协作应用中的 For-Each 操作

Matthew Weidner
maweidne@andrew.cmu.edu 卡内基梅隆大学 美国宾夕法尼亚州匹兹堡

BenitoGeordie
bg31@rice.edu 莱斯大学 美国得克萨斯州休斯顿

Ria Pradeep
rpradeep@alumni.cmu.edu
卡内基梅隆大学
美国宾夕法尼亚州匹兹堡

Heather Miller
heather.miller@cs.cmu.edu
卡内基梅隆大学
美国宾夕法尼亚州匹兹堡

arXiv:2304.03141v1 [cs.DC] 6 Apr 2023

摘要

无冲突复制数据类型（CRDT）支持对应用数据的协作访问。我们介绍了一种新型 CRDT 操作——对 CRDT列表执行逐操作，并展示其在协作应用中的应用。我们的逐操作会将给定的变更应用于列表中的每个元素，包括并发插入的元素。这通常能够以一种原本需要定制 CRDT 算法才能实现的方式保留用户意图。我们给出了将逐操作应用于协作富文本编辑器、协作食谱编辑器和幻灯片编辑器的示例。

***CCS*概念：** • **计算理论 → 分布式算法**； • **以人为中心的计算 → 协作与社会计算系统和工具**。

关键词： 协作, CRDT, 并发

1 引言

可变值列表在协作应用中很常见。例如幻灯片编辑器中的幻灯片列表，或富文本编辑器中的富文本字符列表（字符及其格式属性）。

为了允许对这些数据进行协作访问，我们希望使用一个无冲突复制数据类型 [11, 15]。假设我们已经有一个表示列表可变值类型的无冲突复制数据类型 C。那么，我们可以构造一个列表 C的 无冲突复制数据类型，用于表示整个列表。

对一组 C 执行的基本操作允许用户插入、删除列表中的单个元素，并对这些元素应用 C 操作。不过，许多用户操作实际上采用“for-each”循环的形式：对于列表中满足某个条件的每个元素，都对该元素应用 C 操作。例如：

- 在幻灯片编辑器中，用户选中第 3 至第 7 张幻灯片，并将背景颜色改为蓝色。这一操作会：遍历每张幻灯片，如果其索引在 [3, 7]，范围内，就将其“背景颜色”属性设为“蓝色”。
- 在富文本编辑器中，用户全选并点击“粗体”格式按钮。这一操作会：遍历每个富文本字符，将其“粗体”格式属性设为“true”。

实现此类For-Each 操作的简便方式，是在发起用户的副本上使用字面量 for-each 循环。

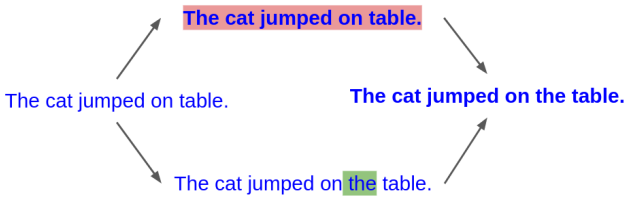

图 1。粗体范围操作（上方）与文本插入操作（下方）并发时的典型用户意图。使用 forEachPrior 操作时，“the”不会显示为粗体。

也就是说，用户的设备会遍历列表的副本，并对各个元素执行 CRDT操作。我们将其称为forEachPrior 操作，因为它作用于在逐操作之前（按因果顺序）插入的元素。

然而，forEachPrior 操作并不总能体现用户意图。图 1 展示了一个经典例子：在富文本编辑器中，如果一位用户将一段文本设为粗体，而另一位用户与此同时在这段文本中间输入文字，那么后者输入的文字也应该显示为粗体。第一位用户副本上的字面 for-each 循环无法做到这一点，因为它无法识别并发插入的字符。

传统上，协作式富文本编辑器使用专门的树结构（例如 Ignetet al. [5]）或在范围两端使用格式标记（例如 Peritext [9]）来实现图1 所示的用户意图。然而，这两种技术都需要仔细分析操作之间的交互，而且它们无法推广到富文本编辑之外。

1.1 贡献

在本文中，我们提出了一种针对 CRDT 列表的新型逐操作。不同于 forEachPrior，它会将给定的变更应用于在逐操作之前或并发插入的每个列表元素。我们将此操作称为 逐操作（不加限定），以区别于 forEachPrior。

图 2 展示了元素的插入操作与 for-each 操作之间可能存在的关系：因果上先于、并发或因果上晚于。我们的 for-each 操作会影响“先于”和“并发”这两类，而 forEachPrior 只会影响“先于”这一类。

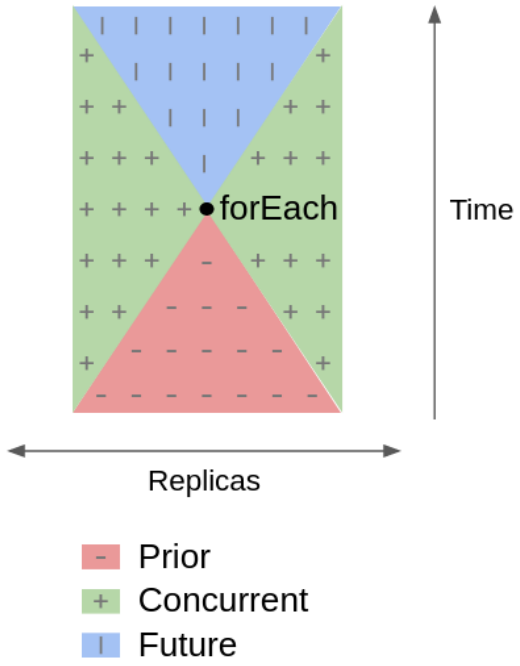

图 2。 forEach 操作的光锥图。

使用我们的 for-each 操作，可以轻松实现图 1 中预期的行为：发出一个 for-each 操作，并执行以下修改：“如果字符在范围内，则将其“粗体”格式属性设为“true””。详情请参见第 4.1 节。

我们希望，Cs 列表和我们的逐操作的预期语义（即用户可见的行为）已经十分清晰。然而，尤其是在将逐操作应用于并发插入的元素时，会出现一些技术问题。第 2 节和第 3 节讨论了这些技术细节，包括我们所有构造的算法。

匆忙的读者可能会直接跳到第 4 节，该节将逐操作应用于示例协作应用。

1.2 背景

我们假设读者熟悉 CRDT操作上的因果顺序 [8]。“（因果上）先前”“并发”和“（因果上）未来”这些术语均指代这一顺序。我们的算法使用向量时钟 [4, 10] 来查询 CRDT 操作之间的因果顺序关系。

在全文中，我们使用基于操作的 CRDT [15], 的术语来描述，尽管我们的构造也可以轻松地改写为基于状态的 CRDT。每个 CRDT 操作都通过生成器 和效应器来描述。生成器负责处理用户在其本地副本上的用户输入，并返回一条消息，以便广播给其他副本。每个副本（包括发送方）都通过将此消息传递给相应的效应器来应用该操作；发送方会与生成器调用一起原子地执行此操作。我们假设确保每条消息在每个副本上都恰好接收一次，并且遵循因果顺序。

2 CRDT 列表

我们首先对*CRDT* 列表进行形式化描述。它基于 Yjs 的 Y.Array 共享类型 [6]。

首先，列表 *CRDT*是一种经典的 CRDT 类型，其外部接口是一个由不可变值组成的列表（有序序列），例如文本文件中的字符 [1, 15]。由于同一个值可能在列表中出现多次，我们使用元素来指代值的唯一实例。列表 CRDT 具有插入和删除元素的操作。

一个*CRDT* 列表 是更通用的无冲突复制数据类型，其中列表值本身是可变 CRDT。具体而言，设 C 是一个基于操作的 CRDT。列表C*s* 的外部接口是一个由 C 类型的可变值组成的列表。该列表上的操作包括：

- 插入(i, σ)：将具有初始值σ—C的一种状态—的新元素插入列表中的索引 i处，位于索引 $i-1$ 和 i处现有元素之间。所有后续元素（索引 $\geq i$）都移至递增后的索引。
- delete(i)：删除索引 i处的元素。所有后续元素（索引 $\geq i+1$）移至递减后的索引。
- apply(i, o): 对索引为 i的元素执行 C 操作 o。所有副本都会按照 C 操作的常规方式更新该元素值的副本（即 C 的一个状态）。并发删除操作可能导致某个副本在删除元素后才收到 apply 消息；在这种情况下，它会忽略该 apply 消息（删除操作“胜出”）。

示例 2.1. 作为一个贯穿全文的示例，请考虑一个协作富文本文件，例如 Google 文档。我们可以使用一个富文本字符无冲突复制数据类型来表示单个字符。其状态是一个二元组 (字符，属性)，其中 字符 是不可变的字符，而 属性 是一个用于格式属性的映射无冲突复制数据类型 [15] 。于是，一个富文本字符无冲突复制数据类型列表 便可模拟整个富文本文件的状态。例如，文本“**ab**”表示为

$$[\{char : \texttt{"a"}, attrs : \{\}\},$$
$$\{char : \texttt{"b"}, attrs : \{\texttt{"bold"}, \texttt{true}\}\}]$$

我们使用 C 和普通列表 CRDT L 构造 Cs 列表。伪代码请参见算法 1。

具体而言，我们假设 L 生成的位置 唯一、不可变，并且取自一个稠密全序 $<$，例如 Logoot 的“位置标识符” [18]。[1] 那么，Cs 列表的实现如下：

状态 元素 (p, σ)，其中 p 是来自 L 的位置，σ 是C的状态，按 p排序。使用该列表的应用通常只查看值 σ，但也可能使用位置，例如光标位置。

[1]如果 L 使用额外状态（例如墓碑）或消息来管理其位置，那么这些内容会隐式地添加到 Cs 列表的状态或消息中。

插入、删除 类似于 $\mathcal{L}$。
应用与 C 类似，但发送给远程副本的消息会标记元素的位置 p。在伪代码中，我们用 C.gen(o, σ) 表示 C 对操作 o 的生成器，并用 C.eff(m, σ) 表示 C 对消息 m的效应器。

1 **每个副本的无冲突复制数据类型状态：**
2 元素：一个元素列表（p， σ），其中 p 是来自 L 的位置， σ 是 C 的状态，按 p排序
3 **查询**元素()
4 **返回** $elts$
5 **更新**插入
6生成器(i, σ)7 $p \leftarrow$ newL位于索引 $i-1$ 和 i 处元素之间的位置 8 返回 (插入 p, σ)9 效应器 (插入 p, σ)10 将 (插入(p, σ) 到 elts 中 11 更新 删除12生成器(i)13 elt $\leftarrow i$中的第 $\leftarrow i$个元素在 $elts$ 中14 返回(删除, elt.p)15 效应器 (删除 p) 16 elt是满足 elt.$p = p$的 elts 中唯一元素，如果不存在则为 null（已删除） 17 如果 elt $\neq$ 为 null则 18 从 elts 中删除 elt 19 更新 应用20生成器(i, o) 21 elt 是 $elts$中的第 $\leftarrow i$个元素22 $m \leftarrow$ C.gen(o,elt .σ)23 返回(应用, elt.p, m) 24 效应器(应用 p, m) 25 elt是满足 elt.$p = p$的 elts 中唯一元素，如果不存在则为 null（已删除） 26 如果 elt $\neq$ 为 null 则 27 elt .$\sigma \leftarrow$C.eff (m, elt.σ) Al

gorithm 1: List of C^{s} 作为一部歌剧 tion-base^{d} 无冲突复制数据类型.

我们在附录 A 中概述强最终一致性的证明（定理 A.1）。

3 逐操作

现在，我们在 Cs 列表上定义新的无冲突复制数据类型操作 遍历。

令 O 表示所有 C 操作的集合。出于技术原因（将在下文第 3.1 节中说明），我们将 for-each

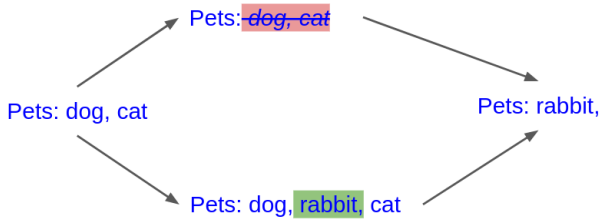

图 3. 范围删除操作（上）与文本插入（下）并发时的典型用户意图：并发插入的文本不会被删除，以避免数据丢失。

限制为纯操作，即当操作生成的消息仅为该操作本身时，该操作就是纯的。形式化地说：

定义 3.1。 令 C.gen(o, σ) 表示 C 的生成器。如果对于所有状态 σ，C.gen(o, σ) $= o$，则操作 $o \in O$ 是纯的 。

Baquero 等人 [2] 表明，许多经典 CRDT 的操作都是纯操作，至少在第 3.1 节讨论的放宽条件下如此。

令 $O_P \subset O$ 表示纯 C 操作的子集。令

$$f : (p, prior) \to O_P \cup \{\text{del}, \text{null}\}$$

这是一个函数，输入为列表元素的位置 p和一个布尔值 $prior$（将在下文说明），并返回以下其中之一：

- $o \in O_P$：要应用于该元素的纯 C 操作。
- del：删除该元素的指令。
- null：表示不执行任何操作的指令。

然后，操作 forEach(f) 会遍历元素，对每个元素应用 f，再执行由f指定的操作。具体来说，它会遍历所有在该逐操作之前因果上插入或与之并发插入的元素，但不会遍历因果上晚于该操作插入的元素。它还会为 f 计算参数 $prior$，该参数表示每个元素在因果上是否先于该操作（true）或与之并发（false）。

示例 3.2。 在富文本文件中，用户将一段文本设为粗体。设$start$和 end分别为该文本范围内第一个字符和最后一个字符之后的位置，因此该范围为 [$start$, end）。定义：

函数 $f(p,\ prior)$**如果** $start \le p < end$ **则返回** ($rich \mapsto rich.attrs$.set("粗体",true))**否则返回** null

然后，forEach(f) 实现了图 1 中预期的行为：范围内的所有字符都会被设为粗体，包括并发插入的字符。

示例 3.3. 在富文本中，用户再次删除一段文本 [起始位置、结束位置）。要仅删除现有字符，我们查询先前状态：

函数 $f(p$、先前状态)

如果 前置且开始 $\leq p <$ 结束 **则**

返回 删除**否则返回** 空值

然后，forEach(f) 实现图 3 所示的行为。请注意，我们也可以改用 forEachPrior 操作，即在发起副本上执行普通循环。不过，forEach(f) 产生的网络流量更少：整个范围只需发送一条 forEach 消息，而不是针对每个被删除的字符分别发送一条删除消息。

算法 2 给出了 forEach 的伪代码实现，下面对此进行说明。

我们首先修改 C 列表，以跟踪每个元素的逻辑插入时间 t——即其发送方的向量时钟条目。[2] 我们还在内部状态中添加一个列表缓冲区 。

当用户调用 forEach(f) 时，其副本会广播 f以及该操作的向量时钟 w。

收到此消息后，副本首先遍历其当前的元素。对于每个元素elt，副本计算$f(elt.p, prior)$并按指示执行，但仅在本地执行；它不会广播任何新消息。这里，$prior$ 表示elt 是否在因果上先于forEach(f)被插入。请注意，我们不会对已在此副本上删除的元素应用 f，包括通过并发删除操作删除的元素（删除操作“获胜”）。

接下来，接收方副本会将消息存储在其缓冲区中。将来，每当副本收到插入消息时，它都会检查该插入操作是否与 forEach(f) 并发。如果是，副本就会计算 $f(elt.p,$ false)，并按照指示执行操作，同样仅在本地执行。请注意，该消息会永远留在缓冲区中，尽管原则上，在所有并发操作都已收到后（即该操作已因果稳定 [2]），便可以将其丢弃。

3.1 OnPure 操作

我们将范围限制在纯操作并非任意之举：即使对于并发元素，我们也需要知道该向 C.eff 传递什么消息。这些元素在发起副本上尚不存在，因此副本无法将它们的状态传递给 C.gen。对于纯操作，我们知道生成的消息就是 o 本身，如第 9 行所用。

实际上，我们可以通过向C.eff传递额外的元数据来放宽纯限制。特别是，我们可以传入f的向量时钟：execute 的第 9 行变为

$$elt.\sigma \leftarrow C.\mathsf{eff}((op, w), elt.\sigma).$$

这不会威胁强最终一致性，因为w 在各个副本之间是一致的。

C 可以使用提供的向量时钟来查询操作的因果顺序。这足以仅使用纯操作 [2] 实现大多数 CRDT。列表 CRDT 的插入操作是一个显著的例外。

[2]有些作者将其称为因果点。

1 每个副本的无冲突复制数据类型状态：

2 元素：元素列表 (p, σ, t)，其中 p 是 L 中的位置，σ 是 C 的状态，$t =$ (senderID, clock) 是一个向量时钟条目；按 p 排序 3 缓冲区：一组配对 (f, u)，其中 f 是 forEach 参数，u 是一个向量时钟条目 4 vc：本地向量时钟，其形式为从副本 ID 到 $\mathbb{N}$ 的函数；初始为全 0 函数 5 replicaID：此副本的唯一 ID 6 函数 execute(f, elt, prior) 7 op $\leftarrow f$(elt $.p$, prior) 8 如果 op $\in O$，则 9 elt$.\sigma \leftarrow$C.eff (op, elt$.\sigma$) 10 否则，如果 op $=$ del，则 11 从 elts 中删除 elt

12 更新插入

*13*生成器*(i, σ)*14 $p \leftarrow$ 新建 L 位于索引 $i - 1$ 和 i 处的各元素中的位置之间15 $v \leftarrow$*vc*的副本

16 $v[replicaID] \leftarrow v[replicaID] + 1$*17* 返回*(*插入 p, σ, v*, replicaID) 18* 效应器*(*插入 p, σ, v*, senderID)19 vc*$[senderID] = v[senderID]$

20 $t \leftarrow$*(senderID* $v[senderID]$*)21* 插入 *(p, σ, t)* 到各元素中 *22//* 遍历并发的 *For-Each* 操作。 *23 for(f, u) in* 缓冲区 *do 24* 并发 $\leftarrow$*(*$v[u.senderID] < u.clock$*) 25 if* 并发*then execute(f,elt, false) 26* 更新*forEach 27* 生成器 *(f)*28 $w \leftarrow$*vc*的副本29 $w[replicaID] \leftarrow w[replicaID] + 1$*30* 返回*(forEach f, w, replicaID) 31* 效应器 *(forEach f, w, senderID)32 vc*$[senderID] = w[senderID]$*33 forelt in elts do 34*前序 $\leftarrow$*(*$w[elt.t.senderID] \geq elt.t.clock$*)35 execute(f, elt, prior)*36 $u \leftarrow$*(senderID* $w[senderID]$*) 37* 将 *(f, u)* 追加到缓冲区 算法 *2*：使用我们的 *For-Each* 操作的 *Cs* 列表

on.

此处未显示的代码块与 Algorith 中的相同

本 1 元素（删除和应用）

3.2 正确性

粗略地说，我们声称算法 2 符合引言中所述的语义。也就是说，无论消息顺序如何，逐操作的f都会恰好应用于因果上先于它的元素和并发元素，并排除已删除的元素。

我们将在附录 A（定理 A.2 和推论 A.3）中给出精确的正确性主张和证明概要。

3.3 其他数据结构

如果忽略列表顺序，但仍为每个元素分配唯一 ID p，For-Each 同样适用。也就是说，我们可以为一组 *CRDT*定义 For-Each 操作，其中每个添加的元素都会被分配一个唯一 ID。

同样，我们可以在 CRDT 值映射上定义 For-Each 操作：设置每个键值对时，为其分配一个唯一 ID，例如 Yjs 的 Y.Map 共享类型 [6]。

然而，我们的构造不适用于 Riak 风格的映射 [3]，其中键的值 CRDT 会在首次使用时创建，而非显式设置：两个用户可能会并发创建同一个键的值 CRDT，从而使选择要应用哪些逐操作 [16, §4]变得复杂。我们预计，在 Kleppmann 和 Beresford 的 JSON CRDT [7],中的 CRDT 列表也会遇到类似问题，因为某个元素可能在删除后重新出现。

完整描述留待未来工作。

4 示例

下面我们从高层次介绍逐操作在协作应用中的示例用法。如第 3 节所述，我们将逐操作写作 forEach(f)，其中

$$f : (p, prior) \to O_P \cup \{\mathsf{del}, \mathsf{null}\}$$

是一个函数，其输入为列表元素的位置 p 以及该元素是否在因果上更早（否则为并发），并输出针对该元素的一条指令：应用一个（纯）操作 $o \in O_P$、删除该元素，或什么也不做。

4.1 富文本编辑器

让我们从引言中介绍的协作式富文本编辑器开始。回顾示例 2.1 和 3.2，我们可以将富文本表示为一个列表，其中包含富文本字符 *CRDT*（字符，属性），其中字符是不可变字符，而属性是用于格式属性的映射 CRDT。给定列表位置起始和结束，定义：

函数 f(p, 先前)**如果** 开始 $\le p <$ 结束 **则返回** (富文本 $\mapsto$ 富文本。属性。设置("粗体",真))**否则返回** 空值

随后，forEach(f) 会将范围 [起点，终点)设为粗体，其预期行为如图 1 所示：范围内的所有字符都会被设为粗体，包括并发插入的字符。

也可以使用闭区间 [开始， 结束′]，而不是半开区间 [开始， 结束)。这里，结束，是原始范围内最后一个字符的位-置，而结束 是最后一个字符之后的位置。区别在于， [开始， 结束)还会格式化范围末尾并发插入的字符，而 [开始， 结束′] 则不会。后一种行为是超链接格式设置 [9] 的典型情况。

其他格式属性也类似。不过，对于删除，通常只删除因果上先于当前操作的字符，如例 3.3 所示。这样更安全，因为删除是单调的（永久性的），意外删除后更难撤销。

请注意，直接使用富文本字符 CRDT 列表会造成内存效率低下，因为每个字符都要存储一个映射 CRDT。不过，可以将这个理论模型用作指导，再实现一种等效但效率更高的 CRDT。例如，可以像 Peritext [9] 一样，仅在 $attrs$ 的状态与前一个字符不同时，才显式存储该状态。

4.2 食谱编辑器

协作式食谱编辑器允许多个用户查看和编辑一道菜的食谱。让我们特别关注配料列表。我们可以将它建模为一个由配料 *CRDT*组成的列表，其中每个配料 CRDT 都包含用于表示其名称和用量的子 CRDT。

假设我们添加一个“缩放食谱”按钮，将每种配料的用量乘以给定值。如果一位用户缩放食谱的同时，另一位用户并发插入一种新配料，那么确保新配料的用量也被缩放就很重要。否则，它的比例就会与其他配料不协调。

要实现这种“缩放食谱”操作，令 s 为缩放值。定义 f如下：

函数 f(p, 先前)
返回 (配料 $\to$配料 .用量 .乘(s))

然后，forEach(f) 会缩放每种食材的用量，包括并发插入的食材。[3]

4.3 幻灯片编辑器

幻灯片编辑器是另一种可以使用 For-Each 操作的协作应用。单张幻灯片可能包含多个图像、形状或文本框。这些对象可以单独编辑，也可以一起编辑。例如，一位用户可能平移（移动）单个对象，而另一位用户同时旋转幻灯片上的所有对象。

为了实现这些平移和旋转，对对象的每次平移都可以表示为一个平移向量。然后，对象的位置由一个列表 CRDT L 表示，其中包含迄今为止进行的所有平移；实际位置是该列表中所有向量的总和。

[3]这里我们假设对“食材用量”无冲突复制数据类型执行 mult(s) 操作。如果还允许 set($value$) 操作，实现起来就并不简单；不过可以使用另一个支持逐操作的列表来实现，具体细节从略。

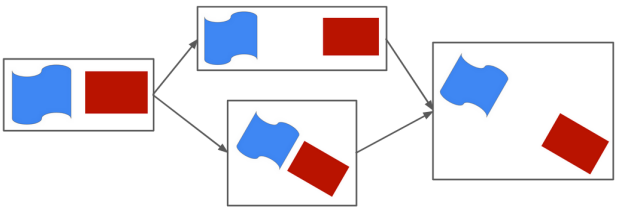

图 4. 典型的用户意图：执行平移操作（上方），同时并发执行群组旋转（下方）；矩形在旋转后的群组内进行平移。

整个幻灯片可以表示为列表 CRDT L′，其中每个元素都是对象 p翻译向量的列表 CRDT Lp 。[4]

用户通过将平移向量追加到对象的列表 L p 中来移动对象p，并通过将 L p 中的对应平移向量乘以旋转矩阵来旋转对象 p 。编辑一组对象时，应对每个对象的列表应用相同的操作。

例如，假设用户将一组对象顺时针旋转 30 度，就像图 4 下方的操作那样。我们希望旋转组中的每个对象。为了使对象在组内保持对齐，我们还希望旋转这些对象的任何并发平移，例如图 4 上方的平移操作。

为实现这一点，令 $updatedObjects$ 表示所选对象的位置 p 集合。定义 g 和 f 如下：

函数 优先 $g(p)$ **函数** $f(p)$ 优先)**如果** $p \in$ 更新的对象 **那么返回** (对象 $\mapsto$ 对象 $\left(\begin{bmatrix} \cos 30 & \sin 30 \\ -\sin 30 & \cos 30 \end{bmatrix}\right)$) **return** $\left(vector \mapsto vector.\text{mult}\left(\begin{bmatrix} \cos 30 & \sin 30 \\ -\sin 30 & \cos 30 \end{bmatrix}\right)\right)$ $forEach$ $(g))$**否则返回** null

然后 forEach(f) 旋转所有 更新的对象 30 度，顺时针方向。旋转的角度也可以存储起来，以便正确渲染旋转后的对象。

5 相关工作

数据流编程和流处理都会对流中的每个“元素”执行操作。与本工作不同的是，它们通常会将逐操作应用于图2中的所有区域，包括因果未来。具体而言，FlowPools[12] 允许在FlowPool（流）开始后发出逐操作；它们会立即将该操作应用于所有现有元素，然后将其存储为并发或未来元素的回调，这与我们的算法（算法 2）类似。

操作转换 [13]允许协作应用中的每个操作对每个对象执行转换并发操作。相比之下，我们允许 for-each 操作转换列表元素（等价于插入操作），但不允许它们相互转换。因此，我们不需要复杂的代数规则来确保最终一致性。

CRDT的半直积 [16] 将两个 CRDT（C_1 和 C_2,）的操作组合到单个 CRDT中。它本质上实现了以下规则：要应用一个 C_2 操作，就以某种方式对每个先前的和并发的 C_1 操作“采取行动”，然后对这些C_1 操作进行归约，以获得当前状态。不过，它并不存储 C_1 操作的字面列表，而只存储它们的归约形式（实际状态）。因此，可以将半直积视为我们带有逐操作的列表的一种经过优化但不太直观的版本。

6 结论

我们对 CRDT 列表进行了形式化定义，并描述了针对该列表的一种新型逐操作。由此得到的 CRDT 对协作应用中的可变值列表进行建模，并支持以下操作：对于列表中的每个元素（包括并发插入的元素），对该元素应用某种操作。我们给出了多个示例，在这些示例中，我们的逐操作比字面意义上的 for-each 循环更符合用户意图。

未来，我们计划在 CollabsCRDT 库 [17]中实现我们的逐操作。

致谢

感谢 James Riely 针对一篇启发了本工作的先前论文 [16] 提出的深刻问题。我们还感谢匿名的 PaPoC 审稿人提出的宝贵意见。Matthew Weidner 获得了由美国海军研究办公室资助的 NDSEG Fellowship。Benito Geordie 获得了由美国国家科学基金会资助的 REU 项目支持。

参考资料

[1] Hagit Attiya、Sebastian Burckhardt、Alexey Gotsman、Adam Morrison、Hongseok Yang 和 Marek Zawirski。2016。《协作文本编辑的规范与复杂度》。载于*2016 年 ACM 分布式计算原理研讨会论文集* （美国伊利诺伊州芝加哥）（*PODC '16*）。计算机协会，纽约州纽约，美国，

259 268 10 1145 2933057 2933090 – 。

[2] 2014 https://doi.org/ 。/ 。Carlos Baquero、Paulo Sérgio Almeida 和 Ali Shoker。。让基于操作的 CRDT 成为基于操作的。载于分布式应用与互操作系统（计算机科学讲义），Kostas Magoutis 和 Peter Pietzuch（编）。Springer，柏林，海德堡，126–

140 10 1007 978 3 662 43352 2_11 。https://doi.org/ 。/ - -

[3] 2015 - - Basho。。Riak 数据类型。http://github.com/basho。[4] Colin J. Fidge。1988。保留偏序关系的消息传递系统中的时间戳。第 *11* 届澳大利亚计算机科学会议论文集 10，1（1988），56–66。[5] Claudia-Lavinia Ignat、Luc André 和 Gérald Oster。2021。通过实时协作增强富内容维基。并发与计算：实践与经验 33，8（2021），e4110。https://doi.org/10.1002/cpe.4110 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4110 e4110 cpe.4110。[6] Kevin Jahns。2022。Yjs。GitHub 仓库。https://github.com/yjs/yjs

[4]由于幻灯片上的对象没有顺序，我们仅将它们的位置 p 用作 ID，而忽略它们的整体顺序。

[7] Martin Kleppmann和 Alastair R Beresford。2017 年。无冲突复制 JSON 数据类型。*IEEE*并行与分布式系统汇刊 28，10（2017 年 4月），2733–2746。https://doi.org/10. 1109/TPDS.2017.2697382 arXiv:1608.03960[8] Leslie Lamport。1978年。分布式系统中的时间、时钟与事件排序。*Commun. ACM* 21，7（1978 年7月），558–565。https: //doi.org/ 10 1145 359545 359563 。/ 。 Geoffrey Litt、Sarah Lim、Martin [9] Kleppmann和 Peter vanHarden-berg。2022年。Peritext：用于协作式富文本编辑的无冲突复制数据类型。*Proc.ACM*人机交互 6，CSCW2，文章 531（2022 年11 月）， 36 10 1145 3555644 页。https://doi.org/。/ [10] 1989 Friedemann Mattern。 。分布式系统的虚拟时间与全局状态。载于并行与分布式算法。North- Holland，215–226。[11] Nuno Preguiça、Carlos Baquero 和 MarcShapiro。2018 年。无冲突复制数据类型 *CRDT*。施普林格国际出版公司，Cham，1–10。https://doi.org/10.1007/978-3-319-63962-8_185-1[12] Aleksandar Prokopec、Heather Miller、Tobias Schlatter、Philipp Haller 和 MartinOdersky。2013 年。FlowPools：无锁确定性并发数据流抽象。载于 并行计算语言与编译器，HironoriKasahara和Keiji Kimura（编）。施普林格柏林海德堡，柏林，海德堡，158–173。[13] Matthias Ressel、Doris Nitsche-Ruhland和 RulGunzenhäuser。1996年。群组编辑器中一种面向集成、面向转换的并发控制与撤销方法。载于*1996* 年*ACM*计算机支持的协同工作会议论文集 （美国马萨诸塞州波士顿） （*CSCW '96*）。计算机协会，纽约，NY，美国， 288 297 10 1145 240080 240305 – 。 [14] https://doi.org/。/ 。 Marc Shapiro、Nuno Preguiça、CarlosBaquero 和 MarekZawirski。2011 年。收敛与可交换复制数据类型的综合研究。研究报告 RR-7506。法国国家信息与自动化研究所– 巴黎-罗康库尔中心；法国国家信息与自动化研究所。 50 00555588 页。 [15] https://hal.inria.fr/inria- Marc Shapiro、Nuno Preguiça、CarlosBaquero 和 Marek Zawirski。2011年。无冲突复制数据类型。载于 第 *13* 届分布式系统稳定性、安全性与安保国际研讨会（*SSS 2011* 386 400 10 1007 978 3 642 24550 3_29 ）。– 。 [16] https://doi.org/ 。/ - - - - Matthew Weidner、HeatherMiller和 Christopher Meiklejohn。2020 年。使用半直积组合与分解基于操作的无冲突复制数据类型。*Proc. ACM* 编程语言 4，ICFP，文章 94（2020 年 8月），27 页。https://doi.org/10.1145/3408976[17] Matthew Weidner、HeatherMiller、Huairui Qi、Maxime Kjaer、Ria Pradeep、Ignacio Maronna、Benito Geordie 和 Yicheng Zhang。2022 年。Collabs。GitHub 仓库。https://github.com/composablesys/ collabs[18] Stephane Weiss、Pascal Urso和 Pascal Molli。2009年。Logoot：点对点网络上用于协作编辑的可扩展乐观复制算法。载于 *2009*年第 *29* 届 *IEEE* 分布式计算系统国际会议。404–412。https://doi.org/10.1109/ICDCS.2009.75

一个正确性证明

定理A.1。 *C* 列表（算法 *1*）满足强最终一致性。

证明概要。只需证明并发操作可交换 [14, 命题 2。2]

任何引用不同位置的操作都显然可以交换。特别是，插入操作与所有类型的操作都可以交换，因为插入操作总是使用一个新位置。对同一位置执行的删除操作显然可以交换。对同一位置执行的应用操作可以交换，因为 C 是一种无冲突复制数据类型。最后，一个删除操作与一个应用操作作用于同一位置时可以交换，因为无论哪种情况，该元素最终都会被删除。□

定理 A.2（算法 2 的语义）。 固定一个算法*2* 副本和一个时间点。设***elt*** 是一个其插入消息已被该副本接收的元素。如果 ***elt***存在于***elts*** 中，则 $elt.\sigma$ 是对其初始状态依次按因果顺序恰好施加以下消息的结果：

(a)所有因对 ***elt*** 应用操作而产生且已接收的消息。***(b)*** 所有（纯）操作 $f(elt.p,$ 真），其中已接收 forEach(f)，且该操作在因果上晚于 ***elt*** 的插入操作。***(c)*** 所有（纯）操作 $f(elt.p,$ 假），其中已接收 forEach(f)，且该操作与 ***elt*** 的插入操作并发。

当且仅当副本已收到针对 ***elt*** 的删除消息，或者上述某个 f 调用返回了del时，该元素会被删除（不再存在于 ***elts*** 中）。

证明概要。类型 (a) 的消息由 apply 的效应器处理。

收到 forEach(f) 时，forEach 的效应器会处理类型 (b) 的操作。注意，根据向量时钟的性质，第 34 行正确地将$prior$ 设为 true。

对于类型 (c) 的操作，有两种情况。如果在收到 for-each 消息之前已收到elt的插入消息，那么情况与上一段类似。

否则，当插入效应器遍历缓冲区时，该操作会被处理。注意，它正确地将$prior$ =设为 false。由于缓冲区 按接收时间排序，该循环会按照因果顺序处理此操作以及其他类型 (c) 的消息。此外，由于elt 是新插入的，该操作会在所有类型 (a) 和 (b) 的消息之前处理；这符合因果顺序，因为所有此类消息都是并发的，或在因果上晚于该操作。

关于删除的说法也类似。 □

推论 A.3. 算法 *2* 满足强最终一致性。

证明。 根据该定理的描述以及 C 是一种无冲突复制数据类型这一事实。 □