A Primer on Memory Consistency and Cache Coherence

内存一致性与缓存同一性入门

Daniel J. Sorin, Mark D.Hill, David A.Wood

翻译：李默 phylimo@163.com

2022.09

译注

1. consistency 和 coherence 一般都统一译作“一致性”，即内存一致性、缓存一致性。但其实二者的“一致性”的英文单词不同，所以其“一致”内涵实际是不同的。本书使用稍微不同的翻译加以区分。

   前者 consistency 字面意思是前后一致，持续的一致，即偏重时序上的行为特点，采用常规“一致性”的译法；

   后者 coherence 字面为协调,和谐,所以更偏重在逻辑上的行为特点,“cache coherence”的机制主要用于描述多个内存访问者对相同内存位置的操作是如何交互的，最理想的是在同一时刻，同一个数据、同一个值（但默认行为上一般不会如此严格，这会以损失性能为代价）。所以 cache coherence 是描述同数据是怎样保持同值的，故这里译作“同一性”。

2. 如无特殊说明，“一致性”将作为缩称代指内存一致性，“同一性”将作为缩称代指缓存同一性。

3. 如无特殊说明，文中如果出现“内存系统”或“内存”，指共享内存系统和共享内存，即多个操作者（如处理器，外设等等）同时会访问操作的内存系统。

目录

摘要

许多现代计算系统及多核处理器芯片在硬件上都支持共享内存。在有共享内存的系统中，不同的处理器核心可能对同一内存地址空间进行读写操作。对这样拥有共享内存的系统，它的内存一致性模型在架构上定义了它的内存组件能被观察到的行为的集合。一致性定义了读写操作的行为规则及它们如何与内存交互作用。为了实现所标称的一致性模型，一般这样的系统同时会使用特定的缓存同一性协议来保证多处缓存的数据时刻保持最新。本入门手册的目的是为读者提供对内存一致性和缓存同一性模型的基本理解，包括技术问题与其解决方案。我们会在书中给出高度抽象的概念，同时也会给一些实际中的例子。

第一章　内存一致性、缓存同一性简介

许多现代计算系统及多核处理器芯片在硬件上都支持共享内存。在有共享内存的系统中，不同的处理器核心可能对同一内存地址空间进行读写操作。这种使用多核心、共享内存的系统设计一般是想追求高性能、低功耗、低成本等等特性。当然，如果没有了基本的正确性，那这些特性也将无价值。共享内存正确性乍看好像很简单，但实际上，正如本书将展示的，共享内存的正确性定义甚至都不是一件简单事，存在很多模糊点，更不用说实现一个“正确”的共享内存。这些问题需要在硬件实现过程弄清楚，因为修复一个硬件 bug 的成本是很高的。学术的人也应该弄清，明确对象，这样才能讨论某个提议的设计是否能正常工作。

在实际研究中我们发现在研究内存正确性时从两维度出发将有助于问题的分析与解决：consistency 与 coherence（一致与同一，下面会对这两个概念进行定义阐述）。当然一个系统并不是必须要区分此二者，只是从我们的经验上看，此二者有助于我们将问题分解，各个击破。实际上，在内存系统实现过程中，普遍采用了这两个维度的分解方法。

共享内存的正确性主要由一致性维度来定义，一致性模型定义了读写如何作用到内存的规则。理想情况下，一致性模型应该是一个简单的易于理解的模型（或规格说明）。然而，定义共享内存的正确性比定义一个单核模型（即无需考虑并发共享）下的内存的正确性要模糊、朦胧地多。在单核模型下，内存正确性很好定义，它有一个唯一正确性结果，可以与所有不正确性的行为明确区分，这是因为在单核模型下，程序的输入具有明确的输出，即使这个 cpu 在内部执行时可能是乱序的，我们依然可以有明确的符合程序逻辑的预期输出。但对共享内存（即可能有多个处理器核心同时读写交互），它需要考虑来自多线程的并发读写，这种情况哪种输出是正确的就不

是唯一的，它可能会允许一些行为（即“正确”行为，是该内存模型下可能发生的行为），同时不允许另外一些行为（即“不正确”，是该内存模型承诺不会发生的行为）。之所以允许“一些”可能行为，而不是只允许“一种“行为，是因为在多线程并发读写的场景中无法明确这些线程在物理上孰前孰后，所以线程间可能有多种物理时序交错。这种正确性多样化使得问题变得复杂，但要想在共享内存、多核系统上编写“正确”的程序，这些问题必须要得到明确地解决。

与内存一致性不同，缓存同一性对软件是不可见的，也不需要可见。但作为支撑一致性模型的一部分，大部分共享内存计算机系统才会使用某种同一性协议（因为大多系统也都使用了缓存）。同一性的目的是为了让共享内存系统的缓存行为和单核模型下的缓存一样对外不可见，即不影响程序的功能。所以，同一性协议的目的是让用户无法从读写的数值结果上感受到缓存的存在，避免缓存的使用引入任何功能性差异（当然，程序员依然可以从时间维度上探测缓存的存在，但这不属于功能性差异）。

一般来说，同一性在支撑一致性模型的过程中扮演非常重要的角色。因此，虽然一致性是本书的第一个主题，我们在第二章还是先对同一性进行一些介绍。本章先对一致性与同一性的关系进行一些必要的介绍，而不是直接深入具体的协议和实现（这些将在后面的章节 6-9 中探讨）。第二章中，我们先使用 SWMR(单写者多读者) 不变式来定义同一性。SWMR 不变式要求在任意时间任一内存位置要么只能在一个缓存中接收写操作（同时也可以在该缓存中接收读操作），要么可以在多个缓存中接受读操作。

1.1　一致性

一致性亦称内存一致性、内存一致性模型，或内存模型。一致性模型定义了什么是“正确的”内存读取写入行为，而且这里的正确性与缓存及缓存同一性无关。为了更通俗地理解为什么需要一致性模型，我们举一个现实世界的例子。比如一个大学在网上发布了它的课程表，其中计算机课程的上课地点是 152 教室。在上课的前一天，该大学管理人员决定将课程改至 252 教室上课。管理人员首先给网站管理员发送了 email 告诉他将网上的课程信息

更新，不久后他发短信给所有注册该课程的学生告知他们课程表更新的事情。不难预见，可能会有这样一种情形：网站管理员太忙碌没有及时更新课表信息，而某位学生在接收短信后立即上网查看“新”课表，此时该学生将看到教室还是 152。虽然几分钟后管理员最终将课程教室更新至了 252，而且这位大学管理人员 (从他的视角看) 也进行了正确的“写”操作顺序，但这位学生观察到的“写”操作顺序却是不同的，导致最终去了错误的教室。一致性模型就是定义在这样的系统中这种行为表现（即管理人员先通知网络管理员更新网络，再通知学生; 而学生则是先收到信息，之后才觉察网络更新）是“正确”的（此时用户需要采取其它措施以获取想要的结果），或是错误的（即系统实现将保证不会出现这种行为表现）。

上述例子使用了多种信息传递介质，但类似的行为完全可能出现在同一介质的内存系统中，尤其系统组成中有多个能够乱序执行的 cpu，或内存系统有 write buffer，具有预取能力，或者多 cache bank 机制。因此，我们需要定义内存系统的正确性，即何种内存行为是被允许的，这样程序员才会根据该正确性定义对系统的输出有正确的期望，内存系统的实现者也会有一个明确的行为约束准则。

内存系统正确性由内存一致性模型给出。一致性模型明确规定了多线程、共享内存环境下被允许的行为集合。对某处理某特定输入的多线程程序而言，内存模型规定了其各个读取操作可能返回的内存值，以及内存可能的最终状态。不同与单线程模式，在多线程模式下，可能有多种行为结果是被允许的，因而理解内存一致性模型也会稍微复杂。

第三章介绍了内存一致性模型的概念，同时介绍“顺序一致性模型”(Sequential Consistency，SC)，这是约束最强也是最直观的一致性模型。章节伊始再次阐述了内存正确性定义的必要性，同时精确地定义什么是内存一致性模型。接着，章节深入讨论 SC 模型。SC 模型规定，一个多线程程序运行的结果应该看起来像它的各个线程各自按程序指令顺序执行，线程间又以某种时间交错方式下分时复用单核心处理器运行的结果一样，很直观、易于理解。在直观印象后，我们用形式化语言正式地对 SC 进行讨论，同时探索了在有缓存机制的情况下较简单较复杂两种方式实现 SC 模型的过程，最终以 MIPS R10000 实例学习作为结束。

第四章，我们在 SC 基础上研究 x86 和 SPARC 处理器所使用的内存模型：全局写序一致性模型 (total store order, TSO) 。TSO 模型的要旨是在处理器将写操作正式提交给缓存之前，先使用一个先入先出的写缓冲区来暂存这些提交的写操作。写缓冲区的引入将违背 SC 模型定义，但由于 TSO 模型可以带来明显的性能收益，所以人们还是允许这样一种内存模型的存在并对其进行了明确定义。在该章中，我们还将探索如何用 SC 的形式化方法来类似地研究 TSO 模型，TSO 模型如何影响内存系统的具体实现，以及对 SC 和 TSO 进行比较。

最后，在第五章我们引入弱序一致性模型（简称弱序模型）。弱序模型提出的动机是那些强约束一致性模型中对大多内存操作的序的限制是不必要的。如果一个线程更新了十项数据，然后用一个同步 flag 进行标记与其它线程同步，程序员其实不关心这十项数据哪项数据先更新哪项后更新，唯一关心的是同步 flag 的标记动作需要在十项数据更新完成后再完成。弱序模型的目标正是提供这种灵活的内存操作序以得到更高的性能或更简单的内存系统实现。阐述了动机后，我们提出了一个名为 XC 的弱序模型示例。在 XC 中，程序员只有使用 FENCE 指定才能得到明确的内存操作顺序，比如在数据更新和同步 flag 标记之间使用 FENCE，以确保 flag 的完成在数据更新之后。之后，该章延续使用前面对 SC 和 TSO 使用过的形式化方法，对 XC 模型进行分析，以及探讨如何实现 XC (包括各种乱序、同一性协议等等)。接着我们讨论了一种很多程序员可以用来避免直接对弱序模型进行推理的做法：如果在程序中加入足够多的 FENCE 指令以保证程序没有数据竞争（data race free, DRF），那么大多弱序模型将表现地类似 SC 模型。在这种无数据竞争的类 “SC 模型” 下，程序员可以兼得（相对）SC 的直观的正确性和 (相对）XC 的高性能。对想深入了解的读者，该章最后将”acquire/release” 做了区分，讨论了写原子性和因果性，还将涉及一些商业产品例子（包括一个 IBM Power 的例子）以及高级语言 (Java 和 C++) 的内存模型。

回想上面大学课程表的例子，我们可以看到 email、网站管理员、短消息这些元素组成的系统其实是一个 “弱序模型”。为了学生由于 “太及时” 查看短信反而去到错误教室这种情况的发生，大学管理人员需要在发送 email

通知网站管理员更新网页和发送短信给学生之间插入一条 FENCE 命令，以保证网站更新完成后学生才会收到短信。

1.2　同一性

同一性即缓存同一性。在有缓存的系统中，多个内存操作者（如处理器、可直接访问内存的外设等等）同时对一块数据区进行访问而且其中有写操作时，如果不精心处理这些并发操作的交互，同一性问题将会发生。举一个与前面用过的课程表类似的例子：一位同学上网查看了课程表信息，看到上课地点是 152 教室（数据读取），记在了自己的本子上 (数据缓存)。接着管理员将上课地点改到 252 教室，并及时更新了网页内容。此时，这位同学自己本子上的数据是过期的了，这时就遇到了“不同一”的场景。如果同学到 152 去，他会发现进错了教室。计算机的世界里（不包括计算架构），“不同一”的例子有网页缓存、使用未更新代码库编程等等。

同一性协议就是为了避免这种获取到过期数据的情况出现，它是所有参与者都共同遵循的一系列规范。同一性协议可能有很多变种形式，但其主要思想都差不多，第六-九章会进行详细讲解。

第六章对同一性协议全景图进行了介绍，为后面章节对特定的同一性协议深入讨论做了个铺垫。这一章会讨论大多数同一性协议都会遇到的共性问题，包括缓存控制器和内存控制器的分布式操作，通用的 MOESI 同一性协议中的状态：修改中 (modified, M)，占有 (owner, O)，独占（exlusive, E），共享 (shared, S)，非法 (invalid, I)。本章的一个重点是介绍用表驱动的方法对同一性协议的状态进行表征，包括稳态与瞬态。瞬态是需要的，因为现代系统极少提供从一种稳态原子转换到另外一种状态的能力（比如，对处于 Invalid 状态的数据进行读操作导致读未命中 (read miss)，在变成 Shared 状态前，需要耗费一点时间等待数据响应）。很多同一性协议的实现复杂性被屏蔽在瞬态的过程中，这点类似于处理器复杂性被屏蔽在微架构状态中。

第七章对商用中占主导地位的同一性协议：嗅探式同一性协议 (snooping cache coherence protocols)，进行讨论。在简单的模型，snooping 协议是比较简单的。当 cpu 核心遇到 cache miss 发生时，它就在共享总线上发起

仲裁，将其请求广播出去。共享总线的设计保证所有其连接的控制器接收到相同顺序的请求广播，这样，分布式的控制器就可以对他们各自的行动进行同步，进而能够保证全系统的状态是能够一致合理的。然而，现代系统中可能有多条总线，且总线不提供原子性处理请求的能力，总线内部还可能有仲裁请求暂存队列，并延迟、乱序发送请求，这些都使得 snooping 协议变复杂，也使得系统中可能出现更多种同一性瞬态。在结尾，第七章将实例学习 Sun UltraEnterprise E10000 和 IBM Power5。

第八章将深入讨论目录式同一性协议 (directory cache coherence protocols). directory 协议比 snooping 协议在多核可扩展性以及异构扩展性方面更有优势。计算机科学领域中有个笑话是，任何问题都可能通过增加中间层来解决。directory 协议有点这个意思：某一个 cache 发生 read miss，将从它的下一级 cache 中获取该内存数据，而下一级 cache 中有个目录监视着哪些 cache 现在占有着哪些内存数据。根据目录项中记录的内容，缓存控制器 (cache controller) 可能会直接回复请求者或者将请求消息继续转交给当前占有该内存数据的 cache。可以看到在 directory 协议中，每条消息都有明确目的地 (没有广播或多播动作)，但由于上述的转交操作，一次原始请求可能产生与内存操作节点成正比的请求消息，从而在系统中产生大量的瞬态。这一章先以一个简单的 diretory 协议开始，继而对其进行优化以使其能处理 MOESI 中的 E 和 O 状态，可以进行分布式操作，更小的等待时延，(limo) 近似 directory entry 表示，等等。该章还探讨了 directory 同一性协议设计本身的一些技术点，包括 directory 缓存技术。最后，我们进行几个实例分析，包括 (较旧) SGI Original 2000 和 (较新) AMD HyperTransport, HyperTransport Assist，以及 Intel QuickPath Interconnect (QPI) 。

第九章讨论一些同一性进阶主题。前面几章在讨论同一性时有意地将场景限制在了简单的系统，以方便进行一些根本性问题的讨论。第九章深入讨论在复杂系统模型下的同一性协议及其优化问题，这些问题是 snooping 和 directory 都会遇到的。章节开始的主题包括指令缓存 (instruction cache)，多级缓存 (multilevel cache)，写直通缓存 (write-through cache)，地址翻译缓存 (translation lookside buffer, TLB)，同一性内存直连 (coherent direct memory access, DMA)，虚拟缓存 (virtual cache)，和多级同一性协议。(limo) 最后，

章节对性能优化问题进行了讨论（比如针对 migratory 共享和 false sharing 进行优化）

1.3　小测验

你可能觉得自己对一致性和同一性有了足够的知识了解，没有必要继续再读本书了。为了验证你的感觉是否正确，我们准备了以下问题小测验。

- 问题 1：在 SC 模型的系统中，处理器核必须以程序指令顺序发送同一性请求。对错?（答案见 3.8）
- 问题 2：一致性模型定义了什么样的同一性请求操作顺序是正确的。对错?（答案见 3.8）
- 问题 3：在进行原子读-改-写 (比如 test and set)，处理器核总是需要与其它处理器核进行通信。对错?（答案见 3.9）
- 问题 4：在同时使用 TSO 模型和多线程处理器核（limo 超线程?）的系统中，
- 问题 5：程序员如果在高级语言一致性模型层面使用了恰当的同步机制，就无需再考虑内存系统上的一致性问题。对错?（答案见 5.9）
- 问题 6：在 MSI snooping 协议中，一个 cache 块只能处于三种状态中的一种状态。对错?（答案见 7.2）
- 问题 7：snooping 协议要求各个处理器核在总线上通信。对错?（答案见 7.6）

虽然答案在本书后面会揭晓，我们鼓励读者能在看答案前试着回答一下。

1.4　本书无法提供什么

本书是一致性、同一性入门。我们期望的是本书第 2-9 章的每一章，以及进阶材料，是一节 75 分钟研究生课程教学量。为此，我们无法讨论所有

一致性、同一性的问题，比如：

- 同步。同一性的目的是让缓存在程序功能上不可见。一致性的目标是使得系统中的所有共享内存像一个内存模块一样。但是程序员需要使用在必要的时候使用锁、barrier 及其它的同步机制来使他们的程序能够正确运行、可用。
- 商用弱序模型。本书没有详细讨论 ARM 和 PowerPC 的所有的内存序微妙行为，但我们会讨论使用什么的机制可以明确得到的什么样的内存序。
- 并发编程。本书不涉及并发编程模型，方法论及其工具。

第二章　同一性基础

本章我们对同一性进行必要的介绍，以能够让大家理解一致性和同一性是如何相互作用的。在 2.1 中我们先介绍我们将在本书贯穿使用的基础系统模型。为了使讲解更简明扼要，我们选择了最简单但又能将主要问题阐述清楚的系统模型。直到第九章我们再进行复杂系统模型下的一致性同一性讨论。2.2 对同一性问题进行了解释，这些问题是同一性所必须的解决的，并讨论了可能导致“不同一”的场景。

2.1　系统模型

在本书中，我们考虑具有多核处理器与共享内存的系统。每个处理器核都可以对所有（物理）内存位置进行读写操作。我们构造的系统模型中只含有一个多核处理器，一个与处理器不共芯片的内存，见图 2.1。这里的多核处理器是指有多个单线程核（即，不包括超线程等技术），每个核都有本地私有数据缓存，而后所有的核还共享有一个末级缓存 (last-level cache, LLC)。如非特殊说明，本书中使用缓存 (cache) 时指的是核的私有缓存，非 LLC。每个核都使物理地址对缓存访问，缓存类型是“写回”(write-back)。核与 LLC 之间通过互通网络来通信。虽然 LLC 缓存是处理器芯片上的组件，它其实可以看作是内存侧的缓存，因此它的引入并未对缓存同一性带来新问题。逻辑上，LLC 的作用只不过是内存的前端，降低内存访问的平均迟延，同时提升内存有效带宽。当然，LLC 还有一个作用是处理器片上的内存控制器。

我们使用的系统模型省略了很多常见的组件及特性，不过它们对本书的讲解无关紧要。这些组件包括：指令缓存 (instruction cache)，多级缓存

(multiple-level cache), 多核共享缓存, 虚拟地址缓存, 地址翻译缓存 (TLB), 内存直连等（DMA）等。模型中也不包括多个多核处理。在本书后面特性组件会被讨论，但暂时我们先忽略以避免引入不必要的复杂性。

2.2　“不同一”的产生

“不同一”的产生都是由于同一个根因：缓存、内存有多个操作者。在现代系统中，这些操作者是处理器核，DMA 引擎，以及一些可以对缓存、内存进行访问的外设。在本书我们主要关注的操作者是处理器核，但要知道实际中其它类型的操作也是存在的。

图 2.2 简单示意了不同一的产生。初始时，内存地址 A 的内存值为 42，核 1 与核 2 都将 A 读到了他们各自的私有缓存中。在时刻 3 时，核 1 将它私有缓存中的 A 的值进行了加 1 操作变为 43，这使得核 2 缓存中的 A 值过期，导致“不同一”。为了避免这种不同一，系统需要实现一套同一性协议来控制核 2 不能在核 1 观察到 43 的时候依然还观察到 42。同一性协议的设计与实现是第 7-9 章的主要内容。

2.3　同一性定义

图 2.2 中“不同一”的场景直观理解应该是属于“错误”的行为，有了直观感觉后，本小节中我们将给出同一性严格的定义。在各种教科书和文献中，同一性的定义有多种多样。我们不打算全部讲解它们，而是给出我们偏向于使用的一个定义，因为它点明了同一性的要旨。在阅读角中，我们讨论一点其它的同一性定义，以及它们与我们的定义的关系。

我们对同一性的定义的基础概念是单写多读（或单写者多读者，single-writer-multiple-reader, SWMR）不变式。对任一内存地址，在任一时刻（注 1），要么只有一个核可以写入（当然这个核也可以读），要么有多个核在进行读取。所以在任一时刻，不能一个核在写，而有另外一个核在读或写。上述描述也可以换一种方式理解：将内存地址的“生命周期”用代 (epoch) 来表征。在一个 epoch 里，要么只有一个核有读写访问权限，要么有多个核

（也可能是 0）同时有只读访问权限。图 2.3 给出了保持 SWMR 不变式的包含 4 个代际的内存地址生命周期变化。

除了 SWMR，同一性还要求内存值的广播是正确的。简单解释下内存值正确性为什么是必需的。比如上面图 2.3 中，虽然 epoch 上 SWMR 不变式得到了保持，但在第一个只读 epoch 中核 2 与核 5 如果对该内存读取到不同的值，那么系统将是“不同一”的。类似地，如果核 1 没有获取到核 3 在该内存 read-write epoch 的时候写入的值，或者核 1，2，3 没有获取到核 1 在该内存 read-write epoch 中写入的值，系统也将进入“不同一”状态。

因此同一性定义需要在 SWMR 不变式的基础上加入值的某种不变式才能完整，这个不变式是关于内存值是如何从一个 epoch 向下一个 epoch 传播：在一个 epoch 开始时，某内存位置的内存值需要等于它最近一次在 read-write epoch 结束时的值。

这些不变式其实有一些不同的等价描述，其中有个著名的是将 SMWR 用令牌来描述。描述如下：对每个内存地址，都配有至少与处理器核数量等量的令牌。对一个内存地址，如果一个核拥有其全部令牌，它可以对内存地址进行写操作，如果一个核有一个或多个令牌，它可以进行读操作。通过这样的设置，可以保证在任一时间，不可能有一个核在对某地址写入而其它核在读或写该地址。

注 1：SWMR 不变式只要求在逻辑时间 (logical time) 域上被遵守，而是物理时间。这个微小的区别使得我们可以做很多优化，有的优化看起来好像破坏了不变式，但实际没有。关于优化部分我们将推迟到后面的章节讲解，如果读者对逻辑时间不熟悉不要担心。

同一性不变式

1. **单写多读不变式**。 Single-Write, Multiple-Reader(SWMR) invariant。对任一内存地址 A，在任一（逻辑）时刻，要么只有一个核可以写入（这个核也可以读），要么有不定数量的核可以读取 A。
2. **值不变式**。在一个 epoch 开始时，某内存位置的内存值需要等于它最近一次在 read-write epoch 结束时的值。

2.3.1　同一性不变式的保持

在前面小节中，我们通过同一性不变式对同一性协议是如何工作进行了直观的介绍。实际上大部分同一性协议的设计内容就是在维持这些不变式，这类协议称为“使无效协议”(invalidate protocol)。一个处理器核在读取某内存位置内容时，它会先发消息给其它核来获取当前该内存位置的值，同时也是为了确保不存在其它核对该内存位置的缓存正处于读写状态。该消息会终止（使无效）所有对该内存位置活跃的读写 epoch，并开始一个新只读 epoch。如果一个核要向某内存位置写入数据，若它还未持有该内存位置的一个可用只读缓存，它将发送消息给其它核来获取当前的内存值，并确保不存在其它核对该内存有合法的只读或读写的缓存状态。这样的消息会终止（使无效）所有对该内存位置正在活跃的读写或只读 epoch，并开启新的读写 epoch。本书第 6-9 章会针对“使无效”类协议上面的这些笼统操作展开详述，但直观表现上它们是一样的。

2.3.2　同一性的粒度

处理器核能够以不同粒度进行读写，一般粒度从 1 字节到 64 字节不等。理论上，同一性协议可以在最小的读写粒度上进行，但实践中同一性协议一般在缓存块 (cache block) 的粒度进行。换句话说，处理器硬件一般提供的是缓存块的同一性。而 SWMR 的操作对象也从对某内存“位置”变成了某内存“块”，也就是说在现实系统中，一般不会发生一个核对某内存块的首字节在写，而另一核同时在对该内存块的中间字节在写入。尽管基于“块”的同一性是普遍的，而且也是我们本书中使用的模型，我们还是要知道更小粒度或更大粒度的同一性协议都是存在的。

> 同一性的类“一致性”描述
>
> 上面我们对同一性定义所使用的不变式是关于处理器核在何种条件下能够拥有何种访问权限，以及数据如何在核间传播的。除此外，还有另外一类对同一性的定义，它的侧重点是读写操作关系本身，有点类似于一致性模型对可感知的读写序行为的定义。

这一类描述中，有一种描述与顺序模型 (SC) 有相通处。我们在第三章中将深入讨论 SC 模型，它要求系统的内存变化过程如同所有线程的所有内存读写操作按照程序指令序以某种时域交错执行的一样（线程间交错，线程内遵守程序指令序），所以它是一种时序上的“全序关系”模型。在 SC 系统中，全序关系中的任一读操作的结果是序关系中最近的一次写操作的结果。用类“SC”的描述方法定义的同一性是这样的：一个具有同一性的系统中，某特定内存位置的变化过程如同所有线程各自按程序指令序对该位置进行读写，并线程间以某种时域交错执行一样。可以看到该定义指出了一致性与同一性的重要区别点：同一性描述的是特定内存位置值的变化过程，而一致性描述不同内存位置值的变化过程尤其不同位置值变化的时间关系。

还有一种对同一性的定义也是使用了两个不变式：(1) 每个写操作最终都会被所有核观察到 (2) 对同内存位置的写操作是串行进行的（即，所有核观察到的该内存位置值的变化过程是相同的）。IBM 在 Power 中使用了类似的定义以使得其硬件实现更加容易，因为该模型允许一个核对某内存位置的写操作先通知到一些核（这些核可以通过读操作观察到值的变化），而后通知到另一些核。

另外一种同一性定义是 Hennessy 与 Patterson 所使用的，它有三个不变式：(1) 内存位置 A 的读返回的该核最近一次写入的值，除非在该核上次写与本次读之间其它核对该位置进行了写操作 (2) 一个核对 A 位置进行了写操作 S，足够长时间后，另外一个核对 A 位置进行读操作将返回 S 写入的值，除非在 S 和这次读之间还有其它写入操作 (3) 对同内存位置的写入是串行的（与前一种定义的不变式 #2 相同）。这组不变式集合的描述很直观，但其中“足够长时间后”可能会引起一些问题，因为它不是一个精确的术语。

上面这些“类一致性”定义和我们在 2.3 中使用的定义是等价的，它们可以用来判断给定的同一性协议是否足以使一个系统“同一”。一个正确的同一性协议会满足所有的定义形式。但是类一致性描述不能给同一性协议设计者太多的直观理解，他们在设计协议的时候考虑的

是规范处理器何时、如何可以访问内存，所以我们认为 2.3 的定义对他们来说更具有启发性。

2.3.3　同一性的作用范围

同一性的定义-无论我们采用哪种定义-都是有特定适用范围的，设计必须清楚什么时候需要考虑同一性、什么时候不需要。我们讨论两个重要的适用范围：

- 同一性适用于所有持有共享内存块的存储结构器件。这些器件包括 L1 数据缓存，L2 缓存，LLC，和主存，另外还包括 L1 指令缓存和地址翻译缓存（TLB）。
- 同一性不属于架构组件（即，同一性不是架构可感知元素）。严格来说，一个系统即使处于“不同一”的状态，只要遵守它的一致性模型要求，那么它依然是“正确”的（因为正确本身就是一致性定义的）。这个问题可能有点玄乎（有点难想象一个现实系统遵守一致性而未保持同一性是怎样的），但它有个非常重要的推论：内存一致性模型对内存同一性模型没有任何约束。尽管如此，如第 3-5 章将讲述的一样，许多一致性模型的具体实现依赖了某种同一性模型来保证其正确性，这也是我们先在本章进行同一性讲解，再后面章节讲一致性的原因。

2.4　参考文献

(limo)

参考文献

[1] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. PhD thesis, Computer System Laboratory, Stanford University, Dec. 1995.

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15–26, May 1990.

[3] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, fourth edition, 2007.

[4] IBM. Power ISA Version 2.06 Revision B. http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf July 2010.

[5] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003. doi:10.1109/ISCA.2003.1206999

第三章　一致性的由来以及顺序一致性模型

本章深入讨论内存一致性模型。一致性模型定义了拥有它的共享内存的“正确”行为，这样程序员就可以依赖此并对程序执行的结果有预估，实现者也可以依此进行有目的的设计。我们先对一致性模型的由来进行介绍(3.1)，然后讲解一致性模型包括哪些内容（3.2)，最后比较一致性与同一性(3.3)。

然后，我们介绍下（相对而言）比较直观的一致性模型：顺序一致性模型（sequential consistency, SC)。SC 非常重要，因为大多数程序员对共享内存的行为期待其实都是 SC，它也是理解更宽松的一致性模型（弱序一致性模型，接下来两章将进行讨论）的基础。我们先介绍 SC 的核心思想（3.5)，再讨论几种 SC 的实现，包括初级实现（limo，3.6，operational mode)，有缓存同一性的简单实现 (3.7)，有缓存同一性的优化版实现（3.8)，和原子操作实现（3.9)。最后我们以 MIPS R10000 为实例讨论 SC，然后提供一些深入阅读学习建议。

3.1　共享内存带来的问题

让了使大家明白“定义共享内存的正确行为”这件事的必要性，我们看表 3.1 中的例子（如非特殊说明，本书使用的例子都假设所有变量的初始值为 0)，表中是两个处理核的执行流。大多数程序员都期待核 C2 的寄存器 r2 获取的值是 NEW。然而，在现今有些系统中，r2 的值可能是 0。

之所以 r2 可能获得值 0，是因为在硬件层面 C1 的写操作 S1 和 S2 可以乱序执行。从本地视角看（即只关注 C1 的执行，不考虑其与其它核的交互)，这种乱序看起来是正确的，因为 S1 和 S2 访问的不同的内存位置。正

面的 sidebar 列举了一些硬件可能乱序的内存访问操作，包括上面的写-写操作。非硬件专家最好心中有数，这些乱序是可能发生的（即，缓存的写操作 buffer 不是写入写出的）。

由于 S1 与 S2 的乱序，系统的执行序列可能是 S2，L1，L2，S1，如表 3.2 表示。该乱序执行依然满足同一性，因为 SWMR 未被破坏，所以同一性不是该问题的原因。

阅读角：内存操作的乱序（或重排序）

本阅读角列举了现代处理器核心对不同内存地址的操作可能乱序执行的方式。对硬件概念不熟悉的读者可以暂时跳过，后面再回来阅读。现代处理器的核心可以乱序执行多个内存访问操作，但以两个内存操作的序关系对核心概念的讲解就足够了。绝大多数情况下，我们只需要对不同地址的内存访问的序进行讨论，因为 SC 模型一般要求一个处理器核中对同一地址的内存操作是以原程序指令顺序进行的。根据内存操作的读、写类型不同，我们将可能的乱序分解为以下有三种：

写-写乱序（store-store reordering）。如果处理器核的写操作 buffer 不是先入先出（FIFO）的，那么对两个不同地址的写操作将有可能被乱序，与他们在原程序中的指令序产生颠倒。这种情况可能发生在前面的写操作产生了缓存未命中，或者后面的写操作与更前面的写操作合并操作了。(limo，跳过一句)。不同内存地址之间的操作乱序对单线程的执行是没有影响的。但在表 3.1 的多线程执行的例子中，核 C1 的乱序执行让核 C2 先于 data 的变化观察到了 flag 变为 SET 的变化。这种乱序即使使用最完美的同一性（如严格地立即值同步的内存）也无法修正。同一性只是让 cache 的缓存效果不被感知，但这里的写操作已经乱序了。

读-读乱序（load-load reordering）。如前所述，现代动态指令调度的处理器核可能以不同于代码指令的顺序执行程序，即乱序执行。在表 3.1 中，核 C2 可能乱序执行读操作 L1 和 L2。对单线程程序而言，这种乱序不会引发什么问题，因为它们指向不同的位置。但在多线程场景下，如表 3.1，核 C2 的乱序可能导致与核 C1 乱序类似的结果，

比如系统的全局序列为 L2，S1，S2，L1，那么 r2 的值也会是 0。如果把判断 B1 去掉，L1 和 L2 之前不再有条件依赖，这种情况会更明显。

读-写乱序（load-store reordering）和写-读乱序（store-load reordering）。来自同一线程的读操作和写操作也可能被乱序（如果它们指向不同的内存位置）。读-写乱序可能导致很多奇怪的（看似不正确）的行为，比如可能导致保护资源的锁释放后再对该资源进行读取。表 3.3 则示例了写-读乱序。如果核 C1 的 S1、L1 及核 C2 的 S2、L2 都乱序，则会产生非常反直觉的结果：r1 和 r2 都为 0。即使处理器核按照指令原顺序执行，写-读乱序依然可能在有 FIFO 式写缓存机制时发生。

读者可能默认硬件不应该让上面的行为或某种行为发生，但在未完整了解硬件真正允许的行为前，不能这样默认。

让我们再看另外一个例子，该例子由 Dekker 互斥锁算法启发而来，如表 3.3 中所示。执行完毕后，r1 和 r2 中的值可能是多少？直觉上，有以下有三种可能的值组合：

- (r1, r2) = (0, NEW)，对应执行序列为 S1, L1, S2, L2
- (r1, r2) = (NEW, 0)，对应执行序列为 S2, L2, S1, L1
- (r1, r2) = (NEW, NEW)，可能执行序列为：S1, S2, L1, L2

令人意外的是，大多数现实处理器，包括 Intel 和 AMD 的 x86 系统，为了提高性能它们了使用先入先出 FIFO 式写缓冲区，这将导致有可能产生值组合 (r1, r2) = (0, 0)。这类似表 3.1 中的乱序问题一样，虽然执行过程都遵守了同一性，依然可能产生结果 (r1, r2) = (0, 0)。

可能有的读者可能觉得上面的例子是不好的编程范式，它的结果本来具有不确定性。但实际上，现代的多核处理器在默认执行下都具有不确定性，我们所知道的所有架构都允许并发线程间产生多种可能的时序交错。确定性执行可能需要（但不是所有时候）是通过在程序中加入适当的同步机制来实现的。因此，我们在定义共享内存的行为时，一定要考虑并发模型的天然不确定性。

而且，内存行为的定义是要考虑所有的程序指令的所有可能的执行序列，即使这些程序本身是错的或故意写得很微妙（如使用非阻塞的同步算法）。但在第 5 章，我们会看到，一些高级语言有时对一些代码的执行是未定义的，比如对可能产生数据竞争（data race）的代码。

3.2　什么是内存一致性模型

上一节的例子展示了共享内存的行为是很微妙的，并通过不同的结果值来说明程序员应该对结果有怎样的期望，以及系统的硬件实现者可以进行可能优化。内存性一致性模式就是用来给程序员期望、给实现者以约束指导。

内存一致性模型，或一致性模型，是对多线程使用共享内存时所允许发生的内存行为的规范详述。一个多线程程序接收某特定输入后，其运行过程中的某条读取指令可能返回什么结果，内存的最终状态可能是什么，皆由内核一致性模型约束。与单线程的场景不同，多线程执行的结果本身就具有多样性、不确定性，这一点我们将在顺序一致性模型的介绍中再次看到（3.4 节及后续章节）。

一般来说，特定的内存一致性模型 MC 提供了规范可以将程序的执行过程区分为遵守、不遵守 MC 两类。这种对程序执行过程的划分反过来对硬件的实现也可以起到分类作用。遵守 MC 的硬件实现上只会产生遵守 MC 的程序执行过程，而非 MC 的硬件上就会产生非 MC 的执行过程。

最后，本书到现在讨论中我们对编程语言的层级一直比较模糊。这里我们先假设程序是硬件指令的集合体，然后我们还假设内存位置的访问是通过物理地址进行的（即，我们暂不考虑虚拟地址和地址翻译）。在第 5 章中，我们会讨论高级语言（HHL）的一些问题，比如，我们会看到变量的寄存器分配机制对高级语言的内存模型的影响有点类似于硬件对程序内存指令的乱序（或重排）。

3.3　一致性与同一性

第二章中我们对同一性通过两个不变式进行了定义，这里再简单回顾下。SWMR 不变式要求在任意（逻辑时间）时刻，一个内存位置要么只能有一个核对其进行读写操作，要么多个核可以对其读操作。数值不变式要求多核间的数据传递是正确的，这样一个核中缓存的数值总是最近一次写入的值。

看起来好像同一性定义了共享内存的行为，但其实并没有。原因是：

- 缓存同一性的目的恰恰是为了使缓存不可见，在多核运行时像在单核 cpu 中一样不产生功能可见的行为差异。如果缓存不可见了，还有什么行为需要定义?
- 同一性一般是针对一个缓存块的，而对多个缓存块的行为交互关系，它未涉及。实际程序会涉及大量的缓存块的读写。
- 内存系统可以没有同一性，甚至没有缓存。

虽然同一性不是必须的，很多内存系统在实现一致性的时候用到了同一性。即使如此，同一性还是可以与一致性解耦的，我们也认为这样是很有用处的。为此，在本章和接下来的两章中，讨论一致性实现的时候 “同一性” 将如同一个子函数调用。比如我们将使用 SWMR，但不必关心它是如何实现的。

总结一下：

- 同一性不等于一致性
- 一致性的实现可以像黑盒一样使用同一性

3.4　顺序一致性模型基础

最直观的一致性模型可能就是顺序一致性模型了（sequential consistency, SC）。SC 是由 Lamport 首次形式化地给出正式定义。Lamport 首先定义了单核的 SC：如果一次单核执行的结果等于其中的指令操作如代码序一样顺序执行的结果，我们称这个核是 SC 的。在此基础上，继 Lamport 继

续提出了多核 SC：一次多核执行的结果等于多核的所有指令以某个全局序列在单核 SC 的模型在运行，且在该全局序列中每个核的指令以其原指令序排列。这个全局指令序（全局序）称为“内存序”。在 SC 中，内存序遵守每个核的指令序，但其它的一致性模型中，全局序有可能不遵守每个核的指令序。

图 3.1 展示了表 3.1 一种可能执行序列。中间的箭头代表了内存序(<m)，而两侧的箭头代表每个核的原始指令序。我们用 <m 表示内存序，op1 <m op2 的含义为在内存序中 op1 在 op2 之前。相似地，我们用 <p 来表征一个核的指令序，op1 <p op2 的含义为在该核的指令序中 op1 发生于 op2 之前。在 SC 中，内存序遵守每个核的指令序。“遵守” 的含义是 op1 <p op2 可以导出 op1 <m op2。在注释中（/*...*/）的值是读取或写入的值。上图中的执行以 r2=NEW 作为结束。更进一步，表 3.1 中的程序的所有可能执行都是以 r2=NEW 结束。这其中的不确定性–在 L1 指令读取到 SET 之前所需要经历的循环次数–是不重要的。

上面的例子展示了 SC 模型中变量值的变化。在章节 3.1 中，如果你期待 r2 必须为 NEW，你可能独立发明了 SC 模型，可能没有 Lamport 定义的那么精确。

图 3.2 中对 SC 模型中值的变化进一步阐释，共列举了表 3.3 中四种可能的执行序列。图 3.2（a-c）展示了遵守 SC 模型的执行序列，也是符合直观常理的，分别对应对最终状态为：(r1, rw) = (0, NEW)，(NEW, 0) 和 (NEW, NEW)。需要说明的是，图 3.2(c) 中展示的只是四种可能导致最终状态为 (r1, r2) = (NEW, NEW) 的序列的一种，即 {S1, S2, L1, L2}，其它 3 种为 {S1, S2, L2, L1}，{S2, S1, L1, L2}，和 {S2, S1, L2, L1}。因此，总共有 6 种遵循 SC 模型的执行序列。

图 3.2(d) 展示了一种不遵循 SC 的执行，它导致结果为 (r1, r2)=(0, 0)。无法构造一个遵守指令序的内存序来产生这样的结果。指令序上要求：

- S1 <p L1
- S2 <p L2

而要产生这样的结果，内存序要求：

- L1 <m S2（这样 r1 才能为 0）
- L2 <m S1（这样 r2 才能为 0）

同时满足这两组约束会产生一个环，即无法在现实中构造这样一个序。图 3.2(d) 中的弧线展示了这个环。

上面讲了 6 种遵循 SC 模型和一种不遵循 SC 模型的执行。这可以帮助我们理解什么是“符合 SC 模型的系统实现”（简称“SC 实现”）：SC 实现允许 6 种执行序列中的一种或多种的存在，但决不允许第 7 种的存在。

我们也可以在上面的例子中看到一致性与同一性的一个关键不同点。同一性适用于同一个 block 的行为，而一致在是不同 block 的基础上定义的。（预告一下第 7 章的内容，我们可以看到 snooping 同一性协议的系统保证了对不同 block 的同一性请求的一种全局性顺序关系，虽然同一性本身仅仅要求对同一个 block 的同一性请求的全局顺序关系。这看起来有点过度约束，但这对 snooping 同一性协议的系统支持 SC 一致性模型是需要的。）

3.5　SC 形式化分析初步

本节进一步精确地定义 SC，这样可以更利于我们将 SC 与接下来要介绍的更弱的内存模型做对比。我们使用 Weaver 和 Germond 的形式化表示法：L(a) 与 S(a) 分别表示对内存位置 a 的读取与写入，<p 与 <m 分别用来分别表示指令序和内存序。<p 是每个核视角而言所看到的进行内存操作的顺序关系（即指令序），<m 是全局内存序中各个内存操作的顺序关系。

一个 SC 模型的执行序列需要满足：

1. 所有核将各自的内存操作按各自的指令序汇总到内存序中，不管它们访问的内存位置相同与否（即 $a = b$ 或 $a \neq b$）。有四种一一对应关系：
   - L(a) <p L(b) ⇒ L(a) <m L(b) /* Load → Load */
   - L(a) <p S(b) ⇒ L(a) <m S(b) /* Load → Store */
   - S(a) <p S(b) ⇒ S(a) <m S(b) /* Store → Store */
   - S(a) <p L(b) ⇒ S(a) <m L(b) /* Store → Load */

2. 每个读取操作获得的结果是内存序中对该位置最近一次写入的值。即: L(a).v = MAX$_{<m}${S(a) | S(a) <m L(a) }.v, 其中 MAX$_{<m}$ 表示 “内存序中最近一次”。

我们将在 3.9 节中深入讨论的原子指令” 读改写” （read-modify-write, RMW）操作对执行序列有进一步的约束。每个 test-and-set（比较并写入）指令要求用于 test 的读取操作和用于 set 的写入操作之间在内存序中逻辑上是连贯的，即：它们之间没有被插入对相同内存位置或不同内存位置的内存操作。

我们将 SC 模型的内存序要求总结一下，列在表 3.4 中。这个表展示在内存模型下什么样的指令序得到了保持、遵守。比如一个核的指令序列中一个读操作在写操作之前（即图中 Operation 1 为读，Operation 2 为写），那么表中的 “X” 即代表这些操作在全局内存序中需要被保持。在 SC 模型中，所有的指令序中的内存操作顺序需要在最终的内存序中被保持；在其它模型中，比如接下来的两章中，有的指令序将不一定被保持（即表 3.4 中有的项不为 “X”）。

一个 “SC 实现” 将只允许满足的 SC 执行序列。严格地说，这是 SC 实现的安全性所在（不引入破坏）。SC 实现还要允许一定的自由度。SC 的实现至少允许一个程序的一种 SC 执行序列。另外防饿死和公平性也是要考虑的，但这超出本书讨论范围。

3.6　简单 SC 实现

有两种 SC 的模型的简单实现，可以进一步让我们理解 SC 模型所允许的执行序列。

单核多线程实现

首先，在一个保序执行的单核上，我们可以为多线程用户程序实现 SC 模型。线程 T1 在核 C1 上执行直到上下文切换到线程 T2。在上下文切换到新线程前，旧线程运行产生的内存操作必须全部完成。这样的模型很容易可以推导得知是符合 SC 模型的。

开关实现

其次，使用多个核 Ci, 一个内存开关，一个内存（如图 3.3），我们可以实现一个 SC 模型。我们假设每个核一次最多通过开关按指令序进行一次内存操作。每个核可以内部使用任何方法进行优化，只要该优化不会打乱通过开关访问内存的顺序。比如核内部可以使用带有分支预测功能的 5 级保序流水线。

开关会选择一个核，让内存响应它的读写请求，然后重复这种选核-响应的操作。开关可以使用任意不至于饿死某 cpu 的策略（如随机策略）来选核。这种实现方式在建构上就满足了 SC 模型的要求。

评估讨论

上面的简单实现的好处是从操作上定义了（1）被允许的 SC 执行序列（2）SC 实现的黄金法则。开关实现还说明了 SC 实现可以在没有 cache 和同一性的情况下实现。

但不好的一点是，这些简单实现的多核扩展性较差。随着核数的增加，上面两个例子的独占式使用 cpu 或开关的操作成为性能瓶颈。这种性能瓶颈会让人有的人误以为 SC 模型无法支持并行，我们在下文将看到其实不是这样的。

3.7　具有缓存同一性的简单 SC 实现

缓存同一性可以使 SC 模型支持无冲突内存操作–两个内存访问如果访问的内存位置相同，且至少一个为写时，我们称他们是冲突的–的完全并行执行。而且，这样的系统概念上是简单的。

在这里，我们先把同一性看作似乎一个黑盒，它的特性是实现第二章的 SWMR（单写者多读者）不变式。这里以 L1 缓存为例给出一些比较直观的、基本的实现细节，比如：

- 使用状态 M（modified）表示某个缓存块 block 正在被某个核进行写（同时可读）操作
- 使用状态 S（shared）表示某个 block 可以被一个或多个核进行读操作
- GetM 和 GetS 用来表示同一性请求，分别表示请求一块正处于 M 和 S

状态的 block

正如将在第 6 章中讨论的，我们不需要对同一性的实现有深入了解。

图 3.4（a）是图 3.3 中 SC 实现的改版，图 3.3 中的开关和内存被一个具有缓存同一性的黑盒内存系统所替代。每个核以其指令序向该内存黑盒提交其内存操作。对同一个核，该内存系统会在完成上一个请求后才开始处理下一个请求 (即串行)。

图 3.4（b）稍微深入打开黑盒内存系统的内部，可以看到每个核与其私有的 L1 缓存相连。该内存系统可以在缓存块 B 的同一性权限状态允许时响应针对缓存块 B 的读写请求（比如 M 和 S 状态可以响应读，M 可以响应写）。而且，该内存系统还可以同时响应多个核，只要 L1 缓存的权限状态允许。图 3.5（a）示意了四个核在准备做内存操作前的缓存状态。这四个内存操作是“无冲突”的，可以被它们各自的 L1 缓存直接响应答复，所以它们可以并发地执行、完成。如图 3.5（b）所示，我们可以在全局序中任意排布这几个操作，而且保持结果是遵守 SC 模型的。更一般地，L1 缓存可以响应的操作都可以并发进行，因为同一性的单写多读者不变式保证了它们不会“冲突”。

评估讨论

我们创造了一种 SC 的实现：

- 完全地利用了缓存的时延与带宽优势
- 可以达到其使用的同一性协议一样的扩展性
- 将实现核的复杂性与实现同一性的复杂性解耦

3.8　优化的具有缓存同一性的 SC 实现

大多实际的 cpu 核的实现比上面的简单 SC 实现要复杂的多。cpu 核可能会使用预取、投机执行、或超线程等技术来提升性能，降低总访问时延。这些技术都是与内存系统界面打交道，这里我们讨论下这些技术如何影响 SC 实现。

非绑定式预取（也叫缓存预取）

对一个块 B 进行非绑定式预取请求，实际是对该块的缓存同一性状态进行更改。一般预取请求由软件、cpu 硬件逻辑、或缓存硬件逻辑发出，用以响应对某缓存块 B 的读取（比如将目标缓存块状态转变为 M 或 S 状态）或写入（比如将目标缓存块状态转变为 M）操作，请求有 GetS 和 GetM。重点是，非绑定式预取绝不会修改 cpu 寄存器状态（即不会直接将值生效到寄存器上）或缓存块的值。非绑定式预取的效应被约束在具有缓存同一的内存块中（如图 3.4 所示），这样，预取对一致性模型的功能影响是透明无感的。只要读写操作以指令序进行，同一性状态的更改顺序是无关紧要的。

非绑定式预取可以不影响一致性模型。这对缓存内部预取（如流缓冲区）和更激进的 cpu 核特性都有帮助。

译者注：非绑定式预取只将缓存块提前预备到需要的状态，数据还在缓存中，缓存状态和内存值也可能再次变化，真正读写时再实际操作，所以仅仅是缓存的准备，故又名**缓存预取**。而绑定式预取是地预取时即确定了数据值作为后续待用，绑定式预取会有数据过期的问题。

可投机执行的处理器核心

考虑这样一个核，它按指令序执行指令，但同时可能做分支预测执行，包括分支中的读写操作，但在分支预测失败时这些效果会被压制、回滚。这些被回滚的读写操作可以做到像缓存预取一样对 SC 模型的正确性透明无感，如同从未发生过。比如：分支预测后面的读取操作被发送给 L1 缓存，该操作可能触发 miss（导致 GetS 语义的预取）或者命中并返回该值到寄存器中。如果该读取操作被回滚，该核会放到所有的寄存器更新，擦除由于读取带来的功能可观测的影响，就像读操作未曾发生过一样。但缓存不会撤消缓存预取的操作效果，因为这是不必要的，而且预取的块还有利于后面该读取操作重新执行时的效率。对写操作而言，cpu 核可能提前发出 GetM 请求，但 cpu 核不会向缓存提交写操作，直到该写操作必然要发生了。

问题 1 回顾：在 SC 模型的系统中，处理器核必须以程序指令顺序发送同一性请求。对错??

答案：错！cpu 核可以按任意顺序发送同一性请求。

动态调度的处理器核心

现代很多 cpu 核可以不按程序指令序，动态调度指令执行，从而获得比静态调度（即严格按照指令序执行）更高的性能。具有该特性的单核处理器必须要维持数据依赖的正确性。但在多核的场景下，动态调度可能引入一个新问题：内存一致性的投机。比如有一个核动态重排序两个读指令的顺序，Load1 和 Load2（比如 Load2 所使用的内存地址先于 Load1 得到）。很多核会投机性地先执行 Load2，然后该核认为这对其它核是无感的，但这可能会违背 SC 模型（从而导致“有感”）。

采用 SC 一致性模型的处理器需要对其投机行为的正确性有验证能力。Gharachorloo 等人提供了两种验证方法。以上面两个读取操作 Load1 Loa2 为例。第一种：当核对 L2 投机执行后，当真正提交给 Load2 命令作为结果时，核需要检查 Load2 所访问的缓存块未离开本核缓存。只要缓存块还在缓存中，其数据值就不可能在投机执行与真正提交之间的时间段发生过变化。cpu 核可以通过跟踪 Load2 所访问的地址，将其与被换出的缓存块以及接收到的同一性请求进行监视比对。如果收到一条针对该缓存块的 GetM，则说明本次投机执行需要作废。

第二种验证方法是当核在真正提交读取的结果时再走一遍投机执行的读取过程（有点像“验算”）。如果这次“验算”的结果与之前投机执行的结果不同，则说明投机执行结果错误，需要抛弃。在本例中，如果对 Load2 操作的验算结果与当初投机执行得到的结果不同，那么说明 L1 和 L2 的读取操作的乱序将造成可观察的执行顺序变化（破坏 SC 一致性），所以需要被抛弃。

动态调度处理器中的非绑定式预取能够动态调度的处理器核很可能会不以指令序来处理读取和存储操作。比如，指令序是读 A，写 B，写 C。cpu 核可能会先发出对 C 的 GetM 非绑定式缓存预取请求，然后再并行地发出 GetS A，GetM B。SC 一致性不会受缓存预取的顺序影响。SC 一致性仅仅要求读取和写入操作（在效果表现上）以指令序访问 L1 缓存。同一性要求 L1 缓存处于合适的状态以接收读取或写入操作。

SC（或其它任意内存模型）的重点是：

- 规定了读写操作（在效果体现上）应用到同一性内存系统的顺序，但

- 没有规定同一性维护活动的顺序

问题 2 回顾：一致性模型定义了什么样的同一性请求操作顺序是正确的。对错?

答案：错!

多线程 (to do)

3.9　SC 模型下的原子操作

要想多线程程序，程序员需要能够在线程间做同步，这些同步一般要涉及成对的原子操作。同步功能就是基于一类“读取-修改-写入”（read-modify-write, RMW）的原子指令来完成的，比如”test-and-set”，“fetch-and-increment”，和“compare-and-swap”。这些操作对于类似自旋锁或其它锁原语是至关重要的。对自旋锁，程序员使用 RMW 指令来原子地读取并判断锁的值是否处于开锁状态（比如值为 0），然后将锁锁上（比如将值写为 1）。为使得这一系列动作是原子的，RMW 的读取和写入操作必须在效果表现像如同在 SC 的全局内存操作中是连续不间断进行的。

在微架构中，实现原子操作在概念上是很简明的。但简单的设计可能会导致原子指令低下的性能。一种正确而又简单的设计是：在进行原子操作时，cpu 核锁住内存系统（即，阻止其它核进行内存访问），然后进行 RMW 操作。这种实现简单，但牺牲了性能。

一种稍微激进的原子操作实现可以利用这样一个洞察：SC 模型仅仅规定了在表象上关于所有内存操作的全局内存序的合法形式。所以，cpu 核执行 RMW 的过程可以是先将缓存准备至 M 状态（如果该核的对应缓存块之前没有在 M 状态下的话）。之后，该核就可以在本地实行读写操作而不需要发送什么同一性请求，直到 RMW 中的 W 执行完，该核才响应关于该缓存块的同一性请求。这种操作不引发什么死锁风险，因为 W 操作确保会完成。

问题 3 回顾：在进行原子读-改-写 (比如 test and set)，处理器核总是需要与其它处理器核进行通信。对错?

答案：错！

更激进的 RMW 实现优化版本可以在不破坏原子性的前提下容忍读取操作和写入操作之间更长的时间间隔。设想这样的场景：缓存块起始处于只读状态。RMW 的读取部分操作可以先行投机执行，同时缓存控制器发出同一性请求来更新缓存块的状态为读写。当缓存块更新为读写状态后，RMW 的写部分再进行执行。只要核对外表现维持了一种原子性的“表象”，其实现就可以称为“正确”的。为了上面这种原子性“表象”是否正确，核必须要有能力判断缓存块是否在读写之间被变换过，这个针对 RMW 投机执行的验证支持与验证 SC 下的投机执行正确性是一样的。

3.10　综合实例：MIPS R10000

MIPS 10000 提供了一种比较纯粹的商业范例，它使用了 SC 一致性，同时带有提供缓存同一性能力的内存结构。这里我们重点看 R10000 在内存一致性方面的实现。

R10000 是一个四路超标量 RISC 处理器核，具有分支预测及乱序执行的能力。它在片上还带有写回模式的 L1 指令缓存和 L1 数据缓存，以及可在片外连接共享 L2 的接口。

该芯片片上系统总线支持四个处理核的缓存同一性，如图 3.6 所示（借自 Yeager[18] 的图 1）。为了同时运行更多的 R10000 的处理器核，如 SGI Origin 2000（将在 Section 8.8.1 详细讨论），系统设计者实现了基于目录的同一性协议，使得 R10000 能够通过片上系统总线和一个特殊的 Hub 芯片连接到一起。无论哪种架构，R10000 的处理器核看到的具有缓存同一性的内存是一部分在片上，一部分不在片上的。

在执行时，R10000 核会（可能投机地）以指令序将读取写入操作发送往一个地址队列的结构中。读取操作会投机地返回它之前的最近一次写入的值，如果没有这种写入操作，则从数据缓存中加载。读取和写入最终都是以指令序进行提交的，然后会从地址队列中移除。在最终提交写入操作时，目标缓存块必须在本核 L1 cache 中处于 M 状态，准备好的写入值必须原子

地写入到缓存中。

重要的是，当一个缓存块由于同一性请求变为不可有状态或者为其它块腾空间被换出时，地址队列中对该缓存块的读取操作及其后的所有操作的投机执行需要被抛弃，然后重新执行。所以，当一个读取操作最终完成时，它所需要的缓存块会在读取执行和读取完成的时间区间内不断地加载到缓存中，这样当最终提交时，它返回的结果就好像是提交时刚执行的一样。又由于写入操作是在提交时才真正写入缓存的，所以 R10000 在逻辑上以指令序将读取和写入操作发送给同一性内存系统，也就是实现了一个上面所述的 SC 内存模型。

3.11　3.11 SC 深入阅读

正面我们从海量的关于 SC 的文献中精选一些供大家阅读参考。

Lamport 定义了 SC 模型。所我们所知，Meixner 和 Sorin 第一个证明了一个系统以指令序向同一性内存系统发送内存操作就足以实现一个 SC 模型，尽管这个结论有一段时间让人们感到不可思议。

SC 可以与数据库的序列化作个对比。此二者的相似处在于他们都要求来自不同操作者的请求需要串行序列化地作用在全局状态上。二者的不同在于它们对各自操作和全局状态有各自特定的预期。在 SC 中，一个操作是一次对易失状态的内存访问，操作不会失败。在数据库序列化中，一个操作是一个可能读写多个数据库项的事务，这个事务需要满足 ACID 原则：atomic–要么全部，要么全不成功，consistent–数据库处于一致性状态，isolated–隔离性，并发的事务相互不影响，以及 durable–操作的效果能在崩溃和掉电后持久保存。

上面我们采用 Lamport 和 SPARC 的方法定义了内存操作的 “全局序”。

limo

最后，要注意一个陷阱。我们前面提到 (section 3.7) 一种检查乱序投机读取是否会被其它核观察到的方法是比较投机执行得到的值和 commit 时该位置的值是否相同。Martin 等人的研究表明这在有内存值预测的情况下不可行的。在值预测时，当一个读取操作执行时，核可以先 “猜测” 读取的

结果。想象下，如果一个核猜测某次对缓存块 X 读取的结果是 A，而实际该位置值为 B。在值预测和重新验证执行期间如果有另外一个核将 X 的值变为了 A。当这个核在对读取操作进行验证提交时，发现预测与实际值相同，于是错误地认为本次预测正确。系统在这种情况下可能会违反 SC 内存模型。这和多线程 ABA 问题 (http://en.wikipedia.org/wiki/ABA_problem) 类似。Martin 等人还提出了几种在有值预测情况下如何验证投机执行正确性的方法 (limo)。本段的目的不是深入讨论 SC 的某个边角情形或方案，而是让您明白内存模型的实现需要证明正确，而不是靠直觉感觉正确。

3.12　参考文献

limo

第四章　全局写保序和 x86 内存模型

全局写保序模型 (total store order, TSO）是一种被广泛实现的内存模型。TSO 被 SPARC 采用，更重要的是，它似乎符合广泛使用的 x86 的内存模型特征。本章使用上一章类似的范式来介绍这种内存模型。首先（section 4.1）我们先通过指出 SC 模型的局限性来给出 TSO/x86 内存模型的动机。然后我们先直观地介绍 TSO/x86（section 4.1），再对其进行形式化地描述（section 4.3），以及它的系统实现（section 4.4）。接着介绍 TSO/x86 的系统如何实现原子指令和实现辅助建立“指令序”的指令。(limo)

4.1　TSO/x86 模型动机初探