# 第六章：Coherence 协议

在本章中，我们回到了我们在第 2 章中介绍的 cache coherence 主题。我们在第 2 章中定义了 coherence，以便理解 coherence 在支持 consistency 方面的作用，但我们没有深入研究特定的 coherence 协议是如何工作的，或者它们是如何实现的。本章先一般性地讨论了 coherence 协议，然后我们将在接下来的两章中讨论特定的协议分类。我们从 6.1 节开始介绍 coherence 协议如何工作的整体情况，然后在 6.2 节展示如何指定 (specify) 协议。我们在第 6.3 节中介绍了一个简单而具体的 coherence 协议示例，并在第 6.4 节中探讨了协议设计空间。

## 6.1 整体情况

Coherence 协议的目标是通过强制执行 (enforce) 第 2.3 节中介绍、并在此重申的不变量来保持 coherence。

1. **单写多读 (Single-Writer, Multiple-Reader (SWMR)) 不变量。** 对于任何内存位置 A，在任何给定的（逻辑）时间，仅存在一个核心可以读写 A、或一些核心可以只读 A。
2. **数据值 (Data-Value) 不变量。** 一个时间段 (epoch) 开始时的内存位置的值，与其最后一个读写时间段 (epoch) 结束时的内存位置的值相同。

为了实现这些不变量，我们将每个存储结构（即，每个 cache 和 LLC/memory）关联到一个称为 **coherence 控制器 (coherence controller)** 的有限状态机。这些 coherence controllers 的集合 (collection) 构成了一个分布式系统。其中，控制器相互交换消息，以确保对于每个块，SWMR 和 Data-Value 不变量始终保持不变。这些有限状态机之间的交互由 coherence protocol 指定。

Coherence controllers 有几个职责。Cache 中的 coherence controller，我们称之为 **缓存控制器 (cache controller)**，如图 6.1 所示。Cache controller 必须为来自两个来源的请求提供服务。在 "**核心侧 (core side)**"，cache controller 与处理器核心连接。控制器接受来自核心的 loads 和 stores，并将 load values 返回给核心。一次 cache miss、会导致控制器发出一个 coherence **请求 (request)**（例如，请求只读权限）、来启动一个 coherence **事务 (transaction)**，其中，这个请求包含了核心所访问位置的块。这个 coherence 请求通过互连网络发送到一个或多个 coherence controllers。一个事务由一个请求和为满足请求而交换的其他消息（例如，从另一个 coherence controller 发送到请求者的数据响应消息）组成。作为每个事务的一部分，发送的事务和消息的类型、取决于特定的 coherence 协议。

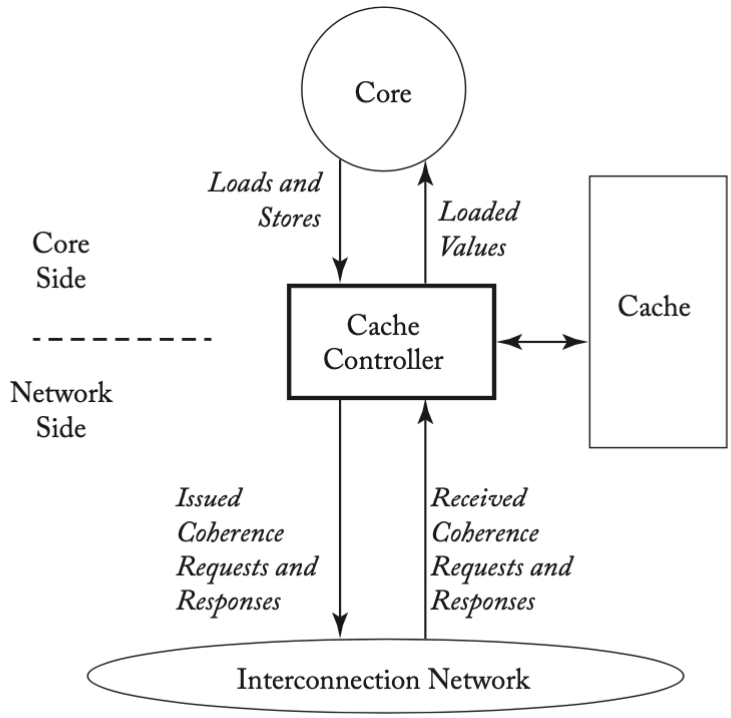


Figure 6.1: Cache controller.

在 cache controller 的 "**网络侧 (network side)**"，cache controller 通过互连网络与系统的其余部分连接。控制器接收它必须处理的 coherence 请求和 coherence 响应。与核心侧一样，如何处理传入的 coherence 消息、取决于特定的 coherence 协议。

LLC/memory 处的 coherence 控制器，我们称之为 **内存控制器 (memory controller)**，如图 6.2 所示。内存控制器类似于缓存控制器，只是它通常只有一个网络侧。因此，它不会发出 coherence 请求（为了 loads 或 stores）、或接收 coherence 响应。其他代理（例如 I/O 设备）的行为，可能类似于缓存控制器、内存控制器、或两者都是，取决于它们的特定要求。

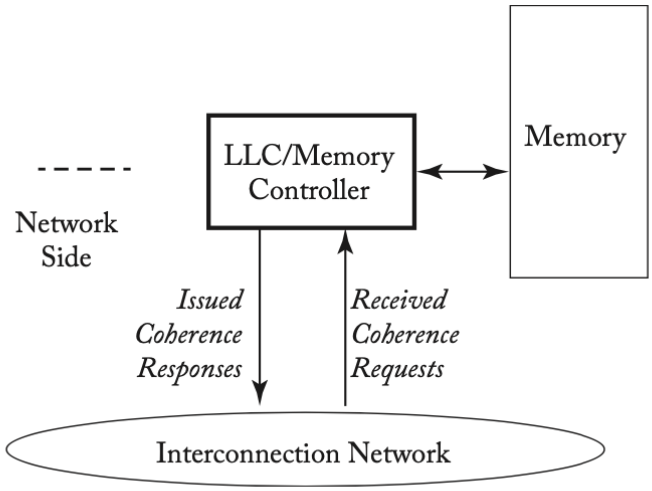


Figure 6.2: Memory controller.

每个 coherence 控制器都会实现一组 (a set of) 有限状态机（逻辑上，每个块都有一个独立但相同的有限状态机），并根据块的状态接收和处理**事件 (events)**（例如，传入的 coherence 消息）。对于到块 B 的、类型 E 的事件（例如，从核心到缓存控制器的 store 请求），coherence 控制器采取的动作 (actions)、是 E 和 B 的**状态 (state)**（例如，只读）的函数。采取这些动作之后，控制器可能会改变 B 的状态。

# 6.2 指定 Coherence 协议

我们通过指定 (specify) coherence 控制器来指定 coherence 协议。我们可以通过多种方式指定 coherence 控制器，但是 coherence 控制器的特定行为适合用表格规范来表示 [9]。如表 6.1 所示，我们可以将一个控制器指定为一张表，其中行对应于块状态 (states)，列对应于事件 (events)。我们将表中的一个 state/event 条目称为一次**转换 (transition)**，与块 B 相关的事件 E 的转换包括

- (a) E 发生时采取的动作 (actions)，和
- (b) 块 B 的下一个状态。

我们将转换格式表示为 “action/next state”，如果下一个状态是当前状态，我们可以省略 “next state” 部分。作为表 6.1 中的转换示例，如果从核心接收到块 B 的 store 请求、并且块 B 处于只读状态 (RO)，则该表显示控制器的转换将是执行动作 “issue coherence request for read-write permission (to block B)”，并将块 B 的状态更改为 RW。

Table 6.1: Tabular specification methodology. This is an incomplete specification of a cache coherence controller. Each entry in the table specifies the actions taken and the next state of the block.

| States | Events | | | |
|---|---|---|---|---|
| | | Load request from core | Store request from core | Incoming coherence request to obtain block in read-write state |
| | Not readable or writeable (N) | Issue coherence request for read-only permission/RO | Issue coherence requests for read-write permission/RW | <No action> |
| | Read-only (RO) | Give data from cache to core | Issue coherence request for read-write permission/RW | <No action>/N |
| | Read-write (RW) | Give data from cache to core | Write data to cache | Send block to requestor/N |

为简单起见，表 6.1 中的示例故意做得不完整，但它说明了表格规范方法捕获 coherence 控制器行为的能力。要指定 coherence 协议，我们只需要完全指定缓存控制器和内存控制器的表即可。

Coherence 协议之间的差异在于控制器规范之间的差异。这些差异包括不同的块状态 (block states)、事务 (transactions)、事件 (events) 和转换 (transitions)。在 6.4 节中，我们通过探索每个方面的选项、来描述 coherence 协议的设计空间，但我们首先来指定一个简单、具体的协议。

# 6.3 一个简单的 Coherence 协议示例

为了帮助理解 coherence 协议，我们现在提出一个简单的协议。我们的系统模型是第 2.1 节中的基线系统模型，但互连网络仅限于共享总线：一组共享的连线 (wires)，核心可以在这些连线上发出消息并让所有核心和 LLC/memory 观察到它。

每个缓存块可以处于两种**稳定 (stable)** 的 coherence 状态之一：**I(nvalid)** 和 **V(alid)**。LLC/memory 中的每个块也可以处于两种 coherence 状态之一：I 和 V。在 LLC/memory 中，

- 状态 I 表示所有缓存都将该 block 保持在状态 I，
- 状态 V 表示有一个缓存将 block 保持在状态 V。

缓存块也有一个单一的**瞬间 (transient)** 状态，即 **IV^D**，将在下面讨论。在系统启动时，所有的缓存块和 LLC/memory 块都处于状态 I。每个核都可以向其缓存控制器发出 load 和 store 请求；当缓存控制器需要为另一个块腾出空间时，它会隐式生成一个 **Evict Block** 事件。缓存中未命中的 loads 和 stores 会启动 coherence 事务，如下所述，以获得缓存块的 valid 拷贝。像本入门书中的所有协议一样，我们假设了一个**写回缓存 (writeback cache)**；也就是说，当 store 命中时，它将 store 值仅写入（局部）缓存、并等待将整个块写回 LLC/memory、以响应 Evict Block 事件。

我们使用三种类型的**总线消息 (bus messages)** 实现了两种类型的 **coherence 事务 (coherence transactions)**：

- **Get 消息** 用于请求一个 block，
- **DataResp 消息** 用于传输一个 block 的数据，
- **Put 消息** 用于将 block 写回内存控制器。

> 译者注：两种事务是指 **Get 事务**、**Put 事务**。

在一次 load 或 store miss 时，缓存控制器通过发送 Get 消息、并等待相应的 DataResp 消息、来启动 Get 事务。Get 事务是原子的，因为在缓存发送 Get 和该 Get 的 DataResp 出现在总线上之间，没有其他事务（Get 或 Put）可以使用总线。在 Evict Block 事件中，缓存控制器将带有整个缓存块的 Put 消息发送到内存控制器。

我们在图 6.3 中说明了稳定 coherence 状态之间的转换。我们使用前缀 "**Own**" 和 "**Other**" 来区分由给定缓存控制器发起的事务的消息、以及由其他缓存控制器发起的事务的消息。请注意，如果给定的缓存控制器具有处于状态 V 的块，并且另一个缓存使用 Get 消息（表示为 **Other-Get**）请求它，则 owning cache 必须用一个块去响应（使用 DataResp 消息，图中未显示）、并转换到状态 I。

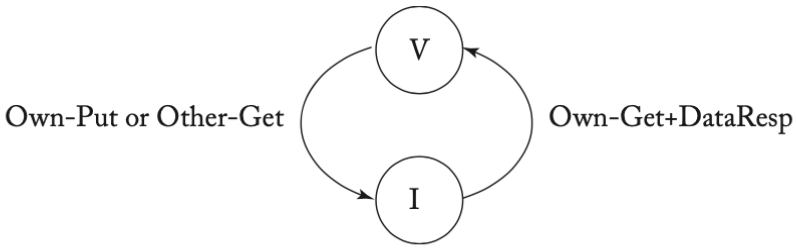


Figure 6.3: Transitions between stable states of blocks at cache controller.

表 6.2 和 6.3 更详细地指定了协议。表中的阴影条目表示不可能的转换。例如，缓存控制器不应该在总线上看到它自己对于一个块的 Put 请求，其中，该请求在其缓存中处于状态 V（因为它应该已经转换到了状态 I）。

Table 6.2: Cache controller specification. Shaded entries are impossible and blank entries denote events that are ignored.

| States | Core Events | | Bus Events | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | Messages for Own Transactions | | | Messages for Other Cores' Transactions | | |
| | Load or Store | Evict Block | Own-Get | DataResp for Own-Get | Own-Put | Other-Get | DataResp for Other-Get | Other-Put |
| I | Issue Get/ $IV^D$ | | | | | | | |
| $IV^D$ | Stall Load or Store | Stall Evict | | Copy data into cache, perform Load or Store /V | | | | |
| V | Perform Load or Store | Issue Put (with data) /I | | | | Send DataResp /I | | |

Table 6.3: Memory controller specification

| State | Bus Events | |
|---|---|---|
| | Get | Put |
| I | Send data block in DataResp message to requestor/V | |
| V | | Update data block in memory/I |

瞬间状态 IV^D 对应于状态 I 中的块，该块在转换到状态 V 之前正在等待数据（通过 DataResp 消息）。当稳定状态之间的转换不是原子的之时，会出现瞬间状态。在这个简单的协议中，单个消息的发送和接收是原子的，但是从内存控制器获取一个块需要发送一个 Get 消息、并接收一个 DataResp 消息，两者之间有一个中间的间隙 (gap)。IV^D 状态指示协议正在等待 DataResp。我们将在 6.4.1 节更深入地讨论瞬间状态。

这种 coherence 协议在许多方面都过于简单且效率低下，但介绍该协议的目的是了解如何指定协议。在介绍不同类型的 coherence 协议时，我们在整本书中都使用了这种规范方法。

# 6.4 Coherence 协议设计空间概述

如第 6.1 节所述，coherence 协议的设计者必须为系统中每种类型的 coherence 控制器选择状态 (states)、事务 (transactions)、事件 (events) 和转换 (transitions)。稳定状态的选择在很大程度上独立于 coherence 协议的其余部分。例如，有两类不同的 coherence 协议，称为**监听 (snooping)**、和**目录 (directory)**，架构师可以根据相同的稳定状态集、设计不同的监听协议或目录协议。我们将在 6.4.1 节中讨论独立于协议的稳定状态。同样，事务的选择也很大程度上独立于具体的协议，我们将在 6.4.2 节讨论事务。然而，与稳定状态和事务的选择不同，事件、转换和特定的瞬间状态，高度依赖于 coherence 协议，不能孤立地讨论。因此，在第 6.4.3 节中，我们讨论了 coherence 协议中的一些主要的设计决策。

## 6.4.1 状态

在只有一个参与者 (actor) 的系统中（例如，没有 coherent DMA 的单核处理器），缓存块的状态是 valid 或 invalid。如果需要区分块是**脏的 (dirty)**，则缓存块可能有两种可能的 valid 状态。脏块具有比该块的其他拷贝更近期的写入值。例如，在具有写回式 L1 缓存的两级缓存层次结构中，L1 中的块相对于 L2 缓存中的陈旧拷贝可能是脏的。

具有多个参与者的系统也可以只使用这两个或三个状态，如第 6.3 节所述，但我们经常想要区分不同类型的 valid 状态。我们希望在其状态中编码缓存块的四个特征：**有效性 (validity)**、**脏性 (dirtiness)**、**独占性 (exclusivity)**、和**所有权 (ownership)** [10]。后两个特征是具有多个参与者的系统所独有的。

- **Validity**：一个**有效 (valid)** 的块具有该块的最新值。该块可以被读取，但只有在它同时是独占的情况下才能被写入。
- **Dirtiness**：就像在单核处理器中一样，如果一个缓存块的值是最新的值、且这个值与 LLC/memory 中的值不同，那么这个缓存块就是**脏 (dirty)** 的，且缓存控制器负责最终使用这个新值去更新 LLC/memory。**干净 (clean)** 一词通常用作脏的反义词。
- **Exclusivity**：如果一个缓存块是系统中该块的唯一私有缓存拷贝，则该缓存块是独占的（注1）（即，除了可能在共享 LLC 中之外，该块不缓存在其他任何地方）。
- **Ownership**：如果缓存控制器（或内存控制器）负责响应对该块的 coherence 请求，那么它就是该块的**所有者 (owner)**。在大多数协议中，始终只有一个给定块的所有者。在不将块的所有权交给另一个 coherence 控制器的情况下，已拥有的块可能不会被从缓存中逐出（由于容量或冲突 miss）、以腾出空间给另一个块。在某些协议中，非拥有的块可能会被静默地驱逐（即，不发送任何消息）。

在本节中，我们首先讨论一些常用的稳定状态（当前未处于一致性事务中的块的状态），然后讨论使用瞬间状态来描述当前处于事务中的块。

> 原书作者注1：这里的术语可能会令人困惑，因为已经有一个称为 “Exclusive” 的缓存 coherence 状态，但还有其他缓存 coherence 状态在此处定义的意义上是 exclusive 的。

> 译者注：这边作者是想说，一个是协议里 “Exclusive” 状态的定义，另一个是缓存块 “exclusive” 性质的定义。

## 稳定状态 (Stable States)

许多 coherence 协议使用 Sweazey 和 Smith [10] 首次引入的经典五态 MOESI 模型的一个子集。这些 MOESI（通常发音为 “MO-sey” 或 “mo-EE-see”）状态指的是缓存中块的状态，最基本的三个状态是 MSI；可以使用 O 和 E 状态，但它们不是基本的。这些状态中的每一个都具有前面描述的特征的不同组合。

- **M(odified)**：该块是有效的、独占的、拥有的，并且可能是脏的。该块可以被读取或写入。该缓存具有块的唯一有效拷贝，且该缓存必须响应对块的请求，并且 LLC/memory 中的该块的拷贝可能是陈旧的。
- **S(hared)**：该块有效，但不独占、不脏、不拥有。该缓存具有块的只读拷贝。其他缓存可能具有该块的有效只读拷贝。
- **I(nvalid)**：该块无效。该缓存要么不包含该块，要么包含可能无法读取或写入的陈旧拷贝。在本入门书中，我们不区分这两种情况，尽管有时前一种情况可以表示为 “不存在 (Not Present)” 状态。

最基本的协议仅使用 MSI 状态，但有理由添加 O 和 E 状态以优化某些情况。当我们想要讨论带有和不带有这些状态的监听和目录协议时，我们将在后面的章节中讨论这些优化。现在，这里是 MOESI 状态的完整列表：

- **M(odified)**
- **O(wned)**：该块是有效的、拥有的、并且可能是**脏**的，但不是独占的。该缓存具有该块的**只读**拷贝，并且必须响应对该块的请求。其他缓存可能具有该块的只读副本，但它们不是所有者。LLC/memory 中的该块的拷贝可能是陈旧的。

- **E(xclusive)**：该块是有效的、独占的、且干净的。该缓存具有该块的**只读**拷贝。没有其他缓存拥有该块的有效拷贝，并且 LLC/memory 中的该块的拷贝是最新的。在本入门书中，我们认为当该块处于独占状态时，它是拥有 (owned) 的，尽管在某些协议中独占状态不被视为所有权 (ownership) 状态。当我们在后面的章节中介绍 MESI 监听和目录协议时，我们将讨论是否把独占的块视为所有者的问题。
- **S(hared)**
- **I(nvalid)**

我们在图 6.4 中展示了 MOESI 状态的维恩图。维恩图显示了哪些状态共享哪些特征。

- 除了 I 之外的所有状态都是有效的。
- M、O 和 E 是所有权 (ownership) 状态。
- M 和 E 都表示独占性，因为没有其他缓存具有该块的有效拷贝。
- M 和 O 都表示该块可能是脏的。

回到第 6.3 节中的简单示例，我们观察到协议有效地将 MOES 状态压缩为 V 状态。

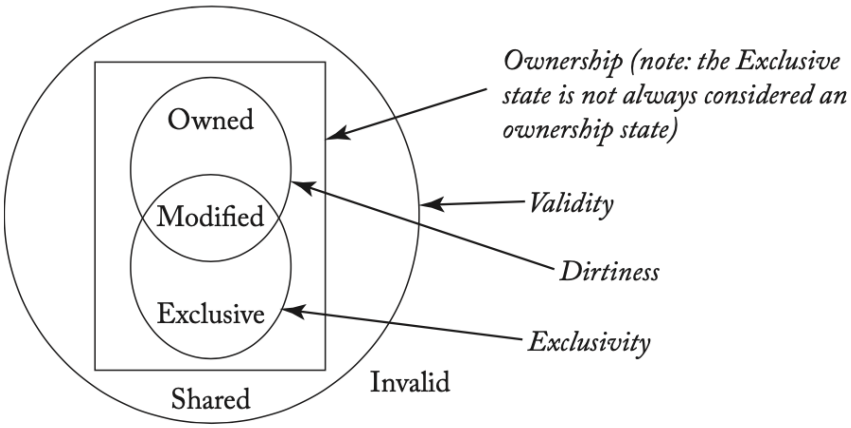


Figure 6.4: MOESI states.

MOESI 状态虽然很常见，但并不是一套详尽的稳定状态。例如，F (Forward) 状态类似于 O 状态，只是它是**干净**的（即 LLC/memory 中的拷贝是最新的）。有许多其他可能的 coherence 状态，但我们在本入门书中，会将注意力集中在著名的 MOESI 状态上。

## 瞬间状态 (Transient States)

到目前为止，我们只讨论了当块没有当前 coherence 活动时出现的稳定状态，并且在提到协议（例如，“具有 MESI 协议的系统”）时，仅使用这些稳定状态。然而，正如我们在 6.3 节的例子中看到的那样，在从一种稳定状态到另一种稳定状态的转换过程中，可能存在瞬间状态。在 6.3 节中，我们有瞬间状态 IV^D（在 I 中，正在进入 V，等待 DataResp）。在更复杂的协议中，我们可能会遇到几十种瞬间状态。我们使用符号 XY^Z 对这些状态进行编码，这表示该块正在从稳定状态 X 转换到稳定状态 Y，并且在发生 Z 类型的事件之前不会完成转换。例如，在后面章节的一个协议中，我们使用 IM^D 来表示一个块先前在 I 中，一旦 D (Data) 消息到达该块，它将变为 M。

## LLC/Memory 中的块状态

到目前为止，我们讨论的状态（稳定的和瞬间的）都与缓存中的块有关。LLC 和内存中的块也有与之相关的状态，有两种通用的方法来命名 LLC 和内存中的块的状态。命名约定的选择不会影响功能或性能；这只是一个规范问题，可能会使不熟悉该约定的架构师感到困惑。

- **以缓存为中心 (Cache-centric)**：在我们认为最常见的这种方法中，LLC 和内存中的块状态是缓存中该块状态的聚合 (aggregation)。例如，如果一个块在所有缓存中都处于 I 状态，则该块的 LLC/memory 状态为 I。如果一个块在一个或多个缓存中处于 S 状态，则 LLC/memory 状态为 S。如果一个块在单个缓存中处于 M 状态，则 LLC/memory 状态为 M。
- **以内存为中心 (Memory-centric)**：在这种方法中，LLC/memory 中块的状态对应于内存控制器对该块的权限（而不是缓存的权限）。例如，如果一个块在所有缓存中都处于 I 状态，则该块的 LLC/memory 状态为 O（不是 I，如在以缓存为中心的方法中那样），因为 LLC/memory 的行为类似于该块的所有者。如果一个块在一个或多个缓存中处于 S 状态，则 LLC/memory 状态也是 O，出于同样的原因。但是，如果一个块在单个缓存中处于 M 或 O 状态，则 LLC/memory 状态为 I，因为 LLC/memory 有该块的无效拷贝。

本入门书中的所有协议都使用以缓存为中心的名称来表示 LLC 和内存中的块状态。

## 维护块状态

系统实现必须维护与缓存、LLC 和内存中的块相关联的状态。对于缓存和 LLC，这通常仅需要将 per-block 的缓存状态进行扩展、至多几位，因为稳定状态的数量通常很少（例如，MOESI 协议的 5 个状态需要每个块 3 位）。Coherence 协议可能有更多的瞬间状态，但只需要为那些有未决 (pending) coherence 事务的块维护这些状态。实现通常通过向未命中状态处理寄存器 (miss status handling registers, MSHR) 添加额外的位，或添加用于跟踪这些未决事务 [4] 的类似结构、来维护这些瞬间状态。

对于内存而言，更大的总容量似乎会带来重大挑战。然而，许多当前的多核系统会维护一个 inclusive LLC，这意味着 LLC 会维护缓存在系统中任何位置的每个块的拷贝（甚至是 “独占” 的块）。使用 inclusive LLC，则内存不需要 **explicitly** 表示 coherence 状态。如果一个块驻留在 LLC 中，则它在内存中的状态与它在 LLC 中的状态相同。如果块不在 LLC 中，则其在内存中的状态为 **implicitly Invalid**，因为 inclusive LLC 的缺失意味着该块不在任何缓存中。侧边栏讨论了在具有 inclusive LLC 的多核出现之前，内存状态是如何保持的。上面对内存的讨论假设了系统具有单个多核芯片，本入门书的大部分内容也是如此。具有多个多核芯片的系统可能会受益于内存逻辑上的显式 coherence 状态。

## 原书侧边栏：在多核之前：在内存中保持一致性状态

传统地，pre-multicore 协议需要维护每个内存块的 coherence 状态，并且它们不能使用第 6.4.1 节中解释的 LLC。我们简要讨论了几种维持这种状态的方法以及相关的工程权衡。

**用状态位扩充每个内存块。** 最通用的实现是向每个内存块添加额外的位、以保持 coherence 状态。如果内存中有 N 个可能的状态，那么每个块需要 log_2(N) 个额外的位。尽管这种设计是完全通用的并且在概念上很简单，但它有几个缺点。首先，额外的位可能会以两种方式增加成本。使用现代的面向块的 DRAM 芯片很难添加两个或三个额外位，这些芯片通常至少需要 4 位宽，而且通常更宽。此外，内存中的任何变化都会妨碍使用商用 DRAM 模块（例如 DIMM），这会显著增加成本。幸运的是，对于每个块只需要几位状态的协议，可以使用修改后的 ECC 代码来存储这些状态。通过在更大的粒度上维护 ECC（例如，512 位、而不是 64 位），可以释放足够的代码空间来 “隐藏” 少量额外的位，同时，还能使用商用 DRAM 模块 [1, 5, 7]。第二个缺点是，将状态位存储在 DRAM 中、意味着获取状态会导致完整的 DRAM 延迟，即使在最新版本的、块存储在其他缓存中的情况下、也是如此。在某些情况下，这可能会增加缓存到缓存 coherence 传输的延迟。最后，将状态存储在 DRAM 中意味着所有状态更改都需要一个 DRAM read-modify-write 周期，这可能会影响功率和 DRAM 带宽。

**在内存中为每个块添加单个状态位。** Synapse [3] 使用的一个设计选项是，使用与每个内存块相关联的单个位来区分两个稳定状态（I 和 V）。很少有块处于瞬间状态，并且可以通过小型专用结构来维持这些状态。该设计是更完整的第一个设计的子集，存储成本最低。

**Zero-bit logical OR。** 为了避免修改内存，我们可以让缓存按需重建 (reconstruct) 内存状态。一个块的内存状态、是关于每个缓存中块状态的函数，因此，如果所有缓存聚合它们的状态，它们就可以确定内存状态。系统可以通过让所有核心发送 "IsOwned?"（注a）来推断内存是否是块的所有者。信号发送到逻辑或门（或门树），其输入数量等于缓存的数量。如果此 OR 的输出为高，则表示缓存是所有者；如果输出低，那么内存就是所有者。该解决方案避免了在内存中维护任何状态的需要。然而，使用逻辑门、或 wired-OR 来实现一个快速 OR、可能很困难。

> 原书作者注a：不要将此 IsOwned 信号与 Owned 缓存状态相混淆。IsOwned 信号由处于所有权状态的缓存置位 (asserted)，该状态包括 Owned、Modified 和 Exclusive 缓存状态。

## 6.4.2 TRANSACTIONS

大多数协议都有一组相似的事务，因为 coherence 控制器的基本目标是相似的。例如，几乎所有协议都有一个事务来获得对块的共享（只读）访问。在表 6.4 中，我们列出了一组常见事务，并且对于每个事务，我们描述了发起事务的请求者的目标。这些事务都是由缓存控制器发起的、以响应来自其相关核心的请求。在表 6.5 中，我们列出了核心可以向其缓存控制器发出的请求，以及这些核心请求如何引导缓存控制器启动 coherence 事务。

Table 6.4: Common transactions

| Transaction | Goal of Requestor |
|---|---|
| GetShared (GetS) | Obtain block in Shared (read-only) state |
| GetModified (GetM) | Obtain block in Modified (read-only) state |
| Upgrade (Upg) | Upgrade block state from read-only (Shared or Owned) to read-write (Modified); Upg (unlike GetM) does not require data to be sent to requestor |
| PutShared (PutS) | Evict block in Shared state[a] |
| PutExclusive (PutE) | Evict block in Exclusive state[a] |
| PutOwned (PutO) | Evict block in Owned state |
| PutModified (PutM) | Evict block in Modified state |

[a] Some protocols do not require a coherence transaction to evict a Shared block and/or an Exclusive block (i.e., the PutS and/or PutE are "silent").

Table 6.5: Common core requests to cache controller

| Event | Response of (Typical) Cache Controller |
|---|---|
| Load | If cache hit, respond with data from cache; else initiate GetS transaction |
| Store | If cache hit in state E or M, write data into cache; else initiate GetM or Upg transaction |
| Atomic read-modify-write | If cache hit in state E or M, automically execute RMW semantics; else GetM or Upg transaction |
| Instruction fetch | If cache hit (in I-cache), respond with instruction from cache; else initiate GetS transaction |
| Read-only prefetch | If cache hit, ignore; else may optionally initiate GetS transaction[a] |
| Read-Write prefetch | If cache hit in state M, ignore; else may optionally initiate GetM or Upg transaction[a] |
| Replacement | Depending on state of block, initiate PutS, PutE, PutO, or PutM transaction |

[a] A cache controller may choose to ignore a prefetch request from the core.

尽管大多数协议使用一组类似的事务，但它们在 coherence 控制器如何交互、以执行事务、这一方面存在很大差异。正如我们将在下一节中看到的，在某些协议（例如，监听协议）中，缓存控制器通过向系统中的所有 coherence 控制器广播 GetS 请求来启动 GetS 事务，并且当前该块的所有者的控制器、会用包含所需数据的消息、来响应请求者。相反，在其他协议（例如，目录协议）中，缓存控制器通过向特定的预定义 coherence 控制器发送单播 GetS 消息来发起 GetS 事务，该 coherence 控制器可以直接响应、或者可以将请求转发到将响应请求者的另一个 coherence 控制器。

## 6.4.3 主要协议设计选项

设计 coherence 协议有许多不同的方法。即使对于同一组状态和事务，也有许多不同的可能协议。协议的设计决定了每个 coherence 控制器上可能发生的事件和转换；与状态和事务不同，没有办法提供独立于协议的可能事件或转换的列表。

尽管 coherence 协议有巨大的设计空间，但有两个主要的设计决策会对协议的其余部分产生重大影响，我们接下来将讨论它们。

### Snooping vs. Directory

Coherence 协议主要有两类：监听和目录。我们现在对这些协议进行简要概述，并将它们的深入介绍分别推迟到第 7 章和第 8 章。

- **监听协议**：缓存控制器通过向所有其他 coherence 控制器广播请求消息来发起对块的请求。Coherence 控制器集体 “做正确的事”，例如，如果他们是所有者，则发送数据以响应另一个核心的请求。监听协议依靠互连网络以 consistent 的顺序将广播消息传递到所有核心。大多数监听协议假定请求按 total order 到达，例如，通过 shared-wire 总线。但更高级的互连网络和更宽松的顺序也是可能的。
- **目录协议**：缓存控制器通过将块单播到作为该块所在的内存控制器来发起对块的请求。内存控制器维护一个目录，该目录保存有关 LLC/memory 中每个块的状态，例如当前所有者的身份或当前共享者的身份。当对块的请求到达主目录时，内存控制器会查找该块的目录状态。例如，如果请求是 GetS，则内存控制器查找目录状态以确定所有者。如果 LLC/memory 是所有者，则内存控制器通过向请求者发送数据响应来完成事务。如果缓存控制器是所有者，则内存控制器将请求转发给所有者缓存；当所有者缓存接收到转发的请求时，它通过向请求者发送数据响应来完成事务。

监听与目录的选择涉及权衡取舍。监听协议在逻辑上很简单，但它们无法扩展到大量核心，因为广播无法扩展。目录协议是可扩展的，因为它们是单播的，但许多事务需要更多时间，因为当 home 不是所有者时，它们需要发送额外的消息。此外，协议的选择会影响互连网络（例如，经典的监听协议需要请求消息的 total order）。

### Invalidate vs. Update

Coherence 协议中的另一个主要设计决策是决定核心写入块时要做什么。这个决定与协议是监听还是目录无关。有两种选择。

- **Invalidate protocol**：当一个核心希望写入一个块时，它会启动一个 coherence 事务以使所有其他缓存中的拷贝无效。一旦拷贝失效，请求者就可以写入块，而另一个核心不可能读取块的旧值。如果另一个核心希望在其副本失效后读取该块，它必须启动一个新的 coherence 事务来获取该块，并且它将从写入它的核获得一个副本，从而保持 coherence。
- **Update protocol**：当一个核心希望写入一个块时，它会启动一个 coherence 事务来更新所有其他缓存中的副本，以反映它写入块的新值。

再一次，在做出这个决定时需要权衡取舍。更新协议减少了核心读取新写入块的延迟，因为核心不需要启动并等待 GetS 事务完成。但是，更新协议通常比无效协议消耗更多的带宽，因为更新消息大于无效消息（地址和新值，而不仅仅是地址）。此外，更新协议使许多 memory consistency models 的实现变得非常复杂。例如，当多个缓存必须对一个块的多个拷贝、应用多个更新时，保持写原子性（第 5.5 节）

变得更加困难。由于更新协议的复杂性，它们很少被实现；在本入门书中，我们将重点介绍更为常见的无效协议。

### 混合设计

对于这两个主要的设计决策，一种选择是开发一种混合设计。有些协议结合了监听和目录协议 [2, 6] 的各个方面，还有一些协议结合了无效和更新协议 [8] 的各个方面。设计空间丰富，架构师不受限于遵循任何特定的设计风格。

## 6.5 参考文献


[1] A. Charlesworth. The Sun Fireplane SMP interconnect in the Sun 6800. In Proc. of the 9th Hot Interconnects Symposium, August 2001. DOI: 10.1109/his.2001.946691. 101

[2] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. IEEE Micro, 27(2):10–21, March/April 2007. DOI: 10.1109/mm.2007.43. 105

[3] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. Electronics, 57(1):164–169, January 1984. 101

[4] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In Proc. of the 8th Annual Symposium on Computer Architecture, May 1981. DOI: 10.1145/285930.285979. 100

[5] H. Q. Le et al. IBM POWER6 microarchitecture. IBM Journal of Research and Development, 51(6), 2007. DOI: 10.1147/rd.516.0639. 101

[6] M.M.K.Martin,D.J.Sorin,M.D.Hill,andD.A.Wood.Bandwidthadaptivesnooping. In Proc. of the 8th IEEE Symposium on High-Performance Computer Architecture, pp. 251– 262, January 2002. DOI: 10.1109/hpca.2002.995715. 105

[7] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp scalable shared memory multiprocessor. In Proc. of the International Conference on Parallel Processing, vol. I, pp. 1–10, August 1995. DOI: 10.1109/hicss.1994.323149. 101

[8] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In Proc. of the 2nd IEEE Symposium on High-Performance Computer Architecture, February 1996. DOI: 10.1109/hpca.1996.501197. 105

[9] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Transactions on Parallel and Distributed Systems, 13(6):556–578, June 2002. DOI: 10.1109/tpds.2002.1011412. 93

[10] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In Proc. of the 13th Annual International Symposium on Computer Architecture, pp. 414–423, June 1986. DOI: 10.1145/17356.17404. 97, 98


## 6.6 译者扩展：一个 MESI 协议小玩具

这个小玩具可以帮助理解 MESI 协议。

https://www.scss.tcd.ie/jeremy.jones/vivio/caches/MESI.htm