使 LL/SC 可写且保持历史独立性

Dante Bencivenga ✉加拿
大卡尔加里大学**Philipp**
Woelfel ✉加拿大卡尔加里大学

arXiv:2611.33067v1 [cs.DS] 24 Nov 2026

摘要

一种针对每个最多具有 k 个未完成LL() 操作的 n 个进程的随机化 m 加载链接/条件存储对象实现，使用 $O(kn+m)$ 个比较并交换和 FAI 对象 [9], , 实现了常数级期望步复杂度，并为诸如 [7] 的算法保持静止历史独立性。与其他一些加载链接/条件存储算法（例如 [5, 6, 25, 26], ）不同，该算法不支持无条件写操作。我们展示了如何添加一个（随机化的）无等待写操作，同时不影响该实现的渐进步复杂度或空间复杂度。

2012年 ACM 学科分类 计算理论 → 共享内存算法；计算理论 → 算法设计与分析；计算理论 → 并发算法

关键词和短语 可写 LL/SC、并发算法、历史独立性、随机化算法、无等待性

资助 *Dante Bencivenga*：加拿大自然科学与工程研究委员会（NSERC）加拿大博士研究生奖学金（CGS-D）578918-2023/198，以及卡尔加里大学 Eyes High 博士生招募奖学金

Philipp Woelfel：加拿大自然科学与工程研究委员会（NSERC）项目编号 RGPIN-2025-04233

目录

1 引言 **2**

2 预备知识 **4**
2.1 原语操作.4 2.2 对手.5 2.3 顺序规范.5 2.4 未完成的 LL() 操作.6

3 实现 6
3.1 修改思路 from [9] .6 3.2 低级描述.9

4 分析 **14**
4.1 对计数器的贡献.15 4.2 锁定标签.19 4.3 保护标签.22 4.4 历史独立性.23 4.5 期望步复杂度.25 4.6 线性化.26

5 结论 **31**

1 引言

异步共享内存算法通常需要使用原子硬件指令来协调其操作。现代处理器如今普遍支持多种此类指令，例如交换、比较并交换和取值并加法：交换指令将一个值写入共享内存字，并返回其紧接之前的值。比较并交换（或 CAS(old, new)）指令仅在共享内存字存储着预期的 old 值时才会生效，此时它会将该值替换为 new。在本文中，我们只需要 CAS() 返回一个表示成功或失败的布尔值，但它有时也可以像一次 Swap() 调用那样返回先前的值。取值并加法指令将给定的整数值加到存储在共享内存字中的值上，并返回先前的值。在本文中，我们只需使用取值并加法将值加一或减一，分别称为取值并递增（FAI()）和取值并递减（FAD()）。

尽管 CAS() 结合读写寄存器已经足以实现任何具有顺序规范的共享对象 [21], ，但由于*ABA* 问题，正确使用它可能并不容易。例如，假设一个共享内存对象存储着值 v_1，某个进程读取了该值，然后尝试将其从 v_1 更改为 v_2。该进程无法得知，从读取到 CAS()期间，对象的值是否一直保持为 v_1 ，也无法得知它是否曾变为 v_3 后又变回来。这种模棱两可的情况就是 ABA 问题，如果不谨慎处理，很容易导致数据结构被错误修改。

加载链接/条件存储 对象通过使用两种操作解决了这个问题。加载链接（LL()）操作会读取并返回对象的值。条件存储（SC(v)）操作可能成功，也可能失败：当且仅当自上一次 LL()以来，没有发生 Write() 或成功的 SC() 时，该操作才会成功，并将值 v 写入对象。否则，该操作将失败（更正式的规范见第 2.3 节）。因此，它能避免 ABA 问题，从而防止一大类难以发现和修正的错误。大多数实现（包括我们的实现）还包含链接验证（VL()）操作，该操作会返回一个布尔值，表示 SC() 是否会成功，而不会改变对象的状态。

遗憾的是，本文所定义的加载链接/条件存储并不存在于现代硬件中。（一些架构提供了一对名为 LL() 和 SC()的指令，但其 SC() 操作即使在内存字未发生任何变化时也可能意外失败，而且对这对指令之间可以执行哪些操作有严格限制。为与“强”加载链接/条件存储区分开来，这些指令有时被称为 RLL/RSC；例如参见 [26]。）因此，人们提出了多种软件实现 [3,4,9,13,15,23,27–29,32, 34–36], ，具备不同的特性，例如支持 Write() 操作 [5,6,24–26], 、实现强线性化 [36],持久性 [25], ，或根据系统中的进程数量进行调整 [15,25,29]。尽管任何 LL/SC 实现都可以通过反复调用LL()，然后调用 SC(v)，直到某次 SC(v) 尝试成功，来支持无锁 Write(v) 操作，但要在同一对象上同时支持无等待 Write() 和 LL/SC 则更加困难。

我们的实现具有最优的空间复杂度和期望步复杂度，使用有界基础对象，除了 LL() 和 SC()之外还支持 Write() 操作，并且 保持静止历史独立性，下面将对此进行说明。

历史独立性 是一项实用的安全属性，在顺序计算领域已得到广泛研究 [1,11,12,14,17–20,33,37,38]，最近也开始在并发计算领域受到关注 [6,7,9,39]。粗略地说，如果某个实现中，操作之间的内存状态仅由所实现对象的抽象状态决定，那么该实现就是历史无关的。在并发计算领域，历史独立性有多种定义方式， [6]对此进行了探讨。其中最严格的定义是完美

历史独立性，它要求共享内存在执行的所有 时刻都由抽象状态决定；但他们指出，对于大多数抽象对象而言，这需要规模大得不切实际的基础对象。

与 [9], 一样，我们转而关注静止历史独立性（QHI）。静止状态是指这样一种系统配置：没有进程处于等待中，即没有进程已发起方法调用但尚未收到响应。若某个实现中，在其任意执行所能到达的所有静止状态下，共享内存的状态都是并发对象抽象状态的函数，则称该实现满足 QHI。由于使用加载链接/条件存储的算法通常不要求在静止配置中存在未完成的 `LL()` 操作（即尚未有匹配的 `SC()` 操作跟随其后的 `LL()` 操作，我们将在定义 2 中对其作形式化定义），我们按照 [9] 采用 QHI 的宽松定义：将 QHI 定义中的静止状态替换为稳定状态，即额外满足没有进程存在未完成 `LL()` 操作的静止状态。因此，在任意可达执行的稳定状态下，共享内存的状态必须是抽象对象状态的函数。满足这一宽松定义的加载链接/条件存储实现（例如我们的实现）称为保持 *QHI*。任何使用保持 QHI 的加载链接/条件存储实现的算法，只要进程空闲时不存在未完成的 `LL()` 操作，就满足 QHI。通常可以在方法响应之前添加“清除链接”或 `CL()` 操作来实现这一点，从而清除未完成的 `LL()` 操作。

很少有 LL/SC 实现还能提供一个具有次-$O(m \cdot n)$ 空间复杂度的 `Write()` 操作，其中 m 是 LL/SC 对象的数量， n 是进程的数量。Jayanti、Jayanti 和 Jayanti 的实现 [25] 具有常数步复杂度，且自适应空间复杂度最优，为 $O(m+n+C)$，其中 C 是给定时刻尚未完成的 `LL()` 操作总数；该实现还可在非易失性内存上持久保存。然而，它使用无界序列号，因此不适用于历史独立算法。Attiya、Bender、Farach-Colton、Oshman 和 Schiller 的实现 [6] 满足完美历史独立性，但使用大小为 $O(n)$ 位的大型基础对象，且仅为无锁实现。

其他论文提出了变换，可为现有 LL/SC 实现添加 `Write()` 操作：Jayanti [24]提出一种变换，通过将固定大小的标签附加到对象的值上，从两个不可写 LL/SC 对象得到一个可写 LL/SC 对象。然而，由于该算法的正确性要求这些标签在静止状态下不得重置，因此该变换无法保持历史独立性。Aghazadeh、Golab 和 Woelfel 提出的另一种更通用的变换，可为共享对象 [2], 添加无条件的 `Write()` 操作；随后，Ben-David、Blelloch、Friedman 和 Wei 优化了其空间复杂度 [8], ，但他们的变换会额外增加 $O(n^2)$ 个基础对象。此外，与我们的实现不同，他们的变换也无法保持历史独立性。

我们的实现基于 Bencivenga、Habashi 和 Woelfel 的实现 [9], ，该实现具备上述特性，但不包含 `Write()` 操作。增加随机化无等待`Write()` 操作以及新的正确性分析，是本文的主要技术贡献。我们的 `Write()` 操作也可以自然地用作 `Swap()` 操作，因为它在硬件 `Swap()` 操作处进行内部线性化（而 [9]中并未使用该操作）。

▶**定理 1**。 设 n 为进程数， k 为每个进程同时未完成的 LL() 操作的最大数量。则存在一种从 m *CAS/Swap* 对象实现 m 加载链接/条件存储对象的线性化且保持 *QHI* 的实现，该实现存储加载链接/条件存储值及标签，共需$\log(kn) + O(1)$ 位，并使用$(m + kn)$ 个 *FAID/CAS* 对象，每个对象需$\log n + O(1)$ 位。该实现支持所实现的方法 LL()、 SC()、 Write()、 Swap()、 CL()、Read()和 VL()，在弱自适应对手下的期望步复杂度为 $O(1)$ 。

由于与 [9], 相比，我们的实现使用了更多基础对象、更大的标签，并且每个实现方法需要执行更多步骤，同时还使用了种类更丰富的原语操作，因此，在不要求无条件写入 LL/SC 对象的情况下，他们的实现更为合适。

与 [9], 的实现类似，我们为实现 LL/SC 字的每个比较并交换对象附加了有界标签，其大小为 $\lceil \log(k+1)n \rceil + 2$ 位。正如 [9, 13] 所论证的，在使用 LL/SC 的已发表算法中， k 至多为 3，因此在实践中，我们的标签最多为 $\lceil \log n \rceil + 4$ 位。由于 64 位堆指针通常将高 16 位和低 3 位设置为零 [30], ，因此可以将带有 19 个标签位的指针存储在一个 64 位字中。因此，当进程数量不超过$2^{15} = 32$时，我们的实现可用于在 *LL/SC* 对象中存储指针值; 768，这与 Linux [31]上的默认最大进程数和线程数相匹配。

支持无条件无等待 Write() 所需的额外复杂性，部分原因在于：为了防止 ABA 问题，正确选择新标签变得更加困难。具体而言， [9]会统计对于 FAID 对象中的每个标签值，有多少个进程执行了尚未完成的 LL() 操作并见过该标签值； SC() 则会随机探测这些使用计数，以找到使用计数为零的标签。这在一定程度上之所以可行，是因为一次成功的 SC() 之前必定会有一个 LL() 操作，因此前一个标签的使用计数非零，从而确保成功的 SC() 始终会改变标签。然而，无条件的 Write()操作没有这一前提，因此需要进一步的机制来确保它始终改变该标签，使并发的 SC() 操作能够失败。因此，我们的使用计数还会统计当前使用给定标签的 LL/SC 对象数量。 SC() 和Write() 会尝试对随机探测到的使用计数执行从零到一的 CAS() ，随后一直保持该使用计数递增，直到 LL/SC 对象不再存储该标签。这使得跟踪使用计数更加复杂，因为递减使用计数的责任可能转移给另一个进程，同时也影响分析的其他几个方面。

本文其余部分安排如下：第 2 节定义共享内存模型和可写 LL/SC 的规范，第 3 节介绍我们的实现并对其作直观解释，第 4 节证明定理 1，第 5 节总结全文。

2 预备知识

我们采用标准的异步共享内存模型，其中 n进程通过对共享对象执行原子共享内存操作来进行通信。在执行过程中的每个时刻（我们简称为“时刻”），恰好有一个共享内存操作发生。

2.1 原语操作

我们使用的原子原语操作包括 Read()、 CAS()、 FAI()、 FAD()和 Swap()。Read() 操作返回共享对象的值。 Swap(val) 操作将val 写入共享对象，并返回其先前的值。 CAS(old, new) 操作将 new 写入共享对象；当且仅当其先前的值为 old 时返回真，

否则保持对象不变并返回假。最后， FAI() 和 FAD()分别递增或递减对象的（整数）值，并返回先前的值。在我们的分析中，我们首先假设，对值 max 执行 FAI() 时会回绕至 0；同样地，对 0 执行 FAD() 时会回绕至 max。我们证明（引理 7 和引理 12(a)），FAID 对象不会发生溢出和下溢，因此这些对象可以具有任意的溢出和下溢行为（包括在溢出前具有更大的可能值范围）。

我们实现中的对象要求对数组 L 的元素使用组合 Read()、 CAS()和 Swap()，对数组 C 的元素使用组合 Read()、 FAI()和 FAD()，对数组 R 的元素使用组合 FAI()、FAD()和 CAS$(0,1)$。（在实际实现中，所有共享内存变量通常都支持相同的一组原子硬件指令，其中通常包括本文所用的全部指令。）

2.2 对手

我们的算法是随机化的，这意味着进程可以随时通过执行随机选择步骤生成一个（私有）随机数。对手通过在执行期间决定下一个采取步骤的进程，来确定进程采取步骤的顺序。我们假设使用与 [9]，中相同的弱自适应对手：进程总是在随机选择步骤之后立即执行其下一次共享内存操作。也就是说，尽管弱自适应对手知道所有过去的随机选择并会据此调整策略，但它无法介入进程的随机选择与随后的共享内存步骤之间。虽然我们采用了随机化，但所有正确性属性（如线性化 [22] 和保持静止历史独立性（QHI））都是确定性地实现的；也就是说，无论做出何种随机选择，这些属性对所有执行都成立。

2.3 顺序规范

本节紧接 [9]，的第 2.1 节，其中除 Write() 之外的所有操作均按该论文中的相同方式定义。

一个抽象对象（也称为类型）是一个五元组 (Q, q_0, I, R, δ)，其中 Q 是状态集合， q_0 是初始状态， I 和 R 分别是操作集合和响应集合，而 δ： $Q \times I \to Q \times R$ 是转移函数。这个五元组也称为该对象的顺序规范，因为它定义了可能的顺序执行集合：当对象处于状态 q 且操作 i 被调用时， $\delta(q, i) = (q', r)$ 定义该操作的响应 r 和由此产生的新状态 q'。

为描述可写 LL/SC 抽象对象，我们采用 [6,9,25] 中的定义。每个状态都是一个二元组 $(v, \mathcal{L})$，其中 v 是对象的值，而$\mathcal{L}$是一个 n 位序列 $(\ell_1, \ldots, \ell_n)$，称为上下文。初始时， $\ell_1 = \cdots = \ell_n = 0$，而 v 的初始值可以任意定义。若 $\ell_p = 1$，我们就说 p 与 LL/SC 对象之间存在开放链接。

A Read() 返回对象的值 v。进程 p调用 LL() （“加载链接”）时，会将 ℓ_p 设为 1，并返回值 v。进程 p 执行 A Write(x) 操作时，会无条件将对象的状态更改为 $(x,$ (0，…，0)），且不返回任何内容。进程 p 执行 SC(x)（“条件存储”）操作时，若 $\ell_p = 1$，则操作成功，并将对象的状态更改为 $(x,$ (0，……，0)）（与 Write(x)类似），然后返回真。若 $\ell_p = 0$，则操作失败，返回假，且不更改对象的状态。进程 p 执行 OperationCL() （“清除链接”）时，只会将 ℓ_p 重置为 0，且不返回任何内容。最后，进程 p 执行操作 VL() （“验证链接”）时，会返回一个布尔值，当且仅当$\ell_p = 1$时，该值为真。

本文实现了一组 m 可写 LL/SC 对象，并将其中第 i 个对象称为 L_i。因此，抽象对象的状态是 m 个独立 LL/SC 对象状态的笛卡尔积。因此，每个操作都多一个参数，用于指明集合中的哪个 LL/SC 对象会受到影响。例如，操作`Write`(i,x) 的行为与上述定义的 `Write`(x) 操作相同，但只影响集合中的第 i 个 LL/SC 对象 L_i。

2.4 未完成的 LL() 操作

我们的实现需要知道进程的数量以及每个进程的未完成 `LL()` 操作数量，下面对此进行形式化定义：

▶ 定义 *2*（未完成的 `LL()` 操作）。当且仅当进程 p 在时刻 t 对 L_i 有一个未完成的 `LL()` 操作时，p 在时刻 t没有对 L_i 的待处理方法调用，并且p 在时刻 t 之前调用了 `LL`(i)，此后既未调用 `Write`$(i,\cdot)$、 `SC`$(i,\cdot)$或 `CL`(i) 调用，也未调用返回假值的 `VL`(i) 调用。等价地，我们称 p 在时刻 t有一个未完成的 `LL`(i)。

未完成 `LL()`操作的定义与 [9], 类似，不同之处在于，对`Write`$(i,\cdot)$ 的调用会清除任何未完成的 `LL`(i)，并且我们不认为对 L_i 有待处理调用的进程具有未完成的 `LL`(i)。

在我们的实现中， k 是每个进程所能拥有的未完成 `LL()`操作数量的每进程上界（而不是像 [9]中那样的全局上界）。因此，我们要求在调用 `LL`(i)时，调用进程对除 L_i之外的对象所拥有的未完成 `LL()` 操作数量少于 k。

与 [9], 一样，我们的实现要求进程在从方法返回之前，如果不再需要未完成的 `LL`(i) 操作，则使用 `CL`(i) 清除这些操作，以实现我们的空间上界并保留 QHI。

3 实现

3.1 修改思路 [9]

从非常高的层面来看， [9] 的不可写 LL/SC 实现通过将第 i 个 LL/SC 对象的值 L_i 与一个非负有界标签一起存储在数组条目 $L[i]$ 中来工作。

数组 R 为每个非零标签 g 设置一个元素： $R[g]$ 存储标签 g 的使用计数，即从某个 L 元素的标签字段中读取标签 g 的未完成 `LL()` 操作数量。（标签零是默认值，因此其处理方式不同，包括不需要使用计数。）如果 `LL`(i) 调用读取了该 $L[i]$. 标签 $= g \neq 0$, 则使用 $R[g]$ 递增。非零使用计数表示某个进程已保护该标签，在受保护期间不应重新使用该标签。 `SC`$(i,\cdot)$ 调用会随机探测 R 的元素，直到找到一个新标签 g'，使其读取 $R[g'] = 0$。使用计数为零表示 g' 未受保护，因此可以安全地将 $L[i]$. 标签更改为 g'，而不会引发 ABA 问题。清除未完成 `LL()` 操作的任何操作随后都会递减其关联标签的使用计数。

另一个数组 C 为每个 LL/SC 对象设置一个元素： $C[i]$ 存储L_i的活动计数，即在 L_i 上有待处理的方法调用或存在尚未完成的 `LL`(i)的进程数。当 $C[i]$ 归零时，重置

将 $L[i]$.标签设为零是安全的（也是保持 QHI 所必需的），因为没有尚未完成的 LL(i)，因此无需防范 ABA 问题。因此，可以大致将非零活动计数视为在静止状态下保护标签零不被使用。

如需更详尽的解释，请参阅 [9] 第3节中的高层和低级描述。

现在，我们来说明为何要修改 [9] 的算法，以支持随机化的无等待 Write() 操作，以及为确保正确性、保持相同的渐近步数和空间复杂度并保留静止历史独立性而必须进行的修改。

我们的实现与 [9] 的实现之间，主要区别在于如何管理数组 R 中的使用计数。与 [9], 一样，当进程对 LL/SC 对象 L_i 执行 LL(i)，并看到非零标签 g 时，它会递增使用计数 $R[g]$。（默认标签 0 没有使用计数，其 ABA 防护由数组 C 中的活动计数提供。）

回想一下，在 [9], 中，SC(i,·) 调用会随机探测 R 的元素，直到对某个 $R[g']$ 读到 0——这意味着在读取时，标签 g' 未受保护。然而，当 SC() 调用尝试对 $L[i]$ 执行 CAS() 时，标签 g' 可能已经受到保护；若操作成功，便会将 $L[i]$. 标签改为受保护标签 g'，从而可能引发 ABA 问题。但是，进程只有在看到 $L[i]$. 标签为 $= g'$ 时才会保护标签 g'，这意味着在尝试对 $L[i]$ 执行 CAS() 之前，$L[i]$ 已发生变化。此外，调用 SC(i,·) 的进程本身也在保护一个标签 g，以防其从 $L[i]$. 标签读取的、尚未完成的 LL(i) 操作。它在比较并交换尝试中使用标签 g，也就是说，CAS() 是从 $(\cdot\ g)$ 变为 $(\cdot\ g')$。因此，CAS() 成功的唯一可能是 $L[i]$. 标签从 g 变为 g'，然后又变回 g，也就是发生了更早的 ABA。由于 SC() 是他们的实现中唯一能够修改 $L[i]$ 的方法，而且它不可能导致首次 ABA，因此它也不可能导致任何 ABA（参见他们的引理 17）。关键在于，他们的论证依赖于只有 SC() 调用能够修改 $L[i]$，并且 $L[i]$ 只能通过 CAS() 操作进行修改。

与 SC(i,·) 一样，新增的 Write(i) 操作也应选择一个新标签，确保所有进程当前尚未完成的 LL(i) 都未曾看到过该标签，以防这些进程的 SC(i,·) 操作出现 ABA 问题。然而，与 SC() 不同，Write() 总是成功，且不需要先执行 LL()，同时仍然是（随机化）无等待的。这意味着，仅仅验证使用计数为零已不足够：进程 p 在执行 Write(i,·) 操作时，可能在读取 $R[g] = 0$ 后休眠；之后，标签 g 可能会在来自另一个 SC(i,·) 或 Write(i,·) 操作的过程中，被用于 $L[i]$. 标签。由于我们要求 Write() 是一种随机化无等待方法，且无条件更新 L_i，因此在其操作过程中，必定存在某个步骤以非零概率更新 $L[i]$，同时将 $L[i]$. 标签更新为一个新的、未受保护的标签。但 Write() 调用在更新 $L[i]$ 之前，仅仅读取过 $R[g] = 0$，并不知道 g 是否仍未受保护，因此任何此类对 $L[i]$ 的更新都可能导致 ABA 问题。（相较之下，SC() 会尝试执行 CAS()；正如前文所述，如果 $L[i]$ 已更改，该操作必定会失败。）

相反，我们的方法是让 p 通过将对零的读取替换为一次 $R[g]$.CAS($0, 1$) 操作，来阻止任何并发的 Write() 或 SC() 操作使用其选定的标签 g。这实际上会锁定该标签，使其仅供 p 使用，直到 $R[g]$ 最终递减回零。按照 [9], 的做法，我们假设进程由弱自适应对手调度，该对手无法在一次随机选择与紧接着的共享内存步骤之间进行干预。因此，随机选择一个新标签 g 后，我们会立即尝试执行一次 $R[g]$.CAS($0, 1$) 操作；只要有足够多未使用的标签可供选择，该操作就会以恒定概率成功。我们将调用进程成功对标签 g 执行 $R[g]$ 锁定的操作称为一次成功的 CAS()（这种锁定不会损害随机化无等待性，因为我们的算法保证

每次随机探测标签时都有恒定的成功概率）。随后， Write(i, u) 调用可以无条件地将这个新标签 g 及其值 u 写入 $L[i]$。我们还需要防止Write() 调用使用与并发的 SC() 调用相同的所选标签，因为 Write()具有无条件执行的特性，这可能导致 ABA 问题。因此，我们修改 SC() 方法，使其针对每个随机选定的标签尝试执行一次 CAS($0, 1$) ，直至成功，做法与Write() 方法完全相同。

为了确保历史独立性，并允许最终安全地回收标签，在系统进入静止状态之前，必须最终递减 $R[g]$ 的这次额外递增。若 SC() 操作失败，调用进程可以立即安全地递减其所选标签的使用计数，因为该标签没有在其他任何地方使用。

然而，考虑一下如果我们在 Write(i,·) 调用结束时，或在成功的 SC(i,·) 调用结束时递减 $R[g]$ ，会发生什么。如果另一个已崩溃的进程保留着 $C[i] > 0$，表明系统处于非静止状态，那么 $L[i]$.标签就不会重置为零。在这种情况下，当 Write() 或 SC() 调用返回时，我们可能会遇到 $L[i]$.标签 $= g$和 $R[g] = 0$ 。在 [9],中，这种情况并不会造成问题，因为只有 SC() 操作能够更改 LL/SC 对象。然而，在我们的情形下，一个新的 Write()调用可能会重新锁定标签 g，使系统进入这样一种配置：一个尚未完成的 Write() 操作正准备将 g 写回 $L[i]$.标签，而 $L[i]$.标签 $= g$。这种情况违反了顺序规范，因为 Write(i,·) 应将 L_i 的所有上下文位设为零，从而导致并发的SC(i,·) 操作失败；而写回相同的值—标签对，对其他进程来说却是不可见的。此外，一个正准备覆盖 $L[i]$ 的 Write(i,·) 调用可能会恢复已更改的值—标签对，这又会导致无法检测到的 ABA，对并发的 LL(i) 操作不可见，无论这些操作保护的是哪些标签。

虽然让 Write() 调用只需在锁定 g 后再次检查 $L[i]$.标签 $\neq g$ 看起来很有吸引力，但在最坏情况下，这需要一个无法保证无等待性的无锁循环。（请回想，我们只防止弱自适应对手在随机选择与紧随其后的下一步之间进行干扰，而后一种做法则要求两个共享内存步骤不受干扰，才能实现随机化无等待性。）

出于这些考虑，锁定非零标签 g 后，只要 $L[i]$.标签 $= g$ 仍然如此，我们就会让它保持锁定。这意味着，递减 $R[g]$ 的责任会从执行 Write() 或成功 SC() 操作的进程转移到下一个将$L[i]$.标签改为其他标签的进程。因此，每当在 $L[i]$ 上成功执行 CAS() 操作并将 $L[i]$.标签从 g 改为其他值后，我们都会添加一次 $R[g]$ 的递减。由于 Write() 操作需要无条件修改 $L[i]$ ，并且还要在覆盖 $L[i]$.标签中存储的先前标签之前获知该标签，因此我们使用 Swap() 指令来读取先前的标签。因此， Write()方法在 Swap()之后从 $L[i]$.标签读取到非零 g 时，也会递减 $R[g]$，而不会丧失随机化无等待性。

由于 Write() 和 SC() 现在会递增使用计数，而递减计数的责任转移到了其他进程，因此分析对使用计数的贡献变得更加复杂。将标签保持为 LL/SC 对象锁定的一个副作用是，现在可以保证每个对象在非零时存储一个唯一的标签（即不同的 LL/SC 对象不会共享相同的非零标签）。这一属性并非正确性、复杂度或保持 QHI 所必需，但与 [9]中的 kn 相比，它确实降低了 R 的元素需要存储的最大值，即 $n + 1$。

由于对象现在也计入使用计数，乍看之下，我们可能需要增加 Ω(m) 个标签，因而需要增加 Ω(m) 个 R 的元素，才能使随机标签的选择仍以恒定概率成功。不过请记住，为了保持 QHI，我们会将每个没有待处理的方法调用或未完成的LL() 操作的对象的所有标签字段重置为零。因此，我们可以证明（引理 22），每个 LL/SC 对象 L_i with a

非零标签字段与某个进程相关联，该进程要么在L_i 上有待处理的方法调用，要么有尚未完成的 LL(i)。由于有 n 个进程，且至多有 kn 个尚未完成的LL() 操作，因此额外使用的标签数至多为 $(k+1)n$。总计而言，计入确保至少一半标签未被使用所需的 2 倍因子，需要 $4(k+1)n$ 个非零标签，仅比 [9]中所需的 $2kn$ 个非零标签多一个常数因子。

此观察结果仅适用于使用保持 QHI 版本的算法：与 [9], 一样，我们可以通过放弃该要求来简化实现，但所需标签的数量将变为 $2((k+1)n+m)$。相反，如果 Write(i,·) 仅在调用进程少于 k 个未完成 LL() 操作时才被调用，那么这一界限会略微降低至 $(4k+2)n$。无论采用哪种实现变体，都总共使用 $\Theta(kn+m)$ 个基础对象。

最后，为了保持 QHI，当活动计数器为零时， Write() 方法必须将标签字段改为零，其使用的机制与 [9]在其 CL() 方法中采用的机制相同。不过，由于 Write() 方法不涉及任何未完成的 LL() 操作，因此不需要像该论文的 CL() 方法那样，执行清除与未完成 LL() 操作相关信息的记账工作。由于 Write() 方法也不防范 ABA 问题，因此无需自行递增（或递减）活动计数器：它只需检查计数器是否为零，如果是，则尝试将 LL/SC 对象的标签字段改为零。因此，我们将 CL()拆分为一个较短的 CL() 方法和一个 cleanup()方法：前者负责清除未完成的 LL() 操作，后者则尝试在进入静止状态前将标签字段重置为零。

3.2 低级描述

本节将介绍并解释我们实现所使用的伪代码，包括其中与 [9] 相比未发生变化的部分。对于细节相同之处，我们将采用类似的方式进行说明。

我们将 LL/SC 对象的值存储在由 L 个 CAS 对象组成的数组中。此外，每个 CAS 对象还存储一个初始化为零的非负有界标签。数组 R 用于统计每个标签的使用情况，因此 $R[g]$ 统计的是以下两者之和：已发起 LL() 操作且读取了标签 g 的进程数，以及标签字段中包含 g 的对象数。使用计数的目的是防止 ABA 问题。数组 C 是 LL/SC 对象的活动计数器，因此 $C[i]$ 跟踪已发起 LL(i) 的进程数；其目的是检测静止状态，从而帮助保持静止历史独立性。

在任何 LL/SC 实现中，进程都需要某种方式来跟踪其未完成的 LL() 操作，以便确定其 SC() 和 VL() 调用是否应当成功。在这里，进程使用本地字典 d，其中针对每个 LL/SC 对象 L_i 存储一个二元组，该二元组由从 $L[i]$ 读取的最后一个值和标签组成；如果进程没有未完成的 LL(i)，则存储空值 $\perp$。（ SC(i,·)、 VL(i) 和 CL(i) 方法在$d[i]=\perp$时都会立即返回，因为它们仅对具有未完成 LL(i) 的进程有意义。）

相对于 [9] 的更改或新增内容以蓝色标示。我们的实现使用$4(k+1)n$ 标签，而不是像 $2kn$ 中那样使用 [9], 标签，或者略多于两倍数量的标签。另一方面，在此版本中，标签从不在不同的 LL/SC 对象之间共享， R 中的元素只需存储不超过 $n+1$ 的值，而不是 kn。除了相对于 [9],增加一个标签字段重置功能 CL() 之外，我们还将其移入一个名为 cleanup() 的新辅助方法中。由于 cleanup() 仅是一个辅助方法，因此不会被直接调用，也就不会在顺序规范中提及。

常量：

- U is the set of values that LL/SC objects can have
- m is the number of LL/SC objects
- n is the number of processes
- k is the maximum number of outstanding LL() operations per process

共享数据：

- $L[i]$ for $i \in \{1, \ldots, m\}$ is a CAS/Swap object storing a pair in$U \times \{0, 1, \ldots,$
- $4(k+1)n\}C[i]$ for $i \in \{1, \ldots, m\}$ is a FAID object storing a number in$\{0, \ldots,$
- $n\}R[g]$ for $g \in \{1, \ldots, 4(k+1)n\}$ is a FAID/CAS object storing a number in
- $\{0, \ldots, n+1\}$

进程本地静态数据：

- d is a dictionary storing keys in $\{1, \ldots, m\}$ and values in $U \times \{0, 1, \ldots, 4(k+1)n\}$

初始化：

- $L[i] \leftarrow (u_i, 0)$ where u_i is an application-dependent initial value from U
- $C[i] \leftarrow 0 R[g] \leftarrow 0$
- d starts empty for each process (all indexing returns $\perp$)

3.2.1 LL() 方法

LL() 方法与 [9]中的方法相同。

由于进程最多只能有一个尚未完成的 LL(i)，第 1 行确保通过内部的 CL(i) 调用清除任何之前尚未完成的 LL(i) （ Write() 的第 15 行也有类似作用）。随后，调用进程 p 在第 2 行递增 $C[i]$ ，以声明某个进程已创建一个尚未完成的 LL(i)，并告知其他进程不要将 $L[i]$.标签重置为零。接下来， p 读取存储在 $L[i]$ 中的值-标签对（第 3 行），并将其存入本地字典d （第 4 行），供之后的 SC(i,·) 或 VL(i) 调用使用。如果标签是默认标签零，则调用可以立即返回（第 5 行），因为 SC() 和 Write() 方法不会选择标签零作为新标签，而非零的 $C[i]$ 可防止标签为零时发生 ABA 问题。否则，p 会读取一个非零标签 g 并递增相应的使用计数 $R[g]$ （第 6 行），然后再次读取 $L[i]$以确认其未发生变化（第 7 行）。

如果第二次读取时 $L[i]$ 没有发生变化，那么我们就称 p 已成功保护该标签，使其不会被重用（正式定义见定义 15），而且 p 会在第 8 行返回，并在线性化于第 7 行。此保护会持续到 p 下一次调用 CL(i)，因为第 35 行会再次递减使用计数。正如我们稍后在 SC() 方法中所讨论的，该算法保证只有在没有进程保护某个标签时才会重用它。

否则， $L[i]$ 在 到 p 于第 7 行读取它时，已经变成了另一个值。如果 $L[i]$.标签已变为零，那么 p 在撤销对 $R[g]$ 的递增（第 10 行）并更新其字典（第 11 行）后即可返回；这同样是因为在 $C[i]$中，非零活动计数保护着标签零。否则，我们利用这样一个事实：如果某个进程的 LL(i) 与 SC(i,·)之间出现过 Write(i,·) 或 SC(i,·)，则 SC(i,·) 必定失败。因此，为满足顺序规范，我们追溯性地将 p的 LL(i) 调用的线性化点（它决定了此次执行的线性化顺序）设为第 3 行的首次读取。（尽管这是 LL/SC 实现中的标准技巧，但这个追溯设置的线性化点使得该 LL() 实现无法达到强线性化 [16]。）第 13 行的 CL(i)调用将 $d[i]$ 设为 $\perp$ （第 36 行），仿佛 p 没有尚未完成的 LL(i)，这迫使 p的

```
LL(i ∈ {1, ....., m})
1  如果 d[i] ≠ ⊥ 则 CL(i)
2  C[i].FAI()
3  (val, tag) ← L[i].Read()
4  d[i] ← (val, tag)
5  如果 tag = 0 则返回 val
6  R[tag].FAI()
7  (val′, tag′) ← L[i].Read()
8  如果 (val, tag) = (val′, tag′) 则返回 val′
9  如果 tag′ = 0 则
10 |  R[tag].FAD()
11 |  d[i] ← (val′, 0)
12 |  返回 val′
13 CL(i)
14 返回 val
```

```
Write(i ∈ {1, . . . , m} u ∈ U)15 如果 d[i] ≠ ⊥ ，则 CL(i)1717 重复
18    tag′ ← 从 {1,. . ., 4(k + 1)n} 中抽取独立均匀随机样本 19 直到
  R[tag′].CAS(0, 1) = 为真 20 (· tag) ← L[i].Swap((u, tag′))21 如果 tag ≠ 0 ，
则 |R[tag].FAD()22 如果 C[i].Read() = 0 ，则 cleanup(i)
```

随后 SC(*i*,·) 在第 23 行失败。（在这种情况下，根据定义， *p* 仍有一个尚未完成的 LL(*i*)，因为使用我们的加载链接/条件存储实现的算法尚不能直接观察到下一个 SC(*i*,·) 将会失败。）

3.2.2 Write() 方法

在一次 Write(*i*) 调用的第 17–19 行中，调用进程 *p* 会反复探测可能标签范围内的随机标签。具体而言，第 18 行从 $4(k+1)n$ 个可能标签的定义域中，独立且均匀地选取一个随机标签值。调用进程尝试将所选标签的使用计数从零递增到一；如果 CAS()失败，就重复进行随机选择，直到最终成功递增某个使用计数。共有 $4(k+1)n$个标签；正如第 3.1 节末尾所论证并由引理 22 所证明的，在任意时刻，使用计数非零的标签至多有 $2(k+1)n$ 个。由于弱自适应对手无法在第 18 行的随机选择与第 19 行的 CAS() 尝试之间介入，因此每次 CAS() 尝试成功的概率至少为 $1/2$。因此，循环迭代次数受参数为$1/2$的几何分布随机变量所支配，其期望至多为 2。

通过对某个非零标签 *g* 执行成功的 $R[g]$.CAS$(0,1)$ *p* 锁定 *g*，使得其他任何Write() 或 SC() 调用都无法使用它，并且只要在 $L[i]$.标签中使用， *g* 就会保持已锁定状态。随后，进程 *p* 在第 20 行调用 Swap()，对 $L[i]$ 写入新值以及已锁定的标签 *g*，这就是该方法的线性化点。就在这一时刻之后，

```
SC(i ∈ {1, . . . , m} u ∈ U)23 如果 d[i] = ⊥ ，则返回 假 2525 重复
26    tag′ ← 从 {1,. . ., 4(k + 1)n} 中抽取独立均匀随机样本 27 直到
  R[tag′].CAS(0, 1) = 真28 result ← L[i].CAS(d[i],(u, tag′))29 如果 result =真且
 d[i]标签 ≠ 0 ，则 R[d[i].标签].FAD()30 如果 result = 假，则
 result ← L[i].CAS((d[i].值, 0), (u, tag′))31 如果 result =假，则 R[tag′].FAD()
32 CL(i)33 返回 result
```

我们称标签 g 已被 L_i锁定，因为 $L[i]$.标签 $= g$。由于 p 已锁定其非零标签g，且 $L[i]$.标签之前的值要么先前已被 L_i 锁定，要么为零，因此可以保证此次 Swap() 操作会将 $L[i]$.标签更改为不同的值。反过来，这又保证其他进程执行的任何未完成 LL(i) 都能检测到该 Write() 已线性化，从而防止未被检测到的 ABA 问题。由于从 Swap()返回的前一个标签不再用于 $L[i]$.标签，因此 p 随后会在第 21 行递减其使用计数，使其在使用计数降至零后最终能够被解锁 并再次使用。（如果其他进程存在已看到标签 g的未完成LL(i) 操作，那么它们会增加使用计数 $R[g]$，因此只有在它们各自调用 CL(i)后，该标签才会被完全解锁。）

最后， p 会检查活动计数器 $C[i]$ 是否为零；如果是，则可以安全地通过调用 cleanup(i)（第 22 行）将标签重置为零，以保留 QHI。请注意， Write(i) 不会递增（或递减） $C[i]$，因为与 SC() 不同，它会无条件地更改 $L[i]$ ，且自身操作无需防止 ABA。

由于此方法在线性化于一次 Swap() 调用时，因此可以将其转换为 Swap()实现：返回第 20 行读取（并忽略）的 $L[i]$.val 先前值。

3.2.3 SC() 方法

第 23 行确保当调用进程没有任何未完成的 LL() 时， SC() 调用会返回假（ CL() 的第 34 行和 VL() 的第 47 行也有类似作用）。第 25–27 行的作用与第 17–19 行相同：将未受保护的标签 锁定 ，供调用进程独占使用。回想一下， $d[i]$ 会存储先前一次 LL(i) 操作中从 $L[i]$ 读取的值-标签对。因此，第 28 行会尝试将存储在 $d[i]$ 中的先前值-标签对，条件存储为新值（ SC()的第二个参数）和新选定的标签。与第 21 行类似，第 29 行会在第 28 行的 CAS() 成功后，解锁先前存储在 $L[i]$ 中的非零标签，使其最终可以被回收。

尽管非零的活动计数器（$C[i]$）会阻止 CL() 调用将 $L[i]$.标签重置为零（如下文所述），但仍存在这样一种边界情况：某个 CL() 调用正准备清除 $L[i]$.标签，而在此之前的某个时刻， $C[i]$ 已递增。因此，第 30 行会从值为零的标签中再次尝试 CAS() ，以防 $L[i]$.标签在前一个 LL(i) 操作的线性化点与第 28 行中的 CAS() 尝试之间被重置为零。如果 CAS() 两次尝试均失败，则第 31 行会解锁新选定的标签，因为该标签并未被成功使用。（在这种情况下，使用计数会立即变为零，因为其他进程都不会看到该标签——第 31 行可以替换为值为零的 Write()，但在代码中这样做不太直观。）

该方法在线程 28 或 30 中最后一次尝试 CAS() 时线性化

```
CL(i ∈ {1、.....、 m})
34 如果 d[i] = ⊥ 则返回
35 如果 d[i].标签 ≠ 0 则 R[d[i].标签].FAD()
36 d[i] ← ⊥
37 如果 C[i].FAD() = 1 则 cleanup(i)
```

```
cleanup(i ∈ {1, . . . , m})
38 (val, tag) ← L[i].Read()
39 如果 tag = 0，则返回
40 R[tag].FAI()
41 如果 L[i].Read() = (val, tag)，则
42     如果 C[i].Read() = 0，则
43         如果 L[i].CAS((val, tag), (val,0)) = 真 则
44             R[tag].FAD()
45 R[tag].FAD()
```

随后在第 32 行调用 CL()，清除调用进程尚未完成的 LL()，并返回其最后一次 CAS() 尝试是否成功。

3.2.4 CL() 和 cleanup() 方法

确认 CL(i) 调用并非无操作后（第 34 行），该调用会通过递减非零标签的使用计数（第 35 行），结束调用进程p对该标签的保护，并清除p中 L_i 的字典条目（第 36 行），从而结束 p 尚未完成的 LL(i)。接下来，p 会递减 L_i 的活动计数器（第 37 行），表明其尚未完成的 LL(i) 已被清除。如果计数器从一递减到零，p 就有责任重置 $L[i]$.标签以确保 QHI，并通过 cleanup() 方法完成重置。

方法 cleanup(i) 会在两种情况下被调用：如前所述，当 $C[i]$ 递减到零时，由 CL(i) 调用；或者当从 $C[i]$读取到零时，由 Write(i) 调用。在这两种情况下，调用进程 p 都有责任在调用 cleanup(i)之前的某个时刻将 $L[i]$.标签重置为零，除非 $L[i]$ 再次发生变化。

第 38、40 和 41 行的工作方式与 LL() 中的第 3、6 和 7 行相同，尝试保护存储在 $L[i]$.中的当前标签 g，防止 p 在发生 ABA 后重置 $L[i]$.标签。如果 g为零，则 p 可以立即返回（第 39 行）。如果 $L[i]$.标签从 g发生变化，则 CL(i) 不再需要重置 $L[i]$.标签，因此只需递减其使用计数，结束对标签 g 的保护（第 45 行）。否则，p 接下来会再次检查活动计数器 $C[i]$ 是否仍为零（第 42 行）。对 $C[i]$ 的第二次检查确保任何更改 $L[i]$.标签的进程都是在 p执行第 42 行之后才进行更改的，而此时 g 仍受到保护。第 43 和 44 行的工作方式与 SC()中的第 28 和 29 行类似：p 尝试将 CAS() $L[i]$.中的 g 标签重置为零；如果成功，p 随后会递减标签 g的使用计数，因为 g 不再使用 $L[i]$.中的标签。（请注意，在成功重置标签的情况下，p 会在第 44 和 45 行递减 $R[g]$ 两次：第一次递减与第 19 或第 27 行中的初始锁相关联的L_i使用计数，第二次则结束 p对该标签的保护。）

Read($i \in \{1, \ldots, m\}$)

46 **返回** $L[i]$.Read().值

VL($i \in \{1, \ldots, m\}$)

47 **如果** $d[i] = \perp$，**则返回** 假
48 **如果** $L[i]$.Read().标签 $\in \{d[i]$.标签,0$\}$ **则返回** True
49 CL(i)
50**返回** 假

3.2.5 Read() 和 VL() 方法

Read() 和 VL() 方法与 [9]中的相同。

一次 Read(i) 调用只需返回 $L[i]$.val 的内容，因为它无需对标签字段执行任何操作。

相反， VL(i) 调用只在第 48 行检查 $L[i]$.tag，并在线性化的读取时刻完成线性化。如果进程 p 在存在未完成的 LL(i) 时调用 VL(i) （这是预期情况，否则它会在第 47 行返回），那么 p 此前已经保护了它读取的标签，并将其存储在 $d[i]$.tag 中。这种保护可防止在 p 存在未完成的LL(i)时出现 ABA 问题，因此，如果 $L[i]$.tag $= d[i]$.tag，那么还可以推知 $L[i]$ 在 LL(i) 和 VL(i)的线性化点之间没有发生变化。 VL(i) 还会检查 $L[i]$.tag $= 0$，原因是第 3.2.3 节中讨论的关于 SC()的同一边界情况。如果 $L[i]$.tag 是不同的非零值，那么 p 与 L_i 之间没有开放链接，因此 p 会在第 49 行调用 CL(i)，以清除其尚未完成的 LL(i)。

4 分析

本节将正式证明定理 1。若干证明在形式上与 [10], 的证明十分相似，但由于添加了 Write() 操作，并对 SC() 和 CL()作出了必要修改，因此整体上有显著变化。

▶ **引理 3.** 当进程从 CL(i)、 Write($i, \cdot$) 或 SC($i, \cdot$) 调用返回时，其值 e $d[i]$为 $\perp$。

此引理对应于 [10], 的引理 1，其证明基本没有变化，只是也适用于 Write($i, \cdot$)。

证明。 对于 CL(i)，如果调用进程 p 在第 34 行返回，则该行的 if 条件可满足此引理。否则， p 将 $d[i]$ 设为 $\perp$，并且在其响应之前不再改变它（包括在 cleanup(i)被调用时）。

对于 Write($i, \cdot$)，如果在调用时满足 $d[i] \neq \perp$，那么调用进程 p 会在第 15 行调用 CL(i)；根据上述内容，这确保了 $d[i] = \perp$ 通过其响应实现。此次调用的其余部分不会再次改变 $d[i]$，包括可能对 cleanup(i)的调用。

对于 SC($i, \cdot$)，如果调用进程 p 在第 23 行返回，那么该引理由该行中的 if 条件得到满足。否则， p 会在第 32 行调用 CL(i)；根据上述内容，这确保了$d[i] = \perp$ 通过其响应实现，然后立即返回。◀

▶ **引理 4.** 如果进程 $d[i] \neq \perp$ p 在 p 上没有待处理的方法调用，那么 p 有一个尚未完成的 LL(i)。

该引理对应于 [10]，的引理 2，其证明大体上没有变化。

证明。只有 LL(i) 能将 $d[i]$ 设为非-$\perp$ 值。根据引理 3， CL(i)、 Write(i,·)和 SC(i,·) 方法通过各自的响应确保 $d[i] = \perp$。如果一次 VL(i) 调用返回假，那么它要么在第 47 行执行 $d[i] = \perp$，要么在第 49 行调用 CL(i)，因此在这两种情况下，它返回假时 $d[i] = \perp$。因此，如果 $d[i] \neq \perp$ 处于空闲状态时发生这种情况，那么 p 已调用 LL(i)，且之后没有调用 Write(i,·)、 SC(i,·)、 CL(i)，也没有发起返回假值的 VL(i) 调用；因此，根据定义 2 p 有一个尚未完成的 LL(i)。◀

4.1 对计数器的贡献

以下定义对应于 [10],的定义 3，但更为复杂，因为现在加载链接/条件存储对象可以对 R的元素作出贡献，而且这种贡献可以在进程与加载链接/条件存储对象之间转移。直观而言，p 对 $R[g]$ 的贡献代表 p最终递减 $R[g]$值的责任。该责任最初由 p对 $R[g]$ 的递增产生（从一个 FAI() 或一次成功的 $CAS(0, 1)$）。当进程为某个对象 L_i安装一个新标签到 $L[i]$. 标签时，它会将该责任转移给 L_i。随后，当$L[i]$.标签被某个进程替换时，该责任会转移给执行替换的进程，而后者将执行对 $R[g]$的递减。

▶**定义 5** （贡献）。给定时刻 t、进程 p 和元素 $C[i]$， p 对j $C[i]$的值作出贡献，当且仅当 j 等于 p 在 t 之前对$C[i]$执行的 FAI() 操作数减去 FAD() 操作数。

对于每个元素 $R[g]$，进程通过对 $R[g]$执行一次 FAI() 操作或一次成功的 CAS$(0, 1)$ 操作来增加其贡献，并通过对 $R[g]$执行一次 FAD() 操作来减少其对 $R[g]$ 的贡献。此外，对 $R[g]$ 的贡献也可以在进程与 *LL/SC* 对象之间转移，而不改变对 $R[g]$的总贡献或其值，具体如下：

- 假设进程 p 在时刻 t 的第 *20* 行执行一次 $L[i]$.Swap(（· g）) 操作，或在第 *28* 行或第 *30* 行成功执行一次$L[i]$.CAS(·,（· g）) 操作。则在时刻t， p对 $R[g]$的值的贡献递减一，而 L_i对 $R[g]$的值的贡献递增一。假设进程 p 在第 *20* 行的一次 $L[i]$.Swap(·) 操作中读取了（· g），或在时刻 t 的第 *28* 行或第 *43* 行成功执行一次 $L[i]$.CAS(（· g），·) 操作。则在时刻 t， p对 $R[g]$的值的贡献递增一，而 L_i对 $R[g]$的值的贡献递减一。

▶**引理 6.** 当进程 p 调用方法或从方法调用返回时（包括辅助方法 cleanup()），

(a) 若 $d[i] = \perp$，则 p 对 $C[i]$的贡献为零；否则 p 对 $C[i]$的贡献为一；*(b)* 对于 $g \in$ *{1, . . . , 4(*$k + 1$*)*n*}*， p 对 $R[g]$ 的总贡献等于 p 字典 d 中标签为 g 的非 $\perp$ 值的数量。在执行的所有时刻，*(c)* 对于 p，对象 L_i 的数量至多为 k；*(d)* p 对 C的每个单独元素的贡献均为零或一；*(e)* p对 R中各元素的总贡献值属于 *{0, . . . ,* $k + 2$*}*，且对每个单独元素的贡献均非负。

该引理对应于 [10]，的引理 4，但 (e) 是一个每进程上界，且包含零这一较低下界；全局下界将在推论 9 中证明。

证明。请注意，根据定义 5，p 对 R 和 C 的贡献，以及其字典 d，在 p 处于空闲状态时不会改变。因此，如果 (a) 至 (e) 在 p 从方法调用返回并进入空闲状态时成立，那么在 p 下一次调用方法之前的时刻，它们也一直成立。

因此，我们通过对进程 p 的方法调用（包括辅助方法 `cleanup()`）进行归纳法，证明 (a) 和 (b)；同时证明 (c) 至 (e) 在 p 的每次方法调用中的所有时刻均成立。

初始时，p 对 C 和 R 中的元素贡献为零，其字典 d 为空（所有值均为 $\perp$），因此在 p 的首次方法调用之前，(a) 至 (e) 均成立。

假设在某个方法调用开始时，(a) 至 (e) 对 p 均成立：我们将证明 (c) 至 (e) 在整个方法调用期间始终成立，并且 (a) 和 (b) 在调用返回时成立。

如果该方法调用是 `Read`(i)，那么 (a) 至 (e) 均成立，因为 `Read()` 不会影响 p 的贡献或其字典 d。

接下来考虑 `CL`(i) 方法。如果该调用在第 34 行立即返回，那么 p 的贡献和 d 都不会改变，因此 (a) 至 (e) 仍然成立。否则，在调用开始时 $d[i] \neq \perp$，因此根据 (a)，p 在调用开始时对 $C[i]$ 的贡献为一。由于 `cleanup`(i) 方法（若有调用）是最后一步，因此只需证明在第 37 行对 $C[i]$ 执行 `FAD()` 后，(a) 和 (b) 立即成立，并且 (c) 至 (e) 一直成立到该递减操作之后。由于 p 在整个方法调用期间不会在 d 中设置任何非 $\perp$ 值，**(c) 仍然成立**。由于 p 在第 36 行将 $d[i]$ 重置为 $\perp$，并在第 37 行递减 $C[i]$，因此递减操作后，p 对 $C[i]$ 的贡献变为零，这**满足 (a)**。特别是，p 的贡献在整个调用期间始终为零或一，因此**(d) 在整个调用期间均成立**。

如果在调用时 $d[i]$.标签 $= 0$，那么 p 不会在第 35 行对 R 作出任何更改，因此（b）和（e）在整个调用过程中都保持为真。否则，在调用时 $d[i]$.标签非零，因此根据（b），p 在调用时至少为 $R[d[i]$.标签贡献 1 个$]$。在这种情况下，p 在第 35 行递减 $R[d[i]$.标签$]$，从而在第 36 行之后再次满足（b）。特别地，由于根据（e），p在调用时对 R的总贡献为非负数；根据（b），其在调用时对 $R[d[i]$.标签$]$ 的贡献为正数；并且它只会递减 $R[d[i]$.标签$]$，因此（e）在整个方法调用过程中都得到满足。因此，两种情况都**满足（b）和（e）**。

接下来，考虑 `cleanup`(i) 辅助方法。由于它不会修改 d 或 C，**（a）、（c）和（d）仍然得到满足**。如果 p 在第 45 行作出响应，那么它会在第 40 行和第 45 行为其局部变量 tag递增随后递减 $R[tag]$ 的值，这两个操作的效果恰好相互抵消。如果 p 在第 43 行执行成功的 `CAS()`，那么（根据定义 5），它会为其局部变量 tag递增其对 $R[tag]$ 的贡献；随后，当 p 在第 44 行执行其下一步时，再递减其对$R[tag]$ 的贡献。由于该方法不会修改 d，并且对R的任意贡献递增都会随后由同一元素的贡献递减抵消，因此，在 p于第 45 行或第 39 行响应时，**（b）成立**。根据（e）和（c），在调用 `cleanup`(i)时，p对 R 中各元素的总贡献至多为 k。由于 p在 `cleanup`(i)中最多将其贡献递增两次，并且对 R 中各元素的所有贡献递减都发生在同一方法中对该元素的贡献递增之后，因此**（e）仍然得到满足**。

由于 `VL()` 方法不会改变 p对 C 或 R 的贡献，也不会改变其字典 d，除非通过其内部的 `CL()` 调用，因此根据上述针对`CL()`和 `cleanup()`的论证，(a)至(e)对 `VL()` 均成立。

接下来，我们考虑 `Write`(i,·) 方法，以及第 22 行可能发生的 `cleanup`(i) 调用。我们知道，**(a)、(c) 和 (d) 在整个方法调用期间仍为真**，因为该方法不会改变 p'd 字典 d 及其对 C 的贡献。根据归纳假设中的 (b) 和 (c)，在调用时，p 对 R 中元素的贡献至多为 k。接着，p's

对 $R[tag']$ 的贡献在第 19 行成功 CAS() 时递增一，并在第 20 行递减一。如果第 20 行从 $L[i]$.标签读取的值 tag 非零，那么 p对 $R[tag]$ 的贡献在第 20 行递增一，并在第 21 行递减一。在这两种情况下，第 22 行之前 p对 R 中元素的贡献都会恢复为调用时的值，**从而在第 22 行之前满足 (b)**。由于在整个方法调用期间，相较于调用时， p对 R 的总贡献增加不超过一，并且随后同一 R元素上的所有递增都会被抵消，**(e) 在整个方法调用期间仍得到满足** 。

SC(i,·) 方法的论证与 Write(i,·) 的论证类似，但需要考虑更多情况。这里，我们需要考虑直到调用 CL(i) 的第 32 行之前的步骤。由于除 SC() 之外的步骤不会改变 d 中的值，也不会改变 p 对 C 的贡献，因此同样可以得出**(a)、(c) 和 (d) 在这些步骤中始终为真**。

如果 SC(i,·) 调用在第 23 行返回，那么 (b) 和 (e) 在整个调用期间也成立，因为该调用不会改变 d 或 R。否则，在 $d[i] \neq \perp$ 调用时，因此我们考虑直到第 32 行为止的各行。根据归纳假设中的 (b) 和 (c)， p 在 SC(i,·) 调用时对 R 的元素贡献一个不超过 k 的非负整数。因此，当 p 通过第 27 行中一次成功的 CAS$(0,1)$ 跳出 while 循环时，它对 $R[tag']$ 的贡献递增 1，而紧接该行之后，它对 R 的总贡献至多为 $k+1$ 。根据定义 5，该贡献会持续到第 28 行或第 30 行中发生一次成功的 CAS() ，或第 31 行发生一次 FAD() 为止。

如果 p 在第 28 行成功执行了 CAS() ，且 $d[i]$. 的标签为 $\neq 0$，那么 p 对 $R[tag']$ 的贡献会递减，而它对 $R[d[i].$标签 $]$ 的贡献会在该时刻递增，这不会改变 p 对 R 的总贡献。恰恰在这种情况下， p 会在第 29 行执行 $R[d[i].$标签 $]$.FAD()，撤销其对 $R[d[i].$标签 $]$ 的贡献递增所产生的效果。通过检查代码和定义 5 可知， p 对 $R[tag']$ 的贡献恰好会在第 28、30 或 31 行之一递减 1，从而使 p 对 R 的总贡献恢复到至多 k。

由于在该方法中， R 的所有元素递增 p 随后都会针对同一元素递减，因此其在第 32 行之前紧邻时刻的贡献与调用时相同，从而在第 32 行之前**满足 (b)**。特别是， p对 $R[tag']$ 的贡献递增一次，随后递减一次，其中该递减可能与对 $R[d[i].$标签$]$ 贡献的递增同时发生，之后再进行相应的递减。因此，由于 p在调用时对 R的总贡献属于$\{0,\ldots,k\}$，并且根据 (e)， p在调用时对 R 的每个元素都贡献非负值，**(e) 在整个调用过程中仍然满足**。

接下来需要证明，对于由 p 发起的 LL(i) 调用，(a) 到 (e) 均成立。根据上述论证，它们在 CL() 调用后仍然成立，因此当 LL(i) 调用到达第 2 行时，它们也都仍然成立。

由于根据引理 3，第 1 行中的 CL(i) 调用确保了 $d[i] = \perp$，因此根据 (a)，在第 2 行之前的紧邻时刻， p对 $C[i]$ 的贡献为零，故紧接着 FAI() 之后，其贡献为一。具体而言，在整个调用期间的所有时刻， p对 $C[i]$ 的贡献只能为零或一，**始终满足 (d)** 。执行第 4 行后，(a) 再次成立，并一直保持成立，直到 p可能在第 13 行调用 CL(i) ；根据上述关于 CL() 的论证，在该调用返回时 (a) 仍然成立，因此**(a) 在来自的任何可能响应时都成立**。（请注意，第 11 行若执行，只会将 $d[i]$ 的值从一个非-$\perp$ 值改为另一个。）回顾第 2.4 节， p 在调用 LL(i)之前，最多只能有 $k-1$ 个尚未完成的、针对 L_i 之外对象的LL() 操作。因此，根据引理 4，在该调用开始之前，至多有 $k-1$ 个对象 $L_j \neq L_i$ 满足 $d[j] \neq \perp$ 对于 p。由于第 1 行之后$d[i] = \perp$ ，因此执行第 4 行后，至多有 k 个非-$\perp$ 值

在 d中，因此**(c) 在整个调用期间始终成立**。

由于 (b) 在 p 执行第 2 行之前立即成立，并且在该时刻 d 中非 $\perp$ 值的数量小于 k，因此可知，第 6 行不会使 p 对 R 在 k 之前的贡献增加。根据归纳假设以及 CL() 保持 (e) 这一事实，(e) 从调用开始一直成立到第 6 行执行之前（或第 5 行中的某个响应）。由此可知，在第 6 行的 FAI() 之后，p 对 $R[tag]$ 的贡献至少为 1，因此在第 10 行可能发生 FAD() 后仍保持非负。因此，在整个 LL(i) 调用期间，p 对 R 的贡献都属于 $\{0, \ldots, k\}$，并且对每个元素的贡献均为非负，足以在整个调用期间 **满足 (e)**。最后，我们可以验证，在每个相关的响应点以及第 13 行对 CL(i) 的可能调用处，(b) 都成立；依据是，在第 1 行之后立即成立，且当时有 $d[i] = \perp$。在第 5 行中，$d[i].$标签 $= 0$ 和 R 未被 p 改动；在第 8 行和第 13 行中，$d[i].$标签 $\neq 0$ 和 $R[tag] = R[d[i].$标签 $]$ 均递增；而在第 12 行中，$d[i].$标签 $= 0$ 以及 $R[tag]$ 的递增被反转。◀

▶**引理 7**。 C 中每个元素的值等于每个进程贡献之和（即没有元素发生溢出）。

该引理对应于 [10], 中的引理 6 的一部分，其中涉及 R 的部分将在稍后的引理 12(a) 中证明。

证明。 根据定义 5，每个 FAI()对 C 中某个元素的操作，都对应于该元素总贡献的一次递增。同样，每个 FAD() 对 C 中某个元素的操作，都对应于该元素总贡献的一次递减。由于没有其他操作会影响 C中元素的值，因此只需证明每个元素的总贡献始终处于范围 $[0, n]$ 内，从而不会导致该元素的值下溢或溢出。根据引理 6(d)，在任何时刻，进程对 C 的贡献要么为零，要么为一。因此，每个进程对 C中每个元素的贡献最多为一；又由于对象不会对 C中的元素作出贡献，因此 C 中任意给定元素的最大贡献为 n。由此可知，C 中每个元素的总贡献始终处于范围$[0, n]$内，这就证明了该引理。◀

▶ **引理 8.** 在时刻 t, $L[i].$标签 $= g \neq 0$ 当且仅当 L_i 对 $R[g]$ 的值贡献 *1*。对于所有 $g' \notin \{$ $L[i].$标签*,*0*\}*，L_i 在时刻 t对 $R[g']$ 的值贡献 *0*。

这是一个新结果，我们用它来确保 Write() 方法的正确性。

证明。根据定义 5，能够改变 L_i 对 R 中某个元素的值所作贡献的，只有第 20、28、30 和 43 行。这些也恰好是能够改变 $L[i]$ 值的唯一几行。根据定义 5 并检查代码可知，对非零标签 g，对象增加其对 $R[g]$ 的贡献的情形，与 $L[i].$标签变为非零标签 g 的情形相同：这两种情况恰好都发生在第 20 行执行 $L[i]$.Swap($(\cdot\ g)$) 操作，或第 28 或 30 行成功执行 $L[i]$.CAS$(\cdot, (\cdot\ g))$ 操作时。类似地，根据定义 5 并检查代码可知，对非零标签 g，对象减少其对 $R[g]$ 的贡献的情形，与 $L[i].$标签从非零标签 g 变更的情形相同：这两种情况恰好都发生在第 20 行执行返回 $(\cdot\ g)$ 的 $L[i]$.Swap$(\cdot)$ 操作，或第 28 或 43 行成功执行 $L[i]$.CAS$((\cdot\ g), \cdot)$ 操作时。上述情形目前还并非互斥：如果对 $L[i]$ 执行一次 Swap() 或成功的 CAS()，使 $L[i].$标签从 g “变为” g，那么根据定义 5 L_i 对标签 g 的贡献会同时递增和递减 1，因此不会发生变化。（我们稍后将在引理 14 中证明这种情况不可能发生，但该结果依赖于本引理。）

初始时，对于所有 i， $L[i]$. 标签均为 $=0$，且没有对象对 R 中的任何元素作出贡献。根据上述观察可知，只有当 $L[i]$. 标签从不同的标签变为 g 时， L_i 才会增加其对 $R[g]$ 值的贡献。类似地，只有当 $L[i]$. 标签从 g 变为不同的标签时， L_i 才会减少其对 $R[g]$ 值的贡献。因此，通过归纳法即可得出引理的第一部分。由于 $L[i]$. 标签只有一个值， L_i 因而一次最多只能对 R 中的一个元素作出贡献，即满足 $L[i]$. 标签为 $=g\neq 0$ 的元素 $R[g]$，由此证明第二部分。◀

▶**推论 9.** 在执行的所有时刻，对任意给定元素C 或 R 的总贡献均为非负；如果任何进程或对象对该元素作出了正值贡献，则总贡献为正。

这一陈述与 [10], 的引理 5 有些类似，但证明方式不同。

证明。 由于只有进程会对 C作出贡献，且根据引理 6(d)，每个进程对每个元素的贡献不是零就是一，因此对 C 的总贡献非负；若任何进程贡献了一，则总贡献为正。关于 R 的陈述则由引理 6(e) 和引理 8 共同推出。◀

4.2 锁定标签

这一可写 LL/SC 实现中的一个重要新概念是锁定标签。 `SC()` 和 `Write()` 方法会尝试锁定下文定义的标签，以确保它们不会选择与并发的 `SC()` 或 `Write()` 调用相同的新标签；正如第 3.1 节所述，这可能导致 ABA 问题。

▶ **定义 10** （锁定标签）。当进程 p 在 标签 $g\in\{1,\ldots,4(k+1)n\}$ 上执行 L_i 时，如果 p 在第 *19* 行或第 *27* 行成功执行了 $\texttt{CAS}(0,1)$，作为 $\texttt{SC}(i,\cdot)$ 或 $\texttt{Write}(i,\cdot)$调用的一部分，则该进程会锁定该标签。当根据定义 *5*，该标签的总贡献变为零时，该标签便会解除锁定。

▶ **引理 11**。当 $L[i]$.标签 $=g\neq 0$时，标签 g 对 L_i而言处于锁定状态。当 `Swap()` 或一次成功的`CAS()` 下一步对 $L[i]$生效时，标签 g 仍保持锁定状态，直到执行该`Swap()` 或 `CAS()` 的进程执行下一步。

证明。初始时，所有标签字段均设为零。因此，若在时刻 t， $L[i]$.标签 $=g\neq 0$，则存在一个时刻 $t'<t$，某个进程 p在第 20 行执行了 `Swap()`，或在第 28 行或第 30 行成功执行了 `CAS()`，从而将 $L[i]$.标签改为 g。令 t' 为此类时刻中最晚的时刻，则在整个区间 $(t',t]$ 内，$L[i]$.标签 $=g$。存在一个时刻 $t_L<t'$， p 在 $R[g]$上于第 19 行或第 27 行成功执行了 `CAS()`，使其从零变为一；根据定义 5 和引理 6(e)，这使 p对 $R[g]$ 的贡献从零增加到一。根据定义 10，p 在时刻t_L锁定标签 g。根据定义 5， p对 $R[g]$ 的贡献在 t_L 到 t'之间保持不变， p的贡献在时刻 t'转移给 L_i，而 L_i的贡献在 t' 到 t之间保持不变。根据推论 9，在整个区间 $(t_L,t]$ 内，对 $R[g]$ 的贡献非负。因此，根据定义 10，标签 g 在整个区间 $(t_L,t]$ 内保持锁定状态。

现在假设时刻 $t^\star$ 是这样一个时刻：在 t 之后， `Swap()` 在第 20 行生效，或第 28 行或第 43 行中的一次成功 `CAS()` 对 $L[i]$生效；并令 q 为执行该步骤的进程。在整个过程中，$L[i]$.标签 $=g$ $(t',\ t^\star)$，根据定义 5， L_i的贡献在时刻 $t^\star$ 转移到 q，且该转移后的贡献一直保持到 q 执行其下一步（即第 21、29 或 44 行）时的时刻 $t_U>t^\star$。根据推论 9，在整个过程中（t_L， t_U）， $R[g]$ 都具有正贡献。引理的最后一部分随即由定义 10 得出。◀

20 保持历史无关性的 LL/SC 可写

►**引理 12**。 在执行过程中的每个时刻，
(a) 中每个元素的值 R 等于每个进程和 *LL/SC* 对象所作贡献的总和；**(b)** 当一个标签已锁定时，没有进程可以再次锁定它；以及**(c)** 每个进程对 R中的每个元素至多贡献一次，除非在第 *20*、*28* 或 *43* 行中，贡献从对象转移给了进程。在后一种情况下，在该进程下一次分别执行第 *21*、*29* 或 *44* 行之前，它对该元素至多可以贡献两次。陈述 (a) 对应于 [10], 中引理 6 的一部分，其中涉及 C 的部分已在引理 7 中证明。

证明。 我们通过对执行进行归纳，将这三个部分作为一个整体不变式来证明。它们在执行开始时都成立，因为最初没有进程或对象对R 作出贡献，也没有标签被锁定。

我们首先证明 (b)：证明如果标签 g 在时刻 t变为已锁定，并且 (a) 在 t之前的所有时刻都成立，那么紧接在 t之前， g 未被锁定。根据定义 10，在标签 g 被锁定的整个区间内，对 $R[g]$ 的总贡献非零。由于 t 是对 $R[g]$执行成功的 $\texttt{CAS}(0,1)$ 的时刻，因此在时刻 t之前紧接着， $R[g]=0$ 。根据 (a)，紧接在 t之前，没有进程或对象对 $R[g]$ 作出贡献，因此紧接在 t之前， g 未被锁定。

接下来我们证明，在假设(a)、(b)和(c)对于时刻 t 之前的所有时刻均成立的情况下，(c)在时刻 t成立。假设进程 p 在时刻 t将其贡献递增至 $R[g]$ （否则，根据归纳假设，(c)立即成立）。如果这是从某个对象转移而来的，即发生在第20、28或43行，那么根据关于(c)的归纳假设，在时刻 t之前， p对每个标签的贡献至多为一。因此，在时刻 t之后， p对每个标签的贡献至多为二。在所有转移情形中， p 都会在下一行（分别为第21、29或44行）将转移来的标签递减，由此可得(c)成立。

因此，接下来考虑 p 在第6、19、27或40行中的某一行直接将 $R[g]$ 在时刻 t递增的情形：在所有情形中，我们将证明，在时刻 t之后， p 对$R[g]$ 的贡献恰好为一。对于第19行和第27行，由于它们是 $\texttt{CAS}(0,1)$ 操作，根据归纳假设，(a)在 t之前立即成立；而且，对 R 的总贡献为零意味着每个进程的贡献均为零（根据推论 9），因此(c)成立。

剩下的（也是最复杂的）情况是： p 在时刻 t 执行 $\texttt{LL}(i)$ 调用中的第 6 行，或执行 $\texttt{cleanup}(i)$ 调用中的第 40 行；我们将这两种情况一并考虑。为证明矛盾，假设在时刻 t 之前， p 对 $R[g]$ 的贡献至少为 1。由此可知，存在一个时刻 $t'<t$ ，此时 p 增加了对 $R[g]$的贡献，却没有在（t'， t）中减少对 $R[g]$ 的贡献。

请注意，第 19、20、28 和 40 行中可能发生的贡献递增，在同一次方法调用中必然分别由第 20、21、29 和 45 行的贡献递减跟随。同样，第 27 行的贡献递增之后，必然恰好在第 28、30 或 31 行之一发生贡献递减。在上述所有考虑的情况中， p 都不会执行第 6 行或第 40 行。因此，在时刻 t 之前，唯一未被抵消的贡献递增发生在第 6 行。此外，该调用必须在第 8 行返回；否则， p对 $\texttt{CL}()$ 调用（发生于第 13 行）的贡献会在第 10 行或第 35 行递减。因此，存在一个时刻 $t'<t$ ，此时 p 在 $\texttt{LL}(j)$ 调用的第 6 行递增了 $R[g]$ ，而且直到时刻 t之前， p对 $R[g]$ 的贡献始终保持为 1。

由于 p 的 $\texttt{LL}(j)$ 调用在第 8 行返回，因此存在一个时刻 $t^\star\in(t',t)$，此时 p 在第 7 行从 $L[j]$. 标签中读取 g。根据引理 11，在时刻 $t^\star$， g 对 L_j 处于已锁定状态。

接下来，我们通过矛盾证明 $L_j \neq L_i$：反设 $L_i = L_j$。由于在整个区间 (t', t) 内，p 对 $R[g]$ 的贡献恰好为 1，且每次方法调用只涉及 d 的一个元素，因此引理 6(b) 意味着，在该区间内 p 的方法调用过程中，$d[i].$标签 $= g$。在时刻 t 执行第 6 行的情况下，p 会在第 6 行之前调用第 1 行中的 $\texttt{CL}(j)$，并在第 35 行递减 $R[g]$，这与 p 在整个区间 (t', t) 内不会递减 $R[g]$ 相矛盾。在时刻 t 执行第 40 行的情况下，回想一下，p 会从 $\texttt{Write}(i, \cdot)$ 或 $\texttt{CL}(i)$ 调用 $\texttt{cleanup}(i)$。若调用来自 $\texttt{Write}(i, \cdot)$，则 p 之前会调用第 15 行，并在该行调用 $\texttt{CL}(i)$，因为该调用发生在包含时刻 t' 的 $\texttt{LL}(i)$ 调用之后，因此在方法调用时 $d[i].$标签 $= g$。随后，$\texttt{CL}(i)$ 调用会在第 36 行将 $R[d[i].\text{标签}] = R[g]$ 递减为 $g \neq 0$，这再次与 p 在整个区间 (t', t) 内不会递减 $R[g]$ 相矛盾。

在时刻 $t^\dagger \in (t^\star, t)$，$p$ 从第 3 行的 $L[i].$标签读取 g。再次运用引理 11，可知在时刻 $t^\dagger$，g 对 L_i 已锁定。但由于在整个区间 $(t', t) \supset [t^\star, t^\dagger]$ 内，p 始终为 $R[g]$ 贡献一个单位，根据定义 10，在该区间内，g 对 L_j 从未解锁。因此，在时刻 $t^\dagger$，g 同时对 L_i 和 L_j 已锁定，这与点 t 之前的 (b) 矛盾，并完成了 (c) 的归纳情形。

为结束证明，我们证明：已知 (b) 和 (c) 在时刻 t 成立，则 (a) 在时刻 t 也成立。根据定义 5，R 中某个给定元素的总贡献，仅当该元素递增（包括通过 $\texttt{CAS}(0, 1)$ 递增）或递减时才会增加或减少。（特别是，在进程和对象之间转移贡献，不会改变对 R 中任何元素的总贡献。）由于推论 9 表明贡献非负，要证明 (a)，只需证明在时刻 t，R 中某个给定元素的总贡献不超过 $n + 1$，因此该元素永远不会溢出。

根据引理 11 和 (b)，在时刻 t，所有 LL/SC 对象对给定标签 g 的总贡献至多为一。如果在时刻 t，没有进程在尚未执行其递减 $R[g]$ 的下一步之前，于第 20、28 或 43 行中的任一行收到转移给它的贡献 g，那么根据 (c)，每个进程对 $R[g]$ 的贡献至多为一。因此，在这种情况下，对 $R[g]$ 的总贡献至多为 $n + 1$，这就意味着 (a) 成立。

为了完成 (a) 的证明，我们将说明，在时刻 t，至多有一个进程收到转移给它的标签 g，但尚未执行后续的递减；并且如果存在这样的进程，那么没有对象对 $R[g]$ 作出贡献。假设在时刻 $t' < t$，某个进程 p 执行第 20、28 或 43 行中的任一行，从 L_i 收到转移给它的非零标签 g，并且直到时刻 t 之后才执行下一行。根据引理 11，g 在整个 $[t', t]$ 期间都保持已锁定。

为证明矛盾，假设 L_j（可能是 $L_j = L_i$）在某个时刻 $t^\dagger \in (t', t)$ 对 $R[g]$ 有贡献；根据引理 8，$L[j].$在此时刻的标签为 $= g$。（如果第二个进程在该区间内的某个时刻被转移了标签 g，那么在转移之前的某个时刻，$L[j].$对某个对象 L_j 的标签为 $= g$，因此这也属于此假设的涵盖范围。）于是存在一些时刻 $t_1 < t_2 < t^\dagger$，使得在时刻 t_1，某个进程 q 在第 19 行或第 27 行锁定标签 g；而在时刻 t_2，q 在第 20、28 或 30 行使标签 $L[j].$变为 g。由于在时刻 t' 之前的紧邻时刻，g 对 L_i 已锁定，因此存在一个时刻 $t^\star < t' < t^\dagger$，此时某个进程 r 将 g 对 L_i 锁定。由于根据 $\texttt{Write}()$ 和 $\texttt{SC}()$ 的代码，进程要么对一个对象执行一次 $\texttt{Swap}()$，要么在锁定标签后最多成功执行一次 $\texttt{CAS}()$ 来设置标签字段，因此可知 $t^\star \neq t_1$。在 $t^\star$ 和 t_1 中较晚的那个时刻与 (b) 矛盾，因为 g 在这两个时刻中较早的时刻被锁定，并且会持续处于已锁定状态，直到至少时刻 t；该时刻晚于这两个时刻。

总之，除了一个带有已转移标签的进程外，每个进程对 $R[g]$ 的贡献至多为 1；此外，要么有一个对象再贡献 1，要么有一个带有已转移标签的进程对 $R[g]$ 贡献 2。因此，在所有情况下，对 R 的任一元素的总贡献至多为 $n + 1$，由此可得 (a)，因为这些元素不会溢出。

◀

►**推论 13**。 当标签 g 被进程 p 为 L_i 锁定时，只有 p 能使 $L[i]$.标签 变为 g。

证明。 假设 $L[i]$.标签 变为 g，发生在时刻 t，由进程 q执行，而 g 由进程锁定p。由于（根据定义 10）只有非零标签才能被锁定，因此在时刻 t 可能执行的代码行是第 20、28 和 30 行。每种情况下，都存在一个时刻 $t' < t$，在该时刻 q跳出循环，执行的是第 19 行或第 27 行。根据定义 10，q 在时刻 t'锁定标签 g，并且直到时刻 t之后才解锁 g。根据引理 12(b)，$q = p$，推论得证。◄

►**引理 14**。 如果在时刻 t 某个进程执行第 *20* 行，或者在第 *28* 行或第 *30* 行成功执行 CAS()，那么 $L[i]$.标签 会在时刻 t从某个标签 g 变为一个新标签 $g' \notin \{g,0\}$。

这个引理类似于 [10], 的引理 18，但其证明使用了标签锁定这一新概念。

证明。 设 p 为执行第 20 行的进程，或在时刻 t于第 28 行或第 30 行成功执行 CAS() 的进程，并设 g' 为 $L[i]$.标签 在该时刻之后紧接着的值。由于 p 在第 18 行或第 26 行从集合 $\{1, \ldots, 4(k+1)n\}$中选择 g'，因此 $g' \neq 0$，所以只需证明 $L[i]$.在 $\neq g'$ 之前紧接着的标签。 t

设 $t' < t$ 为 p 在第 19 行或第 27 行跳出循环的时刻。由于 $R[g] = 0$紧接在 t'之前，根据引理 8 和引理 12(a)，可得 $L[i] \neq g$ 在 t'。根据定义 10，g' 在时刻 t'被 p 锁定，并一直保持锁定，直到它将自己的贡献递减至 $R[g]$；根据定义 5，这发生在时刻 t 之后紧接着（将其贡献转移给 $L[i]$）。由于在时刻 t'，$L[i]$.标签 $\neq g'$，根据推论 13，可得$L[i]$.标签 $\neq g'$ 在整个 $[t', t)$ 期间成立，由此证明完毕。◄

4.3 保护标签

► **定义 15**（受保护标签和强受保护标签）。 如果满足以下任一条件，则标签 g 在时刻 t 被进程 p 针对 *LL/SC* 对象 L_i 变为<*strong*>受保护</*strong*>：
（**a**） p 在时刻 t从第 3 行或第 7 行的标签 $L[i]$.读取零，在此情况下 $g = 0$；或（**b**） p 在时刻 t从 g 第 7 行或第 41 行的标签 $L[i]$.读取 t，使得其局部变量对 (val, tag)（分别从第 *3* 行或第 *38* 行的标签 $L[i]$.读取）等于时刻 t的值 。
如果标签在一次 LL(i) 调用期间对于 L_i 变为<*strong*>受保护</*strong*>，那么对于L_i 而言，它也会被 p <*strong*>强保护</*strong*>。在这种情况下，进程 p 提供的保护和强保护将在调用进程下一次调用 CL(i)时结束。否则，标签会在一次 cleanup(i)调用期间对于 L_i 变为<*strong*>受保护</*strong*>，此时它不会被进程 p<*strong*>强保护</*strong*>，并且 p 提供的保护会在该调用响应时结束。

如果某个标签在时刻 t 被任意进程为 L_i保护或强保护，我们也会简称其为在 L_i中受保护或强保护，而不提及执行保护的进程。

与 [10] 的定义 9 相比，唯一的区别是，在 CL()方法中保护标签的操作移到了 cleanup() 辅助方法中。

► **引理 16**。 如果某个标签在时刻 t对 L_i 受到强保护，那么在时刻 t， $C[i] \neq 0$ 。

该引理对应于 [10], 的引理 10，其证明几乎完全相同。

证明。 根据定义 15，只有当进程 L_i 在 p 中进行一次 LL(i) 第 3 行或第 7 行的调用时，标签才能变为强保护状态，而这两行都发生在 p 递增 $C[i]$ 第 2 行之后，且在其下一次调用 CL(i)之前。由于只有 CL(i) 才能递减 $C[i]$（在第 37 行），因此 p对 g 的强保护 L_i 会在递减 $C[i]$之前结束。根据定义 5，

p 在其强保护 g 期间始终对 $C[i]$ 有贡献。由引理 6(d) 和引理 7 即可得出该引理。◀

▶**引理 17**。 如果进程 p 在时刻 t于 L_i 上没有待处理的方法调用，那么当且仅当 $d[i]$.标签 $= g$ 在 p的字典 d 中于时刻 t时， p 才会为 L_i 强保护标签 g 。

这对应于 [10], 的引理 11，且证明基本相同。

证明。我们首先注意到，根据定义 15， p 在没有于 L_i上待处理的方法调用时仍能为 L_i 保护标签的唯一情况，是它将标签保护作为一次 LL(i) 调用的一部分。特别地，这意味着 p 也会强保护该标签。此外，只有 LL(i) 能将$d[i]$ 设置为非-$\perp$ 值（在第 4 行和第 11 行），且只有 CL(i) 能将 $d[i]$ 重置为 $\perp$ （在第 36 行）。根据定义 15， CL(i) 的一次调用会结束 p为 L_i 的任何标签提供的（强）保护；并且根据引理 3， $d[i] = \perp$ 会在 CL(i) 响应时结束。因此，剩下要证明的是： LL(i) 调用当且仅当 $d[i]$.在该调用的响应处的标签为 $= g$ 时，才会在响应时为 L_i 强保护标签 g 。

为证明这一点，我们考虑进程 p 对 LL(i) 调用的五种可能响应点。假设根据归纳法，引理在调用开始前成立（基例成立，因为 $d[i] = \perp$ ，且进程在 $t = 0$ 不保护任何标签）。如果在调用 LL(i) 的调用开始前， p 为 L_i 强保护某个标签 g，那么调用开始前的标签 $d[i]. = g$ ，因此 p 在第 1 行调用 CL(i) 。于是，在第 1 行之后， p 不再为 L_i 保护任何标签，并且 $d[i] = \perp$。

如果 p 在第 5 行或第 12 行响应，那么 p 分别在第 3 行或第 7 行从 $L[i].$ 标签读取到零。根据定义 15(a)， p 为 L_i 强保护标签零，并在响应前分别于第 4 行或第 11 行将 $d[i].$ 标签设为零。

如果 p 在第 8 行响应，那么 p 会在第 7 行从 $L[i]$ 重新读取其局部变量（val， tag），此时它已在第 4 行将 $d[i]$ 设为（val， tag），此后未再更改。根据定义 15(b)， p 在第 7 行强保护 $d[i].$ 标签，此时 $L[i] = d[i]$。

否则， p 会在第 14 行作出响应，此时在第 13 行调用的 CL(i) 调用会将 $d[i]$重置为 $\perp$ ，并结束 p 对 L_i的任何标签保护。◀

▶**引理 18**。 假设在 $L[i]$ 上的时刻 t发生了一次成功的 CAS() ，使得在 t之后，标签 $\neq 0$ 立即处于该状态。在时刻 t，某个进程 p 执行SC(i,·) 调用中的第 *28* 行或第 *30* 行。从其调用开始，进程 p 会为 L_i 强保护某个标签 $g^\star$ ，且至少持续到时刻 t。

这是对 [10] 中引理 12 的一个稍简单版本。请注意，引理的表述假定 CAS() 操作恰好在时刻 t发生，因此排除了在第 20 行发生Swap()的情况。

证明。由于只有第 28 行和第 30 行的 CAS() 操作可以将 $L[i]$.标签更改为非零值，因此时刻 t 就是某个进程 p 执行这两行之一的时刻。由于 p 不会在第 23 行返回，根据引理 17，当 p 调用 SC(i,·)时，它会为 L_i 强保护 $g^\star = d[i]$.标签。此外，由于 p 在调用SC(i,·) 之后、在时刻 t成功执行 CAS() 之前，并未调用 CL(i) ，根据定义， L_i 会持续受到强保护，至少直到时刻 t。◀

4.4 历史独立性

在确立了我们的实现所具有的实用性质后，现在可以证明我们的结果了，首先从保持 QHI 性质开始。该结果将在定理 21 中证明，其表述与 [10], 的定理 15 相同，但证明它需要更多工作。

24 使保持历史无关性的 LL/SC 可写

►**引理 19**。 如果在时刻 t，没有进程在 L_i 上存在待处理的方法调用，并且没有进程强保护 L_i 的标签，那么在时刻 t， $C[i] = 0$ 。如果在时刻 t，没有进程在任何对象上存在待处理的方法调用，并且没有进程强保护任何标签，那么对于所有$g \in \{1、\ldots\ldots、4(k+1)n\}$， $R[g] = 0$ 。

证明。假设在时刻 t，没有进程在 L_i 上有待处理的方法调用，并且没有进程强保护任何用于 L_i 的标签。若某个进程满足 $d[i] \neq \perp$，则对于某个标签 $g \in$，有标签 $= g$，其中 $g \in \in \{0, \ldots, 4(k+1)n\}$。根据引理 17， p 会在时刻 t 强保护 g：因此，在时刻 t，所有进程都满足 $d[i] = \perp$。于是，引理 6(a) 和引理 7 推出，在时刻 t 有 $C[i] = 0$。如果在时刻 t，没有进程有任何待处理的方法调用或强保护任何标签，那么引理 6(b) 和引理 12(a) 推出，对于所有 g，都有 $R[g] = 0$。◄

►**引理 20**。 如果在时刻 t ，没有进程在 L_i 上有待处理的方法调用（可能存在第 *44* 行或第 *45* 行的待处理下一步除外），并且没有进程强保护用于 L_i 的标签，那么$L[i]$.标签 $= 0$ 在时刻 t。

该引理对应于 $[10]$, 的引理 14，但需要考虑更多情况。

证明。 对于 $t = 0$，该引理为 $L[i]$.真，因为对于每个 $i \in \{1, \ldots\ldots, m\}$， $L[i]$.标签都初始化为零。该 `Read`(i) 方法不可能处于待处理状态，也不会影响调用进程保护哪些标签或 $L[i]$.标签的值，因此本引理无需考虑该方法。类似地，根据引理 17，响应于第 23、34 或 47 行的方法调用不会影响调用进程强保护哪些标签或 $L[i]$.标签的值。在对象 $L_j \neq L_i$ 上的方法调用同样不会影响调用进程是否强保护某个标签L_i 或 $L[i]$.标签的值。注意，到目前为止提到的所有方法调用，要么不是针对 L_i的，要么在返回前至多包含一个共享内存步骤。因此，它们都不会改变进程是否有针对 L_i的待处理方法调用。由此，不失一般性，我们可以假设时刻 t 要么是本段未涵盖的方法的响应时刻，要么紧接在执行第 43 行之后。

通过检查代码以及根据定义 15 和引理 17，调用进程在响应时刻响应而未强保护标签 L_i 的其余方法调用恰好是

(a) 一个 `LL`(i) 在第 14 行响应的调用，**(b)** 一个 `SC`(i,·) 在第 33 行响应的调用，**(c)** 一个 `CL`(i) 在第 34 行之后响应的调用，**(d)** 一个 `VL`(i) 在第 50 行响应的调用，以及**(e)** 一个 `Write`(i,·) 调用。

在第 14、33 或 50 行响应的情况都紧接在一次 `CL`(i) 调用之后。因此，我们还可以假设，时刻 t 要么是响应时刻，要么是对来自 `CL`(i) 或 `Write`(i,·) 调用 c 的第 43 行的执行。

根据引理 19，若 $C[i] = 0$ 在时刻 t 是响应点，则 $C[i] = 0$ 在该时刻的值为 $C[i]$。由于第 44 行和第 45 行不会影响 $C[i]$，并且在所有情况下，从第 45 行到调用进程变为空闲之前都没有共享内存操作，因此还可得出：如果 t 执行了第 43 行，则 $C[i] = 0$。由于调用 c 未在第 34 行响应，它要么在第 37 行递减 $C[i]$，要么是一个 `Write()` 调用。因此，在 t 之前存在一个时刻，此时要么 $C[i]$ 被递减至零，要么某个 `Write`(i,·) 调用执行第 20 行。（从代码可知，这两种情况都不会恰好发生在时刻 t。）令 t' 为时刻 t 之前最后一个这样的时刻。由于只有第 37 行会递减 $C[i]$，因此要么某个 `CL`(i) 调用 c' （可能是 $c' = c$）在时刻 t' 将 $C[i]$ 递减至零，要么某个 `Write`(i,·) 调用 c' 在时刻 t' 执行第 20 行。

令 p 为执行调用 c' 的进程。根据 t' 的定义，我们还可得知，在区间 $(t', t]$ 内，没有 `Write`$(i,\cdot)$ 调用执行第 20 行，并且在整个该区间内 $C[i] = 0$。如果 c' 是一个 `CL`(i) 调用，那么 p 会在第 37 行从 `FAD()` 读到 1，随后调用 `cleanup`(i)。否则，c' 是一个 `Write`$(i,\cdot)$ 调用，因此它在区间 $(t', t]$ 内执行第 22 行时读取 $C[i] = 0$，所以也会调用 `cleanup`(i)。由于在时刻 t，除了可能涉及第 44 行或第 45 行之外，没有任何进程在 L_i 上有待处理的方法调用，因此到时刻 t，p 的调用已经执行第 43 行或已作出响应。

由于 $C[i] = 0$ 在整个区间 $(t', t]$ 内，根据引理 16，整个区间 $(t', t]$ 内都没有标签受到强保护。因此，根据引理 18，在 $(t', t]$ 内，$L[i]$ 上不会发生成功的 `CAS()`，从而将 $L[i]$.标签设为非零值。由于唯一其他可能将 $L[i]$.标签更改为非零值的行是第 20 行，而该行在 $(t', t]$ 内不会执行，因此 $L[i]$.标签在该区间内不会变为非零值。具体而言，如果 $L[i]$.标签在该区间内发生变化，那么它只会因第 43 行中一次成功的 `CAS()`而变化；该操作会将 $L[i]$.标签改为零，而不改变 $L[i]$.val。因此，只需证明在 p 的 `cleanup`(i) 调用期间的某个时刻，$L[i]$.标签 $= 0$。

如果 p 从其在第 39 行的 `cleanup`(i) 调用返回，或者第 41 行的 if 条件为假，那么 $L[i]$.标签会在 $(t', t]$ 中的某个时刻发生变化，因此 $L[i]$.标签会在该时刻变为零，从而证明该引理。否则，我们知道 p 在第 42 行从 $C[i]$ 读取到零，因此执行第 43 行的 `CAS()`。如果 `CAS()` 成功，那么 $L[i]$.标签会在其之后立即 $= 0$，因此根据时刻 t，引理得证。否则，$L[i]$ 会在 t' 之后且在 `CAS()` 尝试之前的某个时刻发生变化，因此我们再次知道 $L[i]$.标签在该时刻 $= 0$。◀

我们的第一个主要定理表明，在稳定状态（定义见第 1 节）下，除了存储在 val 字段中的加载链接/条件存储值之外，所有共享内存都被设置为默认值零，这意味着我们的实现保持静止历史独立性。

▶**定理 21**。 如果在时刻 t，每个进程都处于空闲状态，且在任何 *LL/SC* 对象上都没有尚未完成的 `LL()` 操作，那么在时刻 t，C、R 的所有值以及 L 的所有标签 字段都为零。

此定理对应于 [10]的定理 15。

证明。 根据引理 4，在时刻 t，每个进程的 d 的所有值都为 $\perp$。根据引理 17，在时刻 t，没有进程强保护任何标签。因此，由引理 19 和引理 20 可得此定理。◀

4.5 期望步复杂度

证明期望步复杂度的核心思路与 [10], 中的相同，但与那篇论文相比，主要的额外难点在于跟踪第 4.1 节和引理 22 中的贡献。

▶**引理 22**。 在执行过程的所有时刻，R 所有元素的总贡献至多为 $2(k+1)n$。

证明。 根据引理 6(e)，每个进程在任意给定时刻对R 中的元素总共贡献至多 $k+2$。

根据引理 8 和 20，对于每个向 R 中某个元素贡献 one 的 LL/SC 对象 L_i，存在一个进程，该进程要么

- 在 L_i 上有待处理的方法调用（但不包括第 44 行或第 45 行的待处理下一步），或
- 者为 L_i 强保护某个标签。

对于在 L_i 上没有待处理的方法调用、且该调用会为 L_i 强保护某个标签的进程，根据适用于该进程的引理 17 $d[i] \neq \perp$ 以及引理 4，该进程有一个未完成的 LL(i)。如 2.4 所要求，每个进程在其没有待处理方法调用的对象上，至多有 k 个未完成的 LL() 操作。因此，每个进程至多可与 $k+1$ 个来自加载链接/条件存储对象的 R 贡献相关联。

然而，根据引理 4、定义 5、引理 6(b)以及对代码的检查，进程能够恰好向 R 的元素贡献 $k+2$ 的唯一情形是

- 该进程有 k 个未完成的 LL(i) 操作，并且
- 它的最后一步是在第 43 行成功执行 CAS()，因此其待处理的下一步是第 44 行。

由于该进程在 L_i上有待处理的方法调用时，LL/SC 对象 i 会向 R 的一个元素贡献，这种情形与上述情形互斥，因为此时它的待处理下一步不可能是第 44 行。因此，每个进程最多可关联 $k+2+k+1-1 = 2k+2$ 次对 R的总贡献。由此，对 R 的总贡献至多为 2($k+1$)n。◀

▶ **定理 23** (期望步复杂度)。 *</style}*除了 Write() 和 SC() 之外，每种方法的步数都有一个常数上界。在弱自适应对手下， Write() 和SC() 方法的期望步复杂度为常数。

该定理对应于 [10]中的定理 7。

证明。 第一部分直接源于以下事实：实现中的唯一循环是 Write() 方法的第 17 至 19 行以及 SC() 方法的第 25 至 27 行。

对于第二部分，回顾一下，在弱自适应对手下，进程 p 进行随机选择后，会立即执行其下一次共享内存步骤。因此，当 p 在第 18 行或第 26 行选择随机标签 g 时，随后会立即尝试在 $R[g]$ 上执行 CAS()，分别对应第 19 行或第 27 行，并且在这两行之间不会有其他进程执行步骤。特别地， $R[g]$ 的值在这两行之间不会发生变化。因此， p在第 19 行或第 27 行将 $R[g]$ 从零改为一的 CAS()，当且仅当 $R[g] = 0$ 在 p 执行第 18 行或第 26 行时成立。于是， p 一旦选择了满足以下条件的标签 g，就会跳出循环：当 p 执行第 18 行或第 26 行时， $R[g] = 0$ 成立；在这种情况下，它会在退出循环时将 $R[g]$ 设为一。由于该随机选择在{1，...，4($k+1$)n} 上均匀且相互独立，因此只需证明，在任意给定时刻， R 中至少有固定比例的索引取值为零。

根据引理 22， R的所有元素的总贡献至多为 2($k+1$)n，而根据推论 9，所有贡献均为非负。因此，在任意给定时刻，R的 4($k+1$)n 个元素中至少有 2($k+1$)n 个元素的贡献为零，并且根据引理 12(a)，其值也为零，从而证明得证。◀

4.6 线性化

我们通过在定理 29 中证明该算法是第 2.3 节顺序规范的线性化实现，来结束分析。与 [10] 相比，需要考虑的情况更多，这是由于 Write() 和 SC() 方法之间存在交互；但锁定标签这一新概念使得一些论证更加简单。

▶ **引理 24**。 如果标签 $g \neq 0$ 在对象 t处的时刻受到保护，则 $R[g] \neq 0$ 在 t处的时刻 .

这对应于 [10]，的引理 16，证明略有修改。

证明。 假设进程 p 在某个时刻保护标签 $g \neq 0$，该标签属于某个加载链接/条件存储对象 L_i，时刻为 $t^\star$。根据定义 15(b)，当非零标签 g 在第 7 行或第 41 行受到保护时，保护它的进程 p 已经递增 $R[g]$（分别在第 6 行或第 40 行），但尚未递减 $R[g]$。根据定义 5，p 因此在时刻 $R[g]$ 贡献一个非零值，时刻为 $t^\star$。

对于第 7 行的情况，根据定义 15(b)， $L[i]$ 在val、 tag时刻与 p的变量对匹配（因此 $L[i]$.标签为 $= tag = g$）；所以， p 在第 8 行返回。如果 p 在第 20、28 或 30 行将其贡献转移到某个加载链接/条件存储对象，从而递减其贡献，那么该递减会抵消同一方法调用中第 19 或 27 行的一次递增。此外，如果 p在第 21、29 或 44 行递减其贡献，那么此前必定有一个来自加载链接/条件存储对象的贡献在第 20、28 或 43 行转移给了 p。因此， p对 $R[g]$ 的贡献不会递减至零，直至 $t^\star$ 之后至少一个时刻；在该时刻， $t^\star$ 会在一次 CL(i) 调用中递减 $R[g]$ 的值，此时 g 在该调用的调用时刻起不再受 L_i 保护；或者在后续调用的第 10 行， LL(i)不再受保护。然而，由于只有 CL(i)能将 $d[i]$ 重置为 $\perp$ （在第 36 行），如果 p 在 $t^\star$ 时刻之后调用 LL(i)，且期间没有调用 CL(i)，那么它会在第 1 行调用 CL(i)。这会使 g 在 p于第 10 行递减 $R[g]$之前，不再受 L_i 保护。

对于第 41 行的情况， p 在第 45 行递减 $R[g]$，也可能在第 44 行递减。如前所述，如果 p 在第 44 行递减 $R[g]$，那么它还会通过第 43 行从 L_i 转移，从而增加其对$R[g]$ 的贡献。因此， p的贡献只能在第 45 行递减至零，此时 p对 g 的保护结束。

因此，在 $t^\star$ 时刻之后，直到 g 不再受保护于 L_i之后， p 不会递减其对 $R[g]$ 的贡献。该引理由引理 6(e)和引理 12(a)得出。◀

►**引理 25** （保护不变式）。 假设标签 g 在 L_i 时刻 t受到保护。那么（**a**） $L[i]$.标签不会在 g 时刻从不同于 g 的值变为 t，并且（**b**）如果 g 在强保护下于 L_i 时刻 t，那么 $L[i]$.标签 也不会在 g 时刻从不同于 t的值变为零。T

该引理对应于 [10]，的引理 17，但 (a) 的证明采用了不同的技术，
u使用了锁定标签这一新概念。

证明。 我们通过反证法证明该引理。假设不变式不为真，并令 t 为一次执行中不变式失效的第一个时刻。因此，在时刻 t，存在一个标签 g，它在 L_i 期间受到保护，并且 $L[i]$.标签 从标签 $g' \neq g$ 变为 g，或者在 g 受到强保护时变为零 L_i。

首先假设 (b) 在时刻 t 失效，因此 g 在时刻 t受到 L_i 的强保护，但 $L[i]$.标签在该时刻从某个标签 $g' \neq g$ 变为零。只有第 43 行能够将 $L[i]$.标签改为零，因为唯一其他能够更改 $L[i]$.标签的行，即第 20、28 和 30 行，都会选择一个非零标签（来自第 18 行和第 26 行）。因此，存在一个进程 p，它在时刻 t于第 43 行成功执行 CAS()，将 $L[i]$.标签从 g'（p 的本地值为 tag）改为零。由于 p 执行第 43 行，它在时刻 t 执行 CAS() 之前立即在第 41 行和第 42 行检查的 if 条件均返回真。因此，存在一些时刻 $t_L < t_C < t$，使得 $L[i]$.标签在时刻 t_L为真，并且在时刻 t_C为 $C[i] = 0$。

根据定义 15(b)， g' 在时刻 t_L开始受到 L_i 保护，并且这种保护持续到 p 执行第 45 行为止，而这发生在时刻 t之后。根据引理 16，在时刻 t_C， g 不受 L_i 强保护。由于根据假设，在时刻 $t > t_C$，g 受 L_i 强保护，因此 g会在某个时刻 $t^\star \in (t_C, t)$ 开始受到 L_i 强保护。根据定义 15， $L[i]$.在时刻 $t^\star$标记为 $= g$。由于在时刻 $t > t^\star$之前， $L[i]$.标记为 $= g' \neq g$，因此存在一个时刻 $t^\dagger \in (t^\star, t)$，在该时刻， $L[i]$.的标签从某个不同于 g' （可能是 g）的值变为 g'。但是，由于 g' 是

在整个（t_L, t] 期间都受到 L_i 保护（回想一下， $t_L < t_C < t^\star < t^\dagger < t$），时刻 $t^\dagger$ 违反了 (a)，这与 t 是该不变式首次失效的时刻相矛盾。

接下来需要证明在时刻 t 情况 (a) 失败时会导致矛盾，因此 $L[i].$标签会在时刻 t 从某个标签 $g' \neq g$ 变为 g，同时 g 在 L_i 期间受到保护。注意，如果 $g = 0$，那么它不可能在第 41 行变为受保护，因为 `cleanup()` 方法会在第 39 行退出。因此，如果 $g = 0$ 受到保护，那么它是在一次 `LL()` 调用期间受到保护的，所以它也受到强保护，从而可由刚刚针对 (b) 得出的结论得到矛盾。因此，不失一般性，我们可以假设 $g \neq 0$。

由于唯一能够将 $L[i].$标签改为非零值的代码行是第 20、28 和 30 行，因此存在一个时刻 $t' < t$，在第 19 行或第 27 行， p 将 $R[g]$ 从零递增为一。根据定义 10，在整个区间 $(t', t]$ 内， g 都由 p 锁定。由于紧接在时刻 t' 之前 $R[g] = 0$，根据引理 24，我们知道紧接在 t' 之前 g 尚未受到保护。根据定义 15，第 19 行和第 27 行不会使标签变为受保护状态，因此在时刻 t'， g 也尚未受到保护，所以 g 会在某个时刻 $t^\star \in (t', t)$ 变为受保护状态。根据定义 15，在时刻 $t^\star$，$L[i].$标签为 $= g$。由于紧接在 t' 之前 $R[g] = 0$，根据引理 8 和引理 12(a)，我们还知道在时刻 t'， $L[i].$标签为 $\neq g$ （回想一下， $L[i]$ 在时刻 t' 不会发生变化）。因此，存在一个时刻 $t^\dagger \in (t', t^\star)$，在该时刻 $L[i].$标签从某个不同于 g 的值变为 g。在时刻 $t^\dagger$ 执行该步骤的进程不是 p，因为在时刻 t' 和 t 之间， p 不会改变 $L[i]$ 的值。由于在时刻 $t^\dagger \in (t', t)$， g 被 p 锁定，这违反了推论 13，从而产生矛盾。◀

▶**定义 26** （函数 lin）。 我们将函数 lin（c） 定义为将执行中的方法调用 c 映射到各个时刻，并将在之后的（定理 *29*）中证明，它们是该实现正确的线性化点。

如果 c 是一次 `LL()` 调用，那么 lin（c） 取决于该方法从哪一行返回。如果它在第 *5* 行或第 *14* 行返回，那么 lin（c） 发生在第 *3* 行。如果它在第 *8* 行或第 *12* 行返回，那么lin（c） 发生在第 *7* 行。

如果 c 是一次 `Read()` 调用，那么 lin（c）发生在第 *46* 行（这是它唯一的步骤）。

如果 c是一次 `VL()` 调用，那么 lin（c） 如果在第 *47* 行返回，则发生在其调用时刻；否则lin（c）发生在第 *48* 行。

如果 c 是一次 `Write()` 调用，那么 lin（c）发生在第 *20* 行。

如果 c 是一次 `SC()` 调用，那么 lin（c） 如果在第 *23* 行返回，则发生在其调用时刻；否则 lin（c） 就是对 L执行的最后一次 `CAS()` 操作的时刻。也就是说，如果第 *28* 行的 `CAS()` 成功，那么 lin（c） 发生在第 *28* 行；否则 lin（c） 发生在第 *30* 行。

如果 c 是一次 `CL()` 调用，那么 lin（c）发生在其调用时刻。

对于包含待处理方法调用的执行， lin（c）对待处理的 `LL()` 调用未定义；对于其他尚未执行与 lin（c）对应行的方法调用，也未定义。

相对于 [10] 中的定义 19，唯一新增的内容是 `Write()` 方法，其中$lin(c)$映射到第 20 行的直观时刻。

▶ 引理 *27*。假设 c 是进程 p发起的 `SC`(i, u) 或 `VL`(i) 调用。该调用返回真，并在时刻 lin（c）将 $L[i].val$ 更改为 u；对于 `SC`(i, u)，当且仅当满足以下条件：*(a)* 在某个时刻 $t < lin$（c），某个标签因 p 而对 L_i 变为强保护状态，并在整个区间（t， lin（c）], 内始终受到 p 的强保护；*(b)* 区间（t， lin（c）） 内，没有任何 `Write`$(i, \cdot)$ 调用执行第 *20* 行的 `Swap()` 操作；*(c)* 区间（t， lin（c）） 内，没有任何 `SC`$(i, \cdot)$ 调用在第 *28* 行或第 *30* 行成功执行 `CAS()`。该引理对应于 [10], 中的引理 *20*，但这里增加了条件 *(b)*。证明与 [10]的证明大体相同。

证明。 我们先证明正向，因此首先假设 c 返回 真。由于c 返回 真，它在第 48 行或第 33 行作出响应。尤其是，它没有在第 47 行或第 23 行返回，这意味着 $d[i] \neq \perp$ 当 p 调用 c时。根据引理 17， $d[i]$.标签在 c的调用时受到强保护，该状态从之前的某个时刻开始，即 $t < lin(c)$ ；根据定义 15，它会一直受到保护，直到 $lin(c)$ 之后。因此，对于某个时刻 $t < lin(c)$ ，条件 (a) 成立。

令 $g^\star$ 为在时刻 t 强保护 p 的标签，即在调用 c 中其 $d[i]$.tag 的值。为证明矛盾，假设 (b) 或 (c) 不成立，因此第 20 行，或第 28 行或第 30 行中的一次成功 CAS() ，发生在 $(t, lin(c))$ 中，并令 t' 为第一个这样的时刻。注意，唯一能够改变 $L[i]$ 值的其他操作发生在第 43 行，该行还会将 $L[i]$.tag 设为零。由于根据定义 15， $L[i]$.tag 在时刻 $t = g^\star$ ，因此在时刻 t' 之前， $L[i]$.tag 必然是 $g^\star$ 或零。根据引理 14，$L[i]$.tag 在时刻 t' 变为不同的非零值；根据引理 25，由于 $g^\star$ 在时刻 t' 受到保护， $L[i]$.tag 不会在该时刻变为 $g^\star$ 。因此， $L[i]$.tag 在时刻 t' 变为某个不同于 $g^\star$ 的非零标签。对 $(t', lin(c))$ 中每次将 $L[i]$.tag 改变的操作（如果有）应用引理 25，可知在整个区间内， $L[i]$.tag 只能变为不同于 $g^\star$ 的其他非零值。但是，由于我们假设 c 返回真，且 $lin(c)$ 发生在第 28 、 30 或 48 行，根据代码可知，在 $lin(c)$ 之前， $L[i]$.tag $\in \{g^\star, 0\}$，这就产生了矛盾。因此，(c) 和 (b) 成立，正向的证明完成。

接下来只需证明逆向，因此假设 (a)、(c) 和 (b) 对 c 均成立。与正向情形类似，令 $g^\star$ 为 p 在整个 $(t, lin(c)]$, 期间强保护的标签，并根据定义 15 得到 $L[i]$. 在时刻 t 的标签为 $= g^\star$ 。根据引理 17，在 c 调用时， p 的 $d[i]$. 标签值为 $g^\star$ ，尤其是 p 不会在第 47 行或第 23 行返回。由于在整个 $(t, lin(c))$ 期间，没有任何能够将 $L[i]$. 标签改为非零值的步骤（即第 20 行，或第 28 行或第 30 行中的一次成功 CAS() ）发生，因此在时刻 $lin(c)$ 之前， $L[i]$. 的标签立即为 $\in \{g^\star, 0\}$。如果 c 是一次 VL(i) 调用，那么该调用返回真，因为 $g^\star$ 是 $d[i]$. 标签在 p 处的值。如果 c 是一次 SC(i,·) 调用，且其第 28 行的 CAS() 成功，那么该调用返回真。否则，第 28 行的 CAS() 失败，因此 p 会在第 30 行执行第二次 CAS() ，而 $lin(c)$ 是这第二次 CAS() 的时刻。由于在 $(t, lin(c))$ 期间， $L[i]$. 标签值唯一可能的变化是从 $g^\star$ 变为零，因此 $L[i]$. 在 p 执行第 28 行时的标签为 $= 0$ ，并且在 $lin(c)$ 之前仍保持为零，从而使第二次 CAS() 成功。◀

▶ **引理 28.** 假设 c 是进程 p 发起的一个 LL(i) 调用，满足 $lin(c) = t$ ，且该调用在时刻 t' 返回。如果某个标签在该调用期间因 p 而对 L_i 变为强保护，则它会在时刻 t 变为强保护。否则，某个进程 $q \neq p$ 会执行调用 c'，该调用要么是 Write(i,·) 调用，要么是带有成功 CAS() 的 SC(i,·) 调用，且该成功 CAS() 发生在时刻 $lin(c')$，满足 $t < lin(c') < t'$。

这对应于 [10], 的引理 21，唯一的区别是，重叠的 Write() 操作也可能导致 LL() 调用无法强保护某个标签。

证明。结合定义 15 和 26，在第 5、8 或 12 行作出响应的 LL() 调用 c 会使某个标签在 $lin(c)$ 处变为强保护，而在第 14 行作出响应的 LL() 调用不会强保护任何标签，从而证明了第一个断言。如果 c 在第 14 行作出响应，则 p的局部变量 $tag \neq 0$ 和 $tag' \notin \{tag, 0\}$。特别地，在时刻 $t^\star \in (t, t')$， p 执行第 7 行，在该行中它从 $L[i]$.标签读取 tag' 。因此，存在某个时刻 $t^\dagger \in (t, t^\star)$，在该时刻 $L[i]$.标签首次变为不属于 $\{tag, 0\}$ 的值。能够将 $L[i]$.标签改为非零值的唯一几行，是第 20 行中的一个 Swap() ，或第 28 或 30 行中成功的 CAS() ；根据定义 26，该操作是 Write(i,·) 或 SC(i,·) 调用的线性化点。因此， $t^\dagger = lin(c')$ 对于

某个 Write(i,·) 调用，或某个 SC(i,·) 调用在 $L[i]$上成功执行 CAS()。此外，由于 c'和c 重叠，因此可得 c'的调用进程 $q \neq p$。◀

▶**定理 29 (线性化)。** 根据顺序规范，*LL/SC* 实现是线性化的，其中 $lin(c)$ 是每次调用 c 的线性化点。

这对应于 [10], 的定理 22，证明基本上没有变化。

证明。 考虑使用此实现执行可写 LL/SC 对象的过程。对于执行中每个待处理调用 c，如果根据定义 26， $lin(c)$ 未定义，则将 c 从线性化中移除。对于每个 CL() 从一个 LL() 内部调用的 SC()或 VL() 调用，也将其从线性化中移除。我们根据 $lin(c)$对剩余调用排序，以此构造线性化。（请记住， cleanup() 是辅助方法，因此不会在执行中被直接调用。所以它不属于线性化的一部分。）

我们通过对按 $lin(c)$ 排序的调用 c 进行归纳，证明该线性化是正确的。具体而言，我们证明每个调用 c 都会返回有效响应，并正确影响系统的解释后抽象状态。我们通过归纳证明，在每个线性化点之前和之后，LL/SC 对象 L_i 的解释后值均为 $L[i]$.val。初始时这一点成立，因为 $L[i]$.val 被初始化为 u_i，即应用程序为 L_i 指定的初始值。回顾顺序规范（第 2.3 节），在顺序执行的给定方法调用 c 之前， L_i 的上下文位 ℓ_p 等于 1（即 p 与 L_i 有一个开放链接），当且仅当

- 在顺序执行中，进程 p 在调用 c 之前调用的 LL(i) 操作 c'，其间不会发生任何
- Write(i,·) 调用，也不会有任何进程发起成功的 SC(i,·) 调用（即返回真值的调用）；此外，在 c' 和 c 之间，进程 p 不会发起 CL(i) 调用。

根据顺序规范还可知，上下文位只会直接影响 SC(i,·) 和 VL(i) 调用的返回值。因此，要证明上下文位的行为正确，只需证明：在相应调用的线性化中，紧接该调用之前，相关上下文位应为 1，当且仅当 SC(i,·) 和 VL(i) 调用返回真值。

如果 c 是 Read(i) 或 LL(i) 调用，那么根据该调用返回 $L[i]$.valat lin（c）这一事实，即可得出其正确性。

如果 c 是 Write(i,u) 调用，那么根据 $L[i]$.val 在 lin（c）之后立即为 $= u$ 这一事实，即可得出其正确性，因为 lin（c）在该调用执行第 20 行时发生，无条件地将 u 写入 $L[i]$.val。如果某个 SC(i,u) 调用 c 返回真值，那么根据定义 26 c，一次成功的 CAS() 会在时刻 lin（c）将 $L[i]$.val 改为 u，因此 $L[i]$.val 在 lin（c）之后立即为 $= u$。根据代码和定义 26，只有当存在调用 c'，且 lin（c'）$= t'$时， $L[i]$.val 才会在时刻 t' 改变；其中 c' 要么是 Write(i,u) 调用，要么是返回真值的 SC(i,u) 调用。在这两种情况下， $L[i]$.val 都会在时刻 t' 变为 u。因此，如果进程 p 发起的 c 是一个 SC(i,u) 或 VL(i) 调用，我们只需证明它返回真值，当且仅当根据前面列出的三个条件，解释的上下文位 $\ell_p = 1$ 应为真。

假设进程 p 发出的 SC(i,u) 或 VL(i) 调用 c 确实返回真，我们将根据以下三个条件证明，在 c 之前发生了 $\ell_p = 1$。根据引理 27(a)，存在一个时刻$t < lin(c)$，在该时刻，某个标签 g 对于 L_i 被 c's 调用进程 p强保护，并且这种强保护至少持续到 $lin(c)$。根据定义 15，只有 p 发出的 LL(i) 调用c' 才能使 p 对 g进行强保护；根据引理 28， g 在时刻 $lin(c')$变为强保护状态。

由于 CL(i) 对 p 的调用会结束其对任意标签的强保护，且 g 从 $lin(c')$ 起至少到 $lin(c)$ 为止一直受到强保护，因此可知 p 不会在 c' 和 c 之间调用 CL(i)。根据引理 27(c) 和 (b)，没有任何 SC(i,·) 调用在 $L[i]$ 上成功执行 CAS()

在区间 ($lin(c')$ $lin(c)$) 内，并且该区间内不会执行第 20 行。由于一次 SC(i,·) 调用 s 若成功执行 CAS() 操作，就会返回真，且在该 CAS()时刻具有 $lin(s)$，因此不存在这样的调用 s，使得 $lin(c') < lin(s) < lin(c)$。同理，由于一次 Write($i$,·) 调用 w 在执行第 20 行时具有 $lin(w)$，因此不存在这样的调用 w，使得 $lin(c') < lin(w) < lin(c)$。因此，这三个条件均得到满足，并且在 $lin(c)$ 之前的紧邻时刻，解释的上下文位为 $\ell_p = 1$，正如所要求的那样。

现在假设，对于进程 p 的一次 SC(i, u) 或 VL(i) 调用 c，紧接在调用 $lin(c)$ 之前的解释的上下文位为 $\ell_p = 1$。那么，根据这三个条件，p 有一次 LL(i) 调用 c'，其间有 $lin(c') < lin(c)$，且 p 不会在 c' 和 c 之间直接调用 CL(i)。此外，不存在从第 28 行或第 30 行的一次 SC(i,·) 调用中成功执行的 CAS()，使得 $lin(c') < lin(s) < lin(c)$；也不存在从第 20 行的一次 Write($i$,·) 调用中执行的 Swap()，使得 $lin(c') < lin(w) < lin(c)$。

不失一般性，假设 c' 是 p 在 c 之前最后一次调用 LL(i)（如果这三个条件对更早的一次 LL(i) 调用成立，那么它们对 c' 也成立）。根据引理 28，标签 g 在时刻 $lin(c')$ 被 p 强保护（因为 p 在调用 c 之前已从 c' 响应）。为了使用引理 27，我们需要证明这种强保护至少持续到 $lin(c)$，尤其要证明 p 不会在 $lin(c')$ 和 $lin(c)$ 之间间接调用 CL(i)。（回想一下，CL() 的间接调用会从线性化中移除。）CL(i) 可能发生间接调用的位置在第 1、13、32 和 49 行，因此我们将分别通过导出矛盾，证明 p 不会在 $lin(c')$ 和 $lin(c)$ 之间执行这些行中的任何一行。

对于第 1 行，由于该行出现在任何可能映射到 $lin$for LL()的行之前，因此它必须属于 c' 之后的一次 LL(i) 调用，这与 p 在 c 之前最后一次进行 LL(i)调用的是 c' 相矛盾。对于第 13 行，根据上述论证，它应该属于调用 c'，但根据定义 15 并检查代码可知，在 c' 期间没有任何标签变为强保护状态，这与某个标签确实在时刻 $lin(c')$ 变为强保护状态相矛盾。第 32 行应属于一次 SC(i,·) 调用，因此我们将证明 p 不会在 c' 和 c 之间进行任何此类调用。如果此类调用返回真，这就与以下假设相矛盾：在线性化中，调用 c' 和 c 之间不存在返回真的 SC(i,·) 调用。如果此类调用返回假，那么根据归纳假设，存在另一次 SC(i,·) 调用 s，它返回真，且有 $lin(c') < lin(s) < lin(c)$，这同样与我们的假设相矛盾。最后这一论证也适用于 p 在 $lin(c')$ 和 $lin(c)$ 之间执行第 49 行的情况，因为该行只会在返回假的 VL(i) 调用中执行。

因此，我们得出：在时刻 lin（c'），标签会因 p for L_i 而受到强保护，并且这种状态至少会持续到时刻 lin（c）。由于在 lin（c'）和 lin（c）之间，既没有来自 SC(i,·) 调用的成功 CAS()，也没有来自 Write(i,·) 调用的 Swap()，根据引理 27，c确实会返回真。

由于 CL() 只影响 ℓ_p，而 ℓ_p 本身仅会通过我们已经讨论过的 SC() 和 VL() 调用变得可见，因此证明完成。◀

定理 21、23 和 29 一起推出定理 1。

5 结论

我们展示了如何支持一种无条件且随机化的无等待 Write() 操作，用于实现保持 QHI 的加载链接/条件存储 [10], ，所使用的基础对象支持交换、比较并交换以及取值并递增/递减，并且具有相同的渐近空间和期望步复杂度。为证明在支持这一更大操作集的情况下结果仍然成立，需要对技术分析进行重大修改。我们的成果为在历史独立并发这一新领域设计更多算法铺平了道路。

32 制作保持历史无关性的 LL/SC 可写

一个自然的开放问题是，在软件中实现更多能够保持历史独立性的原语操作。

参考文献

1 Umut A. Acar、Guy E. Blelloch、Robert Harper、Jorge L. Vittes 和 Shan Leung Maverick Woo。《将静态算法动态化，并应用于动态树和历史独立性》。载于第 15 届 ACM-SIAM 离散算法研讨会（SODA）论文集，第 531–540 页，2004年。URL: http://dl.acm.org/citation.cfm?id=982792.982871。 2 Zahra Aghazadeh、Wojciech Golab 和 Philipp Woelfel。《使对象可写》。载于第 33 届 ACM 分布式计算原理研讨会（PODC）论文集，PODC '14，第 385–395 页，2014年。 doi:10.1145/2611462.2611483。 3 Zahra Aghazadeh 和 Philipp Woelfel。《有界对象的无界标记上界》。载于第 30 届国际分布式计算研讨会（DISC）论文集，第 442–457 页，2016年。 doi:10.1007/978-3-662-53426-7_32。 4 James H. Anderson 和 Mark Moir。《大型对象的通用构造》。载于第 9 届国际分布式算法研讨会（WDAG）论文集，第 972 卷，第 168–182 页，1995年。 doi:10.1007/BFB0022146。 5 James H. Anderson 和 Mark Moir。《多对象操作的通用构造》。载于第 14 届 ACM 分布式计算原理研讨会（PODC）论文集，第 184–193 页，1995年。 doi:10.1145/224964.224985。 6 Hagit Attiya、Michael A. Bender、Martín Farach-Colton、Rotem Oshman 和 Noa Schiller。《历史无关的并发对象》。载于第 43 届 ACM 分布式计算原理研讨会（PODC）论文集，第 14–24 页，2024年。 doi:10.1145/3662158.3662814。 7 Hagit Attiya、Michael A. Bender、Martín Farach-Colton、Rotem Oshman 和 Noa Schiller。《历史无关的并发哈希表》。载于第 57 届 ACM 计算理论年会（STOC）论文集，第 1283–1294 页，2025年。 doi:10.1145/3717823.3718283。 8 Naama Ben-David、Guy E. Blelloch、Michal Friedman 和 Yuanhao Wei。《故障持久内存上的无延迟并发》。载于第 31 届 ACM 算法与体系结构并行性研讨会（SPAA）论文集，第 253–264 页，2019年。 doi:10.1145/3323165.3323187。 9 Dante Bencivenga、Homa Habashi 和 Philipp Woelfel。《高效的随机化 LL/SC：保持历史独立性》。载于第 40 届国际分布式计算研讨会（DISC）论文集，LIPIcs 第 397 卷，第 12:1–12:24 页，2026年。 doi:10.4230/LIPIcs.DISC.2026.12。 10 Dante Bencivenga、Homa Habashi 和 Philipp Woelfel。《高效的随机化 LL/SC：保持历史独立性》。arXiv 预印本，2026年。 arXiv:2608.12946。 11 Michael A. Bender、Jon Berry、Rob Johnson、Thomas M. Kroeger、Samuel McCauley、Cynthia A. Phillips、Bertrand Simon、Shikha Singh 和 David Zage。《持久存储上的反持久性：历史无关的稀疏表和字典》。载于第 35 届 ACM SIGACT-SIGMOD 数据库系统原理研讨会论文集，第 289–302 页，2016年。doi:10.1145/2902251.2902276。 12 Guy E. Blelloch 和 Daniel Golovin。《具有应用的强历史独立哈希》。载于第 48 届 IEEE 计算机科学基础研讨会（FOCS）论文集，第 272–282 页，2007年。 doi:10.1109/FOCS.2007.36。 13 Guy E. Blelloch 和 Yuanhao Wei。《LL/SC 与原子复制：仅使用指针宽度 CAS 实现常数时间、空间高效的方案》。载于第 34 届国际分布式计算研讨会（DISC）论文集，第 5:1–5:17 页，2020年。 doi:10.4230/LIPIcs.DISC.2020.5。 14 Niv Buchbinder 和 Erez Petrank。《获得历史独立性的上下界》。Information and Computation，204(2)：291–337，2006年。 doi:10.1016/J.IC.2005.11.001。

15 Simon Doherty、Maurice Herlihy、Victor Luchangco 和 Mark Moir。《为 64 位应用程序带来实用的无锁同步》。载于第 23 届 ACM 分布式计算原理研讨会（PODC）论文集，第 31–39 页，2004年。 doi:10.1145/1011767.1011773。 16 Wojciech Golab、Lisa Higham 和 Philipp Woelfel。《线性化实现不足以支持随机化分布式计算》。载于第 43 届 ACM 计算理论年会（STOC）论文集，第 373–382 页，2011年。 doi:10.1145/1993636.1993687。 17 Daniel Golovin。《B-treap：B 树的一种唯一表示替代方案》。载于第 36 届国际自动机、语言与程序设计学术讨论会（ICALP）论文集，第 487–499 页，2009年。 doi:10.1007/978-3-642-02927-1_41。 18 Daniel Golovin。《B 跳表：B 树的一种更简单的唯一表示替代方案》。arXiv 预印本，2010年。 arXiv:1005.0662。 19 Jason D. Hartline、Edwin S. Hong、Alexander E. Mohr、William R. Pentney 和 Emily Rocke。《刻画历史无关数据结构》。载于第 13 届国际算法与计算研讨会（ISAAC）论文集，第 229–240 页，2002年。doi:10.1007/3-540-36136-7_21。 20 Jason D. Hartline、Edwin S. Hong、Alexander E. Mohr、William R. Pentney 和 Emily C. Rocke。《刻画历史无关数据结构》。Algorithmica，42(1)：57–74，2005年。 doi:10.1007/S00453-004-1140-Z。 21 Maurice Herlihy。《无等待同步》。ACM 编程语言与系统汇刊，13(1)：124–149，1991年1月。 doi:10.1145/114005.102808。 22 Maurice P. Herlihy 和 Jeannette M. Wing。《线性化：并发对象的一种正确性条件》。ACM 编程语言与系统汇刊，12(3)：463–492，1990年。 doi:10.1145/78969.78972。 23 Amos Israeli 和 Lihu Rappoport。《强共享内存原语的不相交访问并行实现》。载于第 13 届 ACM 分布式计算原理研讨会（PODC）论文集，第 151–160 页，1994年。 doi:10.1145/197917.198079。 24 Prasad Jayanti。《从 LL/SC 实现 CAS 及反向转换的完备常数时间无等待实现》。载于第 12 届国际分布式计算研讨会（DISC）论文集，《计算机科学讲义》第 1499 卷，第 216–230 页，1998年。 doi:10.1007/BFB0056485。 25 Prasad Jayanti、Siddhartha Jayanti 和 Sucharita Jayanti。《支持动态加入的可写 LL/SC 和 CAS 持久算法》。载于第 27 届国际分布式计算研讨会（DISC）论文集，LIPIcs 第 281 卷，第 25:1–25:20 页，2023年。 doi:10.4230/LIPICS.DISC.2023.25。 26 Prasad Jayanti 和 Srdjan Petrovic。《LL/SC 变量的高效实用构造》。载于第 22 届 ACM 分布式计算原理研讨会（PODC）论文集，第 285–294 页，2003年。 doi:10.1145/872035.872078。 27 Prasad Jayanti 和 Srdjan Petrovic。《多字 LL/SC 变量的高效无等待实现》。载于第 25 届分布式计算系统国际会议（ICDCS）论文集，第 59–68 页，2005年。 doi:10.1109/ICDCS.2005.29。 28 Prasad Jayanti 和 Srdjan Petrovic。《高效实现大量 LL/SC 对象》。载于第 9 届分布式系统原理国际会议（OPODIS）论文集，第 17–31 页，2005年。 doi:10.1007/11795490_4。 29 Prasad Jayanti 和 Srdjan Petrovic。《高效实现由未知数量进程共享的 LL/SC 对象》。载于分布式计算国际研讨会（IWDC 2005）论文集，第 45–56 页，2005年。 doi:10.1007/11603771_5。 30 Chaitanya Koparkar。《指针标记入门》。XRDS：十字路口，ACM 学生杂志，29(1)：66–68，2022年。 doi:10.1145/3558200。 31 Linux man-pages 项目。proc_sys_kernel(5)：控制一系列内核参数，2026年2月。6.19 版，访问于2026年11月22日。URL： https://man7.org/linux/man-pages/man5/proc_sys_kernel.5.html。

34 保持历史无关性的 LL/SC 可写

32 Victor Luchangco、Mark Moir 和 Nir Shavit。非阻塞 k——比较单次交换。在第 25 届 ACM 算法与体系结构并行性研讨会（SPAA）论文集，第 314–323 页，2003 年。 doi:10.1145/777412.777468。33 Daniele Micciancio。无感知数据结构：密码学应用。在第 29 届 ACM 计算理论年会（STOC）论文集，第 456–464 页，1997 年。doi:10.1145/258533.258638。34 Maged M. Michael。使用 64 位 CAS 的实用无锁和无等待 LL/SC/VL 实现。在第 18 届国际分布式计算研讨会（DISC）论文集，第 144–158 页，2004 年。 doi:10.1007/978-3-540-30186-8_11。35 Mark Moir。非阻塞同步原语的实用实现。在第 16 届 ACM 分布式计算原理研讨会（PODC）论文集，第 219–228 页，1997 年。 doi:10.1145/259380.259442。36 Fatemeh Naderi-Semiromi 和 Philipp Woelfel。基于 CAS 的强线性化 LL/SC。在第 43 届 ACM 分布式计算原理研讨会（PODC）论文集，第 443–453 页，2024 年。 doi:10.1145/3662158.3662800。37 Moni Naor、Gil Segev 和 Udi Wieder。历史无关布谷鸟哈希。在第 35 届国际自动机、语言与程序设计学术讨论会（ICALP）论文集，第 631–642 页，2008 年。 doi:10.1007/978-3-540-70583-3_51。38 Moni Naor 和 Vanessa Teague。反持久性：历史无关数据结构。在第 33 届 ACM 计算理论年会（STOC）论文集，第 492–501 页，2001 年。 doi:10.1145/380752.380844。39 Julian Shun 和 Guy E. Blelloch。用于确定性的阶段并发哈希表。在第 26 届 ACM 算法与体系结构并行性研讨会（SPAA）论文集，第 96–107 页，2014 年。 doi:10.1145/2612669.2612687。