图 2** JoinActors 库在编译时和运行时的运行概览。

2.3 Scala 3 中的元编程

Scala 3 支持通过inline 修饰符定义在编译时于调用点内联的函数和参数。内联函数还可用于在编译时分析、构造并返回代码片段，从而成为用于代码生成的宏 [34]。对于代码片段的分析与构造，Scala 3 同时提供了引用与拼接机制 [32] 和反射 API [33]：前者是类型安全的，但表达能力较为有限，而后者的类型检查较弱（即可能允许生成类型不正确的代码），但对代码检查与变换提供了更细粒度的控制。在内部，JoinActors同时使用这两种机制，不过在第 4.2.1 节中，我们将主要举例说明其对反射 API 的使用。

3 JoinActors 库概览及其用法

现在，我们将更详细地概述 JoinActors 在编译时和运行时的运行方式（第 3.1 节），并提供一个更完整的用法示例（第 3.2 节）。

3.1 JoinActors 在编译时和运行时

JoinActors 库的整体架构与用法如图 2 所示 :

1. 使用 JoinActors 的程序员会编写一个 **Actor**，声明其接受的消息类型（与清单 1 类似）。在图 2 中，这种类型以闭包络表示。Actor 的主体通过 receive 宏指定（该宏由 JoinActors 提供，并在第 4.2.1 节中讨论）。宏的第一个参数应为一段 偏函数 代码，即一个 **match**表达式，其各个分支定义了 Actor 将处理的连接模式。在图 2 中，连接模式的子句

以打开的信封表示，因为连接模式可以检查每条消息的具体类型和载荷。2. 在编译时，receive宏会检查作为参数提供的 **match** 表达式，并提取生成连接模式匹配代码所需的信息。3. 代码生成后，Actor 的主体会被替换为一段代码，用于实例化匹配器算法（即步骤 1 中提供给receive 的第二个参数），并传入步骤 2 中提取的连接模式匹配信息。（详情见第 4 节。）4. 运行时，Actor 会逐条从邮箱中取出消息，并将其转发给由 匹配器对象实例化的 匹配器算法。5. 匹配器对象处理收到的消息，并将其存储在内部。图 2 中，这些消息以打开的信封表示，说明消息的类型和值现在都已完全可用；数字表示消息接收的顺序，这与公平匹配（第 2.2 节）有关。6. 匹配器对象随后会根据用户提供的连接模式定义，对存储的消息进行求值。7. 所选的匹配器算法决定匹配器对象的工作方式。图 2 展示了匹配器为有状态树匹配器类的实例，并为每个连接模式维护一棵匹配树（如示例 2.1 所示）的情况。

3.2 示例：工厂车间监控

现在，我们通过一个包含大量复杂连接模式的应用程序，来展示 JoinActors 的用法。清单 2 展示了工厂车间监控应用程序的（简化）代码（这是对 [17] 中开篇示例的扩展）。该应用程序接收描述车间相关事件的消息（第 1 行）：

1. 机器报告故障 以请求维护。故障 消息包括机器 ID、唯一的请求 ID 和时间戳；2. 工人修复故障。修复 消息包括机器 ID、请求 ID 和时间戳；3. 定期标记 消息用于标示时间流逝，并据此判断维护请求是否已在所需时限内得到处理。标记消息包括时间戳以及该标记所跟踪的一组故障 ID。

该应用程序会检测到达其邮箱的消息组合，并报告异常或服务质量违规。为此，该应用程序有时会生成并发送一些RecentFix 消息到自己的邮箱——这些消息与修复事件类似，并保存有关近期已修复故障的信息。

JoinActors 提供的连接模式匹配支持，可用于声明式地指定此类消息组合，如清单 2 所示：
故障快速复发： 第一个连接模式（第 7–9 行）匹配一串消息，这些消息表示某台机器最近刚修复，但随后同一台机器又在 30 分钟内发生故障。

延迟维护： 第二和第三个连接模式（第 14–22 行）描述了这样的场景：报告了故障，但未在 10 分钟内修复。如果工人只

清单 2 使用 JoinActors 库构建的简化工厂监控应用。该应用详见第 3.2 节。

```
类型 事件 = 故障 |修复 |标记 |...

定义 监控(匹配器算法: MatcherFactory) = Actor[事件 | RecentFix | 关闭, Unit类型] {
    接收 { self => {

        // 故障快速复发
        情况 RecentFix(mid1, rid1, ts1) &:&故障(mid2, rid2, ts2)
                    如果mid1 == mid2 &&rid1！= rid2&& ts2 >= ts1 && ts2 - ts1 < 三十_最小值 =>
            自身 ! 故障(mid3, rid3, ts3) // 重新将最新故障入队
            println(s"机器 ${mid1} 在维护后 30分钟内发生故障！")
            继续

        // 延迟维护
        case 故障(_, rid1, ts1) &:&修复(mid, rid2,ts2)如果rid1 == rid2 && ts2 - ts1 >= TEN_分钟 =>
            println(s"故障 ${rid1} 仅在 ${(ts2- ts1)/ONE_MIN} 分钟后才修复！")
            self ! RecentFix(mid2, rid2, ts2) // 自我通知：某故障最近已被修复
            继续
        case 故障(mid1, rid1, ts1) &:& 标记(ts2,tracked)
          如果 ts2 - ts1 >= TEN_MIN&& !tracked.contains(rid1) =>

            self ! 故障(mid1, rid1, ts1) // 将未处理的故障重新入队
            self ! 标记(ts2, tracked + rid1) // 重新入队该标记，并跟踪未处理的故障
            println(s"故障 ${rid1} 未被接手达 ${(ts2- ts1)/ONE_MIN} 分钟！")
            继续

        // 及时维护
        case 故障(_, rid1, ts1) &:&修复(mid2,rid2,ts2)如果 rid1 == rid2 && ts2 - ts1 < TEN_MIN =>。
            self ! RecentFix(mid2, rid2, ts2) // 自我通知某故障最近已修复
            继续 // 故障在 10 分钟内得到处理，无需报告

        // 一段时间后清除旧标记和近期修复
        情况 标记(ts1, _) &:&标记(ts2,tracked2)如果 ts2- ts1 >= 2 * TEN_MIN =>
            自身 ! 标记(ts2, tracked2) // 仅重新入队最新的标记
            继续
        情况 RecentFix(_, _, ts1) &:& 标记(ts2,tracked2)如果 ts2- ts1 >= 4 * TEN_MIN =>
            自身 ! 标记(ts2, tracked2) // 仅重新将标记入队
            继续

        情况 关闭() => 停止(())
      }
    }(匹配器算法)
}
```

在超过该阈值后修复故障，或者如果故障已超过10分钟未得到处理（根据一条标记 消息），则应用程序报告服务质量违规。为避免针对同一故障和同一标记重复报告，第二个连接模式的守卫（第18行）要求该故障尚未在标记 消息中被跟踪；随后，应用程序将一条经过修改的标记 重新入队，以跟踪该故障（第20行）。

及时维护：第四个连接模式（第25-27行）仅在故障于10分钟内得到修复时记录一条RecentFix。

过时事件清理：第五和第六个连接模式（第30-35行）会在一段时间后丢弃标记和RecentFix 事件；每当收到新的标记时都会进行此项检查。

最后，最后一个连接模式（第37行）在收到关闭消息时停止监控器。

请注意，清单 2 的传统基于 actor 的实现需要逐条处理每条传入消息，并包含专门的代码来跟踪消息组合（包括可能出现的乱序传递，例如，可能先观察到修复，再观察到相应的故障）。这会大幅增加代码复杂度和出错风险。此外，清单 2 中的模式通常会使用相同的消息类型，因此一条消息的到达可能激活多个模式：在这种情况下，JoinActors 会以确定性的方式选择最公平的 匹配，消耗最早的消息 [17]。在本例中，这有助于确保第 30–35 行中处理掉最早的标记和RecentFix 事件。

4 JoinActors 的设计与实现

下面介绍JoinActors 库的设计（第 4.1 节）和实现（第 4.2 节）。我们首先重点说明设计决策如何影响该库的功能和可扩展性，然后介绍其实现，并着重阐述 Scala 3 元编程设施的运用。

4.1 设计与可扩展性

JoinActors 库的 API 兼顾易用性和表现力：程序员可以使用类似于标准 Scala 3 模式匹配的语法来指定连接模式，如清单 1 所示。这是通过receive宏实现的，它要求第一个参数是一个类型正确的 Scala 3（偏）函数，并且包含**match** 表达式。

清单 3 展示了使用JoinActors API 的 Actor 的通用结构。在构造函数中Actor[M,T]，类型参数 M 表示该 Actor 可以接收的消息类型，而 T 表示该 Actor 的返回类型。[3]

[3]更准确地说：生成一个 Actor 会创建一个 Future[T]；Actor 终止时，会以类型为T的值完成该 future。

清单 3 使用JoinActors 库实现的 Actor 的一般结构。

```
Actor[M, T] {
    receive { (self : ActorRef[M]) => {
            case P1 如果 g1 => e1
            case P2 如果 g2 => e2
            ...
            情况 Pn 如果 gn =>e n
        }(匹配器算法)
    }
}
```

清单 4 使用清单 5 中的特质表示 receive 宏的类型签名。

```
内联 定义 receive[M, T] (内联 闭包： ActorRef[M] => 偏函数[任意, 结果[T]])(内联 匹配器算
法： MatcherFactory)： 匹配器[M, 结果[T]]
```

构造函数 Actor[M,T]接受一个参数：一个 receive 宏实例，该宏将一段代码块封装起来。给定 Actor 的自身引用（类型为 ActorRef[M]），这段代码指定要匹配的连接模式。在清单 3 中，每个 Pi 表示一个针对类型为 M 的消息的连接模式，并且可能返回类型为 T 的值。消息类型 M 通常定义为类型联合，或由 case类或枚举变体组成的密封族；[4] 此外，每个 Pi 都是由以下形式的消息构成的“合取”：M_1(...) &:& M_2(...) &:& ... &:& M_m(...)（如清单 1 所示），其中每个 Mj（$j \in 1..m$）都是一种消息类型（通常是 M 的子类型），而运算符 &:& 由 JoinActors 提供。请注意，合取中的消息数量在不同模式间可能有所不同。每个 gi 都是一个守卫，即一个布尔表达式，可以引用相应连接模式 Pi 中绑定的变量。每个 ei 都是一个表达式，在相应连接模式和守卫满足时执行；每个 ei 都必须返回 Continue（以继续执行 Actor）或 Stop(result)，其中 result 是一个表达式，其类型为 Actor 的返回类型 T。

receive 的第二个参数，即 matcherAlgorithm，指定连接模式匹配算法。此参数为不同算法提供统一的实例化机制，这些算法都必须实现 **Matcher** 特质。

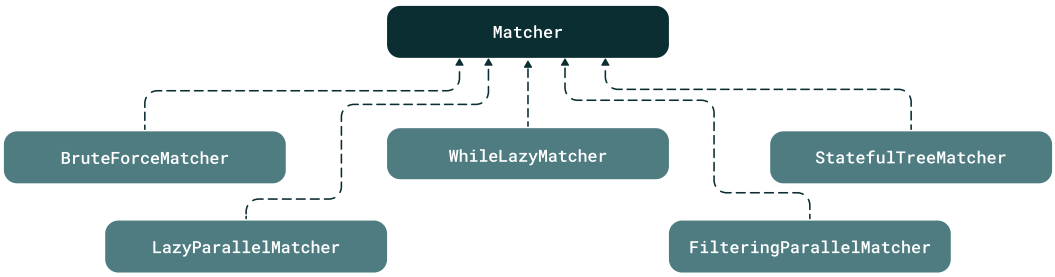
更准确地说，receive 宏的类型签名如清单 4 所示。它的第一个参数是清单 3 中所述偏函数的（代码），该偏函数定义了包含连接模式、守卫和表达式的**case** 子句。第二个参数是**MatcherFactory** 特质的一个实例（见下文清单 5），用于实例化所选的匹配算法。执行receive 宏时，它会从第一个参数中提取连接模式信息，并将其传递给matcherAlgorithm，以实例化实际的**Matcher**（更多详情请参见第 4.2.1 节）。

[4] Scala 3 枚举表示有限备选项（和类型），而 case类定义积类型；通常将它们组合在密封特质下，以提供封闭和。

■ **清单 5** 用于连接模式匹配器的 Matcher 特质，以及用于构造这些匹配器的MatcherFactory 特质。

```
特质 Matcher[M,T]:
    def apply(mbox: Mailbox[M])(self: ActorRef[M]): T

特质MatcherFactory:
    def 应用[M, T]: List[Join模式[M, T]] => 匹配器[M,T]
```



■ **图3** JoinActors 提供的部分连接模式匹配算法概览。

使用新的匹配算法扩展 JoinActors 我们设计了匹配算法的接口，旨在使JoinActors 库模块化且可扩展。图3概述了JoinActors 中已提供的一些匹配算法（并在第 4.2.2 节中进行了讨论）。

要提供新的匹配算法，程序员应当：

1. 定义一个包含算法逻辑的类，实现 匹配器 特质（清单 5）；
2. 创建一个负责实例化新匹配器类的伴生对象，通过扩展 MatcherFactory 特质（清单 5）。然后可以将此伴生对象作为第二个参数传递给 receive 宏（参见清单 4）。

如清单 5 所示，匹配器 通过将自身应用于邮箱 mbox （一个用于存放传入消息的线程安全阻塞队列），并使用 Actor 自己的“地址”self来运行；发生匹配时，匹配器应执行相应的代码（即清单 3 中的e_i），并返回其结果（类型为T）。

4.2 实现

现在我们介绍JoinActors 库的实现。我们首先说明代码生成过程（第 4.2.1 节），然后介绍该库包含的一些连接模式匹配算法及其优化（第 4.2.2 节）。

4.2.1 代码生成宏

receive 宏会检查作为第一个参数传入的inlined 闭包的带类型的抽象语法树（其类型为 ActorRef[M] =>PartialFunction[Any, Result[T]], ，如清单 4 所示）。通过检查，可以提取每个 case 的连接模式、守卫和右侧表达式，如清单6所示。闭包的主体预期

- **清单 6** receive 宏的简化片段：使用 Scala3反射 API 检查作为第一个参数传入的闭包代码（签名见清单 4）[33]。第 4 行中，self 是闭包参数（如清单 3 所示），而body 是闭包主体。第 6 行中，cases 是闭包主体中模式匹配**case**的列表；每个 case 都会被分解，以提取 case 模式、守卫和右侧表达式（分别对应清单 3 中的P_i、g_i和e_i）；这些内容用于第 9 行生成表示连接模式的代码。

```
1 closureAST 匹配 2 case Inlined(_, _, Inlined(_, _, Block(_, Block(List(stmt),_)))) =>3 stmt 匹配 4
case DefDef(_, List(TermParamClause(List( self ))),_, Some(Block(_, Block(List( body ),_)))) =>5 body
匹配 6 case DefDef(_, _, _, Some(Match(_, cases ))) =>7 cases.flatMap { 8 case CaseDef( pattern,
guard, rhs ) =>9 generateJoinPattern[M,T](pattern,g ,rhs,self) 10 }
```

- **清单 7** 表示连接模式的数据类型。 LookupEnv 是一个环境，它将消息模式中出现的变量映射到它们的值。

```
1 case 类Join模式[M, T](
2     守卫: LookupEnv => Boolean, // 用于计算守卫的闭包rhs: (LookupEnv, ActorRef[M]) => T, // 用于
3     执行此连接模式 RHS 的闭包size: Int, // 匹配此连接模式所需的消息数量matcherUtils: MatcherUtils[
4     M] // 用于匹配器实现的工具（例如用于计算 LookupEnv）
5
6 )
```

成为 Match 实例（清单 6 第 6–9 行），即表示模式匹配结构的 Scala 3 抽象语法树节点。模式匹配结构中的每个**case** 都包含三个组成部分：消息模式、可选的守卫和右侧（分别对应清单 3 中的P_i、g_i和e_i）。

如第 4.1 节所述，消息模式的形式为M_1(...) &:&M_2(...) &:&...&:&M_m(...)，其中每个M_j（对于$j \in 1..m$）都是一种消息类型。因此，receive 宏需要从模式中提取消息类型，包括消息负载的类型。这项提取工作由generateJoinPattern 完成（清单 6 第 9 行）；它生成用于实例化清单 7 所示JoinPattern 类的代码。其中，清单 7 中的字段 MatcherUtils[M] 封装了一组用于辅助匹配算法的实用工具。下面的示例 4.1 概述了生成代码的形式，随后我们将进一步阐述generateJoinPattern的实现。

示例 4.1（连接模式的代码生成）。 清单 8 展示了支付服务示例（清单 1）中第一个连接模式生成的代码。守卫和rhs 函数都将一个LookupEnv 实例作为参数，该实例将载荷标识符映射到从所选邮箱消息中提取的值。守卫 是

■ **Listing8** 第1段中第一个连接模式在receive 宏执行后的转换结果（为便于阅读，略作简化）。

```
Join模式[PaymentEvent,Unit类型](
  守卫 = (env: LookupEnv)  =>
            (环境("id1")。asInstanceOf[Int] == 环境("id2")。asInstanceOf[Int]) &&
            (env("id2").asInstanceOf[Int] == env("id3").asInstanceOf[Int]),
  右侧  = (环境:  LookupEnv, 自身:  ActorRef[PaymentEvent]) => {
            核心服务!  PaymentSucceeded(env("id1").asInstanceOf[Int])
            log(s"支付服务已处理支付请求 ${env("id1").asInstanceOf[Int]}")
            继续
  },
  大小 = 3
  matcherUtils = MatcherUtils[M](...)
)
```

在此环境中求值，以确定连接模式是否匹配所选消息：请注意，清单 1 中守卫的代码已被改写为使用 LookupEnv 实例 env，因此，例如，清单 1 中的变量 id1现在通过 env("id1").asInstanceOf[Int] 进行访问。5 当守卫 求值为**true**时，会使用与守卫相同的 LookupEnv 调用**rhs**，以执行连接模式右侧指定的计算；再次注意，rhs的代码也已像守卫一样进行了改写。■

生成一个 Join模式 实例，通过generateJoinPattern 函数 generateJoinPattern（清单 6 第 9 行）负责执行示例 4.1 和清单 8 中所概述的代码生成。对于从传递给 receive 宏的代码中提取的每个 CaseDef，generateJoinPattern 都会执行进一步分析，以确定该 CaseDef 表示一元消息模式（例如，**case**A(x) **=>** ...），还是由 **&:&** 运算符组合而成的 n元模式（例如，**case** A(x) **&:&** B(y) **=>** ...）。在这两种情况下，它都会委托给专用生成器，以恢复消息和负载类型信息，跟踪由模式绑定的变量，并合成用于执行运行时消息匹配的实用函数。这些函数会组装成一个 MatcherUtils[M] 实例（清单 7 第 5 行和清单 8 第 11 行），匹配算法可以使用该实例处理泛型类型为 M 的传入消息。MatcherUtils[M] 实例包含两个关键组件：

1. 一个按消息类型对模式位置进行分组的映射；以及

[5]向下转型 **asInstanceOf**[Int] 是必需的，因为 LookupEnv 会将绑定变量映射到类型为 Any 的值。这是因为消息负载（因此也包括由消息模式绑定的变量）可能具有不同的类型，所以 LookupEnv 只能使用 Any 作为唯一可保证的超类型。检查连接模式的抽象语法树时，receive 宏会推断向下转型的类型。例如，env("id1") 会被向下转型为 Int，因为在清单 1 中，变量id1 由消息模式PaymentRequested(id1)绑定，其载荷类型为 Int。

■ **表 1** 图3中的连接模式匹配算法所实现的优化。

匹配器类型	优化描述
暴力匹配器	缓存高效数据结构
有状态树匹配器	缓存高效数据结构
WhileLazyMatcher	缓存高效数据结构、高效迭代、惰性分支
惰性并行匹配器	缓存高效数据结构、高效迭代、惰性分支，并行分支
过滤并行匹配器	缓存高效数据结构、高效迭代、惰性分支，并行分支、守卫的部分求值

2. 一个将每个模式位置分别关联到以下内容的映射：*(i)* 一个用于识别传入消息确切类型的函数，以及*(ii)* 一个用从传入消息中提取的载荷值填充**LookupEnv**的函数。6

这种组织方式使匹配器实现能够先缩小可能匹配传入消息的模式位置范围（取决于消息的类型），然后提取载荷值。

4.2.2 连接模式匹配算法与优化

现在，我们重新梳理图3中列出的连接模式匹配算法，以讨论它们相较于 [17]，实现了哪些优化。 [17]，仅包含 **暴力匹配器** 和 **有状态树匹配器** 的早期版本（如第 2.2 节所述）。这些优化汇总于表1，并在下文中加以说明；更多详情见 [22]。值得注意的是，JoinActors的模块化设计使我们能够逐步实现并评估这些优化。我们在第5节比较了不同算法，并重点说明了不同优化在不同工作负载下如何带来不同的性能表现。

缓存高效数据结构 早期版本的 JoinActors 在 [17] 中使用不可变链表集合（**List**、**Queue**）。这种设计带来了 $O(n)$ 随机访问、较差的缓存局部性和装箱开销。我们将这些结构替换为**ArraySeq**，它提供连续的数组支持表示，访问复杂度为 $O(1)$，并针对基本类型采用专门的非装箱存储。具体而言，**Int** 值直接存储在 **Array**[**Int**],中，从而消除了装箱、指针间接寻址以及相关的垃圾回收开销。这项优化提升了缓存效率和整体性能，并统一应用于所有匹配器，包括 **暴力匹配器** 和 **有状态树匹配器**。

高效迭代 所有匹配算法都包含性能关键循环。最初的 JoinActors在 [17] 中使用 Scala **for**-each 循环，这类循环以效率低下而闻名，因为它们会被脱糖为分配闭包的代码，带来显著开销 [9]:

[6]每个 **MatcherUtils**[**M**] 实例还提供一个根据连接模式守卫的分析得出的过滤谓词：它用于第 4.2.2 节中讨论的守卫的部分求值。

■ **列表 9**　使用不透明类型和（内联）扩展方法实现高效的**for**-each 循环。

```
不透明 类型FastIterable[T] =IterableOnce[T]

扩展[T] (it: IterableOnce[T])
    def fast: FastIterable[T] = it
```

```
扩展[T](e：FastIterable[T])
    内联定义 foreach[U](内联 f:T =>U): Unit类型 =
        valit = e.迭代器
        当 it.hasNext 执行 f(it.next())
```

```
值 数字 = List(1, 2, 3)

对于 x <-数字执行
    f(x)
```

⇒会被反糖化为 ⇒

```
值数字 = List（1，2，3）
valforeachFunc = new Function1[Int,Unit类型]{
    def apply(x: Int): Unit类型 = f(x)}
numbers.foreach(foreachFunc)
```

因此，我们希望将这类 **for**-each 循环替换为 **while** 循环，它们会编译为高效的字节码。为了在替换时不牺牲代码清晰度，我们利用了 Scala 3 的若干特性：受 [3, 9]，启发，我们引入了一种 不透明类型FastIterable[T]（列表 9），它包装了 IterableOnce[T]，且不会产生运行时开销，并通过 扩展方法 fast提供显式的选择加入转换。随后，我们定义了一个 内联扩展方法 foreach；当脱糖后的 **for**-each 循环在类型为 FastIterable[T] 的表达式上调用该方法时，它会展开为一个 **while** 循环：

```
值 数字 = List(1, 2, 3)

遍历 x <-numbers.fast 执行
    f(x)
```

⇒展开为 ⇒

```
值数字 = List（1，2，3）
值它 = numbers.iterator

当it.hasNext 执行
    f(it.next())
```

惰性分支 如第 2.2 节和图 1 所述，有状态树匹配器算法 [17] 会维护一种树状状态，用于记录部分匹配；随着新消息到达，该状态会不断更新（“分支”），同时确保对树进行深度优先遍历时，总能得到最公平的匹配（如果存在）。有状态树匹配器会针对每条新消息对整个匹配树进行分支，然后进行深度优先遍历，以识别完整匹配、检查其守卫，并报告最公平的匹配（如果存在）。我们实现了另一种惰性 分支策略：新消息到达时，按深度优先顺序对匹配树进行分支；每当新消息完成一个部分匹配，就立即评估其守卫；如果结果为**真**，则停止分支并报告成功匹配。这种短路优化避免了匹配树不必要的增长。我们在WhileLazyMatcher、惰性并行匹配器和过滤并行匹配器中实现了该策略。

并行分支 我们通过将匹配树划分为 n 个子树来并行处理其分支，每个子树由一个独立线程处理（其中 n 是系统中央处理器支持的硬件线程数量）。这些分区按照公平性构建并排序，即对于任意两个分区 p 和 p'，如果 $p < p'$，那么在 p 中找到的任何匹配都比在 p'中找到的任何匹配更公平。当新消息到达时，每个线程都会对其分区执行惰性分支（如上所述）；每当某个线程报告匹配时，正在对公平性较低的分区进行分支的线程就会被中断。当所有线程终止后，系统会选择所报告的匹配中最公平的一个（如果有）。此优化在 惰性并行匹配器 和 过滤并行匹配器中实现。

守卫的部分求值 通过快速识别不可能满足守卫的传入消息，可以提高连接模式匹配的性能。为此，我们确定过滤子句：当守卫是多个谓词的合取时，如果某个谓词仅依赖于单个消息类型的载荷，且该消息类型在模式中只出现一次，我们就将该谓词归类为过滤子句。例如，考虑以下连接模式：

```
1 Foo(x : Int) &:& Bar(y : Int) &:& Bar(z : Int) 如果 x == 10 && y == 20 && z == x => ...
```

此处，x == 10 是一个过滤子句：它仅依赖于 Foo(x) 的载荷，且该消息类型在模式中只出现一次。过滤子句可在相应消息到达时立即求值——如果某条消息使其过滤子句不成立（例如，消息 Foo(42) 使 x == 10不成立），则可以忽略该消息（无需对匹配树进行分支），因为它不可能属于任何能够满足守卫的消息组合。请注意，谓词 y == 20 与 z == x 不是过滤子句，因为它们依赖于 Bar(y) 和 Bar(z) 的有效载荷——也就是说，它们依赖于在模式中多次出现的消息类型 Bar。因此，当消息 Bar(42) 到达时，即使它使谓词 y == 20不成立，我们也不会丢弃它；事实上，该消息仍可能属于某个能够满足谓词 z == x 的消息组合（如果 x == 42），因此丢弃 Bar(42) 是不正确的。与 [17], 相比，我们扩展了 receive 宏，以分析守卫的抽象语法树，提取其过滤子句，并生成可应用于传入消息的过滤闭包。此类过滤闭包由过滤并行匹配器使用，这一优化可以减小匹配树的规模及其分支成本。

5 通过对比基准测试进行评估

现在，我们通过测量其连接模式匹配算法的性能，评估改进后的 JoinActors 库。我们的评估大部分基于 [17]中最初提出的基准测试套件，用于评估连接模式匹配的性能。之所以选择这一基准测试，是因为据我们所知，这是唯一专门用于评估连接模式匹配性能的基准测试。具体而言，[17] 中的基准测试系统地改变了模式大小、守卫复杂度和传入消息流量（其中包含数量不同的乱序消息或不可匹配消息）。此外，该基准测试还展示了最坏情况分析，表明匹配时间可能随 Actor 邮箱中的消息数量呈指数增长：这说明优化对于实现合理的性能至关重要。评估包括合成基准测试（第5.1节）、对智能家居监控场景的改编（第5.2节）以及有界缓冲区基准测试（第5.3节）。对于每项基准测试，我们都会测量执行固定数量连接模式匹配所需的时间，以及每秒匹配次数的吞吐量；作为比较，结果还包括使用 [17]中介绍的 JoinActors 版本测得的基线。此外，在第5.4节中，我们将JoinActors与一个流行的参与者库 Apache Pekko 进行了比较。

此次评估还凸显了JoinActors 库设计的模块化特性，使我们能够在不同匹配算法之间轻松切换，并系统地比较它们的性能。此外，这种模块化设计还借助一个广泛的测试套件得以充分利用：在固定若干连接模式和输入流量后，该测试套件会在不同匹配算法之间切换，并检查每种算法是否都产生与 [17]中的原始实现相同的匹配结果。这对于实际应用也很重要：我们的评估（与 [17]中的评估一样）表明，连接模式匹配算法的效率高度依赖于具体的工作负载以及连接模式的形状。

实验设置 我们在一台配备英特尔酷睿 i7-1185G7 中央处理器（4 核 / 8 个线程，4.8 GHz）和 32 GB 随机存取存储器的计算机上运行实验，操作系统为 Pop!_OS 22.04。我们使用了采用默认设置的 OpenJDK 25，最大堆大小为 16 GB，并使用 Scala 3.7.3。每项基准测试前都会进行预热阶段：使用相同的基准测试逻辑运行 5 次，以便 Java 虚拟机执行即时编译和垃圾回收调优等优化。

5.1 合成基准测试

这些基准测试用于测量单个 Actor 的性能，该 Actor 尝试使用且仅使用一种连接模式来匹配所接收的消息。消息依次发送到 Actor 的邮箱，中间没有延迟。计时从发送第一条消息时开始；随后，Actor 执行 10 次连接模式匹配（根据传入流量的内容，这些匹配必定会发生），然后停止计时。Actor 使用的连接模式大小（即完成该模式所需的消息数量）为 1 到 5。我们有两种合成基准测试变体，详见下文。

无守卫的连接模式匹配性能在此基准测试中，该 Actor 具有一个不含条件守卫的连接模式。例如，一个大小为 5 的无守卫连接模式的 Actor 如下所示：

```
Actor[...,...] {2 接收 { self => {3 情况 A() &:& B() &:& C() &:&
D()&:& E()    => ...4 } } }
```

该基准测试评估两种工作负载：一种情况下，Actor 只接收可由图 4 中连接模式匹配的“干净”消息；另一种情况下，还包含无法匹配的额外“噪声”消息。图 5 展示了结果。总体而言，对于“干净”工作负载的基准测试，我们优化的匹配器比性能最佳的基线快[17] 至多 3.8倍；对于包含噪声消息的基准测试，则快至 31 倍；对于不带守卫的连接模式，“更简单”的有状态匹配器（即 WhileLazyMatcher 和 有状态树匹配器）表现优于其他匹配器。（关于相较于 [17], 的性能提升详情，请参见附录 B 中的表 2 和表 3。）值得注意的是，在“干净”的可匹配消息上，基线无状态暴力匹配器 比基线有状态树匹配器 快 [17],，这是因为

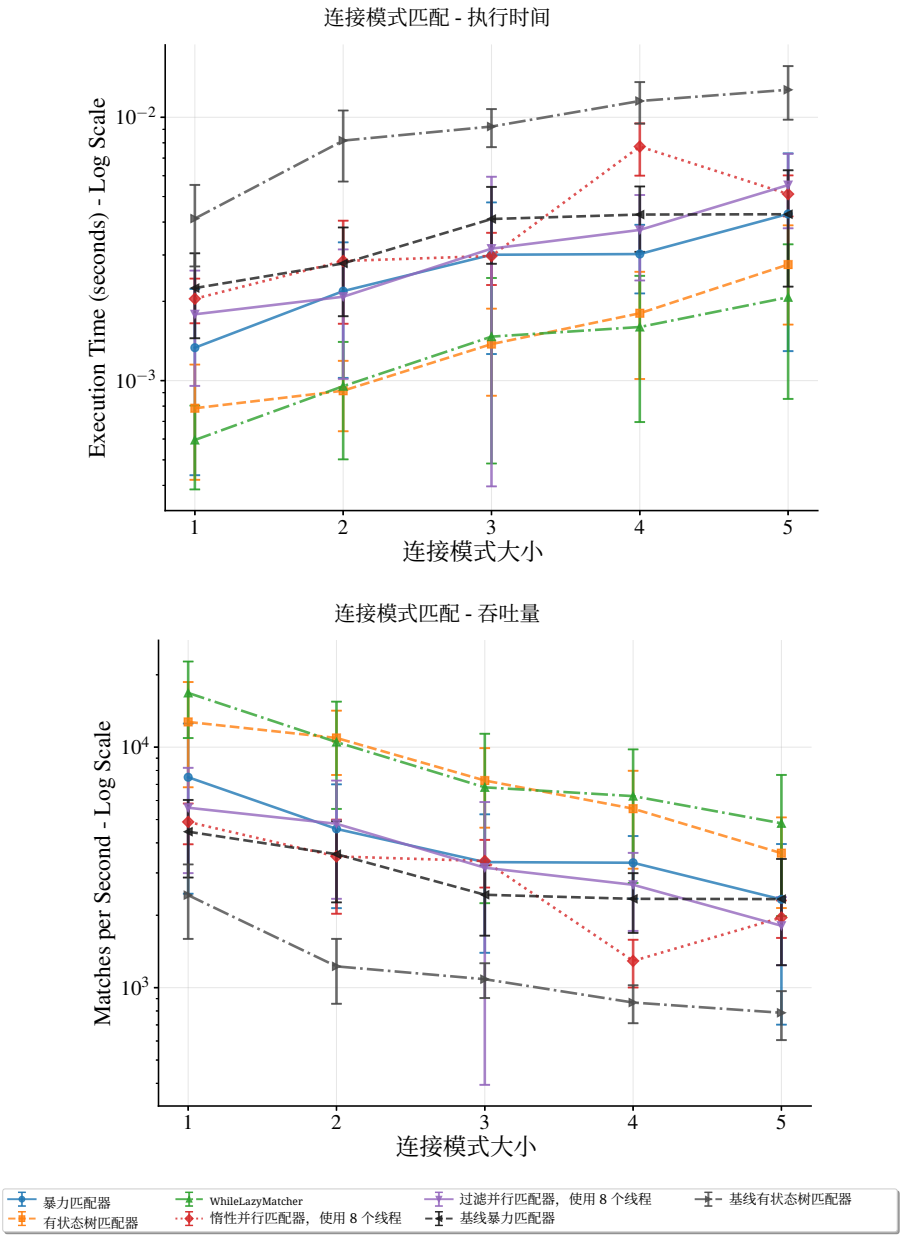


图 4 无守卫时连接模式匹配的性能：使用“干净”的可匹配消息。图中展示了 5 次重复的平均值，误差棒表示标准差。

状态维护的开销所致——但我们的优化带来了相反的结果，因为它们显著降低了状态维护的成本。在接下来的基准测试中也能观察到类似现象。

带守卫的连接模式匹配性能 在此基准测试中，Actor 有一个包含条件守卫的连接模式。例如，具有大小为 5 的带守卫连接模式的 Actor 如下所示：

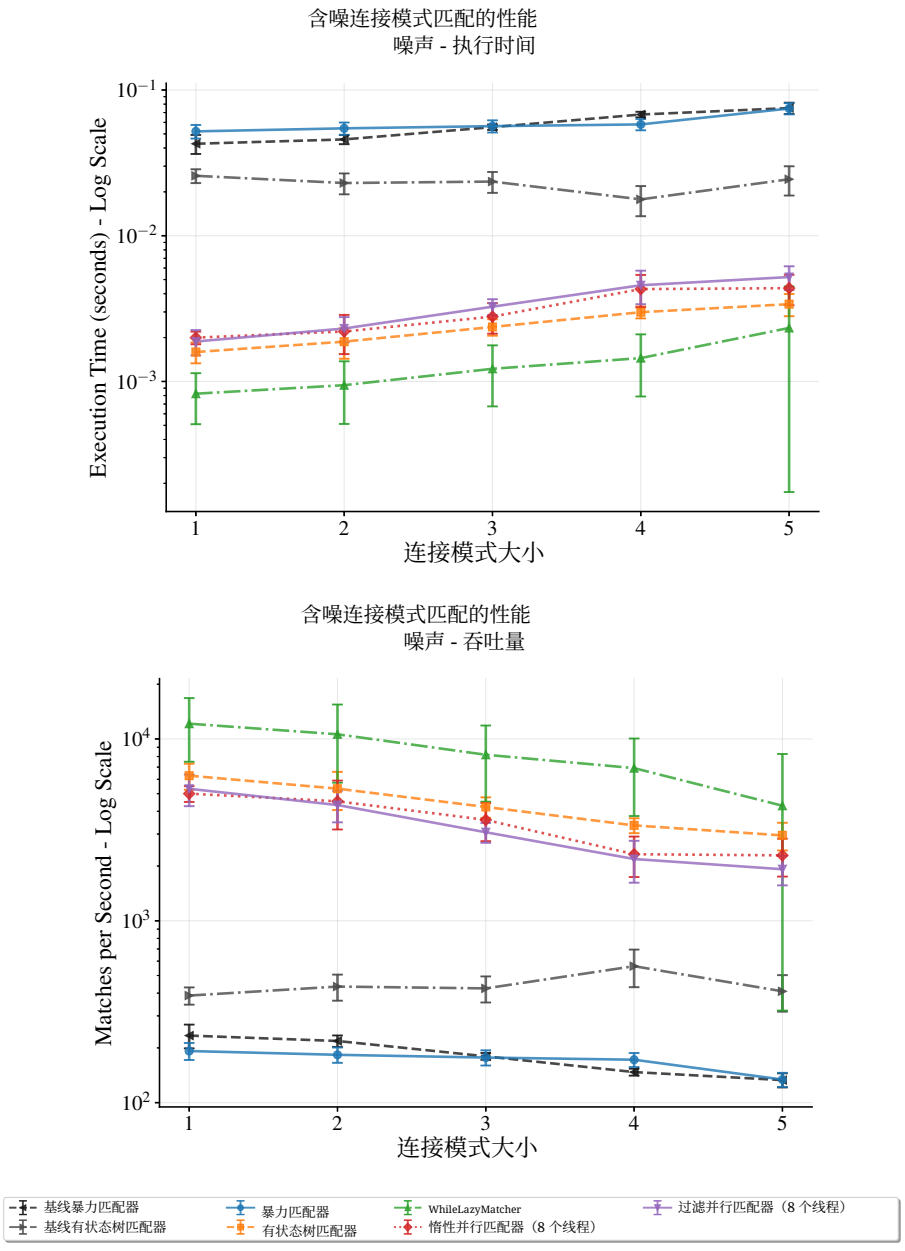


图 5 不带守卫的连接模式匹配性能：包含无法匹配的“噪声”消息。图中显示 5 次重复的平均值，误差棒表示标准差。

```
Actor[...,...] {接收{self => {情况 A(x) &:& B(y) &:& C(z) &:& D(w) &:& E(a) 如果 x == y && y ==z && z == w&& w ==a => ...} } }
```

该基准测试评估三种工作负载：
1. 一串“干净”的可匹配消息；
2. 一条包含额外“噪声”消息的流，这些消息的类型无法被匹配（例如，类型为 F的上述 Actor 消息）；以及3. 一条包含“噪声”消息的流，这些消息的类型可能被连接模式匹配，但其有效载荷不满足守卫。

图6、图7、图 8展示了所得测量结果。同样，我们的优化的匹配器总体上优于来自 [17]的表现最佳的基线，在“干净”工作负载上最高可达 4.25 倍，在“有噪声”工作负载上最高可达 22 倍，在带有不匹配的载荷的工作负载上最高可达 9 倍。（更多细节见附录 B 中的表 4、表 5 和表 6。）当“噪声”由无法匹配的消息类型构成时，较简单的有状态匹配器（即 WhileLazyMatcher和 有状态树匹配器）优于其他匹配器——但当“噪声”由带有不匹配的载荷的消息构成时，并行匹配器（即 惰性并行匹配器和过滤并行匹配器）则领先；请注意，过滤并行匹配器 较慢，因为该守卫不包含任何过滤子句。

5.2 智能家居基准测试

本基准测试改编自 [29] 中提出的一项案例研究（并由 [17] 复用）：一个 Actor 对智能家居监控器进行建模，该监控器接收来自多个家用设备的消息，并通过带有守卫的非平凡连接模式识别特定组合。值得注意的是，此基准测试中的连接模式存在显著重叠，因为许多模式共享相同的消息。7 这种重叠会影响匹配性能，因为单条消息可能在多棵匹配树中产生部分匹配（使用有状态匹配器时），或者可能导致多个模式同时得到满足，从而需要比较它们以找出最公平者。（关于此基准测试的更多细节见附录A。）

该基准测试衡量输入消息包含不同数量“噪声”时的匹配性能，即携带随机（且很可能无法匹配）有效载荷的消息。在图 9 中，x 轴显示与每组3 条可匹配消息一同发送的“噪声”消息数量。同样，随着“噪声”消息数量的增加，我们优化的匹配器的性能最高可达 [17] 中最佳基线的 58 倍。（详细分解见附录 B 中的表 7。）过滤并行匹配器优于其他算法：这是因为这些模式包含过滤子句，可以快速丢弃“噪声”消息，如代码清单10所示。

[7]第5.3节中的有界缓冲区基准测试也体现了连接模式之间的重叠，但此类模式比智能家居基准测试中的模式少得多，也简单得多。

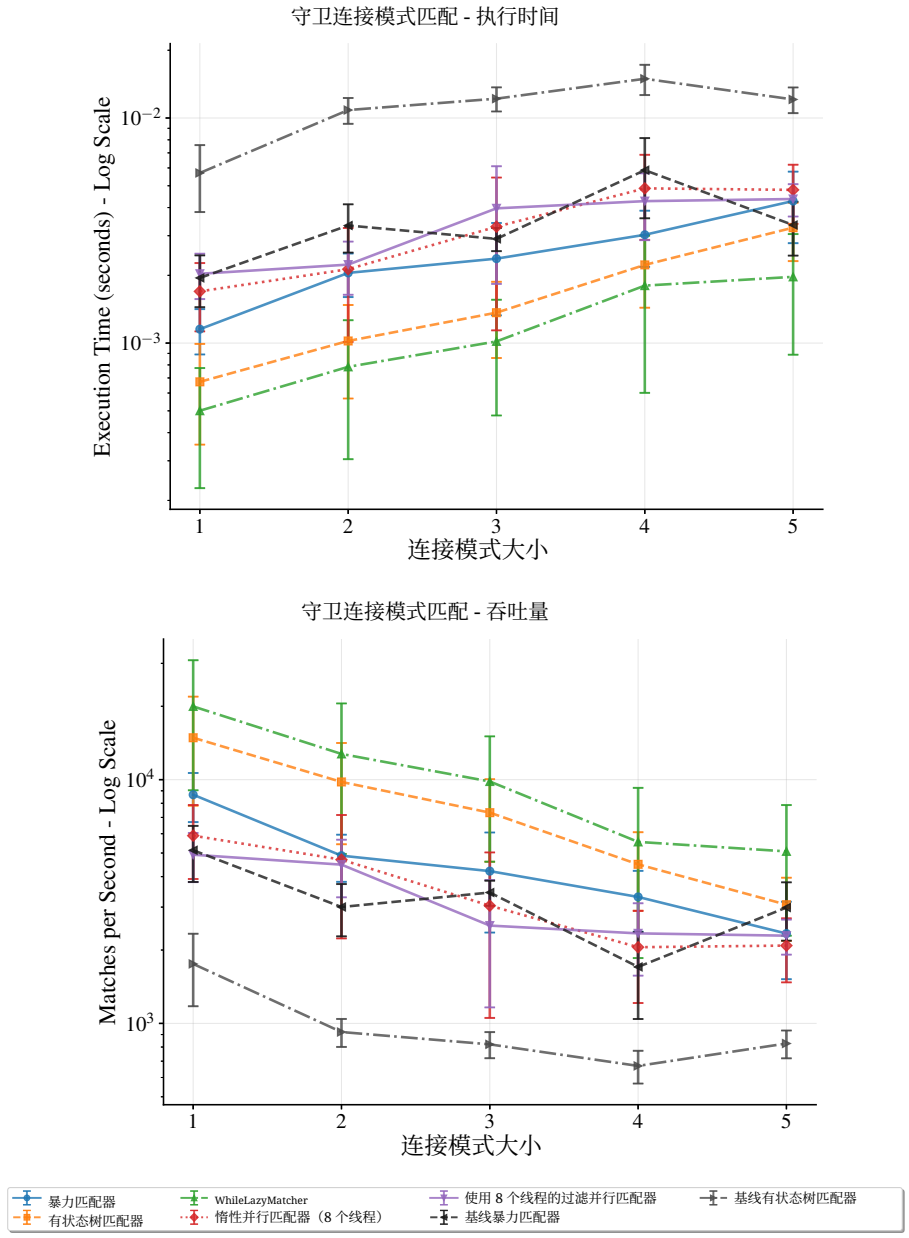


图6 带守卫的连接模式匹配性能：使用“干净”的可匹配消息。图中展示了 5 次重复的平均值，误差棒表示标准差。

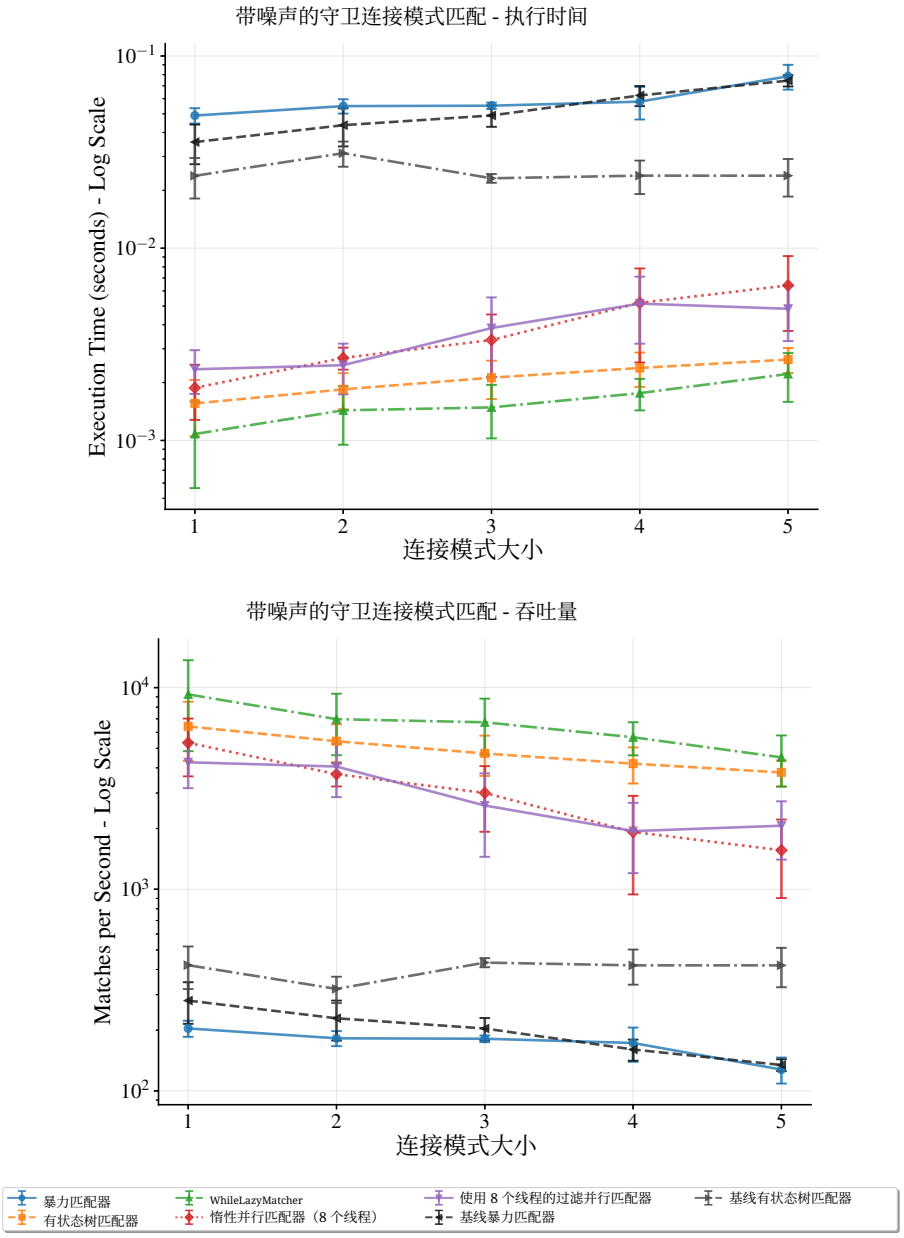


图 7 含守卫的连接模式匹配性能：包含无法匹配的“噪声”消息。图中显示 5 次重复的平均值，误差棒表示标准差。

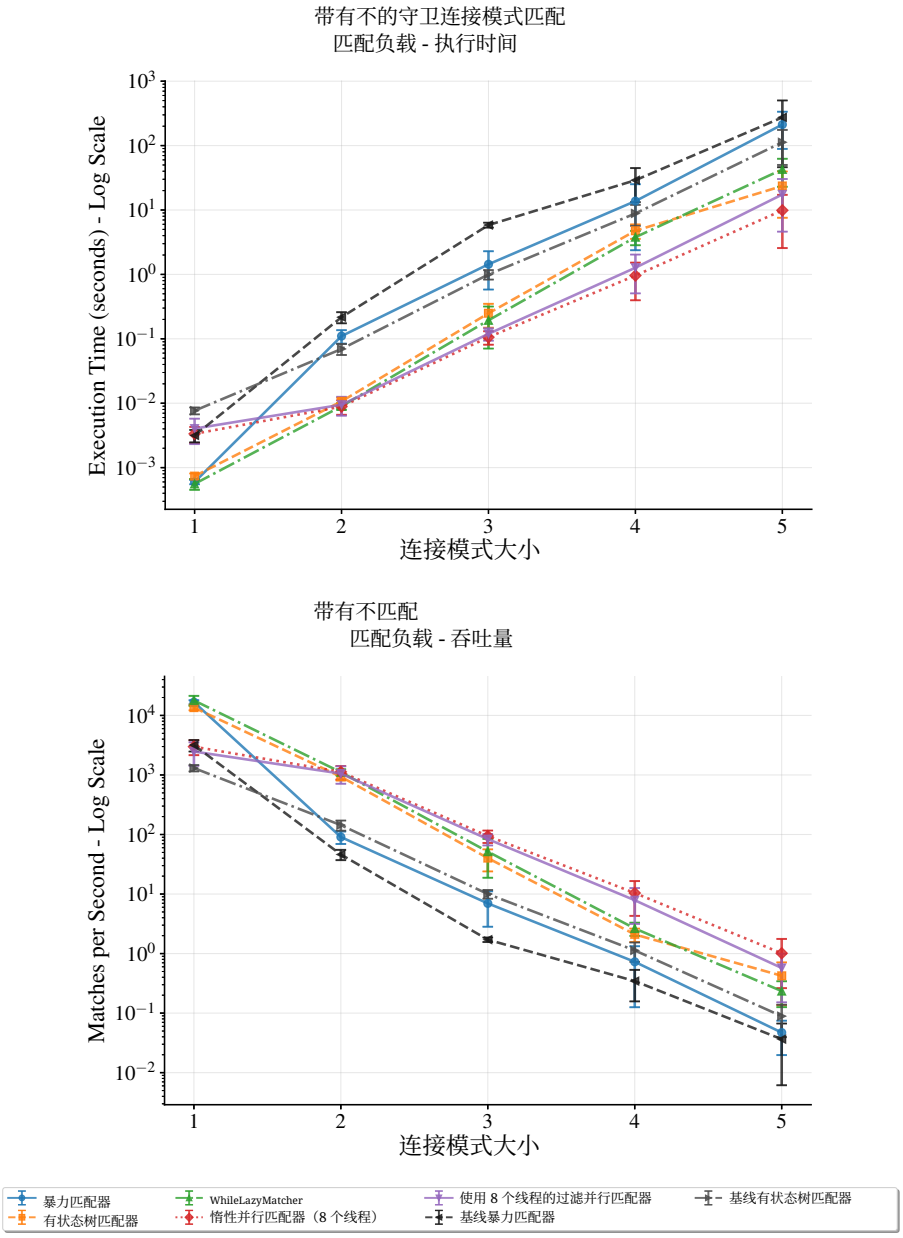
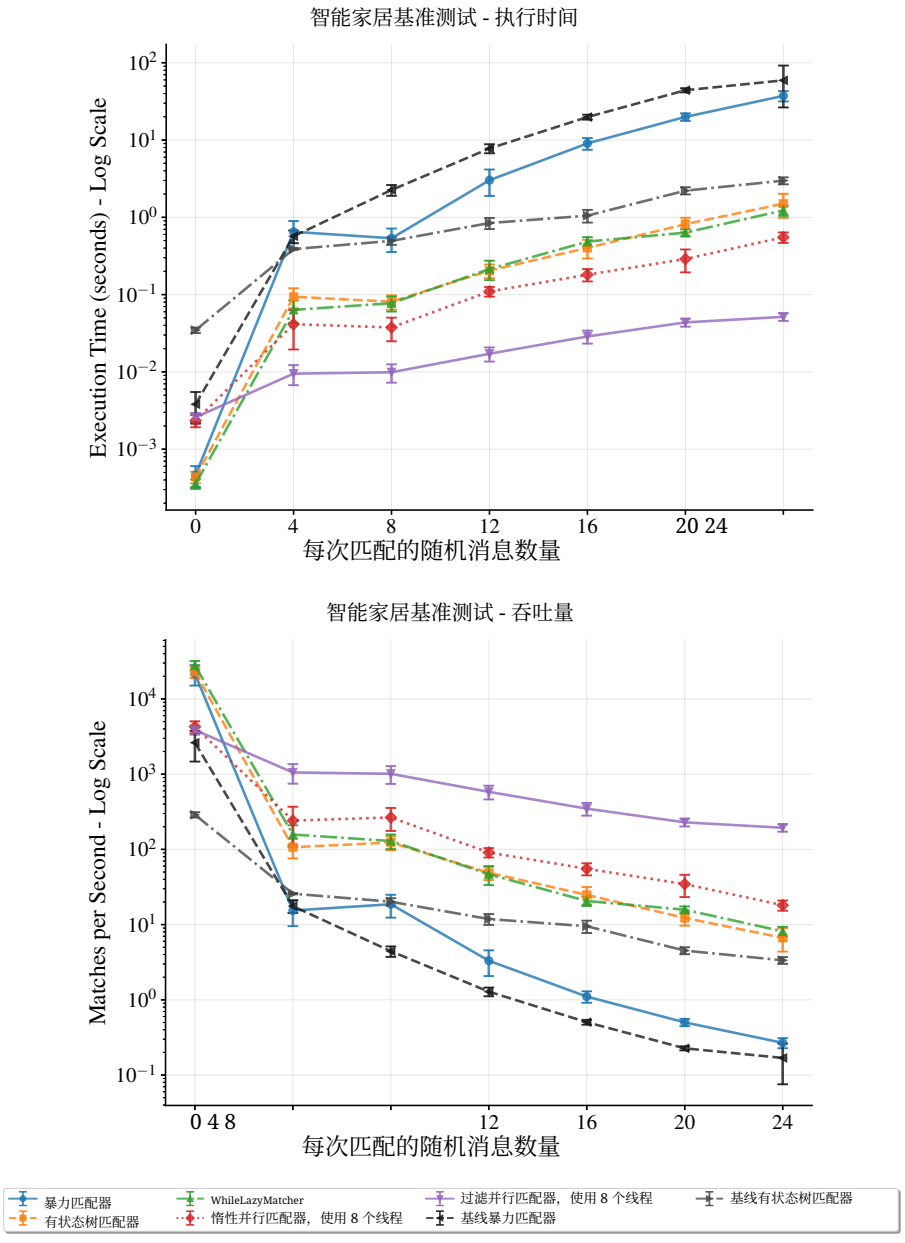


图 8 带守卫的连接模式匹配性能：使用不匹配的载荷。图中显示了 5 次重复的平均值，误差棒表示标准差。



■ **图9** 智能家居基准测试的性能。图中显示了 5 次重复的平均值，误差棒表示标准差。

列表10 智能家居监控基准测试中的简化连接模式：它匹配两条 运动 消息和一条接触 消息，并根据其有效载荷（包括检测到事件的房间）检查房屋是否有人居住。值得注意的是，守卫occupiedHome 具有谓词合取的结构，而依赖于 cRoomo 的谓词是一个过滤子句，可供 过滤并行匹配器 （第 4.2.2 节）利用。例如，匹配器会立即丢弃类似 接触(..., "gate_door",...) 的消息，因为它会将 cRoom 绑定到 "gate_door"，从而使过滤子句 cRoom =="front door"失效。_

```
1 inline def occupiedHome = (..., mRoom0: String, mRoom1: String, cRoom: String ) =>2 ... &&
mRoomo == "前_门" && cRoom =="前_门" && mRoom1 == "入口_大厅" 3 4 ... 5 case 运动(...., mRoom0:
String, ...)&:& 接触(..., cRoom: String, ...) &:&运动(..., mRoom1: String, ...) 6 if occupiedHome(...,
mRoom0, mRoom1, cRoom) => ...
```

5.3 生产者-消费者有界缓冲区基准测试

此基准测试（最初属于 Savina 套件 [20]，后来在 [17] 中进行了改编）模拟了经典的生产者-消费者问题：生产者和消费者都是 Actor，而缓冲区参与者通过连接模式协调它们之间的交互：缓冲区参与者跟踪缓冲区是否已满或为空，在数据不可用时将消费者加入队列，在缓冲区已满时将生产者加入队列，并在缓冲区有可用空间时通知生产者。此基准测试评估了JoinActors 在高并发和高竞争下的性能。

图 10 展示了缓冲区大小为 1000、包含 10 个生产者和 10 个消费者时的基准测试结果。在此情况下，我们的优化的匹配器同样优于 [17] 基线，最高约达 277 倍；详细分解请参见表 8。WhileLazyMatcher 实现了最佳性能，这符合预期，因为此基准测试使用的是相当简单的不带守卫的模式，无法利用并行算法的优化。

5.4 将 JoinActors 作为简单参与者库的评估

在本节中，我们将 JoinActors 与两个不支持连接模式匹配的参与者实现进行性能比较。我们的目标是评估 JoinActors 的连接模式匹配机制所引入的开销，方法是将其应用于仅需对每条传入消息进行简单模式匹配即可处理的情况。

作为比较的参考，我们选择了广受欢迎的参与者库 Apache Pekko[8]（不支持连接模式）。我们采用了第三方 Pekko 实现，其中包含参与者基准测试套件 Savina 的部分基准测试 [20]:

[8] https://pekko.apache.org/ （访问日期：2026-02-12）。

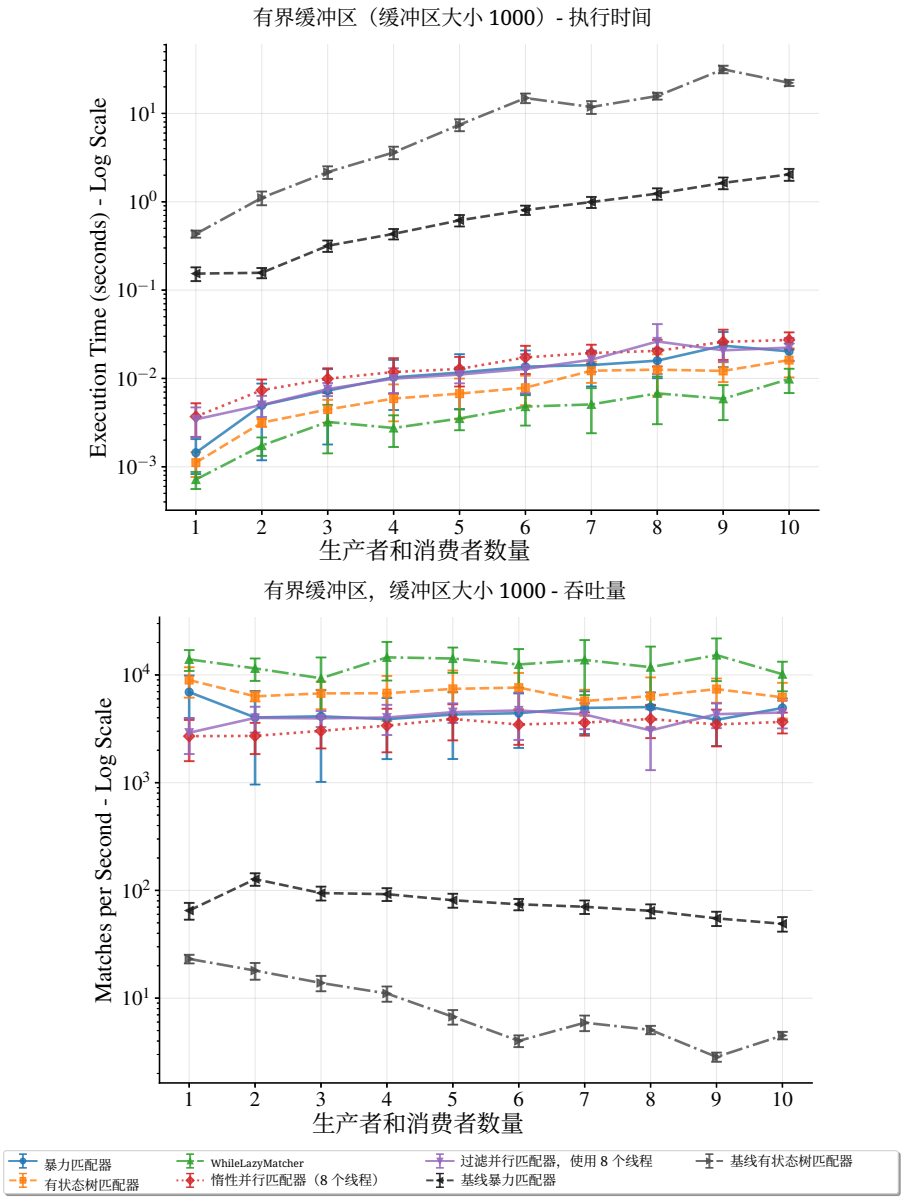


图10 缓冲区大小为 1000、包含 10 个生产者和 10 个消费者的有界缓冲区基准测试性能。图中展示了 5 次重复的平均值，误差棒表示标准差。

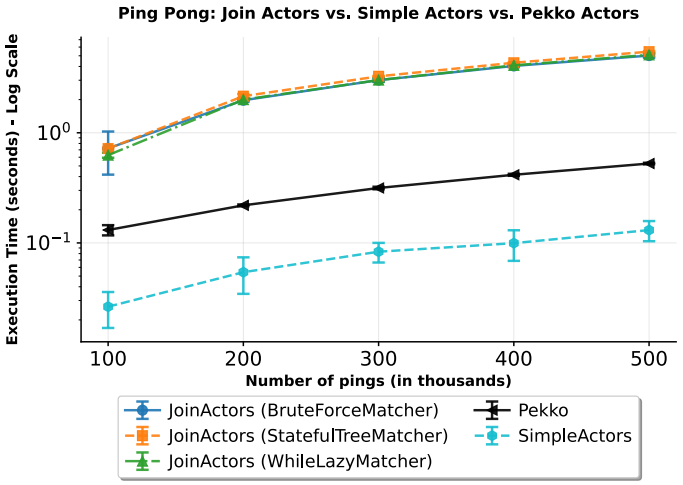


图 11 使用JoinActors 连接模式的乒乓演员基准测试性能。图中展示了 5 次重复运行的平均值，误差棒表示标准差。

具体而言，我们从 [27]的制品中选取了一些实现。9 随后，我们将每个选取的基准测试转换为最接近的Actor 等效版本，并在JoinActors下使用不同的连接模式匹配算法。最后，我们还使用JoinActors所采用的相同最小运行时，通过SimpleActor实现了每个选定的基准测试。SimpleActor 是一种精简的演员实现，只对每条传入消息执行标准模式匹配，与 Pekko 类似（见附录 C）。其理由是，使用简单模式匹配处理单条传入消息时：

1. Pekko 实现为广泛使用、经过优化且运行时功能强大的参与者库提供了性能参照。2. 基于SimpleActor的实现近似体现了使用最小运行时可达到的最佳性能。3. 基于JoinActors Actor的实现展示了连接模式匹配机制相较于SimpleActor所引入的开销。比较结果如下。

乒乓基准测试（图 11）两个 Actor 彼此来回发送消息，基准测试测量完成固定数量消息交互所需的时间。该基准测试高度依赖消息匹配性能，并展示了JoinActors的连接模式匹配机制带来的开销；其速度慢于 Pekko 和SimpleActor。

变色龙基准测试（图 12）多个变色龙（Actor）通过相遇时改变颜色来彼此互动，基准测试测量所需时间

[9]此类第三方基准测试实现也可在 https://github.com/dplyukhin/savina 上获取（访问日期：2026-02-12）。

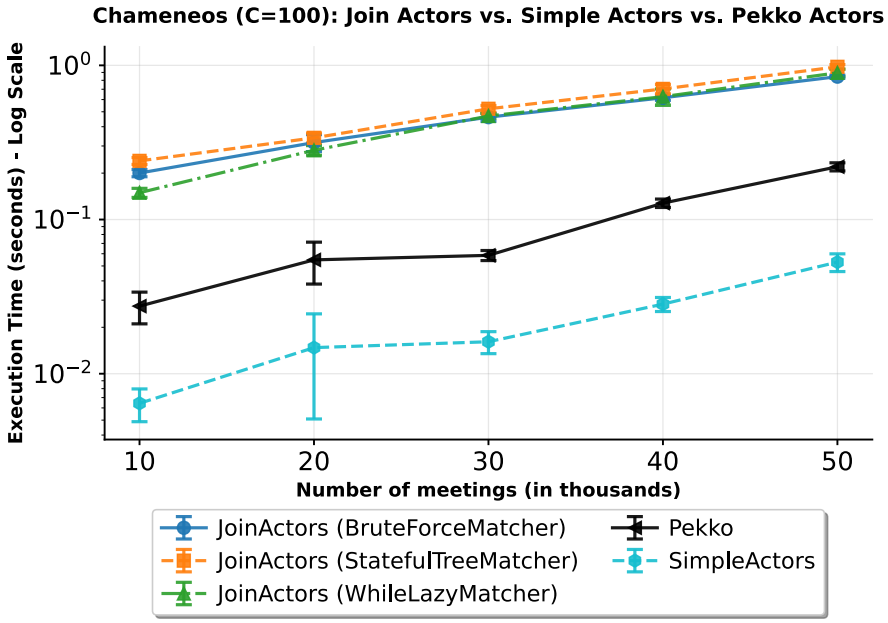


图 12 使用 JoinActors 连接模式的变色龙 Actor 基准测试性能。图中显示了 5 次重复的平均值，误差棒表示标准差。

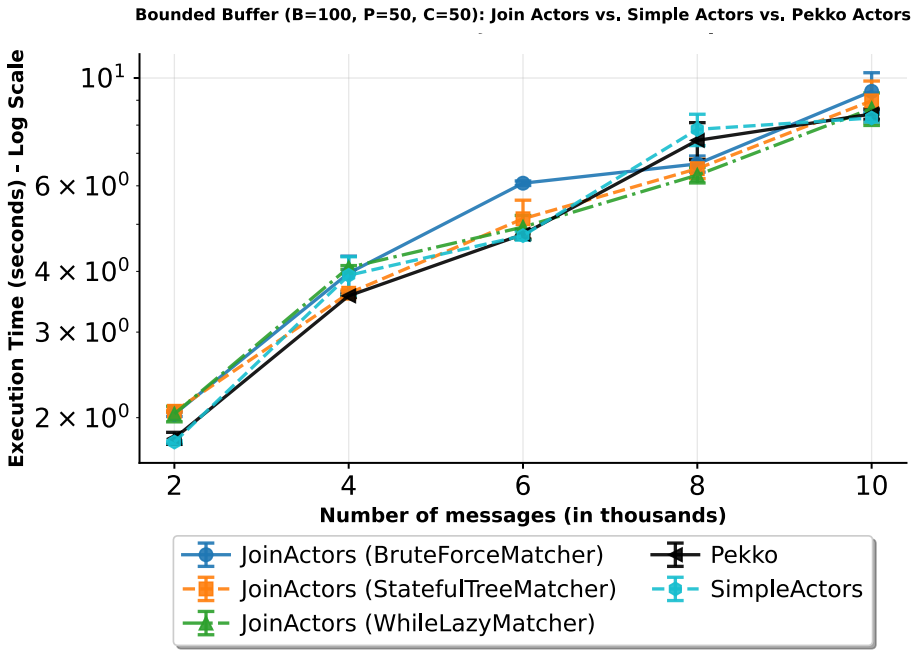


图 13 使用 JoinActors 连接模式的有界缓冲区 actor 基准测试性能。图中显示了 5 次重复的平均值，误差棒表示标准差。

以执行固定次数的会合。与乒乓基准测试类似，该基准测试高度依赖消息匹配性能。

有界缓冲区基准测试（图 13）该基准测试实现了一个有界缓冲区 actor，用于协调多个生产者和消费者 actor。[10] 该基准测试测量生产者和消费者通过缓冲区处理固定数量消息所需的时间；缓冲区大小设为 100，包含 50 个生产者和 50 个消费者。该基准测试受生产者 actor 与消费者 actor 之间的线程竞争影响，因此原始消息匹配性能的影响不如其他基准测试显著。因此，图中显示 Pekko 与JoinActors的 Actor和 SimpleActor表现相近。

总之，我们的测量结果表明，对于执行简单的逐条消息匹配的 actor：

1. JoinActors 的连接模式匹配机制会带来开销；在高度依赖匹配速度的基准测试中，这种开销尤为明显（图 11 和图 12）。[11]2. 如果线程竞争等消息匹配性能以外的因素变得重要，那么JoinActors的连接模式匹配机制所带来的开销影响较小（图 13）。3. 逐条消息进行匹配时，连接模式匹配算法的选择无关紧要，因为我们优化的算法均未能胜过暴力匹配器。（为清晰起见，图中仅展示JoinActors提供的部分匹配算法，其余算法的性能相近或更差。）

6 相关工作

自连接演算 [13] 问世以来，它已启发了跨多种编程语言的广泛实现 [2, 7, 10, 12, 14, 15, 16, 21, 23, 25, 26, 29, 30,31, 35, 36]。最初的实现严格遵循原始连接演算，并不支持对消息负载进行模式匹配；随后， [24] 将连接模式扩展为支持常量值匹配，而 [23] 又将这一思想推广到代数数据结构。之后， [16,26, 29]对更具表现力的模式匹配形式进行了研究。不同于 [23], , 这些系统支持带有守卫 的连接模式，其形式类似于 Erlang、Scala 或 F# 等函数式编程语言中的守卫。守卫的引入提高了连接模式的表现力和可用性，但也增加了实现的复杂性。为支持守卫，[16] 采用无状态的蛮力策略，

[10]请注意，这种有界缓冲区的实现与第5.3节中的实现不同：后者旨在使用连接模式来匹配多个生产者/消费者消息的组合，而 Savina 版本则一次只进行简单的单条消息匹配。[11]原则上，我们的 receive 宏（第 4.2.1 节）可以识别给定的一组连接模式是否一次只匹配一条消息，从而保持相应代码不变，进而实现与SimpleActor类似的性能。不过，我们尚未实现这一优化。

而 [26] 维护部分匹配的缓存，以减少冗余计算。相比之下， [29] 采用了 RETE 算法的一种变体 [11], , 维护一个由部分匹配组成的判别网络 ——尽管这一实现并未公开。后来， [17] 引入了新的匹配算法（有状态和无状态），实现了确定性且公平的 匹配策略。

与本文最相关的工作是连接演算和连接模式匹配的语言级嵌入：

1. JoCaml [12] 通过连接模式和消息传递并发原语扩展了 OCaml。其运行时采用线程，并通过消息传递而非共享内存实现同步。

2. JErlang [26] 通过基于连接的同步扩展了 Erlang，以解决该语言对多进程协调的原生支持有限的问题。JErlang 以库的形式实现，但需要使用经过修改的 Erlang虚拟机，其中集成了基于 RETE 的优化，以提高效率。

3. Sparrow [29]是通过宏嵌入 Elixir 的领域特定语言。它将 Actor 模型从单消息匹配扩展到多消息匹配，并借鉴了复杂事件处理框架中的技术。它采用经过调整的 RETE 算法来实现连接模式匹配。

4. Scala Joins [16] 为 Scala 2 提供了基于库的实现，利用可扩展的模式匹配，主要面向基于 Actor 的应用。

在连接演算中，连接模式采用非确定性匹配策略进行解释：当多个消息组合均可用时，会以非确定的方式选取其中一个。上述实现（ [17] 除外）要么未指定匹配策略，要么采用首次匹配策略 [35]。 [5],对此问题作了部分处理：该文引入了并行对象监视器 （POM），通过调度器 协调并发对象，并由调度器实现特定的消息选择策略。具体而言， [5] 探讨了 POM 在抑制连接模式匹配非确定性方面的潜在应用：文中概述了一种有界缓冲区的实现，该实现基于 Polyphonic C# 的一个变体 [2] （它扩展了 C#，加入了受连接模式启发的构造），并使用 POM 调度器确保按到达顺序选择读写请求，而非以非确定方式选择。相比之下， [17] 引入并形式化了公平、确定性的连接模式匹配策略，该策略已在JoinActors的首个概念验证版本中实现。在本文中，我们重新审视并改进JoinActors，全面介绍其模块化、可扩展的设计；我们还为其扩展了额外的连接模式匹配算法（仍基于 [17]中的公平匹配策略），并展示了性能的显著提升。

7 结论与 Future 工作

我们介绍了JoinActors，这是一个模块化且可扩展的参与者库，为 Scala 3 带来了带守卫的连接模式。我们展示了它对 Scala 3 元编程功能的高级运用，使JoinActors 能够将带守卫的连接模式实现为常规 Scala 代码，而无需修改编译器或语言本身。

借助JoinActors的模块化特性，我们还介绍了新型优化连接模式匹配算法，与 [17] 中介绍的早期Join-Actors 原型相比，性能有了显著提升。我们的工作表明，JoinActors可作为研究试验平台，便于进一步探索连接模式匹配算法并试验新功能。

未来工作 基于 JoinActors 的模块化和可扩展架构，我们确定了若干值得进一步研究和开发的方向。

首先，我们计划进一步研究优化。例如，可以对receive 宏进行优化，使其能够检测连接模式是否每次只匹配一条消息（即其行为类似于对单条消息进行简单模式匹配），并在此类情况下绕过连接模式匹配机制，从而使 JoinActors 的**Actor** 性能可与 **SimpleActor**和 Apache Pekko（第 5.4 节）相媲美。此外，并行化在选定的基准测试中带来了可测量的性能提升：如图 9 所示，**惰性并行匹配器** 和**过滤并行匹配器**取得了最佳结果。这一结果表明，即使不采用**过滤并行匹配器**的守卫过滤，仅并行化本身也能相较于顺序算法带来性能收益。然而，其他基准测试表明，并行化的有效性高度依赖于工作负载，并不具有普适性。对当前匹配算法开展深入的性能分析（例如使用性能分析工具）可以进一步洞察其行为，从而指导实现选择和理论理解——事实上，本文介绍的优化正是通过迭代使用性能分析数据来识别最有前景的性能提升方向而选定的 [22]。开展消融研究（利用JoinActors的模块化设计，系统地添加或移除单项优化）还可以进一步确定，根据工作负载的不同，哪些优化对连接模式匹配性能的贡献最大。

扩展和优化JoinActors 的另一个灵感来源是关于复杂事件处理（CEP）的文献 [4, 6, 28]。一方面，CEP 中的一些技术可用于进一步优化连接模式匹配（不过，这类优化可能需要调整，才能满足我们对公平匹配的要求）。另一方面，JoinActors 可为采用动态可重配置规则执行 CEP 奠定基础：这是因为，JoinActors 的 receive 宏（通过generateJoinPattern生成，详见第 4.2.1 节）所生成的连接模式匹配结构也可以在运行时实例化（尽管JoinActors 目前尚未提供实现此功能的文档化 API）。

JoinActors 的未来版本可以探索一些改进，例如对代数数据类型进行递归模式匹配、限制 Actor 邮箱容量以加强资源控制，以及对复杂守卫进行更精细的部分求值。

得益于JoinActors的模块化设计，可以通过实现新的连接模式匹配算法来探索上述大多数研究方向；这些算法可以与现有算法并存，具体如第 4.1 节所述。

我们还计划研究如何将JoinActors的设计移植到其他语言。Rust 是一个自然而然的候选，因为它注重安全性和性能，拥有先进的类型系统，并支持通过宏进行元编程。

数据可用性声明 本文评估的制品包含所有重要的代码示例和脚本，可用于复现基准测试结果。你可以在 [19] 找到配套制品。

致谢 本研究受到达格施图尔研讨会 21372 上关于“连接模式/同步：下一代”[8, p. 54]的讨论启发。我们感谢 Hadi Abdalla、Arooj Chaudhry、Bence Gattyan、Mohamed Ali Resho、Bozhi Lyu、Leonard Theisler 和 Andrei Popa 共同设计了清单 1 所引用的原始支付系统。本研究部分由以下项目资助：Horizon Europe 项目 101093006（TaRDIS）、PRIN PNRR 项目 DeLICE（P20223T2MF），以及意大利 MUR 为 GSSI 提供的 2023—2027 年卓越系资助。

A 智能房屋监控示例

智能房屋监控示例（清单 11）中的连接模式呈现出以下行为：

照明控制： 每当收到表示运动检测和低环境照度的消息时，就打开灯光；**存在检测：** 识别表示居民到达或离开的消息序列，并作出相应反应。

列表11 一个使用JoinActors 库的智能家居监控与控制器。该 Actor 通过开启灯光以及检测居民到达或离开来管理家用设备

```
. 1 def smartHouseExample(matcherAlgorithm: MatcherFactory)  =2 var lastNotification = Date(0L) 3 var
last
Seq
mSt
lSta
Stri
o ==
Dat
stat
19 A
则打
valu
batl
Dat
检测
Con
t2: I
last
Stri
Boo
mR
正在
```

B 与 ECOOP'24 基线相比的性能提升

参见表 2、表 3、表 4、表 5、表 6、表 7 和表 8。

表 2 无守卫时连接模式匹配的性能比较。

算法	与基线相比	连接模式大小				
		1	2	3	4	5
	基线性能（每秒消息数）					
基线暴力匹配器	（参考）	4451 每秒消息数	3589 每秒消息数	2434 每秒消息数	2341 每秒消息数	2335 每秒消息数
基线有状态树匹配器	（参考）	2424 每秒消息数	1226 每秒消息数	1084 每秒消息数	867 每秒消息数	786 每秒消息数
	改进倍数					
暴力匹配器	对比基线暴力匹配器	1.68 倍	1.27 倍	1.37 倍	1.41 倍	1.00 倍
	相较于基线有状态树匹配器	3.09x	3.73x	3.07x	3.81x	2.96x
有状态树匹配器	对比基线暴力匹配器	2.86 倍	3.04 倍	2.99 倍	2.37 倍	1.55 倍
	对比基线有状态树匹配器	5.25 倍	8.91 倍	6.70 倍	6.40 倍	4.61 倍
WhileLazyMatcher	对比基线暴力匹配器	3.78x	2.93x	2.80x	2.67x	2.07x
	对比基线有状态树匹配器	6.93x	8.56x	6.28x	7.22x	6.14 倍
过滤并行匹配器	对比基线暴力匹配器	1.26 倍	1.34 倍	1.30 倍	1.14 倍	0.77 倍
	对比基线有状态树匹配器	2.31 倍	3.92 倍	2.91 倍	3.09x	2.30x
惰性并行匹配器	相较于基线暴力匹配器	1.10x	0.98x	1.38x	0.55x	0.84x
	与基线有状态树匹配器相比	2.02x	2.87x	3.10倍	1.49倍	2.49倍

Table 3 Performance comparison for join pattern matching without guards with noise

Algorithm	vs Baseline	Join Pattern Size				
		1	2	3	4	5
Baseline Performance (messages per second)						
Baseline BruteForceMatcher	(reference)	234 mps	218 mps	180 mps	147 mps	133 mps
Baseline StatefulTreeMatcher	(reference)	388 mps	435 mps	425 mps	563 mps	410 mps
Improvement Factors						
BruteForceMatcher	vs Baseline BruteForceMatcher	0.82x	0.84x	0.98x	1.17x	1.01x
	vs Baseline StatefulTreeMatcher	0.50x	0.42x	0.42x	0.31x	0.33x
StatefulTreeMatcher	vs Baseline BruteForceMatcher	26.87x	24.41x	23.51x	22.72x	22.16x
	vs Baseline StatefulTreeMatcher	16.19x	12.25x	9.95x	5.94x	7.20x
WhileLazyMatcher	vs Baseline BruteForceMatcher	51.90x	48.58x	45.54x	46.93x	32.26x
	vs Baseline StatefulTreeMatcher	31.27x	24.39x	19.26x	12.28x	10.48x
FilteringParallelMatcher	vs Baseline BruteForceMatcher	22.76x	19.84x	17.07x	14.85x	14.44x
	vs Baseline StatefulTreeMatcher	13.71x	9.96x	7.22x	3.88x	4.69x
LazyParallelMatcher	vs Baseline BruteForceMatcher	21.40x	20.80x	19.96x	15.77x	17.21x
	vs Baseline StatefulTreeMatcher	12.89x	10.44x	8.44x	4.13x	5.59x

Table 4 Performance improvement factors for guarded join pattern matching.

Algorithm	vs Baseline	Join Pattern Size				
		1	2	3	4	5
	Baseline Performance (messages per second)					
Baseline BruteForceMatcher	(reference)	5133 mps	3003 mps	3443 mps	1705 mps	2988 mps
Baseline StatefulTreeMatcher	(reference)	1754 mps	922 mps	821 mps	669 mps	827 mps
	Improvement Factors					
BruteForceMatcher	vs Baseline BruteForceMatcher	1.69x	1.62x	1.22x	1.94x	0.78x
	vs Baseline StatefulTreeMatcher	4.94x	5.29x	5.14x	4.93x	2.83x
StatefulTreeMatcher	vs Baseline BruteForceMatcher	2.90x	3.26x	2.13x	2.63x	1.03x
	vs Baseline StatefulTreeMatcher	8.47x	10.62x	8.93x	6.71x	3.71x
WhileLazyMatcher	vs Baseline BruteForceMatcher	3.89x	4.25x	2.85x	3.26x	1.70x
	vs Baseline StatefulTreeMatcher	11.38x	13.83x	11.98x	8.30x	6.14x
FilteringParallelMatcher	vs Baseline BruteForceMatcher	0.96x	1.49x	0.73x	1.37x	0.77x
	vs Baseline StatefulTreeMatcher	2.81x	4.86x	3.07x	3.50x	2.77x
LazyParallelMatcher	vs Baseline BruteForceMatcher	1.15x	1.56x	0.88x	1.20x	0.70x
	vs Baseline StatefulTreeMatcher	3.36x	5.09x	3.71x	3.07x	2.52x

Table 5 Performance improvement factors for guarded join pattern matching with noise messages.

Algorithm	vs Baseline	Join Pattern Size				
		1	2	3	4	5
Baseline Performance (messages per second)						
Baseline BruteForceMatcher	(reference)	280 mps	229 mps	204 mps	160 mps	134 mps
Baseline StatefulTreeMatcher	(reference)	420 mps	320 mps	433 mps	419 mps	419 mps
Improvement Factors						
BruteForceMatcher	vs Baseline BruteForceMatcher	0.73x	0.80x	0.89x	1.08x	0.95x
	vs Baseline StatefulTreeMatcher	0.49x	0.57x	0.42x	0.41x	0.30x
StatefulTreeMatcher	vs Baseline BruteForceMatcher	22.90x	23.67x	23.12x	26.16x	28.24x
	vs Baseline StatefulTreeMatcher	15.28x	16.92x	10.89x	10.01x	9.05x
WhileLazyMatcher	vs Baseline BruteForceMatcher	33.01x	30.44x	33.01x	35.37x	33.56x
	vs Baseline StatefulTreeMatcher	22.03x	21.76x	15.55x	13.53x	10.75x
FilteringParallelMatcher	vs Baseline BruteForceMatcher	15.20x	17.72x	12.76x	12.11x	15.37x
	vs Baseline StatefulTreeMatcher	10.15x	12.67x	6.01x	4.63x	4.93x
LazyParallelMatcher	vs Baseline BruteForceMatcher	18.99x	16.24x	14.73x	12.00x	11.62x
	vs Baseline StatefulTreeMatcher	12.68x	11.61x	6.94x	4.59x	3.72x

Table 6 Performance improvement factors for guarded join pattern matching with non-matching payloads.

Algorithm	vs Baseline	Join Pattern Size				
		1	2	3	4	5
	Baseline Performance (messages per second)					
Baseline BruteForceMatcher	(reference)	3167 mps	46 mps	2 mps	0 mps	0 mps
Baseline StatefulTreeMatcher	(reference)	1306 mps	144 mps	10 mps	1 mps	0 mps
	Improvement Factors					
BruteForceMatcher	vs Baseline BruteForceMatcher	5.28x	1.96x	4.06x	2.11x	1.29x
	vs Baseline StatefulTreeMatcher	12.81x	0.63x	0.70x	0.64x	0.53x
StatefulTreeMatcher	vs Baseline BruteForceMatcher	4.28x	20.26x	23.44x	6.08x	11.61x
	vs Baseline StatefulTreeMatcher	10.37x	6.51x	4.01x	1.85x	4.77x
WhileLazyMatcher	vs Baseline BruteForceMatcher	5.65x	24.11x	30.10x	7.66x	6.42x
	vs Baseline StatefulTreeMatcher	13.69x	7.75x	5.15x	2.34x	2.64x
FilteringParallelMatcher	vs Baseline BruteForceMatcher	0.78x	22.84x	48.84x	22.84x	15.72x
	vs Baseline StatefulTreeMatcher	1.90x	7.34x	8.35x	6.97x	6.46x
LazyParallelMatcher	vs Baseline BruteForceMatcher	0.94x	24.43x	55.08x	30.18x	27.75x
	vs Baseline StatefulTreeMatcher	2.29x	7.85x	9.42x	9.21x	11.40x

Table 7 Performance improvement factors for smart house benchmark.

Algorithm	vs Baseline	Number of Random Messages per Match						
		0	4	8	12	16	20	24
Baseline Performance (messages per second)								
Baseline BruteForceMatcher	(reference)	2620 mps	18 mps	4 mps	1 mps	1 mps	0 mps	0 mps
Baseline StatefulTreeMatcher	(reference)	289 mps	26 mps	20 mps	12 mps	10 mps	5 mps	3 mps
Improvement Factors								
BruteForceMatcher	vs Baseline BruteForceMatcher	8.24x	0.88x	4.21x	2.57x	2.19x	2.21x	1.59x
	vs Baseline StatefulTreeMatcher	74.80x	0.60x	0.93x	0.28x	0.12x	0.11x	0.08x
StatefulTreeMatcher	vs Baseline BruteForceMatcher	8.75x	6.07x	27.92x	38.06x	49.26x	54.13x	39.49x
	vs Baseline StatefulTreeMatcher	79.40x	4.14x	6.13x	4.12x	2.61x	2.72x	1.99x
WhileLazyMatcher	vs Baseline BruteForceMatcher	10.69x	8.97x	29.21x	36.33x	40.86x	69.60x	48.17x
	vs Baseline StatefulTreeMatcher	97.00x	6.11x	6.41x	3.94x	2.17x	3.49x	2.43x
FilteringParallelMatcher	vs Baseline BruteForceMatcher	1.47x	59.98x	228.52x	453.02x	690.27x	1009.59x	1148.28x
	vs Baseline StatefulTreeMatcher	13.38x	40.88x	50.18x	49.10x	36.58x	50.68x	57.83x
LazyParallelMatcher	vs Baseline BruteForceMatcher	1.63x	13.74x	60.07x	70.79x	109.53x	152.61x	107.18x
	vs Baseline StatefulTreeMatcher	14.79x	9.37x	13.19x	7.67x	5.81x	7.66x	5.40x

Table 8 Performance improvement factors for bounded buffer benchmark.

Algorithm	vs Baseline	Number of Producers and Consumers									
		1	2	3	4	5	6	7	8	9	10
Baseline Performance (messages per second)											
Baseline BruteForceMatcher	(reference)	65 mps	127 mps	94 mps	92 mps	81 mps	74 mps	71 mps	65 mps	55 mps	49 mps
Baseline StatefulTreeMatcher	(reference)	23 mps	18 mps	14 mps	11 mps	7 mps	4 mps	6 mps	5 mps	3 mps	5 mps
Improvement Factors											
BruteForceMatcher	vs Baseline BruteForce Matcher	106.58x	31.65x	43.71x	42.08x	52.91x	59.39x	70.11x	77.84x	69.61x	100.78x
	vs Baseline Stateful TreeMatcher	299.68x	223.55x	298.42x	351.61x	638.74x	1100.59x	835.15x	991.31x	1346.13x	1096.15x
StatefulTreeMatcher	vs Baseline BruteForce Matcher	137.84x	49.74x	71.37x	73.25x	91.57x	102.71x	81.24x	98.40x	134.28x	126.55x
	vs Baseline Stateful TreeMatcher	387.58x	351.34x	487.22x	612.00x	1105.57x	1903.36x	967.72x	1253.07x	2596.75x	1376.49x
WhileLazyMatcher	vs Baseline BruteForce Matcher	214.50x	90.40x	98.61x	157.74x	175.24x	168.17x	195.45x	182.54x	277.78x	207.33x
	vs Baseline Stateful TreeMatcher	603.14x	638.53x	673.16x	1317.92x	2115.71x	3116.33x	2328.17x	2324.60x	5371.96x	2255.08x
FilteringParallelMatcher	vs Baseline BruteForce Matcher	44.71x	31.36x	41.85x	43.60x	55.69x	62.97x	61.00x	47.20x	78.55x	91.48x
	vs Baseline Stateful TreeMatcher	125.72x	221.51x	285.68x	364.27x	672.31x	1166.99x	726.56x	601.08x	1519.10x	994.98x
LazyParallelMatcher	vs Baseline BruteForce Matcher	41.55x	21.35x	32.02x	36.66x	48.16x	46.60x	51.23x	60.30x	63.06x	74.89x
	vs Baseline Stateful TreeMatcher	116.83x	150.77x	218.61x	306.27x	581.40x	863.63x	610.18x	767.95x	1219.46x	814.60x

C 原型参与者实现

清单 12 展示了JoinActors库中提供的原型演员实现。

清单 12 演员实现用于 JoinActors：**Actor** （支持连接模式 n
匹配）以及**SimpleActor** （不支持连接模式匹配）。

```
enum结果[+T]:

 case停止(值: T)
 case 继续


导入结果.*

类 Actor[M,T](私有值 匹配器：匹配器[M, 结果[T]]):
 私有值 邮箱：Mailbox[M] = Mailbox[M]
 私有值 self                = ActorRef(邮箱)

 定义 启动():(Future[T], ActorRef[M]) =
  值承诺 = Promise[T]

   执行上下文.execute() => 运行(promise))


   (promise.future, self)


 @尾递归
 private def 运行(promise:Promise[T]): Unit类型 =
  matcher(mailbox)(self) 匹配
   case 继续          => 运行(promise)
   case 停止(value) => promise.成功(value)


class SimpleActor[M, T](privateval f: ActorRef[M] =>偏函数[Any,结果[T]]):

 private val 邮箱:Mailbox[M] = Mailbox[M]
 私有 val 自身 = ActorRef(邮箱)

 def 启动():(Future[T], ActorRef[M]) =
  val Promise = Promise[T]

   执行上下文.execute() => 运行(promise))


   (promise.future, self)

 @尾递归
 私有 def 运行(promise：Promise[T]): Unit类型 =
  f(self).applyOrElse[M, 结果[T]](mailbox.take(), _ => 继续）匹配
   case 继续 => 运行(promise)
   情况 停止（值） => promise.success(value)
```

参考文献

[1] Gul A. Agha。《Actors：分布式系统中并发计算的一种模型》。载于：*MIT Press* 人工智能丛书。1986年。doi：10.7551/mitpress/1086.001.0001。[2] Nick Benton、Luca Cardelli 和 Cédric Fournet。《C# 的现代并发抽象》。载于：*ACM Trans. Program. Lang. Syst.* 26.5（2004），第769–804页。doi：10.1145/1018203.1018205。[3] *Scala 3* 中的 *C* 风格 *for* 循环。网址：https://august.nagro.us/scala-for-loop.html（访问于2025-09-17）。[4] Federico Cabitza、Marcello Sarini 和 Bernardo Dal Seno。《DJess——用于在普适计算领域部署分布式推理系统的上下文共享中间件》。英文。载于：普适服务国际会议论文集，*Icps '05* 2005（2005），第1506416页。doi：10.1109/PERSER.2005.1506416。[5] Denis Caromel、Luis Mateu、Guillaume Pothier 和 Éric Tanter。《并行对象监视器》。载于：*Concurr. Comput. Pract. Exp.* 20.12（2008），第1387–1417页。doi：10.1002/CPE.1261。[6] *CLIPS*：构建专家系统的工具。网址：https://www.clipsrules.net（访问于2026-01-01）。[7] Silvia Crafa 和 Luca Padovani。《面向类型状态编程的化学方法》。载于：*2015*年 *ACM SIGPLAN* 面向对象编程、系统、语言与应用国际会议论文集（*OOPSLA 2015*，*SPLASH 2015* 的一部分），美国宾夕法尼亚州匹兹堡，*2015*年*10*月*25*日至*30*日。由 Jonathan Aldrich 和 Patrick Eugster 编。ACM，2015年，第917–934页。doi：10.1145/2814270.2814287。[8] Mariangiola Dezani、Roland Kuhn、Sam Lindley 和 Alceste Scalas。《行为类型：连接理论与实践（Dagstuhl 研讨会 21372）》。载于：*DagstuhlReports* 11.8（2022）。由 Mariangiola Dezani、Roland Kuhn、Sam Lindley 和 Alceste Scalas 编，第52–75页。ISSN：2192-5283。doi：10.4230/DagRep.11.8.52。[9] 为简单的 *for* 推导式生成更高效的代码。网址：https://github.com/scala/scala3/issues/8880（访问于2025-09-17）。[10] Fabrice Le Fessant 和 Luc Maranget。《连接模式的编译》。载于：*ElectronicNotes inTheoretical Computer Science* 16.3（1998）。由 Uwe Nestmann 和 Benjamin C. Pierce 编，第205–224页。doi：10.1016/S1571-0661(04)00143-4。[11] Charles Forgy。《Rete：解决多模式/多对象匹配问题的快速算法》。载于：*Artif. Intell.* 19.1（1982），第17–37页。doi：10.1016/0004-3702(82)90020-0。[12] Cédric Fournet、Fabrice Le Fessant、Luc Maranget 和 Alan Schmitt。《JoCaml：一种用于并发、分布式和移动编程的语言》。载于：高级函数式编程：第*4*届国际学校，*AFP 2002*，英国牛津，*2002*年*8*月*19*日至*24*日，修订版讲义。由 Johan Jeuring 和 Simon

L. Peyton Jones。《计算机科学讲义》第2638卷。Springer，2002年，第129–158页。doi:10.1007/978-3-540-44833-4_5.[13] Cédric Fournet和Georges Gonthier。《反射式CHAM与Join演算》。载于：*POPL'96*会议记录：第*23*届*ACM SIGPLAN-SIGACT*编程语言原理研讨会，研讨会论文，圣彼得堡海滩，佛罗里达州，美国，*1996*年*1*月*21*日至*24*日。Hans-Juergen Boehm和Guy L. Steele Jr.编。ACM出版社，1996年，第372–385页。doi：10.1145/237721.237805.[14] Cédric Fournet、Georges Gonthier、Jean-Jacques Lévy、Luc Maranget和Didier Rémy。《移动代理演算》。载于：*CONCUR'96*：并发理论：第七届国际会议，比萨，意大利，*1996*年*8*月*26*日至*29*日，论文集*7*。Springer。1996年，第406–421页。doi：10.1007/3-540-61604-7_67.[15] Rosita Gerbo和Luca Padovani。《Java中的并发类型状态导向编程》。载于：编程语言方法：面向并发与通信中心软件，*PLACES@ETAPS 2019*论文集，布拉格，捷克共和国，*2019*年*4*月*7*日。Francisco Martins和Dominic Orchard编。第291卷。EPTCS。2019年，第24–34页。doi：10.4204/EPTCS.291.3.[16] Philipp Haller和Tom Van Cutsem。《使用可扩展模式匹配实现Join》。载于：协调模型与语言：第十届国际会议*COORDINATION 2008*，挪威奥斯陆，*2008*年*6*月*4*日至*6*日，论文集。Doug Lea和Gianluigi Zavattaro编。第5052卷。《计算机科学讲义》。Springer，2008年，第135–152页。doi：10.1007/978-3-540-68265-3_9.[17] Philipp Haller、Ayman Hussein、Hernán Melgratti、Alceste Scalas和Emilio Tuosto。《Actor的公平Join模式匹配》。载于：第*38*届欧洲面向对象编程会议（*ECOOP 2024*）。Jonathan Aldrich和Guido Salvaneschi编。第313卷。《莱布尼茨国际信息学会议录》（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2024年，17:1–17:28。ISBN：978-3-95977-341-6。doi：10.4230/LIPIcs.ECOOP.2024.17.[18] Carl Hewitt、Peter Bishop和Richard Steiger。《人工智能的通用模块化ACTOR形式体系》。载于：第三届国际人工智能联合会议论文集。IJCAI'73。美国斯坦福：Morgan Kaufmann出版社，1973年，第235–245页。网址：https://www.ijcai.org/Proceedings/73/Papers/027B.pdf （访问于2026年2月15日）。[19] Ayman Hussein、Ioannis Karras、Philipp Haller、Hernán Melgratti、Alceste Scalas和Emilio Tuosto。*JoinActors*：带连接模式的模块化 *Actor* 库（制品）。2026年。doi：10.5281/zenodo.18411860.[20] Shams M. Imam和Vivek Sarkar。《Savina——Actor基准测试套件：支持对Actor库进行实证评估》。载于：第四届基于*Actor*、代理与去中心化控制的编程国际研讨会论文集。AGERE! '14。美国俄勒冈州波特兰：计算机协会，2014年，第67–80页。ISBN：978-1-4503-2189-1。doi：10.1145/2687357.2687368。

[21] G. Stewart Von Itzstein和Mark Jasiunas。《在Java中实现高级并发》。载于：计算机系统架构进展：第八届亚洲太平洋计算机系统架构会议*ACSAC 2003*，日本会津若松，*2003*年*9*月*23*日至*26*日，论文集。Amos Omondi和Stanislav Sedukhin编。第2823卷。《计算机科学讲义》。Springer，2003年，第151–165页。doi：10.1007/978-3-540-39864-6_13.[22] Ioannis Karras。《连接模式匹配的扩展与优化》。2025年。doi：10.48550/arXiv.2512.04876.[23] Qin Ma和Luc Maranget。《编译连接模式中的模式匹配》。载于：并发理论国际会议。Springer。2004年，第417–431页。doi：10.1007/978-3-540-28644-8_27.[24] Martin Odersky。《函数式网络》。载于：欧洲编程研讨会。Springer。2000年，第1–25页。doi：10.1007/3-540-46425-5_1.[25] Luca Padovani。《无死锁的类型状态导向编程》。载于：艺术、科学与工程编程 2.3（2018年）。doi：10.22152/programming-journal.org/2018/2/15.[26] Hubert Plociniczak和Susan Eisenbach。《JErlang：支持连接的Erlang》。载于：协调模型与语言：第十二届国际会议*COORDINATION 2010*，荷兰阿姆斯特丹，*2010*年*6*月*7*日至*9*日，论文集*12*。Springer。2010年，第61–75页。doi：10.1007/978-3-642-13414-2_5.[27] Dan Plyukhin、Gul Agha和Fabrizio Montesi。《CRGC：Pekko中的容错恢复型参与者垃圾回收》。载于：*ACM*编程语言会议录 9.PLDI（2025年），第945–969页。doi：10.1145/3729288.[28] Mark Proctor。《Drools：用于复杂事件处理的规则引擎》。载于：具有工业相关性的图变换应用——第四届*AGTIVE 2011*国际研讨会，匈牙利布达佩斯，*2011*年*10*月*4*日至*7*日，修订选编及特邀论文。Andy Schürr、Dániel Varró和Gergely Varró编。第7233卷。《计算机科学讲义》。Springer，2011年，第2页。doi：10.1007/978-3-642-34176-2_2.[29] Humberto Rodríguez-Avila、Joeri De Koster和Wolfgang De Meuter。《基于复杂事件处理技术、面向Actor 模型的高级连接模式》。载于：艺术、科学与工程编程 5.2（2021年）。doi：10.22152/programming-journal.org/2021/5/10.[30] Claudio V. Russo。《Visual Basic连接模式》。载于：第*23*届*ACM SIGPLAN*面向对象编程、系统、语言与应用会议（*OOPSLA 2008*）论文集，*2008*年*10*月*19*日至*23*日，美国田纳西州纳什维尔。Gail E. Harris编。ACM，2008年，第53–72页。doi：10.1145/1449764.1449770.[31] Claudio V. Russo。《Joins并发库》。载于：声明式语言的实际应用：第九届国际研讨会*PADL 2007*，法国尼斯，*2007*年*1*月*14*日至*15*日。Michael Hanus编。第4354卷。《计算机科学讲义》。Springer，2007年，第260–274页。doi：10.1007/978-3-540-69611-7_17。

[32] *Scala 3Quotes*。https://docs.scala-lang.org/scala3/guides/macros/quotes.html。（访问日期：2025-05-26。）[33] *Scala 3 Reflection*。https://docs.scala-lang.org/scala3/guides/macros/reflection.html。（访问日期：2025-05-26。）[34] Nicolas Stucki、Jonathan Immanuel Brachthäuser 和 Martin Odersky。《使用生成式与分析式宏进行多阶段编程》。载于：第*20*届 *ACM SIGPLAN* 生成式编程：概念与实践国际会议论文集。GPCE 2021。美国伊利诺伊州芝加哥：计算机协会，2021年，第110–122页。ISBN：978-1-4503-9112-2。DOI：10.1145/3486609.3487203。[35] Martin Sulzmann、Edmund S. L. Lam 和 Peter Van Weert。《具有多头消息接收模式的 Actor》。载于：协调模型与语言，第*10*届国际会议 *COORDINATION 2008*，挪威奥斯陆，*2008*年*6*月*4*日至*6*日，论文集。Doug Lea 和 Gianluigi Zavattaro 编。第5052卷。计算机科学讲义。Springer，2008年，第315–330页。DOI：10.1007/978-3-540-68265-3_20。[36] Aaron Joseph Turon 和 Claudio V. Russo。《可扩展Join模式》。载于：第*26*届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用年会论文集，*OOPSLA 2011*（*SPLASH 2011* 的一部分），美国俄勒冈州波特兰，*2011*年*10*月*22*日至*27*日。Cristina Videira Lopes 和 Kathleen Fisher 编。ACM，2011年，第575–594页。DOI：10.1145/2048066.2048111。

关于作者

Ayman Hussein （ayhu@dtu.dk）是丹麦技术大学 DTU Compute 的博士生。https://orcid.org/0009-0005-6173-0976

Philipp Haller (phaller@kth.se）是瑞典斯德哥尔摩瑞典皇家理工学院的副教授。https://orcid.org/0000-0002-2659-5271

Ioannis Karras (yaniskarras@gmail.com)是丹麦技术大学（DTU Compute）的计算机科学与工程理学硕士。

https://orcid.org/0009-0006-6920-111X

Hernán Melgratti(hmelgra@dc.uba.ar)是布宜诺斯艾利斯大学的兼职教授，也是计算机科学研究所（ICC）的研究员。该研究所隶属于 CONICET 和布宜诺斯艾利斯大学（UBA）。

https://orcid.org/0000-0003-0760-0618

Alceste Scalas （alcsc@dtu.dk）是丹麦技术大学 DTU Compute 的副教授。
https://orcid.org/0000-0002-1153-6164

Emilio Tuosto （emilio.tuosto@gssi.it）是格兰萨索科学研究所（GSSI，意大利拉奎拉）的计算机科学正教授。

https://orcid.org/0000-0002-7032-3281