Adaptable TeaStore：一种编排式方法*

GiuseppeDePalma Saverio Giallorenzo Ivan Lanese Gianluigi Zavattaro博洛尼亚大学和法国国家信息与自动化研究所，意大利博洛尼亚
{giuseppe.depalma2,saverio.giallorenzo,ivan.lanese,gianluigi.zavattaro}@unibo.it

Adaptable TeaStore 最近被提出作为可适应微服务架构的参考模型。它包含不同的配置，以及要求在这些配置之间进行转换的场景。我们介绍了一个基于 AIOCJ 实现的 Adaptable TeaStore。AIOCJ 是一种编舞语言，可用于编程能够在运行时适应不同条件的多方系统。遵循编舞传统，AIOCJ 通过构造确保通信在适应前、适应期间和适应后均正确（例如，不会发生死锁）。适应是动态的，并且只需在运行时完整指定适应场景。我们使用 AIOCJ 对 Adaptable TeaStore 进行建模，以展示这一方法的优势及其当前局限，并就未来如何改进这一范式（尤其是 AIOCJ 语言）提出建议，使其更好地契合现实世界中的云架构。

1 引言

Adaptable TeaStore 最近被提出， [2] 作为可适应微服务架构的参考模型。它扩展了面向静态微服务架构的 TeaStore 参考模型 [2] ，支持多种配置以及会触发配置间转换的众多适应场景。

在本文中，我们使用 AIOCJ（Jolie 中面向交互的可适应编舞） [7, 6, 9], 这一可执行语言，将 Adaptable TeaStore 规范建模为可适应编舞；该语言支持对可适应多方系统进行编程。

遵循 AIOCJ 中编舞规范与编程的传统， [15, 14, 17], 单个构件即可定义一个多方系统，并且可以通过投影操作，从该构件推导出参与该系统的每个组件的代码。更确切地说，AIOCJ 背后的理念是将系统构造为由参与者 以及可能存在的外部服务组成的组合。参与者可以执行复杂的通信模式，并且会受到适应的影响。此外，参与者可以调用外部服务，而外部服务能够执行编排之外的计算（例如访问数据库）。对这些服务的调用必须终止，并将一个值返回给调用方（即参与者）。外部服务并非在 AIOCJ 中定义；只要支持通过 AIOCJ 所支持的通信协议之一进行调用，它们就可以使用任何语言编写，例如通过套接字传输 SOAP。在适应过程中，参与者将一致地执行实现适应后行为的新代码（即“原始编排中不存在的”代码），并可能重新配置它们之间以及它们可访问的服务之间的连接，同时改变它们使用这些连接和服务的方式。

编排式方法从构造上确保了通信的相关正确性属性，例如无死锁。具体而言，对于 AIOCJ，这些属性在适应之前、期间以及

*本研究部分获得以下项目支持：法国国家研究署 SmartCloud 项目 ANR-23-CE25-0012；由 PRIN（意大利 MUR）和下一代欧盟框架资助的 PRIN 项目 FREEDA（CUP：I53D23003550006）；通过 MUR 获得下一代欧盟欧洲计划资助的 PNRR CN HPC - 第9工作组 - LEONARDO 创新资助 - TASI - RTMER 项目；以及 INdAM - GNCS 2024 项目 MARVEL （代码 CUP E53C23001670001 ）。

G. De Palma 和 S. Giallorenzo（编）：第一届可适应云架构研讨会，EPTCS 438，2025年，第 79–99 页，
doi:10.4204/EPTCS.438.5

© G. De Palma 等。本文采用知识共享署名许可协议授权。

之后都成立。如前所述，AIOCJ 程序的一个显著特点是，它们能够适应编写编排时未曾预料到的行为。具体来说，编写可适应的编排时，需要指定代码中哪些部分将来可能发生变化。替换这些可适应部分的新代码，以及触发适应的条件定义，都可以在之后添加——尤其是在原系统运行期间。适应通过指定适应规则来实现，这些规则可根据系统及其执行上下文/环境的状态应用。单条适应规则可以协同地更改多个参与者的代码。

由于我们展示的是 AIOCJ 在 Adaptable TeaStore 上的应用，且重点在于对案例研究架构的建模（而非其计算部分等），因此我们对所需服务采用最简实现，并将其视为（在 AIOCJ 中的）外部服务。[1] 为此，我们遵循一种编程风格： [10] AIOCJ 参与者是可适应连接器，负责协调外部服务，并辅以适应规则，使这种协调能够根据不断变化的需求进行调整。

2 可适应茶店：概述

有关完整细节，请参阅官方可适应茶店规范 [2] 。本节将简要概述可适应茶店架构，介绍其中涉及的组件，以及我们为 AIOCJ 实现选择的部署模式。图 1 展示了规范 [2] 中可适应茶店架构的简化示意图。

Adaptable TeaStore 的架构由 5 个主要服务组成：WebUI、身份验证、持久化、图像提供者和推荐器。WebUI 服务是用户的入口，并负责协调所有其他服务。持久化服务作为数据库之上的一层。我们将其部署在与本地提供者不同的提供者上，并通过本地缓存使其可访问。当它不可用时，则由本地静态数据库（功能有限）替代。WebUI 使用持久化服务来检索和存储数据，而身份验证服务则使用该服务检索用户数据，再将数据传递给 WebUI。

图像提供者和推荐器都连接到持久化服务。不过，它们只在启动时需要此连接（虚线）。由于这种启动同步与本文重点讨论的适应场景无关，因此我们在实现中省略了这一部分。图像提供者必须为每种产品生成一张图像，并且与持久化服务一样在远程运行，通过本地缓存连接。必要时，可以由本地静态图像替代图像提供者。身份验证服务提供身份验证功能，并可依赖外部用户单点登录服务，通过 Google 或 Facebook 等方式进行身份验证。与启动同步流程类似，为了聚焦于应用程序的主要行为，我们在实现中省略了对单点登录交互的建模。

推荐器会向用户推荐可能感兴趣的产品。推荐器服务并非基本服务，可以完全禁用。全功率版本的推荐器使用机器学习技术，根据特定的用户偏好进行推荐；而低功耗版本则可以替换它，该版本基于通用的物品流行度进行推荐。同样，图像提供者和身份验证服务也可以采用不同的模式提供。

[1]原则上，虽然可以在 AIOCJ 中实现整个架构，但为了实现关注点分离，我们采用这样一种风格：在 AIOCJ 中实现组件之间的协调，并使用服务实现组件的业务逻辑。

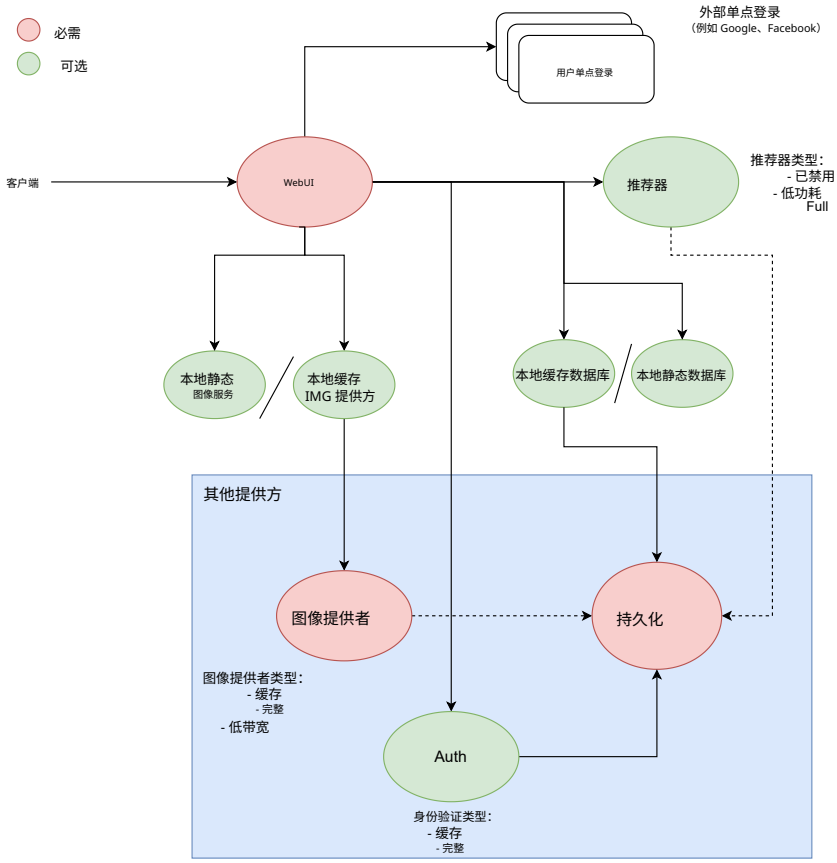


图 1：可适配 TeaStore 架构图。

3 AIOCJ 中的“Barebone” TeaStore

我们首先介绍最基础版本的（可适配）TeaStore 的 AIOCJ 实现，并将其称为“Barebone” TeaStore。Barebone 版本仅包含为用户提供最低限度功能的基本服务和交互，即由 WebUI、本地静态图像和本地静态持久化服务提供的功能。本节先介绍一个不可适配版本的 Barebone TeaStore，随后几节将介绍并完善该实现，加入必要的 AIOCJ 构造，使程序具备适配能力，并能够涵盖大部分 Adaptable TeaStore 架构规范。

我们在清单 1 中展示了 Barebone TeaStore 的 AIOCJ 代码，重点关注第 7–25 行中有趣的部分，该部分位于`aioc` 作用域内，由花括号界定。其中，我们可以看到一段编排，描述了四个参与者的全局行为：用户（标记为 `U`，充当客户端）、WebUI `W`、持久化服务 `P`和图像服务 `I`。该编排通过 `getInput` 功能来表示用户交互；该功能会向用户显示提示，以便用户输入数据。具体而言，编排中的第一个操作（第 8 行）涉及用户提供地址（在此原型中，用户直接输入地址值；而在实际应用程序中，

1 从 " socket :// localhost :8001 " 引入 getPageInfo，并使用 " soap " 2 从 " socket :// localhost :8002 " 引入 getPageImg，并使用 " soap " 3 从 " socket :// localhost :8000 " 引入 compilePage，并使用 " soap " 4 5 前言 { starter : W } 6 7 aioc { 8 用户地址 @U =getInput (" 输入地址 "); 9 getPage : U(address) -> W(address); 10 { 11 { 12 getPageInfo : W(address) -> P(address); 13 info @P =getPageInfo (pid); 14 getInfo : P(info) -> W(info) 15 }16 | 17 { 18 getPageImg : W(address) -> I(address); 19 img@I =getPageImg (address); 20 getImg : I(img) -> W(img) 21 } 22 }; 23 page @W = compilePage (info , img); 24 getPage : W(page) -> U(page) 25 } 清单 1：Barebone TeaStore 版本的 AIOCJ 编排。

用户希望访问的 TeaStore 页面（其 Web 界面将负责提供这类数据）；用户输入的数据存储在其局部变量 address 中。由于 AIOCJ 程序描述位于不同参与者处的数据，因此使用 @符号表示法标明给定变量归属哪个参与者——就 address 而言，它是 address@U。接下来，我们看到编排中参与者之间的第一次交互：用户将地址发送给 WebUI。按照编舞模型的惯例， [15], 所有 AIOCJ 交互都有标签——此处的标签为 getPage。记法 U(address) -> W(address) 表示 U 将其局部变量 address 中的值发送给 W，而 W 将该值存储在同名的局部变量中。请注意第一条和第二条指令之间的分号 ;。这个符号表示顺序组合运算符，规定左侧指令必须先于右侧指令执行。AIOCJ 程序描述的是一个分布式系统。因此，为确保顺序组合中的指令严格按序执行，AIOCJ 编译器会确保按顺序组合的两条指令至少有一个共同参与者——例如，U 是前两条指令的共同参与者——从而保证编排中全局动作之间因果关系的忠实实现。编译为相互独立的可执行文件后，这种编排会变为分布式系统，且这些文件仅通过消息传递进行通信。然后，我们在第 10 行和第 22 行打开一个作用域，以此实现分叉-汇合模式，让 WebUI 与持久化服务和图像服务之间的交互并行执行。我们通过两个内部作用域实现该模式的分叉部分（分别位于第 11–15 行和第 17–21 行），并使用并行组合运算符 | 将它们组合起来。这两个作用域并行执行，终止后会在第 22 行外层作用域闭合处汇合，继续执行更大的顺序组合。

```
1 作用域 @W{2 页面@W =compilePage（info,
img）3 }属性 { N. 标签 ="page-compiler"};
```

Lis图 2：适配版 Barebone TeaStore 的 AIOCJ 编排摘录 .

在内部作用域中，webUI 分别与Persistence 和Images 服务交互，以获取页面的文本内容，并将其保存在 `info` 变量中，该变量属于webUI；页面图像则保存在 `img` 变量中，该变量属于 webUI。在每个作用域中，我们分别可以找到与 TeaStore 服务可用 API（或其包装器）交互的 Persistence 和Images 参与者，它们分别通过 `getPageInfo`和`getPageImage`获取页面内容（服务 API 的名称与交互的同名标签是不同的概念）。在 AIOCJ 中，开发者可以通过 `include` 指令引入 API 的可用性。该指令位于 AIOCJ 程序开头，例如清单 1 第 1–3 行，其中可以指定服务的地址和通信媒介（例如，`"socket://..."`）以及数据格式（例如，`"soap"`）。

会合之后，WebUI 继续调用 compilePage API 获取页面，并将其发送给用户。

结束这个示例时，我们注意到前言子句出现在第 5 行，开发者可用它指定配置参数，例如参与者的地址。在这个最简示例中，我们让 AIOCJ 编译器分配本地地址（用于本地执行示例；不过 AIOCJ 支持一般的分布式执行），并且只指出在编排的参与者中，谁是发起者——实现会合过程需要这一信息；在此过程中，作为发起者的参与者（本例中是webUI）会等待所有其他参与者与其联系，然后通知它们可以开始编排。

适应结构化本节最后，我们将展示如何在 AIOCJ 程序中引入结构化的分布式适应。具体而言，我们考虑这样一种情况：未来可能会有其他用于编译页面的 API 可用。

得益于作用域的[2], AIOCJ 允许开发者在运行时调整编排中参与者的行为，方法是集成甚至在它们开始执行后才编写的代码。

为了说明这一特性，我们需要通过将清单 1 中的第 23 行替换为清单 2 中的代码，来修改编排。一般来说，适应作用域 包含两个元素：一个控制器，用@符号标注在作用域 旁边——在清单 2 中，控制器是webUI——以及作用域的主体，它包含在花括号中，用于界定编排中的一段内容；该内容可在运行时通过应用 AIOCJ 规则而发生变化——下一节将对此进行讨论。

在运行时，控制器是负责选择适应规则适用于作用域的参与者。虽然这个示例很简单，也就是说，作用域中的唯一参与者也是控制器，但一个作用域可能涉及多个参与者，控制器会协调这些参与者，确保它们都遵循选定的适应规则中规定的同一段编排。当协调器到达

[2]AIOCJ 支持两种类型的作用域：使用`{ ... }` 语法的通用作用域，以及使用`scope { ... }` 语法的适应作用域。本文中，我们通过语法高亮来区分示例中的这两种形式。提及可适配作用域时，关键字`scope` 会以高亮形式显示；提及通用作用域时，则在普通文本中使用“作用域”一词。

一个作用域的起点时，会向 AIOCJ 的运行时查询要应用的适应规则。AIOCJ 的运行时包含多个规则仓库，每个仓库包含一条或多条适应规则，并且这些仓库可能随时连接或断开。运行时依次查询可用的规则仓库，而各仓库则会检查其中每条规则的适用条件。运行时会应用首条适用条件成立的规则（如果有），并将每个参与者的代码发送给协调器，再由协调器分发给相关角色。每个角色中的新代码都会取代原有代码。如果没有规则适用，协调器就会通知其他角色执行原有代码。该协议确保了适配的一致性，这是因为只有一个确定适应过程如何展开的“真实”来源（即协调器）。事实上，由于规则可能在运行时发生变化（可能出现或消失），只有协调器负责查看哪些规则可用，便解决了关于规则可用性以及相关适应代码分发的固有不一致问题（角色可能会在不同时间和地点看到不同的规则集合）。

开发者可以将 属性关联到 作用域，使适应规则的应用更加精确。例如，在列表 2 中，我们添加了属性 标签，并以 N[3][4], 为前缀，用于标记该作用域承载着 「页面编译器」的代码。作用域属性旨在描述作用域的当前实现，包括功能属性和非功能属性。此类属性由程序员声明，系统仅使用它们来评估适应规则的适用条件，从而判断给定的适应规则是否可以应用于给定的作用域。

我们进一步通过图 2 展示 Barebone TeaStore 编排的（本地）执行截图，使描述更加具体。

图中展示了一个多终端设置，说明了使用 AIOCJ 构建的服务如何执行。具体而言，我们展示了用户输入地址后 Barebone TeaStore 的行为。

在图 2 中，会弹出一个消息窗口，显示用户交互的结果：一款名为“伯爵茶”的茶产品、其描述，以及一张经过 Base64 编码的关联图像。此输出来自编排中对持久化服务和图像提供者服务的调用。

每个终端窗口都会运行一个独立的 Jolie [18] 服务（通过“jolie”命令启动，该命令会执行 Jolie 程序的解释器）。我们可以看到两类服务：一类是模拟或封装现有 Adaptable TeaStore 服务行为的服务（例如持久化服务，即最上方的窗口）；另一类是由 AIOCJ 编译器自动生成的服务。后一类既包括实现编舞参与者逻辑的服务（图 2 中未显示），也包括构成 AIOCJ 应用程序（分布式）运行时环境的服务（例如，允许管理员指定应用程序执行上下文参数的环境服务，图 2 中也未显示）。

输入地址后，参与者彼此之间以及与外部服务（通过其 API）交互，以获取所请求页面的内容、进行编译，并将结果显示给用户。

使用作用域实现适应 总而言之，AIOCJ 提供了 作用域，用于指定应用程序中哪些部分可以适应。因此，决定将代码的哪些部分括在 作用域中，是一个重要且并非简单的决定。直观而言，应将 作用域括住代码中

[3]最初，这个前缀代表“非功能”，意指 作用域级属性应当反映非功能属性。然而在实践中，适应决策往往取决于交织在一起的功能关注点和非功能关注点。[4]为 作用域级属性/变量添加前缀并非必要，因为它们在语法和上下文上都与其他变量（例如 作用域控制器中的变量）有所区别。相反，在规则中，前缀至关重要，因为规则也可以在适用条件中考虑其他类型的属性。为保持对称，AIOCJ 也要求在作用域中使用该前缀。

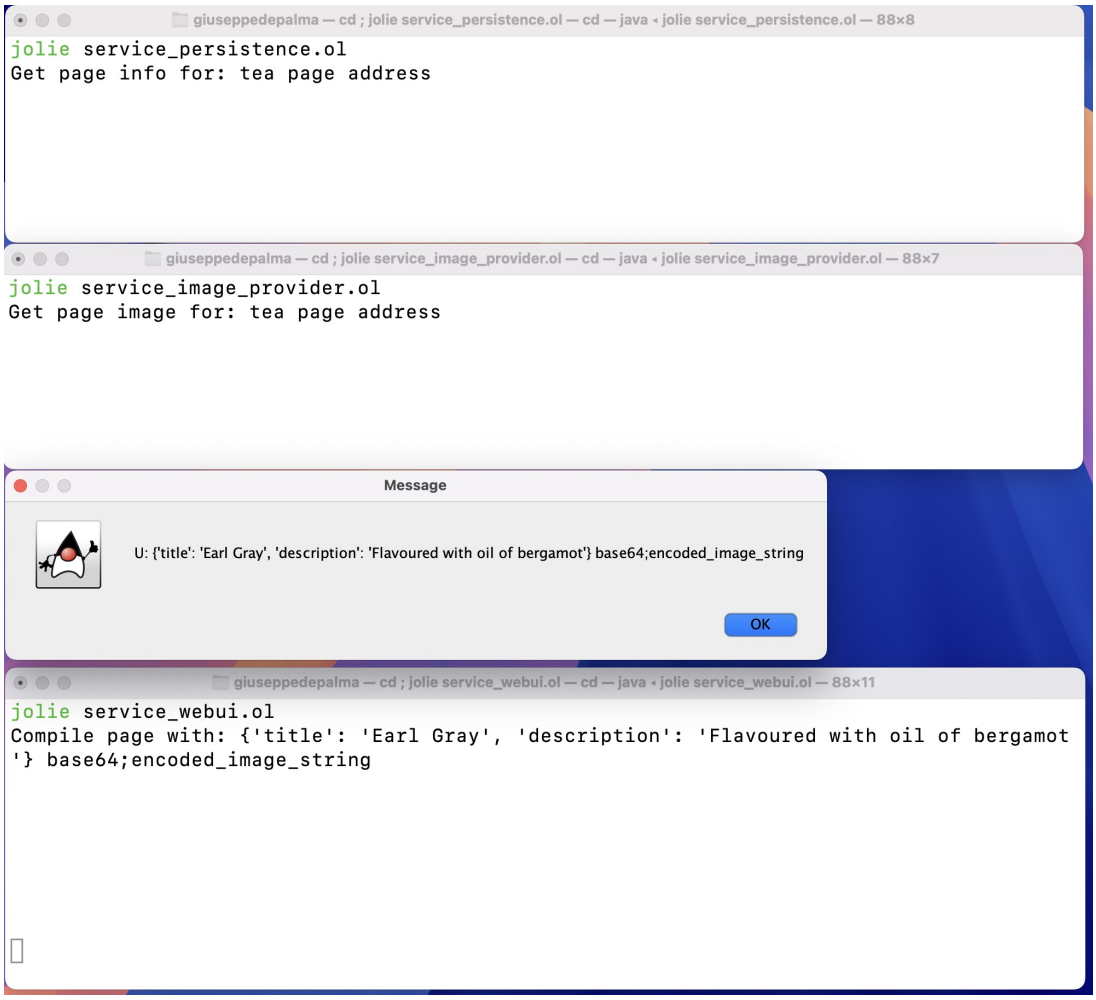


图 2：Barebone TeaStore 编排的执行。

未来可能需要适应的部分。这些部分包括例如可能需要根据业务需求变化而调整的业务规则建模代码，也包括依赖于位置的部分，以及出于性能或安全原因而需要关注的部分。

当然，可以通过采用一个作用域包围整个编排，或者采用多个覆盖所有指令的作用域，来解开这个戈尔迪之结。然而，这两种方案都有明显的缺点。一个包罗万象的作用域会取代整个编排，这意味着必须将适应视为整体变更，并整合所有相关的适应方面（例如，合并彼此正交的功能方面与非功能方面，如安全和可用性），几乎无法支持模块化。此外，这种适应只能在编排启动时执行。逐指令作用域也很难算得上解决方案，因为一次适应可能涉及多个作用域，而且目前 AIOCJ 并未提供内置方式来组织多作用域适应行为——正如下一节将讨论的，Adaptable TeaStore 需要这种协调方式，而我们通过簿记隐式实现了它。

1 { 2 { 3 getPageInfo : W(address) -> P(address); 4 info @P =
getPageInfo (pid); 5 getInfo : P(info) -> W(info) 6 }7 |8 { 9 get
PageImg : W(address) -> I(address); 10 img @I =getPageImg (address);
11 getImg : I(img) -> W(img) 12 }13 |14 作用域 @W { skip } prop { N.
tag ="推荐器" } roles { P, U } 15 }; 清单 3：Adaptable TeaStore 基础版本，仅含推荐器

4 小节 借助可适配推荐器实现 TeaStore 编排

如前所述，Adaptable TeaStore 支持可选的推荐器，该推荐器有两种模式：低功耗模式基于物品流行度，全功率模式则利用机器学习和用户偏好。

要在 AIOCJ 中支持此类适应（如同任何其他适应），我们需要两个要素：一个作用域，用于在代码中指定适应应发生的位置；以及一个或多个适应规则，用于指定发生适应时应使用哪些新代码。值得注意的是，在原始编排的执行期间，适应规则可以在运行时更改（添加或移除）。

在这个特定案例中，我们使用两个作用域。第一个在前一节中介绍，与页面编译有关；它允许利用推荐器中的信息为用户生成页面，其中包含推荐内容。另一个见清单 3 第 14 行，用于插入对推荐器的实际调用——指令`skip` 表示不执行任何操作。请注意，这个作用域引入了一个额外的关键字 `roles`，它指出在适配后的代码中，可能参与其中但未出现在（即未执行任何操作）作用域主体中的角色。具体而言，将`User` 和`Persistence` 添加到作用域 `roles`后，适应规则除了可以指定涉及`Web UI`（它出现在作用域主体中）的交互外，还可以指定涉及这些角色的交互。

还要注意，我们将这个新的作用域并行地（|）放置在页面信息和页面图像的计算过程中，如上一节所述。另一种做法是仅使用上一节所述的用于页面编译的作用域，并添加一条引入推荐器的规则。虽然这种做法在概念上更简单，但效率较低，因为只有在页面信息和图像信息的计算完成后，推荐计算才会开始。

现在，我们从第一条开始介绍这两条适应规则，它们分别列于列表 4 和列表 5 中。

适应规则的主要构建块由 `do` 子句标识。该子句指定执行适应时需要运行的代码，以替换待适应的 作用域 中的代码。不过，规则还包含其他元素。首先，它们可以包含额外的外部服务（例如 `getTopItems`）和额外的参与者，后者由关键字 `newRoles`引入。最后，关键字 `on` 用于引入规则的适用条件：当某个 作用域 在执行过程中得到满足时

```
1 规则 { 2 引入 getTopItems，来自 " socket :// localhost :8001 "，使用 " soap " 3 引入
processRecommendations，来自 " socket :// localhost :8003 "，使用 " soap " 4 5 newRoles : R 6 当 {
N. tag  ==" 推荐器 " and E. 推荐器  =="低 - 功耗 " } 7 执行 { 8 getPopularProducts : R () ->P (); 9
items @P  =getTopItems ( 10 , " popularity " ); 10 popularProducts : P( items ) ->  R( items ); 11
recommendations @R  =processRecommendations ( items ); 12 recommendedProducts : R(
recommendations ) ->  W( recommendations ); 13 recommender @W  =  true 14 } 15 }
```

列表 4：推荐器服务低功耗模式的规则。

执行时，所有可用规则都会按任意顺序检查是否适用。第一个条件计算结果为 true 的规则将被应用。条件可以引用作用域所描述的属性，前缀为 N，例如 N.tag == "recommender"；也可以引用环境所描述的属性，前缀为 E，例如 E.recommender == "low-power"。前者旨在确保规则应用于“正确的”作用域（即与所提及的属性或待添加功能相关的作用域），而后者允许规则的适用性取决于环境条件，例如，当环境中的可用电量有限时，列表 4 中的规则便会适用。

列表 4 中的规则包含两个外部服务，分别提供 getTopItems 和 processRecommendations 功能，位置分别为 "socket://localhost:8001" 和 "socket://localhost:8003"，二者均使用 SOAP 协议。此外，该规则将新角色 R（作为推荐器服务的协调器）引入编排中。当两个条件同时满足时，该规则便会激活：作用域的标签等于 "recommender"，且环境变量 recommender 被设为 "low-power"。触发后，该规则会实现低功耗推荐流程：推荐器向持久化服务请求热门产品，随后持久化服务调用 getTopItems 函数以获取最热门的产品。持久化服务将这些热门产品发送回推荐器，推荐器使用 processRecommendations 函数对其进行处理。随后，推荐器将处理后的推荐发送给 WebUI 服务，该服务会将其推荐器标志设为 true，以表明这些推荐已可用。

虽然前一条规则允许在低功耗模式下计算推荐信息，但还需要第二条规则，以便在页面编译时使用这些信息。该规则见列表 5。值得注意的是，该规则展示了适用条件中的第三种可选方式：使用负责管理适应的角色的局部变量，例如 推荐器 == `true`，其中 推荐器 是角色 `W` (WebUI) 的一个变量。此处，推荐器 用作记账变量，以确保只有在列表 4 中的一条规则已被应用时才执行适应。这种机制可以让不同的适应规则保持同步，不过方式较为临时。

如上一节所述，我们通过图 3 所示的屏幕截图来展示可适应编排的执行过程。该截图捕捉了应用适应规则、采用低功耗版本的推荐器之后的一个执行步骤。具体而言，我们注意到，所得

```
1 规则 { 2 从 " 套接字 :// localhost :8000 " 以 " SOAP " 引入 compilePageWithRecommends 3 4 当 { N.
tag   =="页面 - 编译器 " 和推荐器   ==true } 5 执行 { 6 页面 @W   =compilePageWithRecommends ( 信息
, 图像 , 推荐 ); 7 推荐器 @W   =   false 8 } 9 }
```

列表 5：编译带有推荐内容的页面所用的规则。

发给用户的消息与图 2 中的类似，不过图 3 中的消息还整合了推荐器提供的建议。

关于适应规则的应用 在执行期间，当运行时进入一个 作用域时，它会检查所有可用规则是否适用（顺序不限），并应用其中第一个条件评估结果为真的规则。具体而言，运行时会依次检查规则，根据系统当前状态逐一判断规则是否适用。由于环境属性可能在规则评估的同时发生变化，因此可能出现竞态条件：某条规则的适用条件在检查时为否，但随后变为是，导致运行时不应用任何规则，即使所有可能状态均已被完整覆盖也是如此[5]。

由于分布式架构具有并发性，这个问题并不容易解决。在检查规则时试图固定环境状态（例如，通过创建快照或锁定环境变量）会适得其反，因为这将无法准确反映系统的实际状态，还可能导致基于不同步状态应用规则。此外，运行时应用规则时系统状态发生变化（可能引发错误），与其他分布式系统故障模式并无本质区别。例如，某个编排可能成功完成适应，但随后在执行过程中外部资源发生故障，从而导致类似的错误情况。我们认为，与其试图消除分布式系统中固有的时间不一致，不如通过错误处理和恢复机制更有效地应对这些竞态条件。难点在于设计能够抵御此类时序问题的适应策略。如需详细了解 AIOCJ 的适配机制及其保证的理论与实现，请参阅其参考介绍 [6]。

5 Barebone TeaStore 编排，采用可适应的身份验证

现在我们来考虑一个更复杂的适应场景，其中加入了身份验证服务。其思路是，如果用户能够完成身份验证／已经通过身份验证，系统便可以根据其偏好为其提供个性化产品、图像和推荐。

[5]考虑适用于数据库连接状态的规则，该状态可以是“主库”、“副本”或“离线”。假设我们进入一个适应作用域，当前状态为“主库”，运行时开始检查一条要求状态为“副本”的规则是否适用。该条件为否定条件，运行时继续检查下一条规则。与此同时，状态变为“副本”，从而阻止另外两条规则适用。结果是没有任何规则适用，尽管系统已覆盖所有数据库状态。

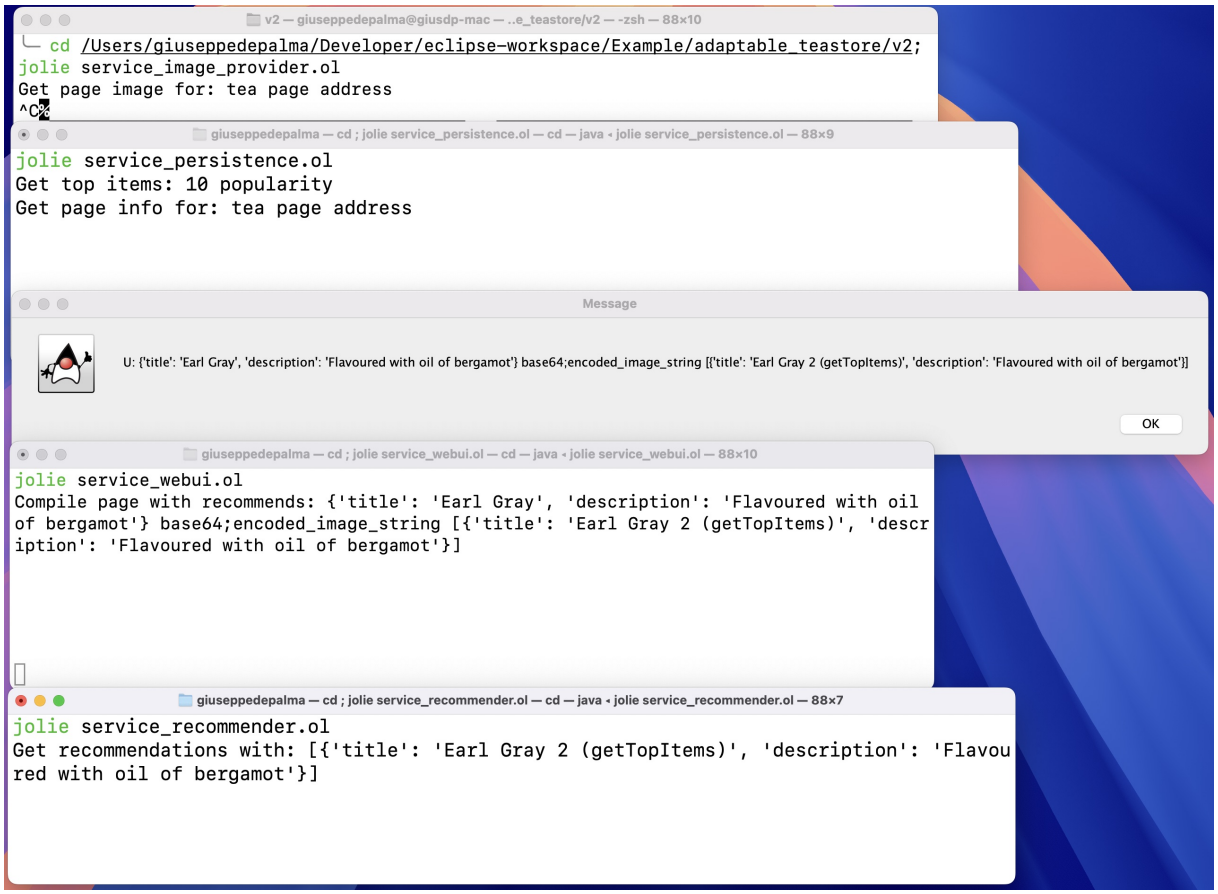


图 3：使用低功耗推荐器版本执行 Barebone Adaptable TeaStore。

不过，我们面临一个棘手的问题：身份验证会在多个环节影响编排，包括可能在运行时添加推荐器等服务的情况。为了协调各方并保持行为一致，我们使用一个由 WebUI 管理的变量令牌，以跟踪是否已完成身份验证（参见清单 6 第 4 行）。我们还添加了多个作用域以支持适应。

提供身份验证的主要规则如清单 7 所述。其适用条件规定，该规则适用于作用域 “auth”，并要求身份验证支持可用（通过检查`E.auth == “available”`）。请注意，只有尚未获取令牌时，该规则才会生效。这些条件体现了这样一个事实：理想情况下，身份验证应只执行一次，并提供一个令牌，供所有需要它的功能使用。

清单 8 展示了用于改进页面信息编译的规则。实际上，如果用户此前已完成身份验证且系统持有其令牌，或者更新了规则第 7 行中的作用域，从而“即时”获取令牌，就会使用需要身份验证的服务 getPageInfoAsLoggedUser。

清单 8 中的规则表明，我们可以在规则内部使用作用域来实现嵌套适配。此处，一旦规则应用并且执行到达清单 8 第 7 行的嵌套作用域，系统就会再次检查适应规则是否可用（具体来说，使用`N.tag = "auth"`）。

```
1 aioc { 2 地址 @U =getInput ( " 输入地址 " ); 3 getPage : U( 地址 ) ->
W( 地址 ); 4 令牌 @W =" 无 "; 5 { 6 作用域 @W { 7 getPageInfo : W( 地址 )
-> P( 地址 ); 8 info @P =getPageInfo ( 地址 ); 9 getInfo : P( info ) -> W(
info ) 10 } prop { N. tag ="页面 - 信息 " } 角色 { U }11 |12 作用域
@W { 13 getPageImg : W( 地址 ) -> I( 地址 ); 14 img @I =getPageImg ( 地
址 ); 15 getImg : I( img ) -> W( img ) 16 } prop { N. tag ="页面 - 图像 " } 角
色 { U }17 |18 作用域 @W { skip } prop { N. tag =" 推荐器 " } 角色 {
U, P } 19 }; 20 作用域 @W { 21 page @W =compilePage ( info , img ) 22 }
prop { N. tag ="页面 - 编译器 " }; 23 getPage : W( page ) -> U( page ); 24 }
```

清单 6：Adaptable TeaStore 基础版本

请注意，清单 8 中的规则独立于身份验证功能的可用性而应用（例如，不存在条件`E.auth == "available"`）。这在身份验证功能可能在计算过程中出现和消失的场景中很有意义，例如，用户在前一阶段完成了身份验证，但身份验证功能随后不再可用，此时我们仍可使用令牌处理用户的请求。

清单 9 为推荐器的全功率版本提供了适应。该规则的逻辑与清单 8 中的规则类似，后者通过嵌套适配在需要时提供身份验证。

请注意，作用域中能够提供适应的部分在清单 6 中是并行的。因此，根据调度方式，可能会执行多次身份验证。尤其是，当授权规则的适用条件全部经过检查，而变量 token 尚未获得 `"none"` 值时，就会出现这种情况。一方面，除了用户需要多次进行身份验证这一负担外，不会造成其他问题。另一方面，若要避免这种情况，就需要将不同步骤顺序化，从而降低效率。

本节最后，我们在图 4 中展示了使用身份验证服务和完整版推荐器完成完整适应后，Adaptable TeaStore 编排的执行截图。从图中可以看到，身份验证服务允许用户登录（可在身份验证服务控制台中看到；由于清单 6 中作用域的并行执行，登录发生了两次）。我们还可以从弹出窗口中的页面编译结果看到，完整版推荐器已执行；其中显示的推荐内容包含 ‘description’: ‘Flavoured with ...’。

```
1 规则 { 2 从 " socket :// localhost :8004 " 引入登录，并使用 " soap " 3 newRoles : A 4 在 { N.tag ==" 身份验证 " and E. auth ==" 可用 " and token ==" 无 " } 5 执行 { 6 credentials @U =getInput ( " 输入凭据 " ); 7 sendCredentials : U( credentials ) -> A( credentials ); 8 token @A =登录 ( credentials ); 9 if ( token !=" 无 " )@A { 10 sendToken : A( token ) -> W( token ) 11 } 12 } 13 }
```

清单 7：身份验证服务的规则。

6 讨论与结论

我们认为，这项工作可作为评估诸如 AIOCJ 之类的语言（以及在一定程度上的编舞式编程方法）在捕捉 TeaStore 架构的关注点与可能性方面能力的有用基准，也有助于指出和讨论受编舞范式启发的实现模式与风格。

临时适应与隐式回归 我们基于 AIOCJ 的方法从一个基本系统开始，在预期可能需要适应的位置策略性地插入适应 作用域。随后，针对可用 作用域和系统环境状态应用适应规则，使用相应的编舞片段（规则中提供）对系统进行适应，以替换这些 作用域。这种方法意味着，对于参与者之间的每个交互流程（即基础编舞的执行），系统都会根据可用规则和当前条件进行适应。

这种方法的一个显著优势是，我们无需指定“回归”适应规则，因为系统总是从原始版本开始执行，并可能根据可用规则及规则适用条件进行适应。如果我们考虑用`aioc` 作用域中的行为，在示例中通过 `while(e)@W{ ... }`[6],进行封装，让整个系统循环运行（只要表达式 `e` 在 `W` 处的计算结果为真），并更忠实地实现 Adaptable TeaStore 的行为，这一点就更具实践意义了——用户可以多次与 WebUI 交互（并请求内容）。考虑完整的 Adaptable TeaStore 实现（清单 6），使用 `while` 循环意味着系统在每次迭代时都会从“基本”版本开始，然后根据系统当前状态进行适应。例如，在某次迭代期间，身份验证服务可能不可用，从而阻止应用涉及该服务的部分规则；而在下一次迭代时，该服务可能恢复可用，进而支持应用相关规则。

[6]从语言的角度来看，与 `if` 类似，`while` 需要指定一个控制器，即一个参与者（在上面的示例中为 `W`），由其决定循环的参与者（包括控制器）是执行 `while` 作用域内的代码，还是继续执行下一条指令。

```
1 rule { 2 include getPageInfo , getPageInfoAsLoggedUser from " socket :// localhost :8001 " 3 with " soap " 4 on { N. tag ==" 页面 - 信息 " } 5 do { 6 getPageInfo : W( address ) -> P( address ); 7 scope @W { skip } prop { N. tag =" 身份验证 " } roles { 用户 }; 8 if ( token !=" none " )@W { 9 sendToken : W( token ) -> P( token ); 10 info @P =getPageInfoAsLoggedUser ( address , token ) 11 } else { 12 info @P = getPageInfo ( address ) 13 }; 14 getInfo : P( info ) -> W( info ) 15 } 16 }
```

清单 8：用于持久交互以获取产品信息的规则。

一般而言，我们可以将 AIOCJ 适配的应用描述为“临时的”（即仅绑定到给定的流程迭代，但对于整体系统状态而言是易变的）；与之相应，这也决定了系统隐式地回归到初始状态。

尽管临时应用适配具有很高的灵活性，但我们注意到，这种模式可能会带来潜在的性能开销，因为适配可能发生多次（例如，在每次迭代时，以及在编排执行的不同步骤中），而且如果规则并非以确定性的方式应用，还可能导致用户体验和系统状态出现“波动”。例如，由于身份验证服务可用，用户起初可能成功登录；但随后，由于规则的应用具有非确定性，启用身份验证服务的规则可能不会生效，用户便可能因先前成功登录而遇到“不一致”的行为。话虽如此，在传统系统中，如果身份验证服务发生故障，用户通常会遇到标准错误（例如 HTTP 500），从而导致与上述情况类似的波动。总的来说，我们认为，探索在这些情境中应用临时适配模式的实际影响，是一项有意义的未来工作，有助于改进 AIOCJ 语言（及其背后的方法）。

集成服务控制器在我们的建模中，有一个重要元素被抽象掉了：在 AIOCJ 层，我们不管理资源配置、扩缩容或其他基础设施问题。目前，AIOCJ 尚未提供一种结构化的方式来指定或推理这些问题；我们设想未来将把它们纳入编排中。

在实践中，可以将 AIOCJ 语言与运行时集成到外部控制器中，用于服务的分配与回收，例如 Kubernetes。这些控制器可以响应各种事件，例如流量增长或节点故障，并通过分配服务、迁移服务等方式作出反应。它们管理的服务是通过我们的编排相互连接的 Adaptable TeaStore 组件。因此，需要一种方法将服务控制器与编舞式实现集成起来，从而将服务可用性、负载均衡及其他类似需求纳入编排式方法。

1 rule { 2 include getPageInfo , processQuery , 3 getPageInfoAsLoggedUser from " socket :// localhost :8001 " with " soap " 4 include 获取查询 , getQueryAsLoggedUser , 5 processRecommendations from " socket :// localhost : 8003 " with " soap " 6 newRoles : R 7 on { 8 N. tag ==" 推荐器 " and E. recommender ==" 完整 - power " 9 } 10 do { 11 getPageInfo : W(地址) -> P(地址); 12 scope @W { skip } prop { N. tag =" 身份验证 " } roles { U }; 13 if (token !=" none ")@W { 14 sendToken : W(token) -> P(token); 15 info @P = getPageInfoAsLoggedUser (地址 , token); 16 getInfo : P(info) -> R(info); 17 sendToken : W(token) -> R(token); 18 query @R =以登录用户身份获取查询 (info , token) 19 } else { 20 info @P =getPageInfo (地址); 21 getInfo : P(info) -> R(info); 22 query @R =获取查询 (info) 23 }; 24 sendQuery : R(query) -> P(query); 25 result @P =处理查询 (query); 26 queryResult : P(result) -> R(result); 27 recommendations @R =processRecommendations (result); 28 recommendedProducts : R(recommendations) -> W(recommendations); 29 recommender @W = true 30 } 31 } 清单 9：完整规则

-低功耗 推荐器服务的服务类型 .

对 AIOCJ 中功能与非功能关注点的建模 上述关于将 AIOCJ 构造与架构控制器集成的讨论，引出了一个更普遍的问题：功能关注点与非功能关注点的分离。后者包括性能、可用性、可扩展性、韧性和可靠性。具体而言，尽管 AIOCJ 并未明确建模这些方面，但它们可能会悄然出现在适应代码中。结合我们的示例，我们会根据是否存在某些组件来适应架构，这一点可从适用条件 `E.recommender == “低功耗”` 和 `E.auth == “可用”` 中看出。这些条件出现在适应规则中，例如，规则会将不同服务类型的处理指定为功能规范的一部分。

因此，目前 AIOCJ 并未提供结构化的方式来直接指定或推理这些非功能属性，尽管它们与适应相关。这种混合悄然引入了一种混合方法：一些非功能方面由编舞式协调，另一些则委托给外部控制器（如 Kubernetes），这进一步说明了在编舞层集成外部控制器的必要性。

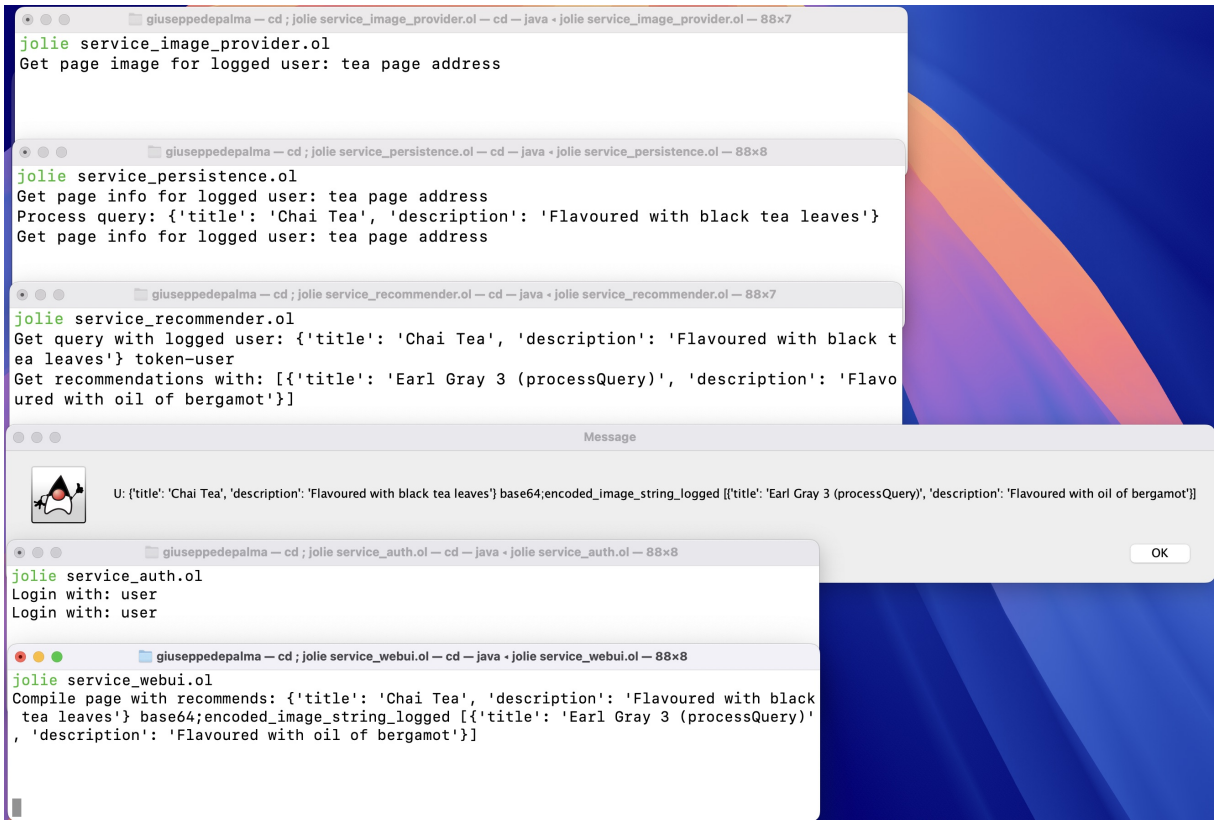


图 4：使用身份验证服务和完整版推荐器的 Adaptable TeaStore 编排的执行。

适配组合复杂性我们对 Adaptable TeaStore 场景和配置的组合性进行建模的另一种方式，是为每个服务配置创建规则，例如，为“身份验证 + 推荐器全功率”“身份验证 + 持久化”等创建规则。这种方法面临的挑战是需要大量规则。事实上，尽管这种替代模式原则上可能产生更易于解释的适应场景（无需根据 作用域和规则的可用性来“推断”可能获得的规则应用组合），但它也可能导致适应规则“爆炸”，具体取决于编排中参与者/服务行为之间的耦合程度。因此，本文选择了规则嵌套方式，例如清单 8 和清单 9 所示。另一个有趣的研究方向是探索在何种上下文中某一种模式可能比另一种更合适，同时考虑定性特征（例如，程序员能否轻松指定可适应系统的预期行为），以及性能（例如，鉴于嵌套适配可能比“扁平”适配涉及更多运行时步骤）。

编舞式服务组合 总体而言，我们观察到 Adaptable TeaStore（与 TeaStore 一样）高度面向编排，WebUI 会编排其他组件提供的服务。本文忠实地对 TeaStore 的行为进行了建模，实现了一种遵循特殊的面向编排的组合模式（大多数交互由 WebUI 集中处理）的编排。未来可以扩展这项工作，提出更符合惯例的 TeaStore 编舞式版本

例如，支持服务之间直接通信，而无需通过 WebUI 路由，例如提高效率。

我们认为，Adaptable TeaStore 的架构模式（集中式编排）与 AIOCJ 建模方法（分布式编舞）之间看似不匹配，这有助于我们展示 AIOCJ 处理非惯用模式的灵活性，说明编舞式方法能够重构面向编排的系统，并且即使在对集中式架构进行建模时，编舞正确性保证（如无死锁）仍然很有价值。

错误处理 为简洁起见，尽管我们没有讨论如何在 AIOCJ 编排中与故障服务交互这一问题，但我们注意到，该语言对服务交互处理的支持较少。

事实上，当编排与外部服务通信时，如果后者发生故障，整个编排就会中断。这种风险是 AIOCJ 模型面临的一项根本性设计挑战。

为了避免因外部服务故障而停止执行，AIOCJ 运行时应包含一个中间服务层（该层不会发生故障），能够处理来自编排的请求，并返回表示操作成功或失败的值。此类中介可根据服务可用性触发适应规则。例如，联系身份验证服务后，若收到表示该服务无法访问的响应，系统便可应用专门针对这一情形设计的适当适应规则。当前模型缺少这一中间韧性层，导致编排与外部服务之间存在脆弱的依赖关系。

或者，也可以引入编舞级错误处理机制，在外部服务发生故障时直接触发适应。这一扩展要求 AIOCJ 纳入异常处理结构（例如 [3]中提出的结构），以便无缝过渡到适应场景。

概括而言，上述讨论揭示了 AIOCJ 等编舞式方法面临的一个更广泛的问题：虽然它们擅长确保正常运行和适应过渡期间的正确性属性，但在面对意外的外部故障时，却难以实现平稳降级。编舞描述与外部服务之间的紧密耦合造成了单点故障，这与现代云架构的韧性目标相矛盾。此外，当适应规则本身依赖于可能发生故障的外部服务时，这一问题会进一步恶化。在这种情况下，原本用于处理需求变化的机制反而容易受到其旨在应对的相同故障模式影响。

以 Adaptable TeaStore 的具体情况为例，当考虑到该架构包含多个服务变体和故障转移场景时，错误处理这一方面尤其明显。虽然 AIOCJ 能够表达服务配置之间的过渡，但它难以处理对促成此类过渡的故障进行检测和管理。

在 AIOCJ 中支持错误处理的一种策略，是利用监控器跟踪外部服务的状态，并将这些信息传递给环境。当外部服务宕机时，环境会获知这一状态变化，并可触发相应的适配。不过，这种监控方法存在固有局限。只有当监控器在作用域 开始时以及执行适应检查时，能够掌握外部服务的最新状态，该方法才能有效发挥作用。更关键的是，如果外部服务在作用域执行期间发生故障，而适应检查已经完成，这种方法便无法提供任何保护。这种时序上的脆弱性造成了一个无法通过 AIOCJ 中传统适应机制解决的故障窗口，凸显了将更强大的错误处理能力直接集成到编舞模型中的必要性。

适配流程 AIOCJ 语言的另一个值得关注的改进方向，是使用记账变量来协调编舞中不同可适配部分的行为。事实上，在我们针对 Adaptable TeaStore 的 AIOCJ 模型中，我们引入了多个变量，用于跟踪不同适应场景下服务的状态和可用性。例如，我们需要变量来记录身份验证服务是否可用，以及当前启用的是哪个版本的推荐器。这些变量并不属于 TeaStore 的概念模型，而是实现层面的构件，用于弥合 AIOCJ 的适配机制与实际系统状态之间的差距。

使用这些记账变量会带来若干问题，例如使编排描述更加复杂、难以理解，因为读者必须在理解实际业务逻辑的同时，在脑海中跟踪这些变量的状态。此外，簿记也可能导致不一致，因为变量在某些执行路径中可能未得到正确更新；这也使静态验证技术更难提出，因为形式化分析必须考虑这些额外的状态元素。

为解决这些局限，可以对 AIOCJ 语言进行改进，直接支持基于服务状态的适应（例如，将规则的运行时交互集成到 Kubernetes 等编排器中），从而使适应条件更具声明性，并与系统的概念模型更加契合。

其他方法 据我们所知，与 AIOCJ 最接近的两种方法都基于多方会话类型 [15]。第一种方法由 Anderson 和 Rathke 提出，涉及动态软件更新 [1]。第二种方法由 Coppo 等人提出，关注自适应系统的监控 [4]。Anderson 和 Rathke 的工作 [1]与 AIOCJ 的主要区别在于，前者针对的是非分布式并发应用程序。实际上，该方法依赖于检查应用程序的全局状态来确保更新安全。单个角色无法执行这种检查，因此在分布式环境中不切实际。此外，Anderson 和 Rathke 的语言 [1] 比我们的语言限制更多，例如要求每对参与者通过专用的一对信道进行交互，并假设在选择结构中既非发送方也非接收方的所有角色在两个分支中的行为相同。Coppo 等人的方法 [4] 也与我们的方法大不相同。具体而言，Coppo 等人 [4] 要求从一开始就提供可能的行为，无论是在类型层面还是进程层面，并使用固定的适配函数在这些行为之间切换。这一区别源于两者所讨论的对象不同：他们讨论的是自适应应用程序，而我们讨论的是从外部更新的应用程序。

我们还回顾了 Di Giusto 和 Pérez [8], 的工作，他们利用类型来确保安全适配。然而，在该工作中，只有在没有活动会话时才能进行更新，而我们则会改变运行中的编排的行为。值得指出的是，与我们的方法不同，以上方法均未实现。

近期，Harvey 等人 [11] 提出了 EnsembleS，这是一种基于参与者的语言，利用多方会话类型为安全运行时适应提供静态的编译时验证。AIOCJ 和 EnsembleS 都旨在确保适应型系统中的通信安全。EnsembleS 通过在编译时检查新发现的组件是否与预定义的会话类型兼容来确保安全。相比之下，AIOCJ 的保证是由构造本身提供的，并允许通过适应规则而非静态类型检查过程来定义动态运行时变更。

拓宽研究范围后，我们发现了 Herry 等人 [12] 和 Wild 等人 [22], 提出的方案，他们利用编排实现应用程序部署的自动化去中心化。这些方法采用全局声明式部署模型，并将其拆分为各参与者的本地部分。随后，基于这些本地模型生成工作流，形成部署编排，以协调本地部署和跨组织数据交换。沿着相似的方向，Philippe 等人 [20] 提出了 Ballet,

这是一种用于重新配置分布式系统的去中心化编排式方法。这些方法与 AIOCJ 的理念一致，都通过编排实现去中心化控制与协调。不过，Herry 等人、Wild 等人和 Philippe 等人的工作聚焦于应用组件的部署与协调。相比之下，AIOCJ 描述的是微服务架构持续进行的动态运行时适应。我们认为，研究如何将 AIOCJ 与这类工作相结合颇有意义，这样可以扩展该语言/运行时对组件部署的支持范围。

在另一项近期研究中，Ortiz 等人提出了一种通过基于事件的 BPMN 片段组合实现微服务编排适应的方法 [19]。在该研究中，使用适应规则目录传播对微服务的局部更改，旨在保持全局流程的功能完整性。然而，由于目录中的规则数量有限，可能会出现没有适用规则的情况，从而导致编排不一致，甚至发生故障。这与 AIOCJ 的编排式方法形成对比：如果没有可用的适应规则，系统会安全地回退到原始行为。

关于分布式软件系统适应技术的更广泛、全面的讨论，特别是侧重于基于组件的方法，我们建议参阅 Coullon 等人近期发表的综述[5]。

最后，我们提及面向方面编程（AOP） [16] 和面向上下文编程（COP） [13]。这两种编程范式都旨在提高代码灵活性。面向方面编程处理横切关注点——即跨越多个软件单元的功能，例如日志记录、安全或事务管理。它不会将代码分散在整个代码库中，而是将其整合到“方面”中，并在“连接点”处织入代码。面向上下文编程则侧重于使程序根据当前执行上下文或环境调整行为。组件不再具有固定行为，而是可以根据用户偏好、系统状态或外部条件等因素动态改变运行方式。Truyen 在 2025年可适应云架构研讨会（WACA）上提出了这些方法的应用 [21] ，旨在将控制逻辑与应用程序解耦、实现细粒度适应、确定系统范围控制，并复用传统的适应策略。在此背景下，AIOCJ 可以通过将“切入点”（即应应用方面的位置）视为空作用域，并将“通知”（即要执行的代码及其执行时机）视为适应规则，来处理日志记录和身份验证等典型的面向方面编程横切关注点。面向上下文编程的层是行为变化的模块化单元，可根据上下文动态激活或停用；而在 AIOCJ 中，则可以通过根据上下文条件应用的适应规则来定义这些层。深入理解 AIOCJ 与面向方面编程和面向上下文编程之间的关系，是未来一项值得探讨的工作。

参考文献

[1] Gabrielle Anderson & Julian Rathke (2012)：消息传递程序的动态软件更新。载于：*APLAS*，*LNCS*7705，Springer，第 207–222 页。[2] Simon Bliudze、Giuseppe De Palma、Saverio Giallorenzo、Ivan Lanese、Gianluigi Zavattaro & Brice Arléon Zemtsop Ndadji (2025)：*AdaptableTeaStore*。载于 Giuseppe De Palma & Saverio Giallorenzo 编：可适应云架构研讨会（*WACA 2025*）论文集，理论计算机科学电子论文集438，开放出版协会，第 1–14 页，doi:10.4204/EPTCS.438.1。[3] Sara Capecchi、Elena Giachino & Nobuko Yoshida (2010)：多方会话中的全局逃逸。载于 Kamal Lodaya & Meena Mahajan 编：*IARCS* 软件技术与理论计算机科学基础年会，*FSTTCS 2010*，*2010*年*12*月*15*日至*18*日，印度金奈，*LIPIcs* 8，Dagstuhl 城堡—莱布尼茨信息学中心，第 338–351 页，doi:10.4230/LIPICS.FSTTCS.2010.338。[4] Mario Coppo、Mariangiola Dezani-Ciancaglini & Betti Venneri (2015)：自适应多方会话。面向服务的计算与应用9(3-4)，第 249–268 页。

[5] Héléne Coullon、Ludovic Henrio、Frédéric Loulergue & Simon Robillard (2023)：基于组件的分布式软件重配置：面向验证的综述。ACM Comput. Surv. 56(1)，doi:10.1145/3595376。[6] Mila Dalla Preda、Maurizio Gabbrielli、Saverio Giallorenzo、Ivan Lanese & Jacopo Mauro (2017)：动态编排：理论与实现。Log. Methods Comput. Sci. 13(2)，doi:10.23638/LMCS-13(2:1)2017。[7] Mila Dalla Preda、Saverio Giallorenzo、Ivan Lanese、Jacopo Mauro & Maurizio Gabbrielli (2014)：AIOCJ：一种用于安全自适应分布式应用程序的编排框架。载于 Benoît Combemale、David J. Pearce、Olivier Barais & Jurgen J. Vinju 编：软件语言工程——第七届国际会议，SLE 2014，瑞典韦斯特罗斯，2014年9月15日至16日，论文集，计算机科学讲义 8706，Springer，第 161–170 页，doi:10.1007/978-3-319-11245-9_9。[8] Cinzia Di Giusto & Jorge A. Pérez (2013)：通过规范化结构化通信实现一致的运行时适应。载于：SAC，ACM，第 1913–1918 页。[9] Saverio Giallorenzo、Ivan Lanese、Jacopo Mauro & Maurizio Gabbrielli (2017)：自适应微服务应用程序编程：AIOCJ 教程。载于 Simon Gay & António Ravara 编：《行为类型：从理论到工具》，River Publishers，第 147–167 页，doi:10.13052/rp-9788793519817。[10] Saverio Giallorenzo、Ivan Lanese & Daniel Russo (2018)：ChIP：一种编排式集成流程。载于 Hervé Panetto、Christophe Debruyne、Henderik A. Proper、Claudio Agostino Ardagna、Dumitru Roman & Robert Meersman 编：《迈向有意义的互联网系统》。OTM 2018 会议——联合国际会议：CoopIS、C&TC 和 ODBASE 2018，马耳他瓦莱塔，2018年10月22日至26日，论文集，第二部分，计算机科学讲义 11230，Springer，第 22–40 页，doi:10.1007/978- 3-030-02671-4 2。[11] Paul Harvey、Simon Fowler、Ornela Dardha & Simon J. Gay (2021)：用于 Actor 语言安全运行时适应的多方会话类型。载于 Anders Møller & Manu Sridharan 编：第 35 届欧洲面向对象编程会议，ECOOP 2021，2021年7月11日至17日，丹麦奥胡斯（线上会议），LIPIcs 194，Dagstuhl 城堡—莱布尼茨信息学中心，第 10:1–10:30 页，doi:10.4230/LIPICS.ECOOP.2021.10。[12]Herry Herry、Paul Anderson & Michael Rovatsos (2013)：编排配置变更。载于：第九届网络与服务管理国际会议论文集，CNSM 2013，瑞士苏黎世，2013年10月14日至18日，IEEE Computer Society，第 156–160 页，doi:10.1109/CNSM.2013.6727828。[13] Robert Hirschfeld、Pascal Costanza & Oscar Nierstrasz (2008)：面向上下文编程。J. Object Technol. 7(3)，第 125–151 页，doi:10.5381/JOT.2008.7.3.A4。[14] Kohei Honda、Nobuko Yoshida & Marco Carbone (2016)：多方异步会话类型。J. ACM 63(1)，第 9:1–9:67 页，doi:10.1145/2827695。[15] Hans Hüttel、Ivan Lanese、Vasco T. Vasconcelos、Luís Caires、Marco Carbone、Pierre-Malo Deniélou、Dim- itris Mostrous、Luca Padovani、António Ravara、Emilio Tuosto、Hugo Torres Vieira & Gianluigi Zavattaro (2016)：会话类型与行为契约基础。ACM Comput. Surv. 49(1)，第 3:1–3:36 页，doi:10.1145/2873052。[16] Gregor Kiczales、John Lamping、Anurag Mendhekar、Chris Maeda、Cristina Videira Lopes、Jean-Marc Loingtier & John Irwin (1997)：面向方面编程。载于 Mehmet Aksit & Satoshi Mat-suoka 编：ECOOP'97——第十一届欧洲面向对象编程会议，芬兰于韦斯屈莱，1997年6月9日至13日，论文集，计算机科学讲义 1241，Springer，第 220–242 页，doi:10.1007/BFB0053381。[17] Fabrizio Montesi (2023)：《编排导论》。剑桥大学出版社，doi:10.1017/9781108981491。

[18] Fabrizio Montesi、Claudio Guidi 和 Gianluigi Zavattaro（2014）：使用 *Jolie* 进行面向服务编程。载于 Athman Bouguettaya、Quan Z. Sheng 和 Florian Daniel 编：*Web* 服务基础，Springer，第 81–107 页，doi:10.1007/978-1-4614-7518-7_4.[19] Jesús Ortiz、Victoria Torres 和 Pedro Valderas（2025）：支持微服务环境中 *BPMN* 片段事件驱动编排的演进。*IEEE Access* 13，第 137827–137849 页，doi:10.1109/ACCESS.2025.3595387.[20] Jolan Philippe、Antoine Omond、Hélène Coullon、Charles Prud'homme 和 Issam Raïs（2024）：使用 *Ballet* 快速编排跨 *DevOps* 重配置：多站点 *OpenStack* 案例研究。载于：*IEEE* 软件分析、演进与再工程国际会议，*SANER 2024*，罗瓦涅米，芬兰，*2024* 年 *3* 月 *12–15* 日，IEEE，第 1–11 页，doi:10.1109/SANER60148.2024.00007.[21] Eddy Truyen（2025）：解耦 *TeaStore* 中的自适应控制。载于 Giuseppe De Palma 和 Saverio Gi- allorenzo 编：可适应云架构研讨会（*WACA 2025*）会后论文集，理论计算机科学电子论文集438，开放出版协会，第 40–57 页，doi:10.4204/EPTCS.438.3.[22] Karoline Wild、Uwe Breitenbücher、Kálmán Képes、Frank Leymann 和 Benjamin Weder（2020）：去中心化跨组织应用程序部署自动化：一种基于声明式部署模型生成部署编排的方法。载于 Schahram Dustdar、Eric Yu、Camille Salinesi、Dominique Rieu 和 Vik Pant 编：高级信息系统工程——第 *32* 届国际会议，*CAiSE 2020*，法国格勒诺布尔，*2020* 年 *6* 月 *8–12* 日，论文集，计算机科学讲义12127，Springer，第 20–35 页，doi:10.1007/978-3-030-49435-3_2。