灵衢®使能操作系统参考设计

文档版本：2.0

发布日期：2025 年 9 月

目录

图目录

表目录

1 概述

1.1 目的

本文描述了操作系统灵衢组件（UB OS Component）的架构、功能及外部接口，主要目标受众包括应用开发者、驱动开发者以及 OS 开发者等。其他任何有兴趣了解操作系统灵衢软件架构的人员也可阅读此文档。

1.2 范围

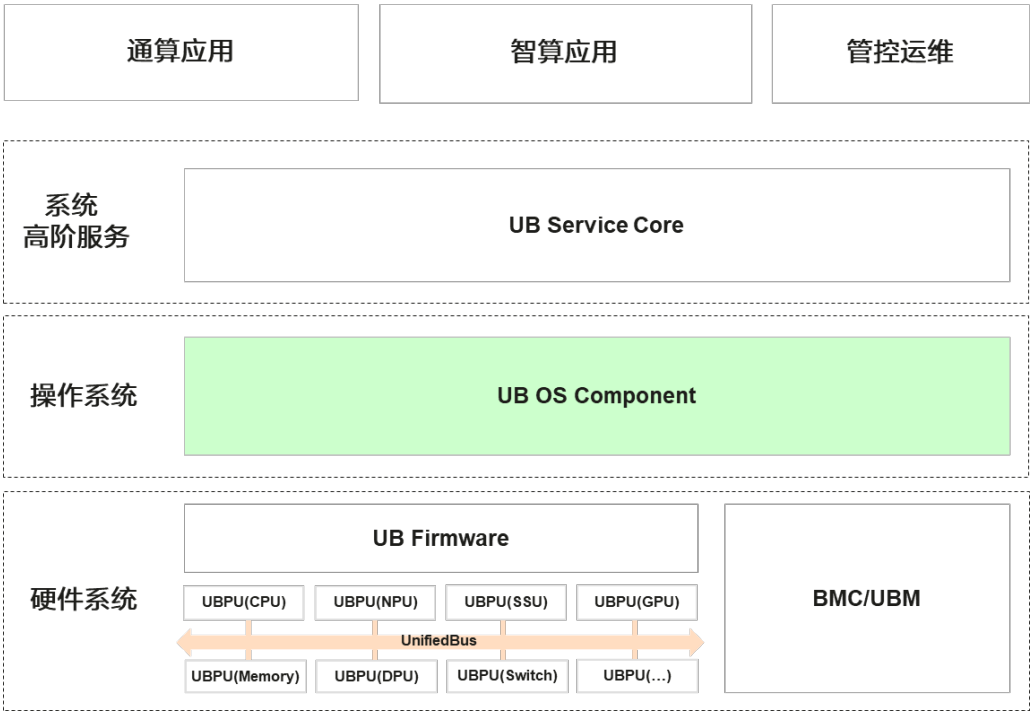

图 1-1 灵衢系统示意图

灵衢系统示意图如上，操作系统灵衢组件（UB OS Component）是在 OS 原有内存管理、通信、设备管理和虚拟化框架上扩展支持灵衢，实现异构硬件统一抽象解耦、统一内存地址空间，支持资源全局调度、计算资源动态组合扩展、设备间高性能通信，释放灵衢硬件能力。

本文档聚焦介绍操作系统灵衢组件架构和接口，包括：

1. 软件架构：介绍操作系统灵衢组件软件架构、功能组件层次。
2. 组件功能：介绍各层内的组件功能，与现有组件的关系。
3. 接口：介绍关键接口设计，典型场景接口使用流程。

注：如需了解灵衢系统参考架构其他层次组件，可从 https://www.unifiedbus.com 网站下载对应文档。

1.3 引用

1. 《灵衢基础规范 2.0》
2. 《灵衢固件规范 2.0》

2 架构

OS 运行在计算节点 CPU 上，为用户业务提供运行环境。操作系统灵衢组件软件架构图如下：

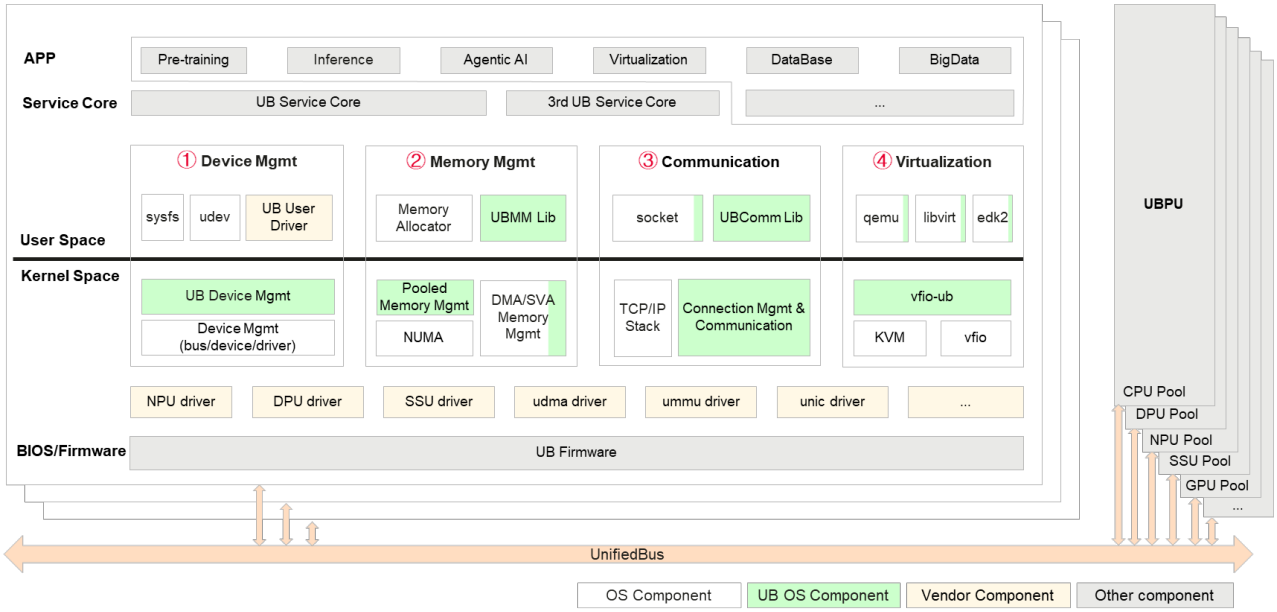

图 2-1 操作系统灵衢组件软件架构

操作系统灵衢组件是在 OS 原有内存管理、通信、设备管理和虚拟化框架上扩展支持灵衢，扩展的 4 个功能分别是：

① **Device Mgmt：**提供 UB 总线、UB 设备管理能力，实现计算节点内 UB 设备热插拔、配置。包含模块：

- UB Device Mgmt：UB 设备管理基于 Linux 的 Device Mgmt（bus/device/driver）模型扩展支持 UB 设备管理，包含 UB 总线驱动、UB Firmware 交互、UB 设备热插拔等，同时提供设备驱动接口用于 UB 设备驱动开发。
- sysfs：UB 设备管理会在 sysfs 生成 UB 设备、驱动、总线信息，用户可通过 sysfs 查看和使用。
- udev：UB 设备管理会生成 uevent 事件，用户可通过已有 udev 工具获取 UB 设备的热插拔状态，实现对 UB 设备的管理。
- UB User Driver：UB 设备用户态驱动，用户可通过此接口将 UB 设备暴露到用户态，供用户态驱动或者虚拟化软件使用。

详细介绍见第 3 设备管理（Device Mgmt）章节。

② **Memory Mgmt：**提供 UB 总线域内内存语义访问能力，实现跨计算节点跨设备内存借用、共享。包含模块：

- DMA/SVA Memory Mgmt：基于 Linux 已有的 DMA/SVA 框架扩展实现对 UMMU 单元的管理，驱动或其他组件可通过本模块将主机内存注册给 UB 设备使用。
- Pooled Memory Mgmt：新增本地内存导出、远端内存导入、共享内存一致性维护等功能。基于 Linux 已有的 NUMA 内存管理框架上支持远端内存上线到 NUMA node 使用，或上线到内存设备（在/dev 下生成内存字符设备）通过 mmap 映射使用。
- UBMM Lib：用户态新增封装 Pooled Memory Mgmt 提供的接口，UBPRM（详见注 1）可通过上述接口实现 Home 和 User 侧的内存管理操作。
- Memory Allocator：兼容已有 glibc/jemalloc/tcmalloc 等内存管理接口，UBPRM 从其他计算节点借用内存并通过 UBMM Lib 上线到 NUMA node 后，应用程序无需修改即可通过 glibc/jemalloc/tcmalloc/llibnuma 等提供的 malloc/free/numa_alloc_onnode 等内存管理接口申请/释放借用的内存。

详细介绍见第 4 内存管理（Memory Mgmt）章节。

③ **Communication：**提供跨计算节点、跨设备通信和远程调用功能。包含模块：

- Connection Mgmt & Communication：实现异步通信的连接管理，同时提供内核态的异步通信接口供其他模块使用。
- UBComm Lib：新增提供灵衢异步通信和统一远程过程调用接口，应用程序可通过此接口实现超节点内节点间通信和函数调用。
- socket：兼容已有 glibc 等提供 socket 接口的库，应用程序无需修改或少量修改（详见注 2）即可通过 socket 接口实现两个节点间通信。

详细介绍见第 5 通信（Communication 章节。

④ **Virtualization：**提供 UB 设备直通虚拟机能力。包含模块：

- qemu：在现有 qemu 上扩展实现 Bus Controller、UB 设备、UMMU 的虚拟化。
- libvirt：在现有 libvirt 上扩展支持 UB 设备的创建、删除等功能。
- vfio-ub：基于 Linux 已有 vfio 框架上新增支持 UB 设备虚拟化。

详细介绍见第 6 虚拟化（Virtualization）章节。

注 1：UBPRM 为超节点内的池化资源管理者，负责资源调度，接受来自用户（例如 k8s 等云管平台、配置管理员手动配置等）的资源分配请求（例如将 A 节点的某个资源分配给 B 节点使用）或自主调度（例如内存资源可由 UBPRM 在超节点内根据用户的调度策略选择借出节点），根据一定的调度策略选择出对应的资源，并调用 UBFM 将资源配置给对应的节点。UBPRM 由用户自行实现，UB Service Core 提供的 UBS Engine 为其中一种实现。

注 2：应用程序无需修改指二进制无需修改，直接可以运行并使用灵衢通信功能。少量修改指通过 preload 或代码少量适配可使用灵衢通信能力。

3 设备管理（Device Mgmt）

3.1 概述

UB 设备管理包括 UBus Driver、vfio-ub 和 ubutils 三个模块，对用户提供 UB 设备管理接口，包括给 UB 设备驱动提供了基本的设备发现、设备注册、中断使能等总线服务，UB 设备直通用户态的服务，以及给用户提供查询 UB 设备信息和配置服务。各类 UB 设备可以注册到 UB 总线上，对用户提供相应的功能。在 OS 内，UB 设备的发现、注册、驱动加载均由设备管理实现，分单机和集群场景：

1. 单机场景：单台服务器主机独立工作，每台服务器独享连接在主机上的所有 UB 设备，单机上的 UBus Driver 完成所有 UB 设备的枚举发现和使能，配合 vfio-ub 提供 UB 设备直通用户态能力。
2. 集群场景：超节点形态下 UBus Driver 提供分配给计算节点上的 UB Entity 发现和使能，并支持池化设备接入和移除，配合 vfio-ub 提供 UB 设备直通用户态能力。

注：UB Entity 分配给计算节点需要 UBPRM 和 UBFM 配合实现。

3.2 功能架构

3.2.1 设备管理架构

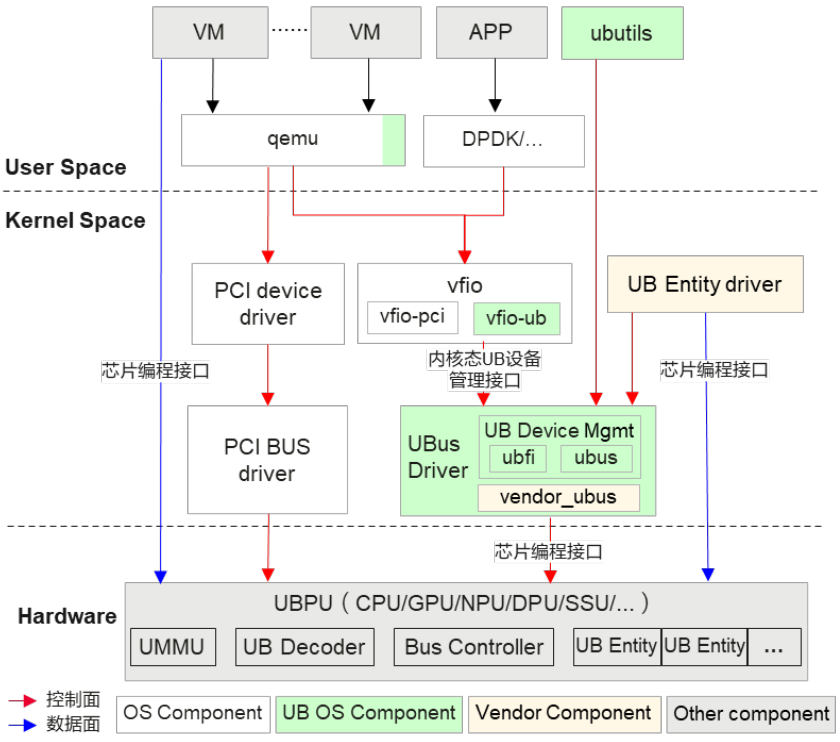

图 3-1 灵衢设备管理架构

- **UBus Driver:** 包含 ubfi、ubus、vendor_ubus 三个模块。其中 ubfi 负责解析 BIOS 上报的 UBRT 表、注册平台设备、生成所需的 Bus Controller 等功能，ubus 基于 Bus Controller 提供 UB 总线的枚举、UB 设备创建、UB 设备管理、错误处理、热拔插、中断等和设备相关的基础服务。vendor_ubus 为厂商差异化实现，负责和厂商硬件交互。
- **vfio-ub：**提供 UB 设备直通用户态能力。
- **ubutils：**提供 UB 设备管理的工具，实现 UB 拓扑查询、配置空间查询及配置等功能。

设备管理关键功能如下：

1. 支持 UB 设备信息解析。ubfi 负责解析 BIOS 上报的 UBRT 信息表，表中记录了 SoC 上与 UB 组件相关的信息，包括寄存器地址、与中断控制器、UMMU 的关联关系、可供 UB 分配使用的系统资源等信息。OS 获取 UBRT 信息表的信息后，解析表项内容，创建并注册 Bus Controller 、UMMU 等设备节点，扫描 Bus Controller port，完成 UB Entity 的设备枚举，建立拓扑关系。
2. 支持设备枚举。UBus Driver 负责将单机范围内的 UB 设备无重复无遗漏地枚举出来。设备枚举功能需要通过邻居发现协议、枚举管理消息和配置访问消息来完成，主要包括拓扑扫描、网络地址分配、路由表配置、EID 分配以及将扫描到的 UB Entity 进行初始化后注册到 UB 总线上进行统一管理。
3. 支持设备资源空间管理。主机获取设备资源空间信息，并将自身地址空间分配给设备。主机驱动通过主机侧地址来访问设备资源空间。CPU 会给 UB 子系统划分地址窗口，总线驱动管理这段地址资源，在 UB Entity 使能时分配设备所需的资源空间，并在设备移除时释放。由于 UB 设备支持热插拔，UB Entity 是用户使用过程中按需使能创建，因此需支持动态的资源分配和释放。
4. 支持设备配置空间访问。UB Entity 支持配置空间寄存器，集中存放了与设备管理相关的能力描述和配置接口。协议定义对配置空间寄存器的读写采用的是配置访问消息，Bus Controller 是配置访问消息的发起者，设备是消息的响应者。Bus Controller 自身也有配置空间寄存器，Bus Controller 自身既是消息的发起者，也是消息的响应者。UBus Driver 通过操作 Bus Controller 提供的软硬件接口（Message Queue）实现对配置空间寄存器的访问。
5. 支持总线设备驱动。UB 总线设备驱动框架采用 Linux 内核的驱动模型来支持对 UB 设备的统一管理，遵从驱动模型的 Bus/Device/Driver 设计，其主要功能有两个：一是为 UB Entity 提供统一的抽象，以及为厂商驱动提供统一的编程接口，达到设备管理的目的；二是 UB 服务管理，提供 UB 组件的公共服务注册/注销接口，如 RAS、热插拔等。
6. 支持设备中断。UB 中断采用消息中断机制，负责建立起 UB 设备到系统的中断通路，包括接入中断控制器、UB 设备的中断申请释放。
7. 支持 Port 管理。UBus Driver 支持 Port 建链、邻居设备等信息的查询以及相关配置。查询与配置操作可以由驱动软件或用户发起。
8. 支持错误检测和上报。UBus Driver 支持《灵衢基础规范 2.0》中定义的故障错误检测、纠正及错误上报机制，依据此来方便准确地定位、纠正、分析错误，增强系统的健壮性和可靠性。

9. 支持 UB 设备信息查询和设置。用户通过 lsub/setub 工具提供给用户查看 UB 组网拓扑、设备配置空间以及改写指定设备配置空间。通过 sysfs 将内核 UB 子系统的信息呈现给用户态，并且针对部分模块开放写权限使得用户可以触发内核执行相关任务。

3.2.2 设备驱动模型

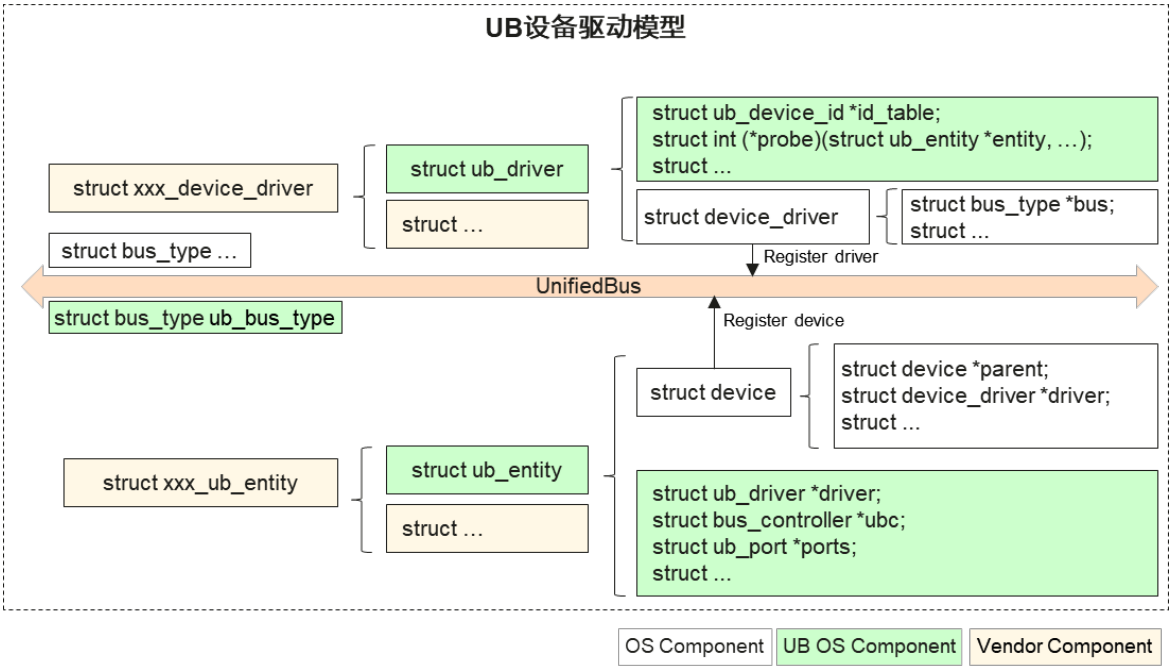

图 3-2 UB 设备驱动模型

UB 设备驱动模型采用 Linux 内核的驱动模型来支持对 UB 设备的统一管理，遵从驱动模型的 Bus/Device/Driver 设计，为 UB 设备提供统一的抽象，为厂商驱动提供统一的编程接口，UB 设备管理模型包括以下三个数据结构：

1. 新增 ub_bus_type 实体，实现驱动模型中 bus_type 定义的操作集合，在 OS 启动阶段将 UB 总线注册到系统中。
2. 新增 struct ub_entity 结构体（包含 struct device），用以描述 UB Entity，协议中每个 UB Entity 都会实例化一个 ub_entity。
3. 新增 struct ub_driver 结构体（包含 struct device_driver），用以描述 UB Entity 驱动。

UB Entity 插入后，会生成对应的 ub_entity，设备管理自动匹配对应的设备驱动 ub_driver，对用户呈现如下设备树：

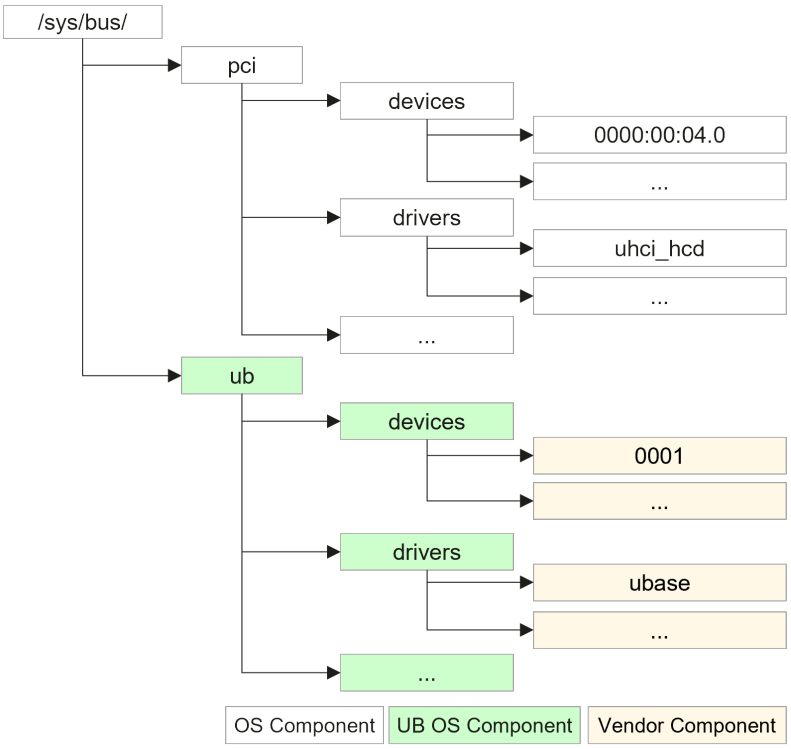

图 3-3 UB 设备树

3.2.3 设备服务模型

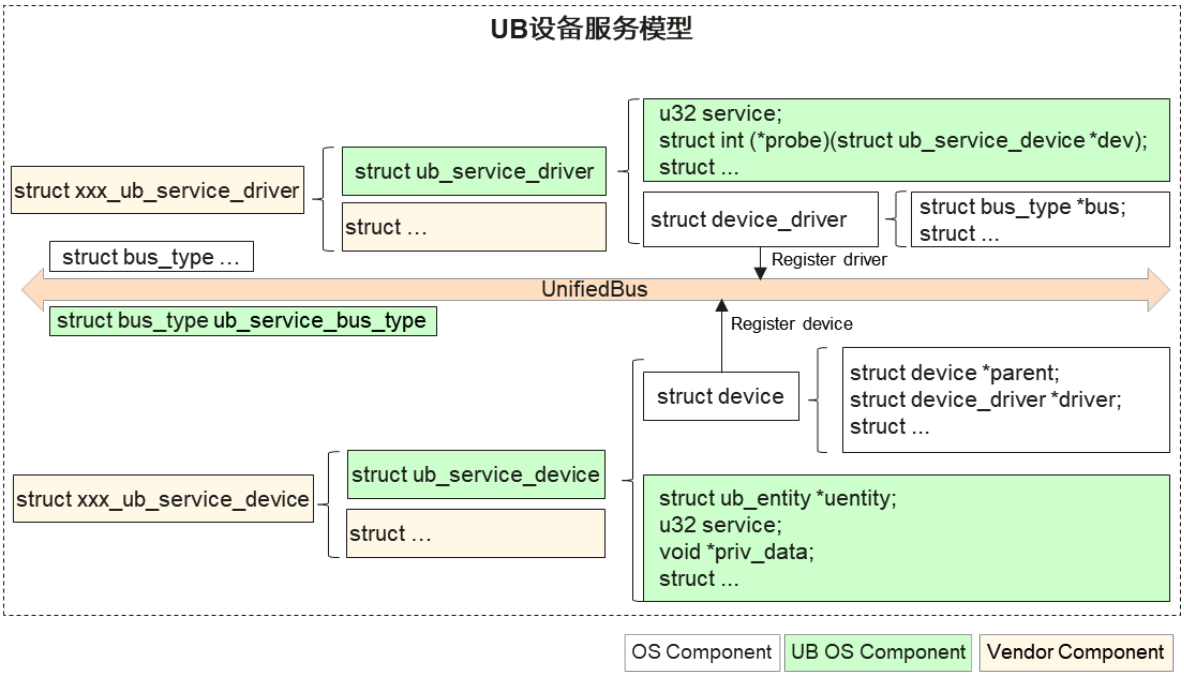

图 3-4 UB 设备服务模型

UB 设备服务管理用于在 UB 组件(如 Bus Controller、UB Switch)驱动中注册公共的 UB 设备服务，比如 RAS、热插拔等。设备服务在 UB 组件驱动中根据组件能力注册到 ub_service 总线，并与 UB 设备服务驱动进行匹配，进入到 UB 设备服务具体的处理流程。UB 设备服务定义三个数据结构：

1. 新增 struct ub_serivce_bus_type 实体，实现 bus_type 定义的操作集合，在启动阶段将 UB 服务总线注册到系统中。
2. 新增 struct ub_service_device 结构体（包含 struct device），用以描述 UB 设备服务，针对协议中 RAS、hotplug 等创建 ub_service_device 实例。
3. 新增 struct ub_service_driver 结构体（包含 struct device_driver），用以描述 UB 设备服务驱动。

框架会对外提供 UB 设备服务驱动注册和注销接口供 UB 组件驱动调用。此框架仅提供服务的注册/注销功能，因此 bus_type 仅定义了 match 操作接口，此接口采用 ID 匹配机制，依据 service 的服务类型进行匹配，对用户呈现如下设备树：

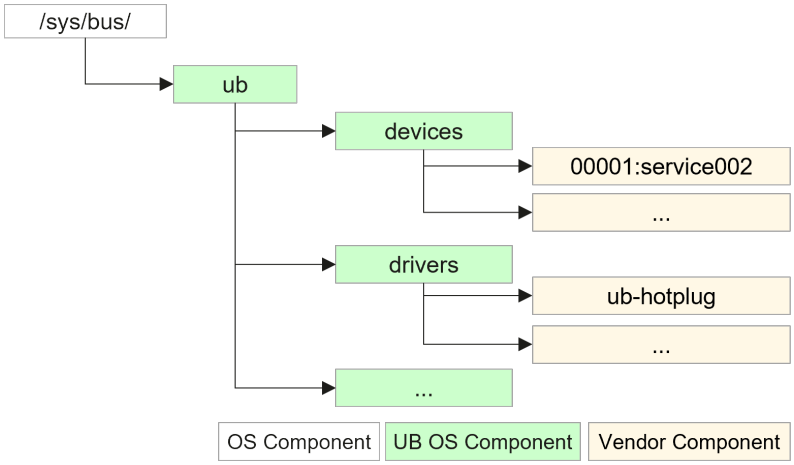

图 3-5 UB 服务设备树

3.3 使用流程

3.3.1 设备驱动

UB 驱动程序通过 ub_register_driver 注册到系统中（此函数一般在驱动模块的 init 函数中调用），一旦 UB 总线探测到一个 UB 设备并匹配到对应的驱动程序，则驱动通常需要执行以下初始化（一般在驱动的 probe 函数中实现）：

1. 调用 ub_set_hostinfo 设置主机信息，例如 EID。
2. 调用 ub_entity_enable 启用 UB Entity。
3. 调用 ub_iomap 映射资源空间。
4. 访问设备配置空间进行设备初始化。
5. 调用 request_irq 注册 IRQ 处理程序。
6. 注册到其他子系统（例如 nvme 或 netdev）上。

当使用完设备后，需要移除设备，驱动需要采取以下步骤（一般在驱动的 remove 函数中实现）：

1. 从其他子系统（例如 nvme 或 netdev）上取消注册。
2. 禁用设备产生的 IRQ，调用 free_irq 释放 IRQ。
3. 调用 iounmap 释放资源空间。
4. 调用 ub_entity_enable 禁用设备。
5. 调用 ub_unset_hostinfo 清除主机信息。

当需要移除整个驱动模块时，则调用 ub_unregister_driver 注销设备驱动（一般在 exit 函数中调用）。

注：上述步骤并非所有驱动程序都需要调用，驱动开发者需要根据自己的设备功能和设计调整初始化和移除步骤。

3.3.2 设备服务驱动

设备服务驱动开发和使用流程如下：

1. UB 服务驱动通过 ub_serivce_driver_register 注册到系统中。
2. 当对应的 UB 组件设备被探测到后，ub_service_bus_type 总线会使用 serviceID 匹配对应的驱动，匹配后，则调用对应驱动的 probe 函数进行服务初始化。
3. 当对应的 UB 组件设备被移除时，则执行对应驱动的 remove 函数。
4. 当需要移除整个驱动模块时，调用 ub_service_driver_unregister 注销驱动。

3.4 维测

1. 总线驱动维测：
   (1) 提供 ubutils 工具给管理员进行 UB Entity 信息和拓扑的查询和配置，其中 lsub 用于查看设备拓扑、设备配置空间，setub 用于配置设备空间。
   (2) 提供 sysfs 接口供管理员进行信息查询，包括 UB Entity 信息（EID/UPI/GUID）等。
2. 设备驱动维测：
   (1) 设备驱动和服务驱动在初始化、移除等动作会打印日志，具体可以通过 dmesg 命令查询。
   (2) 用户态可通过/sys/bus/ub 和/sys/bus/ub_service 两个目录查询设备树、设备属性或控制设备（因设备而异）。

4 内存管理（Memory Mgmt）

4.1 概述

本章描述了在 UB 组成的系统互联拓扑下内存访问相关的软件架构和软件接口。

现有总线采用 Host-Device 模型，以 Host（CPU）为中心，Host 管理每一个 Device。在这个模型之下，Host 访问 Device 内存、Device 访问 Host 内存、Device 访问 Device 内存共三类访存路径有单独实现。UB 统一上述访存模型，根据一个访存发起流程将相关的节点分为 User 和 Home，User 可以通过同步（Load/Store）或者异步（DMA）的方式访问 Home 侧内存资源。

4.2 功能架构

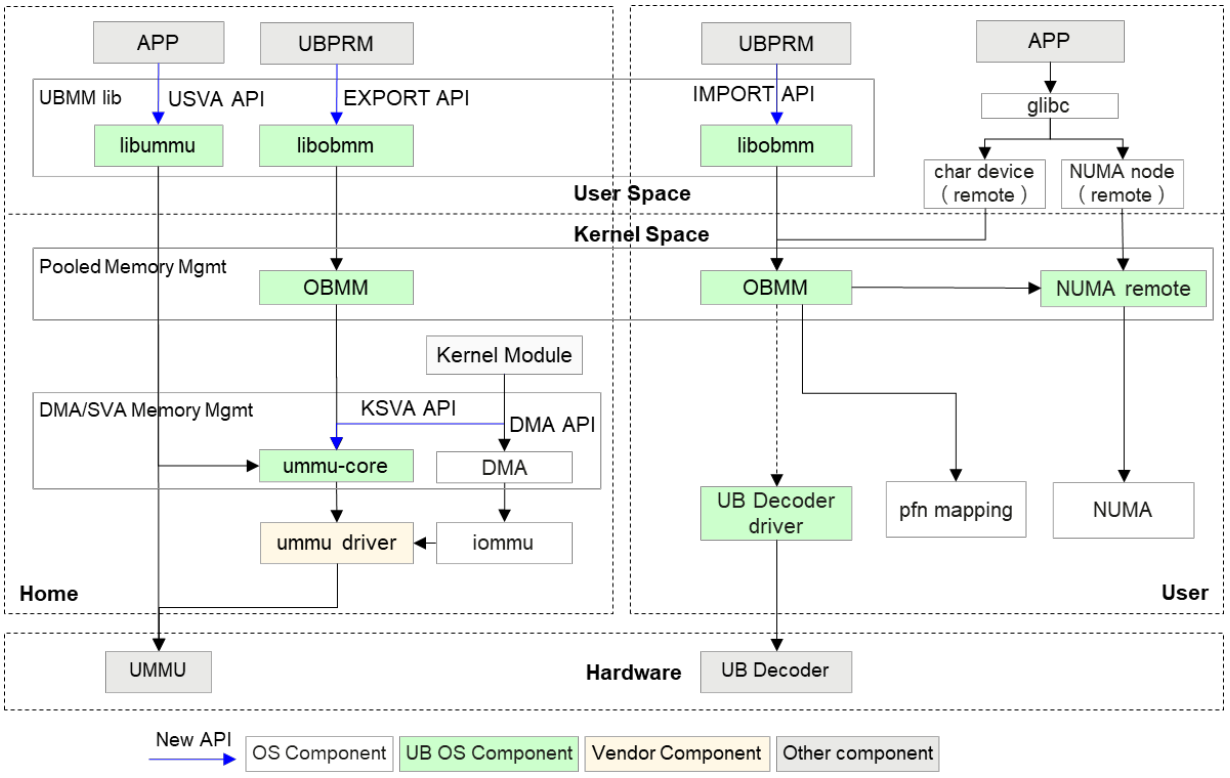

图 4-1 UBMM 软件架构

- **libummu:** 提供用户态 TokenID 申请和释放接口，同时维护《灵衢基础规范 2.0》中定义的权限表。
- **ummu-core：**提供基础的 TokenID 管理和《灵衢基础规范 2.0》中定义的高级能力的配置和管理。
- **ummu driver：**和 UMMU 硬件交互的驱动，实现 ummu-core 定义的底层能力，保证 Home 侧的内存通路。

- **libobmm/OBMM:** 提供内存拆借接口（EXPORT 和 IMPORT）、一致性维护接口（一致性维护接口仅在内存共享场景使用）。
- **UB Decoder driver**：用来配置 UB Decoder 硬件，保证 User 侧的 Load/Store 通路。

注：ummu driver 和 UB Decoder driver 分别用于配置 Home 侧和 User 侧的数据通路，同时 Home 和 User 之间的链路需要 UBFM 配置正确的路由信息才可确保联通。

4.2.1 访存目的方（Home 侧）

在《灵衢基础规范 2.0》中，访存目的方需要将本地的内存资源，注册到 UMMU，并生成访存凭据多元组{EID，TokenID，UBA}。OS 提供四类接口完成上述操作：

- **USVA（User Shared Virtual Address）接口：**用户态进程将进程指定的 VA 地址段导出给其他设备或者节点使用。USVA 增加额外的权限表限制，设备仅能访问指定的用户态 VA 地址段。
- **KSVA（Kernel Shared Virtual Address）接口：**内核态将内核中的指定 VA 地址段导出给其他设备或者节点使用。KSVA 增加额外的权限表限制，设备只能访问被允许的内核 VA 地址范围。
- **DMA（Direct Memory Access）接口：**内核态将指定的物理内存映射为 UBA，导出给其他设备或者节点使用。
- **EXPORT 接口**：从本地导出指定大小的内存资源供其他设备或者节点导入后使用。

4.2.2 访存发起方（User 侧）

访存发起方会使用访存凭据通过不同的方式发起访问，具体包括：

- **异步访问**：使用 Home 侧提供的访存凭据多元组通过传统 DMA 引擎或 URMA 引擎发起异步的 DMA 操作。
- **Load/Store 同步访问**：调用 **IMPORT 接口**将访存凭据多元组配置到 UB Decoder 硬件中，配置完成后，对于指定物理地址的 Load/Store 访存请求会经过 UB 总线完成对 Home 侧的访问请求。

IMPORT 接口以 EXPORT 接口导出的内存资源为入参，支持两种上线和使用方式：

1. 将内存上线到本地 OS 的 NUMA node，由用户态进程通过标准的内存申请接口（例如 glibc/jemalloc 的 malloc/free）申请使用。
2. 将内存上线到 char device，用户态进程直接通过 mmap 的方式映射后使用。

4.3 使用流程

4.3.1 EXPORT/IMPORT

EXPORT 和 IMPORT 用于导出和导入内存，通常使用场景为内存借用。在超节点内，可以将内存富余节点的内存 EXPORT 出来，然后 IMPORT 给内存不足或需要使用大内存做性能优化的节点。

内存借用流程需要由 UBPRM 进行决策，得到 Home 侧的 EID 信息、User 侧的 EID 信息以及需要借用的内存大小。如下流程为决策之后的使用流程：

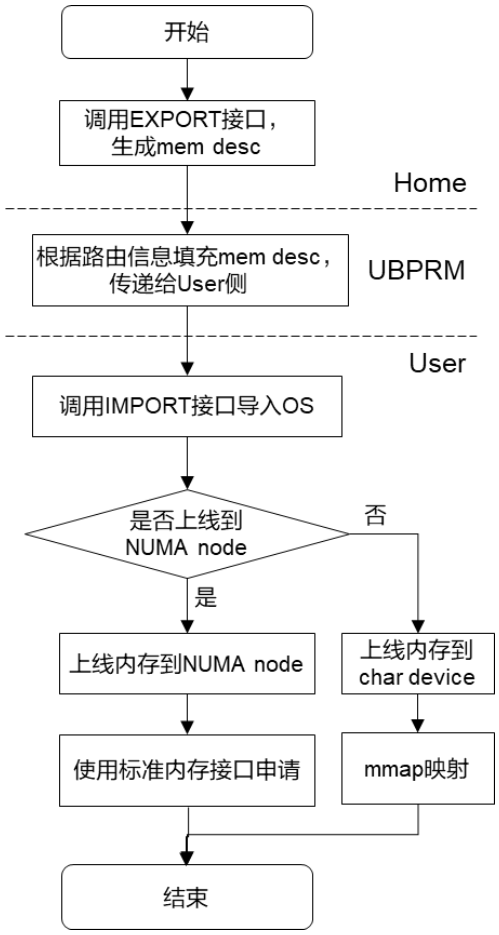

图 4-2 内存借用流程

1. Home 侧：调用 obmm_export 接口，指定 EID 和内存大小信息，生成 mem desc 中的 UBA 和 TokenID 信息，{dest EID，UBA，TokenID}描述 Home 侧出借内存信息。
2. UBPRM：填充 mem desc 中的{src EID，scna，dcna}以及不同 vendor 实现的私有信息。
3. User 侧：调用 obmm_import 接口将 mem desc 导入到 OS 中。
4. User 侧：如果选择上线到 NUMA node，则可使用标准的内存接口申请与使用内存（如 malloc，mbind，numa_alloc_onnode，move_pages 等）。
5. User 侧：如果选择上线到 char device，则调用 mmap 接口映射后使用。

4.3.2 OWNERSHIP

当多个节点需要共享数据进行读写时，则可以由一个节点 EXPORT 出一段内存，其他节点 IMPORT 后使用，形成超节点内的内存共享。当节点间硬件不具有缓存一致性时，多个 User 同时读写会存在数据一致性风险，需要通过软硬协同机制保证数据一致性。

OWNERSHIP 用来管理一段内存的持有状态，每一段内存在每个节点均拥有独立的本地 OWNERSHIP 状态。OWNERSHIP 分为三个状态：

1. Invalid：本节点不能对内存进行读写操作。
2. Write：本节点可以对内存进行读操作或者写操作。
3. Read：本节点可以对内存进行读操作。

用户需要根据内存读写的诉求主动转换 OWNERSHIP 状态，保证软件对于内存的读写符合上述 OWNERSHIP 状态的定义。OWNERSHIP 状态转换中执行的操作有两种：

1. Clean：将 dirty 状态的 CacheLine 写回内存，CacheLine 依然保留。
2. Invalidate：失效指定的 CacheLine，后续 CPU 使用直接从内存读取。

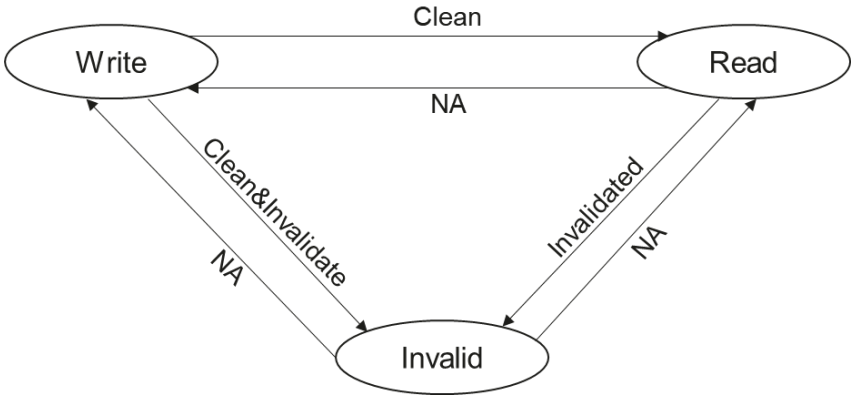

图 4-3 OWNERSHIP 状态转移图

OWNERSHIP 状态转移操作说明：

- Write -> Invalid: 执行 Clean&Invalidate 操作，确保修改的数据写回内存，并且 CacheLine 已经失效。
- Write -> Read: 执行 Clean 操作，确保修改的数据写回内存，同时保留 CacheLine 加速后续的 read 操作。
- Read -> Invalid: 执行 Invalidate 操作，确保 CacheLine 失效，且不影响内存数据。
- 其他（图 4-3 中标注 NA 的状态迁移）：不作处理。

OBMM 提供 obmm_set_ownership 接口实现上述 OWNERSHIP 机制。用户在使用共享内存时，需保证同一时刻只能有一个共享内存的使用方进行写操作。除此之外，针对一段内存读写的前后均需要使用 obmm_set_ownership 维护数据的一致性。

对于共享内存的使用，Home 侧调用流程同内存借用步骤，User 侧调用 obmm_import 接口将内存上线到一个设备（char device），用户通过 mmap 的方式获取 VA 地址，直接使用远端内存。详细流程如下：

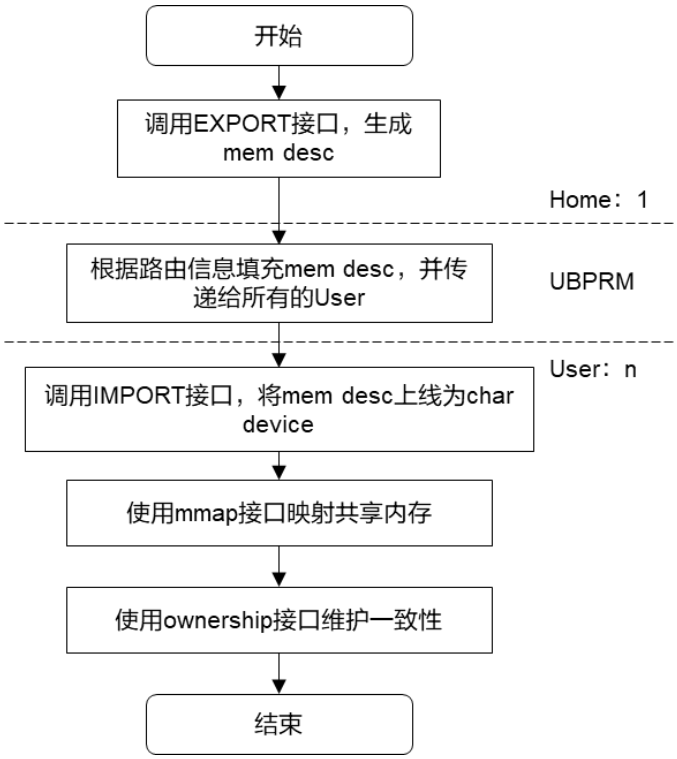

图 4-4 内存共享流程

1. Home 侧：调用 obmm_export 接口，指定 EID 和内存大小信息，生成 mem desc 中的 UBA 和 TokenID 信息，{dest EID，UBA，TokenID}描述 Home 侧出借内存信息；
2. UBPRM：根据 Home 和每一个 User 的拓扑和路由信息填充 mem desc。
3. User 侧：调用 obmm_import 接口将 mem desc 上线到 char device。
4. User 侧：通过标准的 open 和 mmap 在用户态空间获取到共享内存的虚拟地址。
5. User 侧：通过 obmm_set_ownership 在读写之前或之后进行 OWNERSHIP 状态变更（仅 Cacheable 模式需要调用）维护一致性。

4.3.3 DMA/KSVA

传统的设备驱动使用 DMA 接口或者 SVA 申请 Home 侧资源，然后通过设备配置空间配置给设备使用。UB 总线下支持同样的方式，使用流程如下：

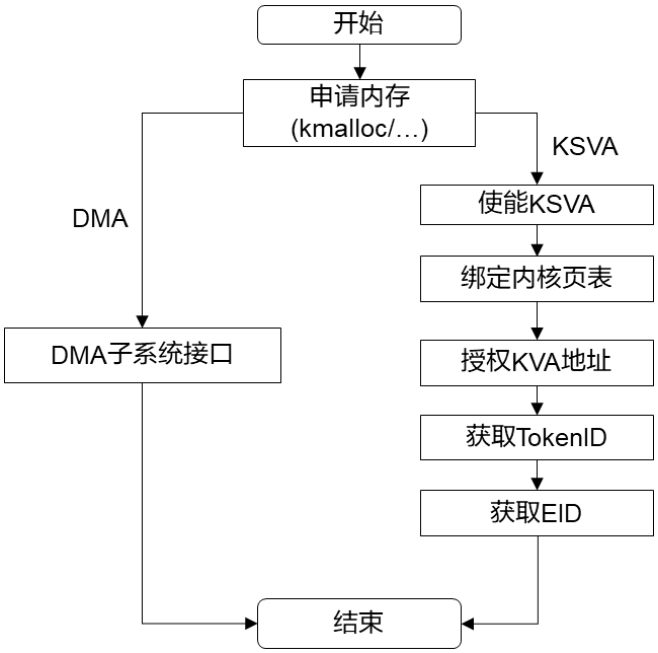

图 4-5 设备使用 Home 侧内存

两种使用方式详细流程如下：

1. 设备驱动通过 DMA 方式使用 Home 侧内存。
   (1) 通过 Linux 的 DMA 子系统接口申请或者直接映射本地内存资源，获得 IOVA（IO Virtual Address）。
   (2) 将 IOVA 作为 UBA 配置到设备寄存器给设备使用。
2. 设备驱动通过 KSVA 方式使用 Home 侧内存。
   (1) 使能 KSVA 特性。
   (2) 通过 KSVA bind 接口绑定内核页表。
   (3) 通过授权接口，授权后续由 User 访问的内存。
   (4) 将{EID，TokenID，UBA，TokenValue}信息配置给设备使用。

4.3.4 USVA

支持从用户态将本进程地址空间的内存资源，生成 UB 内存描述符信息{EID，TokenID，UBA}（该场景下 UBA 等于 VA），实现用户态设备驱动。

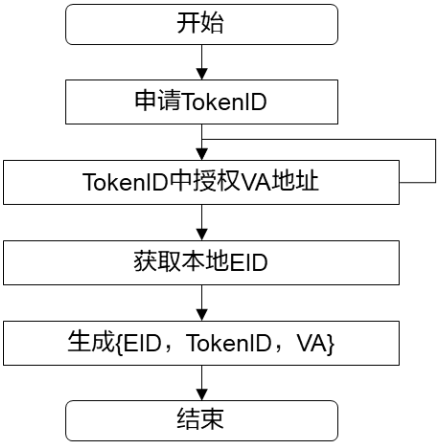

图 4-6 用户态创建 UBMD

1. 通过 ummu_allocate_tid 接口，申请 TokenID。
2. 通过 ummu_grant 接口，将进程的虚拟地址授权给申请的 TokenID，该步骤可以重复多次。
3. 获取本节点（主机节点或设备节点）的 EID 信息。
4. 生成完整的{EID，TokenID，UBA}信息，供其他节点或者设备使用。

4.4 维测

内存管理继承 Linux 生态传统的 ftrace 和 kprobe 等维测手段。除此之外，新增如下的 sysfs，辅助用户维测：

1. 所有创建的 TokenID 在 /sys/devices/platform/ummu_tid_root 和 /sys/devices/platform/ubmem_tid_root 目录下有独立的目录，描述对应 TokenID 的详细信息。
2. 所有创建的 memid 在/sys/devices/obmm 目录下有独立的目录，描述对应的 memid 的详细信息。

4.5 扩展组件 UBTurbo

4.5.1 简介

UBTurbo 在 OS 内提供性能加速功能，用户可根据需要参考使用，包括如下能力：

1. 由于远端内存访问时延大于本地内存，业务需规划远端内存的使用场景避免性能波动过大。UBTurbo 提供内存冷热迁移参考组件 SMAP，核心思路是利用应用访问时空局部特征，基于芯片提供的硬件增强冷热识别能力（需具体芯片支持远端内存冷热识别功能）进行页面数据迁移，将热数据放到本地内存，将冷数据放到远端内存，端到端加速应用性能。
2. 在分级内存场景，由于远端内存访问时延大于本地内存，业务需综合规划使用远端内存的位置及比例。UBTurbo 提供了远近比例调整功能，可以通过业务的内存时延敏感度，冷热分布情况，动态调整业务使用远近端内存的比例。

4.5.2 模块架构

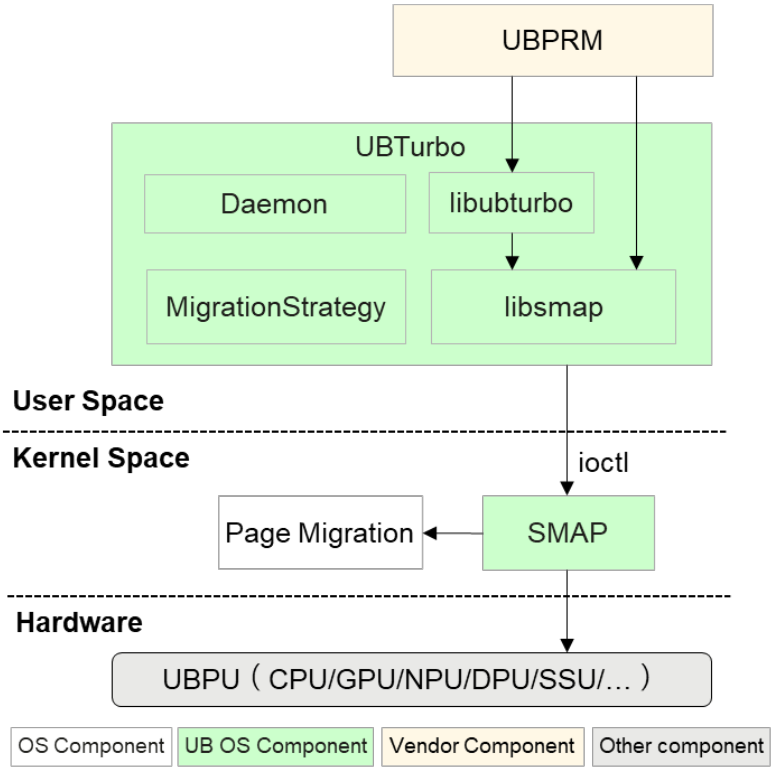

图 4-7 UBTurbo 模块架构

- libubturbo：用户态 lib 库，提供分级内存远近比例调整及 SMAP 等核心功能北向 API。
- libsmap：通过采集的冷热信息进行数据迁移，提升分级内存场景性能。
- Daemon：UBTurbo 进程服务。
- MigrationStrategy：提供资源迁移调度策略。
- SMAP：提供页面迁移内核执行能力，采集芯片提供的冷热数据信息。

4.5.3 使用流程

4.5.3.1 SMAP

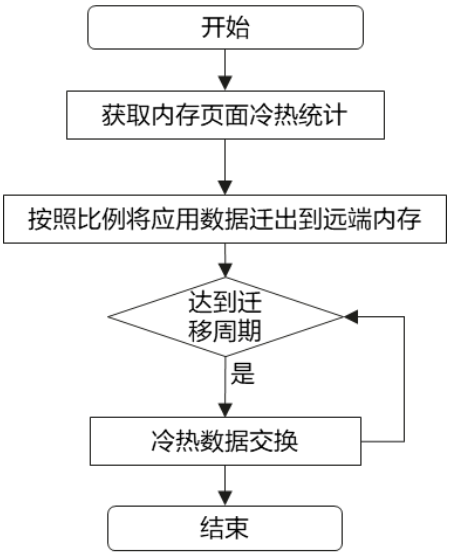

图 4-8 SMAP 冷热识别和数据迁移流程

1. 通过 SmapInit 接口初始化 SMAP 组件，并启动内存页面冷热统计。
2. 通过 SetSmapRemoteNumaInfo 接口和 SmapMigrateOut 接口，按照比例将应用数据迁出到远端内存。
3. SMAP 提供参考策略，可针对注册到迁移进程白名单的应用进程，周期性进行冷热数据交换，实现特定配比下的最优访存性能。
4. 当上层应用需要迁回远端使用内存时，通过 SetSmapRemoteNumaInfo 接口与 SmapMigrateBack 接口，将应用的数据迁回本地。

4.5.3.2 分级内存远近比例调整

借用远端内存后，可以通过 libubturbo 控制 UBTurbo 的分级内存加速能力，根据内存时延敏感度，数据冷热分布情况等，综合决策进程数据迁移到远端内存的位置和大小，并根据数据的冷热变化动态调整进程使用远近端内存的比例，以达到最佳性能与资源利用率。

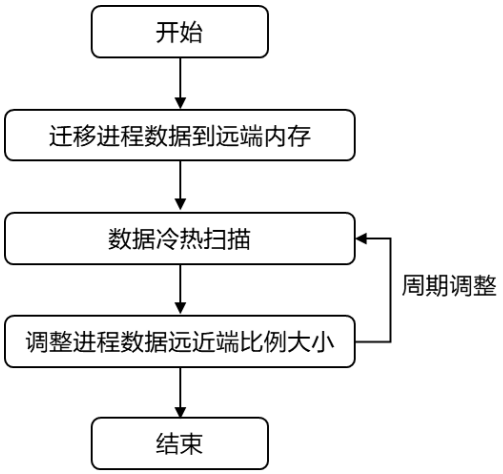

图 4-9 分级内存远近比例调整流程

使用流程：

1. 调用 OutSetWaterMark，初始化场景配置。
2. 调用 OutMemMigrate，提交可使用远端内存的进程 id，以及进程可迁出的最大比例。
3. 调用 SmapMigrateOut 等接口迁出进程数据到指定远端内存上。
4. SMAP 根据进程数据冷热分布以及业务性能要求，动态调整进程使用远近端内存比例大小。

5 通信（Communication）

5.1 概述

灵衢通信库（UMDK）是分布式通信软件库，为数据中心网络、超节点内、服务器内的卡与卡之间提供高性能的通信接口，使能和释放 UB 硬件能力。

5.2 功能架构

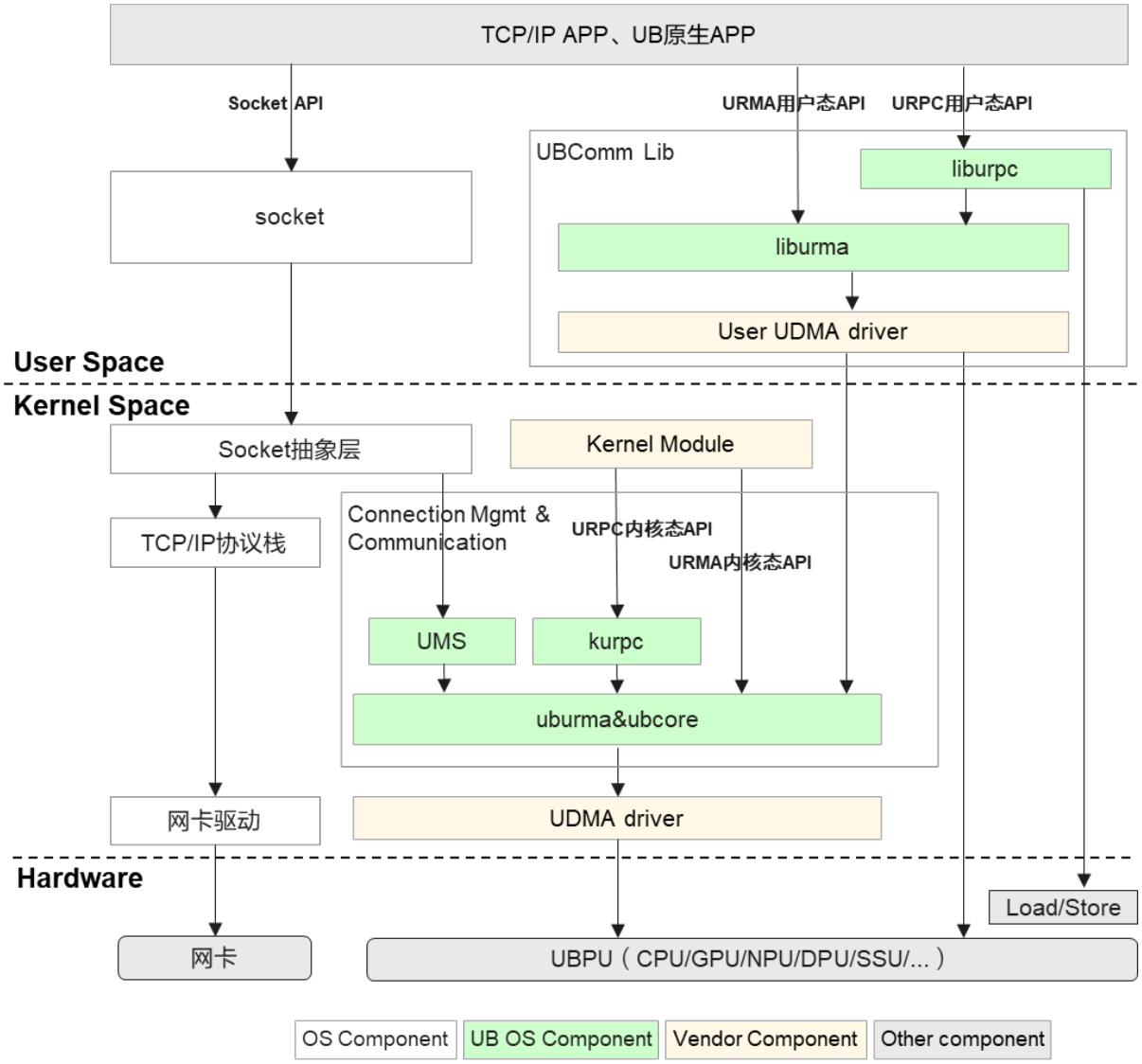

图 5-1 UMDK 功能架构

- **UDMA driver&User UDMA driver：**由硬件供应商提供的设备驱动，可分别在用户态和内核态接入 UMDK。
- **uburma&ubcore&liburma：**包含内核态和用户态两部分，提供了单边、双边、原子操作等远端内存操作方式，是应用之间通信的基础。提供两类接口，一是北向应用编程接口，为应用提供通信 API，二是南向驱动编程接口，为驱动开发者提供接入 UMDK 的 API。
- **liburpc&kurpc：**统一远程过程调用，提供 RPC 语义通信接口。
- **UMS：**对接 Socket 抽象层，实现 Socket 兼容能力。

5.3 URMA

5.3.1 简介

在灵衢平等架构服务器、机架和集群内，CPU、GPU、NPU 等算力单元在处理不同任务时各有所长，为了充分发挥异构算力优势，高效完成计算任务，对通信存在如下诉求：

1. 异构算力直接通信，提升计算系统效率：异构算力以平等方式横向扩展，各组件之间可以直接通信，互相调用，支持在细颗粒度任务上做并行处理或调用。
2. 支持大规模高性能计算通信：提高节点内、节点间通信效率，支持大规模高性能计算通信。
3. 灵活路由，提升网络利用率和业务实时性：支持多路径乱序传输机制、高效的拥塞控制机制，释放数据中心带宽能力，提升网络利用率，保障业务实时性需求，提升计算集群效率。

URMA 旨在完成两个 UB Entity 之间通信，包括单边 DMA 访问、双边消息发送和接收。URMA 具备平等访问、弱事务序、无连接通信等特性：

1. 平等访问：任意异构算力设备之间可通过 URMA 实现免 CPU 介入直接通信。
2. 无连接：URMA 在应用程序之间可复用 UB 传输层提供的可靠服务，在应用之间无需建立端到端的连接，大幅降低通信资源开销。
3. 弱事务序和多路径：URMA 通信组件支持通过应用程序配置来指定任务的保序行为，允许乱序执行和乱序完成，避免任务头阻塞问题，使能底层多路径传输，显著提升整体执行效率。

5.3.2 模块架构

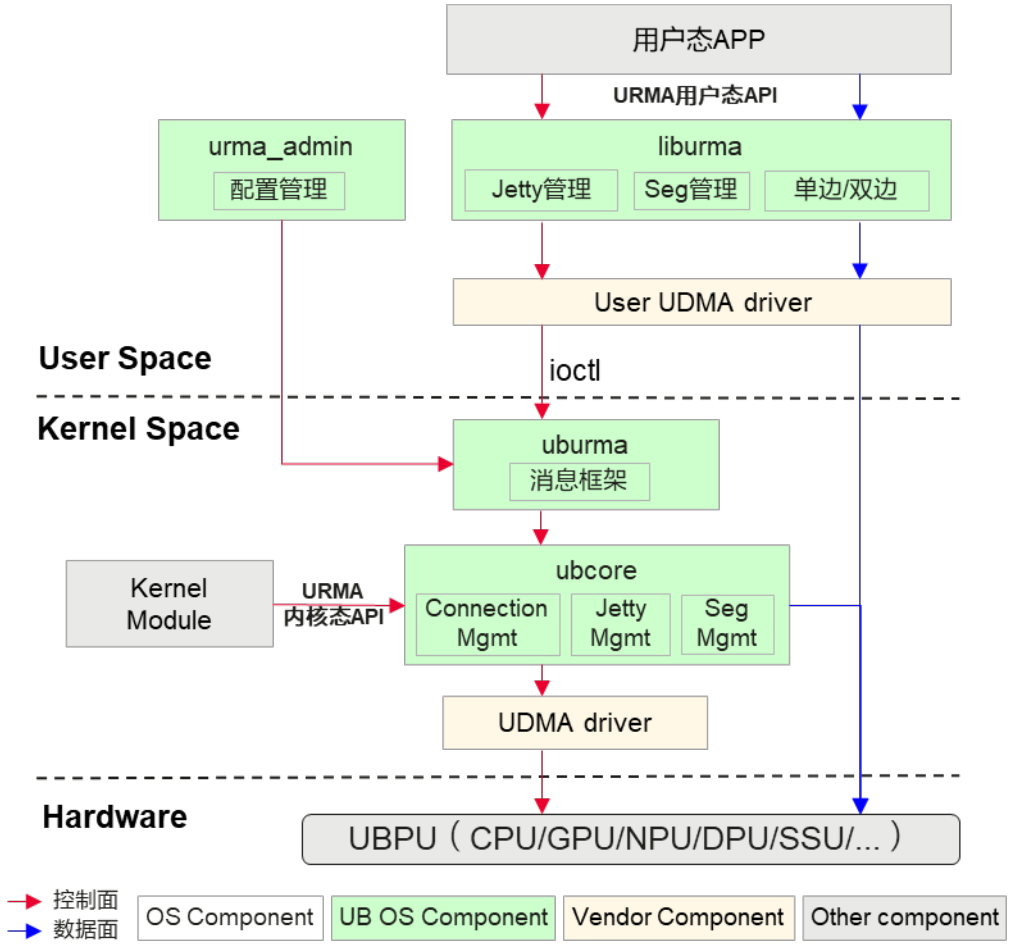

图 5-2 URMA 模块架构

- **liburma**：用户态通信库，提供 Jetty 管理、Segment 管理、数据面接口等核心功能的北向 API。
- **urma admin**：查询和配置工具，用于管理环境上的 urma 设备。
- **uburma**：用户态库与内核态核心层的消息通信框架。
- **ubcore**：负责建链、Jetty，Segment 等资源申请、状态管理等核心功能。
- **UDMA driver&User UDMA driver**：由硬件供应商提供的设备驱动，可分别在用户态和内核态接入 UMDK。

5.3.3 使用流程

URMA 整体使用流程分为如下几个步骤：

1. 创建通信所需的 Jetty。
2. 创建写入读取数据的 Segment。
3. 采用双边、单边、原子等类型发起通信。
4. 读取相关的完成记录。

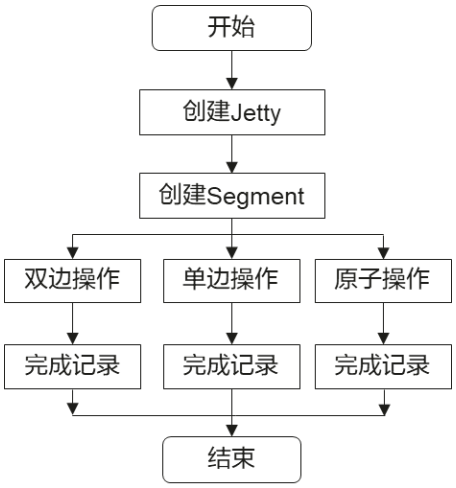

图 5-3 URMA 使用流程

5.3.3.1 Jetty 管理

URMA 执行资源管理通过 Jetty 进行。Jetty 主要用于消息语义接收、发送以及内存语义的命令下发。根据用途的不同，Jetty 可细分为 Jetty For Sending（JFS）、Jetty For Receiving（JFR）、Jetty For Completion（JFC）、Jetty For Completion Event（JFCE）等。在进行具体的 read、write、send、receive 等操作前需要建立相关的 Jetty 资源，后续的 read、write、send、receive 等操作都依赖创建的 Jetty 资源。

Jetty 通过 token 机制进行访问控制。管理面在创建 Jetty 时指定 TokenValue，并通过用户的安全通道交换到发送方，发送方在 import Jetty 的时候会将 TokenValue 配置给硬件 UB 设备。硬件 UB 设备在数据面报文发送时会携带此 TokenValue，接收方收包时会对到达的访问请求做 Token 校验，通过后才允许访问。

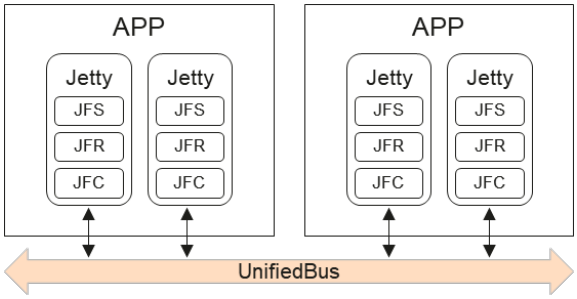

图 5-4 Jetty 管理

使用流程：

1. 调用 urma_create_jetty 创建本地 Jetty。
2. 调用 urma_import_jetty 导入远端 Jetty。

5.3.3.2 Segment 管理

Segment 是一段连续的 VA 地址空间，同时需分配物理内存来对应到一个 Segment。

单边操作（详见第 5.3.3.4 章节）时通过 Token 机制保护内存安全。管理面在注册 Segment 时指定 TokenValue，并通过安全通道交换到发送方，发送方在 import Segment 的时候会将 TokenValue 配置给硬件 UB 设备。硬件 UB 设备在进行数据面单边访问时会携带此 TokenValue，接收方会对到达的单边请求做 Token 校验，通过后才操作内存。

使用流程：

1. 使用本地内存：urma_register_seg 申请 VA，注册 Segment。
2. 释放本地内存：urma_unregister_seg 注销 Segment。
3. 使用远端内存：urma_import_seg 导入远端内存，获取 target_Segment 和 UBA。
4. 释放远端内存：urma_unimport_seg 注销远端 Segment。

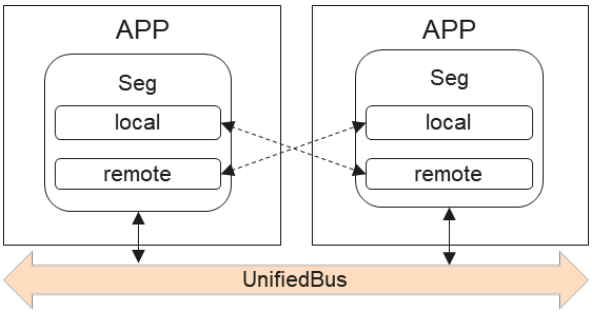

图 5-5 Segment 使用模型

5.3.3.3 双边操作

消息语义提供了双边 Message 服务。消息语义是异步非阻塞，消息接收端需要显式地接收消息，接收完成后读取消息继续其他处理。UMDK 支持一对多消息语义，从同一个 Jetty 向不同的 Jetty 发送消息，这些 Jetty 可能位于不同的远端节点或进程。UMDK 双边操作对本端和远端均支持连续内存和非连续内存。

1. 消息发送流程：
   (1) 调用 urma_post_jetty_send_wr 通过 JFS 提交一个发送请求。
   (2) 调用 urma_poll_jfc 轮询发送请求，轮询完成后，用户才能重新使用（修改或释放）发送消息缓存。
2. 消息接收流程：
   (1) 调用 urma_post_jetty_recv_wr 提交一个接收请求，将本地接收缓存添加到 JFR 中。
   (2) 调用 urma_poll_jfc 轮询接收请求，轮询完成后，用户才能从接收缓存中读取消息内容。

注：为了提高吞吐量服务端可以批量提交多个接收请求。每次成功接收到一个消息后，向 Jetty 的 JFR 补充新的接收请求，或者当 JFR 的接收请求数低于某个阈值时，向 JFR 补充新的接收请求。

5.3.3.4 单边操作

UMDK 单边操作提供了 read/write 语义，使用时需要给出本地的地址和对端的地址。进行单边操作时只有本端进程在操作，不需要对端的应用感知。双边操作一般用于传输一些控制信息，单边操作适用于传输大量的数据，实现大规模数据的搬移等。单边操作缓存支持本端连续内存、非连续内存和远端连续内存。读写流程：

1. 调用 urma_post_jetty_send_wr 提交一个读或写的请求至先前注册好的 JFS。
2. 调用 urma_poll_jfc 进行轮询，查看 jfc 中是否有 cqe 到来，当 urma_poll_jfc 返回值大于 0 时，即表示轮询到有 cqe,表示此次读操作完成。请求完成后，用户才能重新使用（修改或释放）发送消息缓存。

5.3.3.5 原子操作

UMDK 原子操作提供了 CAS 和 FAA 语义，使用时给出知道本地的地址和对端的地址。

CAS 原子操作语义为远端地址的原值与比较值进行比较，如果结果为真则修改远端地址上的数据为新值。否则不修改远端地址的值。

FAA 原子操作的语义为将远端地址的值修改为原值和新值相加的结果，并返回原值。

原子操作流程同上述第 5.3.3.4 章节。

5.3.3.6 完成记录

上述单边、双边和原子操作均为非阻塞方式，操作返回成功仅表示命令已经添加到发送或者接受队列，并不意味着已经全部完成。完成记录（completion record）用来描述操作完成信息。操作完成后，硬件会将完成记录写到 JFC 完成队列中。当用户轮询 JFC 时，读取完成队列的完成记录返回给用户。支持以轮询或中断方式获知操作是否已经完成：

1. 轮询方式应用于低时延场景，用户通过不断查询完成记录，获取操作的执行状态以进行下一步操作。用户调用 urma_poll_jfc 以轮询方式查询完成记录。轮询是一种非阻塞的查询完成记录的方式，如果完成队列为空，则用户获取不到完成记录。

   注 1：用户通过完成记录的状态字段得知操作是否成功完成，如果出错，完成记录的状态字段反应出操作出错的原因。

   注 2：完成长度 completion_len 表示已经成功执行的数据长度，例如发送长度或接收消息长度；该字段只针对数据的接收一侧有效，例如当执行 URMA_OPC_READ 操作或者 URMA_OPC_RECV。

   注 3：如果完成记录为 JFS 类型，说明数据已经发送完成，后续用户可以修改或释放操作对应的本地缓存。

   注 4：如果完成记录为 JFR 类型，表示用户可以从接收缓存中读取消息。

   注 5：如果 notify_data 标志位使能，则完成记录中还携带了立即数。

   注 6：用户通过完成记录的 user_ctx 等于操作上下文（例如 urma_read api 中的 user_ctx 参数），关联到具体某个操作。

2. 中断方式应用于低频通信场景，用户线程以睡眠状态等待完成事件，CPU 开销小，当完成事件发生时，URMA 将唤醒等待的线程。使用流程：
   (1) 调用 urma_rearm_jfc 使能完成事件。
   (2) 提交发送操作（包括单边、双边、原子等）。
   (3) 调用 urma_wait_jfc 阻塞等待一个完成事件，返回产生完成事件的 JFC，同时默认去使能 JFC 完成事件。
   (4) 循环调用 urma_poll_jfc 读取完成记录，直到没有新的完成记录为止。
   (5) 回到步骤（1）重新开启事件。

5.3.4 维测

urma_admin 工具：URMA 框架通过 sysfs 呈现设备不同粒度资源的属性，包含 Jetty 资源的各种属性、端口的状态等。urma_admin 工具可以通过命令行直接查询设备属性和状态，并可以配置部分属性，例如更新 eid、查询设备属性等。

urma_perftest：用于测试时延和带宽测试的性能工具。涵盖收发、读、写、原子操作等四类语义，每种语义支持时延和带宽测试。

5.4 URPC

5.4.1 简介

在灵衢平等架构服务器、机架和集群内，URMA 提供了 CPU、GPU、NPU 之间平等的内存语义通信，但是在功能层存在不同的协议，XPU 与 XPU 之间进行相互访问操作需要协议转换开销。

灵衢统一远程过程调用（URPC）旨在定义 UBPU 之间互访协议。灵衢体系下，任意硬件组件之间的工作关系，均可由 URPC 描述为基于内存对象的函数调用关系。URPC 有两个特点：

1. 平等函数调用：去 CPU 中心化，任一类型 UBPU 可以直接发起对其他类型 UBPU 的函数调用。
2. 引用传参：支持 Worker 基于参数引用(参数数据地址)发起参数数据搬移 。

5.4.2 模块架构

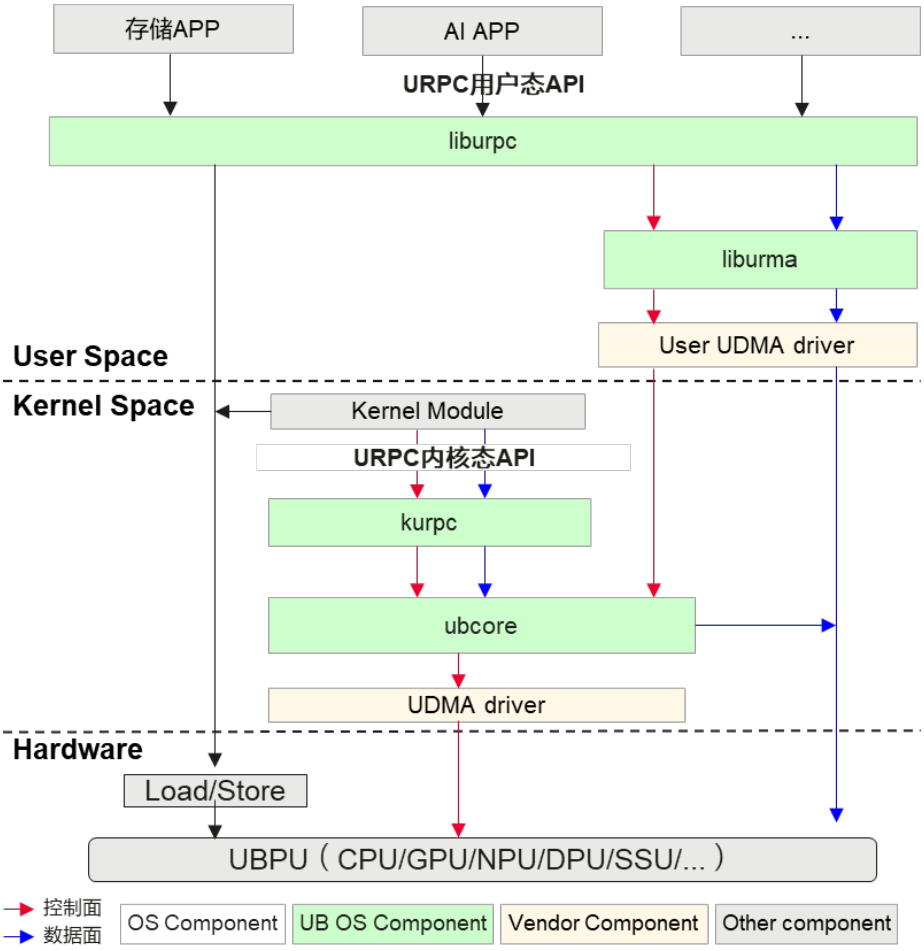

图 5-6 URPC 模块架构

- **UDMA driver&User UDMA driver：**由硬件供应商提供的设备驱动，可分别在用户态和内核态接入 UMDK。
- **ubcore&liburma：**包含内核态和用户态两部分，详见第 5.3 章节。
- **kurpc&liburpc：**包括内核态和用户态。该模块提供易用、统一、高效的远程过程调用语义实现数据中心通信，为应用提供了北向 URPC 编程接口。

5.4.3 使用流程

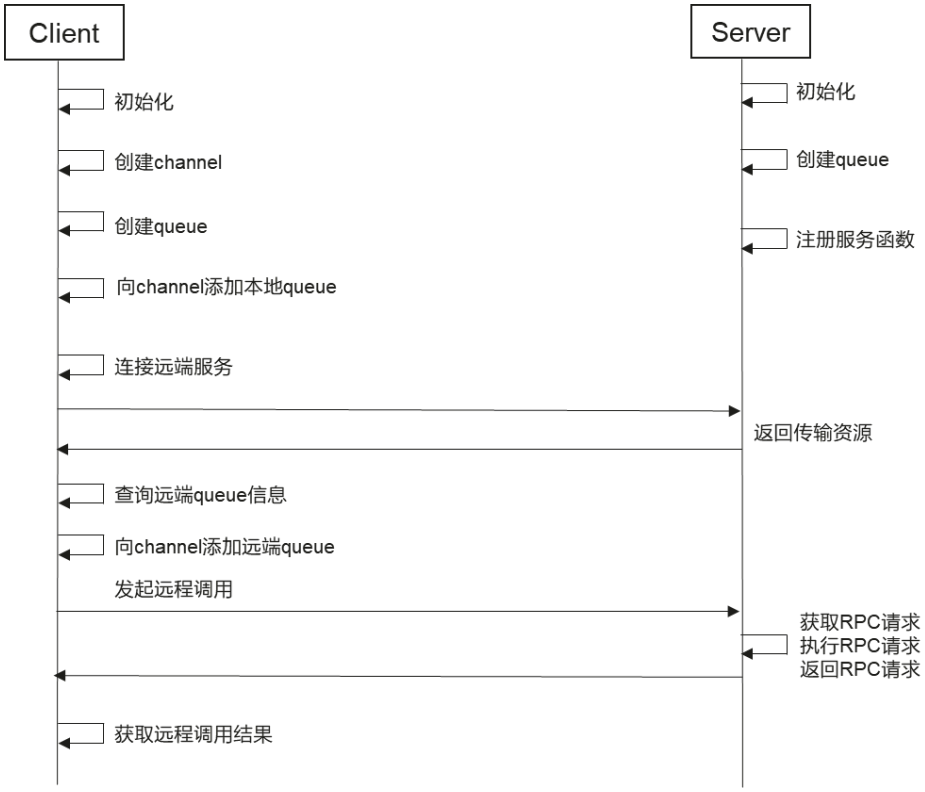

图 5-7 URPC 使用流程

详细步骤如下：

1. urpc_init：client&server 初始化必要的资源。
2. urpc_func_register：server 注册定制函数，获取函数 id。
3. urpc_channel_create: client 侧创建一个 channel。
4. urpc_queue_create: client&server 侧创建 queue，基于传入的模式初始化对应传输层资源（Jetty 或 Memory）。
5. urpc_server_start: server 侧资源创建完毕，启动监听线程，允许 client 来 attach。
6. urpc_channel_queue_add: client 侧将本地 queue 加入 channel。
7. urpc_channel_server_attach: client 侧 channel 挂载到一个 server 上，将 client queue 对应的信息交换给 server，在 server 侧对 client 侧的 queue 建链；server 侧的 queue 信息以及函数信息交换回 client。
8. urpc_channel_queue_query: client 侧从 channel 中查询从 server 交换过来的远端 queue 信息。
9. urpc_channel_queue_add: client 侧将远端 queue 加入 channel，在 client 侧对 server 侧的 queue 建链。
10. urpc_func_call: client 侧 APP 调用该 API 发起一个函数调用。
11. urpc_fun_poll: worker 调用该 API 执行 poll 获取 req（参数）、func_id 等函数执行的必要信息。

12. urpc_func_exec: worker 调用该 API 基于 func_id 和 req(参数)执行定制函数，回调注册的定制函数，获取 rsp。
13. urpc_func_return: worker 调用该 API 将 rsp 返回 client 侧。
14. urpc_fun_poll: client 侧 APP 执行该 API 获取 rsp 以及可以释放的 req，处理 rsp 并释放 req。

5.4.3.1 Channel 创建

URPC Channel 是在 Client、Server 之间无损传递 URPC Request/Ack/Response 的抽象通道。Client 通过 URPC Channel 向 Server 发送 URPC Request，并从同一个 URPC Channel 读取 Server 返回的对应 URPC Ack/Response。

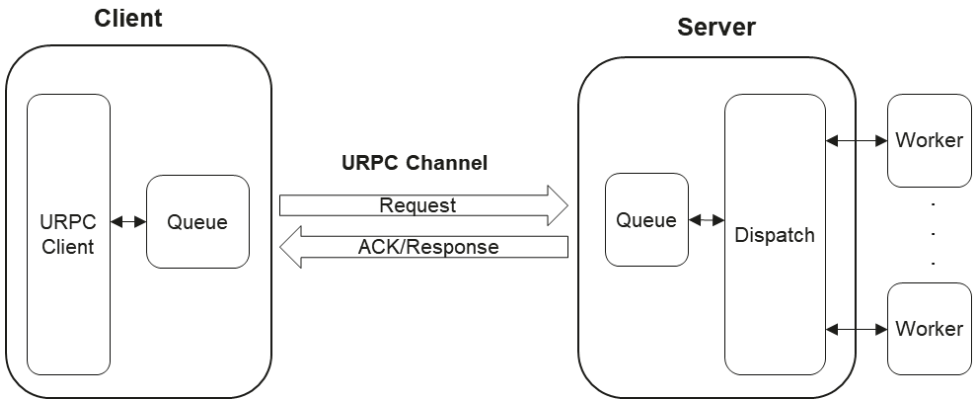

图 5-8 Channel 逻辑图

使用流程：

1. 调用 urpc_channel_create 创建一个 Channel。
2. 调用 urpc_channel_server_attach 连接一个远端 Server。
3. 调用 urpc_channel_queue_add 添加本地 queue 和远端 queue 到 channel 中。

5.4.3.2 Queue 创建

Queue 是一份源端或者目的端传输资源，不同场景可映射不同的实体（Jetty）。使用时 Client 和 Server 都要调用 urpc_queue_create 创建 Queue。

5.4.3.3 函数注册

Server 端调用 urpc_func_register 注册函数。

5.4.3.4 函数调用

1. 调用 urpc_func_call 发起函数调用。
2. 调用 urpc_func_poll 获取执行结果。

5.4.3.5 函数执行

1. 调用 urpc_func_poll 进行请求接收。
2. 调用 urpc_func_exec 执行一个已注册的函数，并获得执行结果。

5.4.3.6 函数返回

调用 urpc_func_return 返回执行结果到发起方。

5.4.4 维测

urpc_admin：urpc 在运行过程中会记录报文统计、内存占用等信息，并可动态开启分段时延统计，通过 urpc_admin 工具可进行相关查询和问题定位。

5.5 UMS

5.5.1 简介

UMS 支持对接 Socket 抽象层，实现 Socket 兼容能力，保持现有 Socket 接口使用高性能灵衢，提升性能。

5.5.2 模块架构

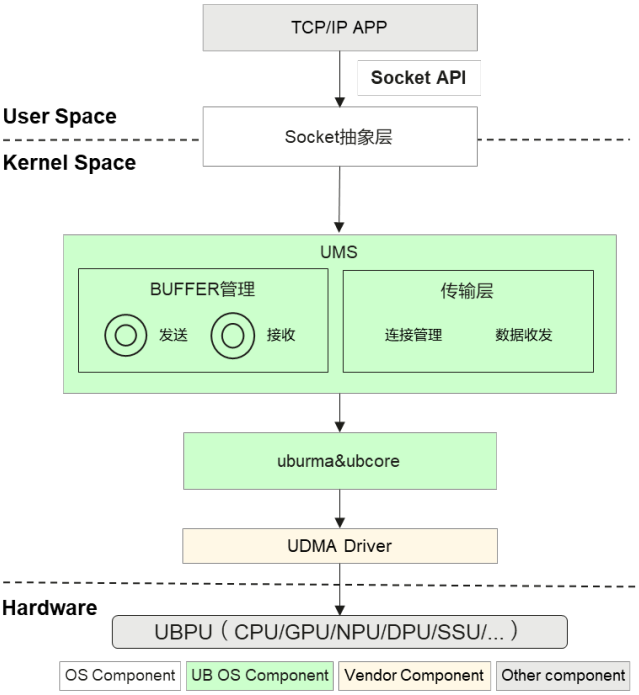

图 5-9 UMS 模块架构

- **UMS**：提供 Socket 兼容接口，在内核态使用 UB 的 URMA Jetty 异步通信或 LD/ST 语义进行通信性能加速。
- **uburma&ubcore**：内核态 URMA 组件，详见第 5.3 章节。

5.5.3 使用流程

应用可以采用 2 种方式使用 ums：

1. 修改应用程序申请 Socket 时的协议族类型。

```
socketfd = socket(AF_INET, SOCK_STREAM, 0) --> socketfd = socket(AF_UB, SOCK_STREAM, 0)
```

2. 修改启动脚本增加 ums_run。

```
ums_run gaussdb -D /data2/lite/data -M primary
```

5.5.4 维测

ums_admin：通过 ums_admin 查询通信链路状态信息，包括 link id、link state、sndbuf cnt、rmb cnt 等。

6 虚拟化（Virtualization）

6.1 概述

本章节主要介绍 UB 总线、UB Entity、Bus Controller 以及 UMMU 的虚拟化实现。在云计算场景中，通过虚拟化将物理机上的硬件资源隔离给多个虚拟机，极大提升了资源利用率。UB 作为新一代高速互联总线，同样支持虚拟化能力。

当前设备虚拟化的主流技术主要包括如下几种：

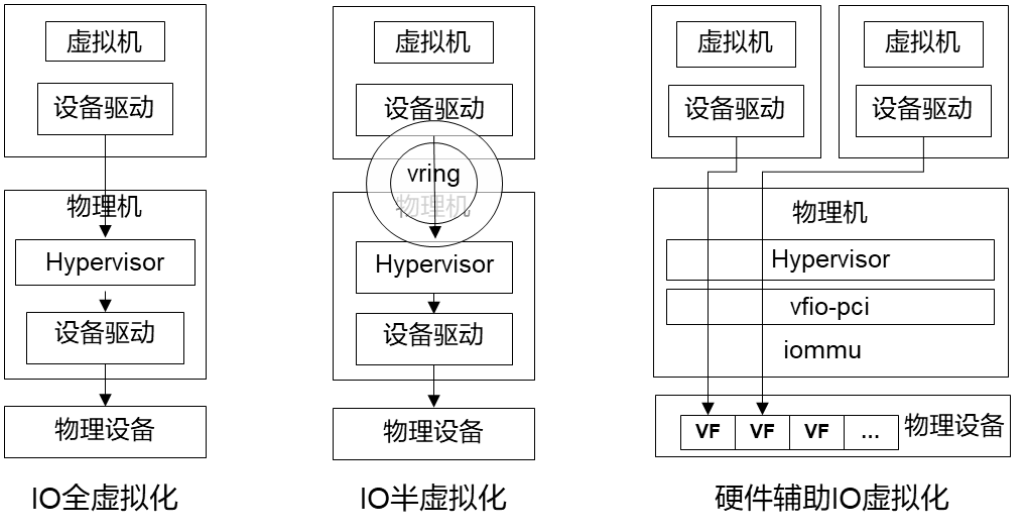

图 6-1 设备虚拟化主流技术

传统架构下每一台服务器都是通过插在其上的网卡连接到 TOR 交换机，所以最大可支持的带宽是固定的，在实际应用过程中会带来一些问题，如下图所示：

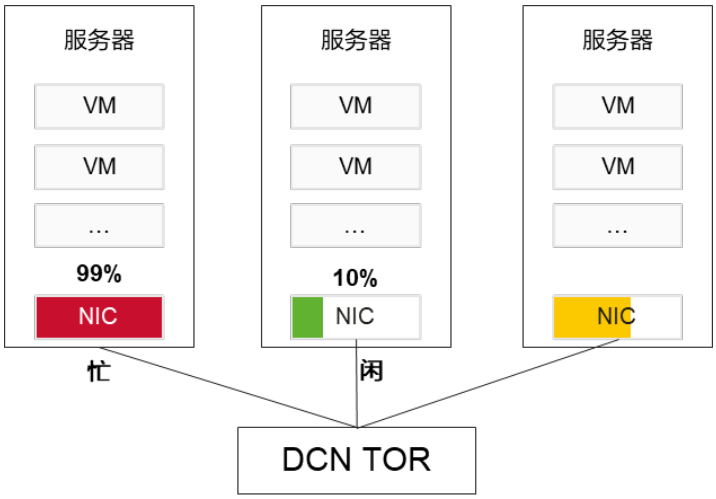

图 6-2 传统架构下网卡流量示意图

- **闲时带宽利用率低：**一般部署业务时会评估一个理论最大带宽诉求，实际运行业务过程中大部分时间都是在低位，偶发业务高峰时期带宽压力才会冲高。这样就导致了带宽利用率低，大量预留带宽资源的浪费。为了提升带宽利用率，公有云厂商一般都会通过虚拟化等技术以带宽超分售卖的形式，将整体物理带宽利用率提升到一个合适的水平。
- **忙时带宽不够用：**在业务高峰时期，带宽压力相对应提升，又会造成带宽不够用的问题。特别是在使用了带宽超分售卖的情况下，所有虚拟机的总承诺带宽超过服务器自身网卡能提供的最大带宽。
- **服务器间带宽压力不均衡：**由于每个服务器的网卡带宽是固定的，每个服务器上业务负载情况的不同会导致彼此之间网卡带宽的压力各不相同，无法给业务提供一个统一的网络服务质量。

在 UB 总线中，UE（UB Entity）作为一个 UB 设备相对隔离的功能单元，UB 虚拟化的功能同样支持通过直通的方式将其直接分配给虚拟机使用，以达到设备原生的性能，UB 设备虚拟化在传统的硬件辅助 IO 虚拟化之上扩展对池化 UB 设备的直通访问。

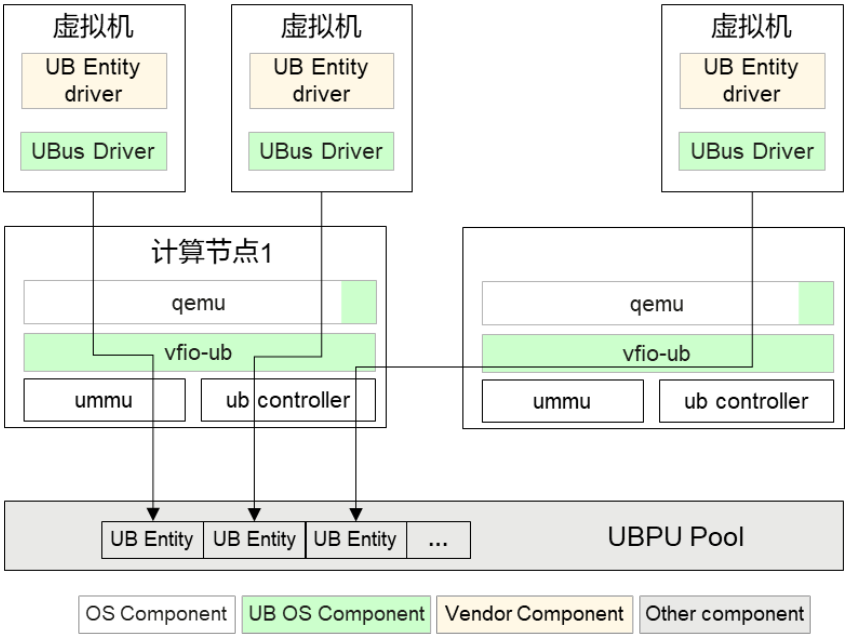

图 6-3 灵衢设备虚拟化技术

通过将网卡、DPU 设备资源池化，可以解决上述问题。所有设备资源统一在设备池中，所有服务器通过共用设备池里的网卡资源，可以根据服务器实际的网络负载情况动态申请使用相对应的网卡资源，达到提升设备资源利用率，避免网络带宽瓶颈。

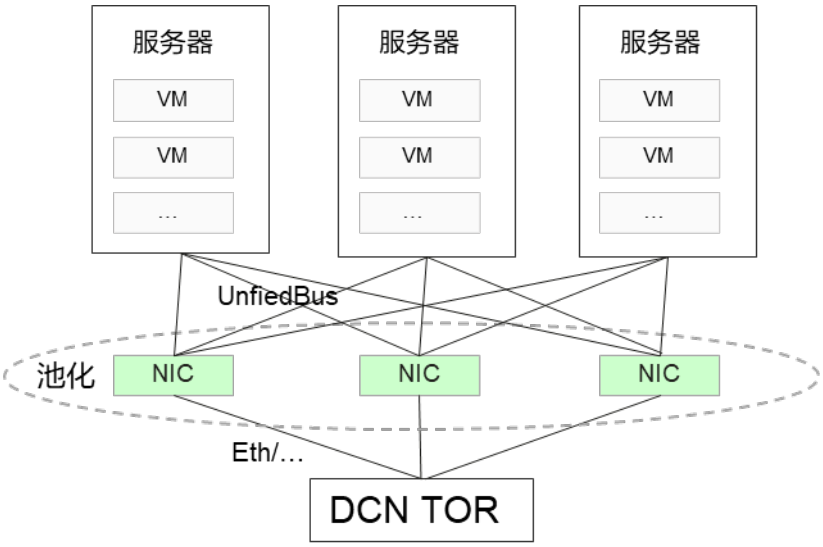

图 6-4 灵衢系统下网卡流量示意图

6.2 功能架构

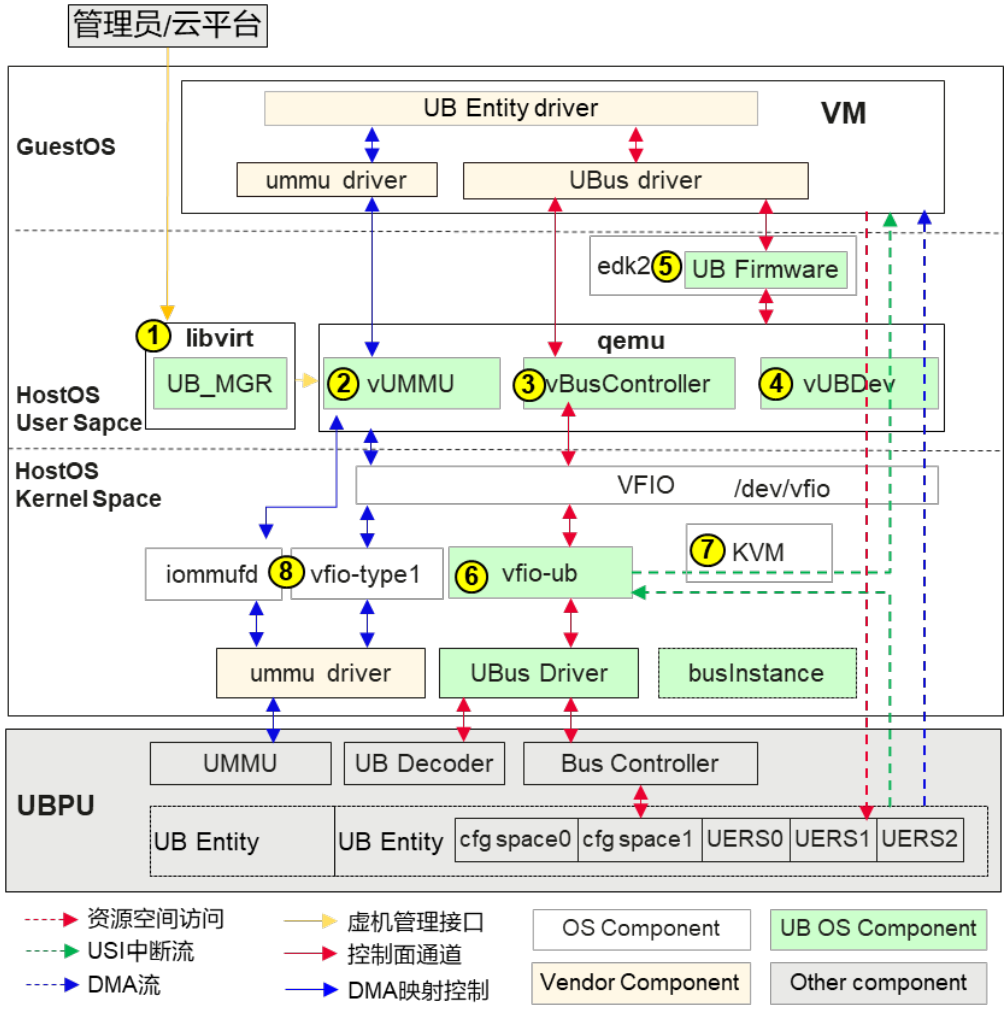

图 6-5 灵衢虚拟化架构

① **UB_MGR**：支持通过 libvirt xml 配置文件创建、管理 UB 虚拟机。

② **vUMMU**：为虚拟机提供 UMMU 模拟，支持 ummu driver 建立 stage1 页表(GVA->GPA)，将页表地址同步到物理 UMMU，完成 UMMU 页表配置。

③ **vBusController**：为虚拟机模拟 Bus Controller，提供虚拟 UB 总线入口，虚拟机内 UB 总线驱动通过 Bus Controller 完成 UB 设备拓扑扫描发现及访问配置等。

④ **vUBDev**：为虚拟机模拟 UB 设备模型，提供完整的 UB 配置空间、资源空间、虚拟拓扑等，完成资源空间映射。

⑤ **UB Firmware**：负责将 Bus Controller 信息、UMMU 信息、可控分配的 UB 资源以及与 UMMU 和中断控制器的映射关系等上报给 GuestOS。

⑥ **vfio-ub**：提供用户态设备管理功能，支持把 UB 设备暴露给用户态 qemu 使用。

⑦ **KVM**: USI 中断重映射，提供虚拟 USI 中断注入或者中断透传。

⑧ **vfio-type1/iommufd**：提供用户空间管理 I/O 页表的 API，供虚拟化完成 UMMU 页表的配置。

6.3 使用流程

UBPRM 将池化设备资源注册到要创建的虚拟机，该过程会为虚拟机在 Host 上创建出 BusInstance，并完成设备注册。

在 UB 总线驱动完成设备注册初始化后，在 Host 上可以通过 lsub 命令获取当前 Host 上的 UB 设备资源列表，然后可以选择将注册给虚拟机的设备通过 XML 配置直通给虚拟机。

```
# lsub
<00002> Class <0200>: Huawei Technologies Co., Ltd. Device <00e0fc>:<a001>
<00004> Class <0a00>: Huawei Technologies Co., Ltd. Device <00e0fc>:<a005>
<00001> Class <0000>: Huawei Technologies Co., Ltd. Device <00e0fc>:<a000>
<00003> Class <0200>: Huawei Technologies Co., Ltd. Device <00e0fc>:<a003>
```

BusInstance 可以通过 lsub 工具或者直接 cat /sys/bus/ub/instance 查询

```
# lsub -b
BusInstance show format: guid type eid upi
00e0fc-0-0-a000-0000-0000000001 Static_Server        00002 7fff
00e0fc-0-0-a000-0000-0000000012 Dynamic_Server   00007 7fff
# cat /sys/bus/ub/instance
   count 0x2, from 0x0, show 0x2
 guid:00e0fc-0-0-a000-0000-0000000001 type:0 eid:00002 upi:7fff
 guid:00e0fc-0-0-a000-0000-0000000012 type:2 eid:00007 upi:7fff
```

dynamic 类型的 BusInstance 是为虚拟机创建的 BusInstance

设备注册后可以通过查询 sysfs 的 instance 确认当前设备所绑定的 BusInstance 信息

```
#cat /sys/bus/ub/devices/00002/instance
0x00007
```

通过 XML 配置 UB 直通设备主要包括如下几部分内容：

1. 虚拟机配置 Bus Controller

   libvirt 的 xml 中对标签进行了扩展，以支持创建 Bus Controller 及 UB 设备的直通。要使用 UB 设备，虚拟机必须要支持 UB 总线，所以需要提供 Bus Controller，可以通过 controller 标签定义一个新的 Bus Controller，最简单的一个 Bus Controller 定义参考如下：

   ```
   <controller type='ub'>
       <ports num='10'/>
       <source>
           <businstance guid='00e-0fc-0-0-a000-0000-000000-0000000012'/>
       </source>
   </controller>
   ```

   ports num 为 Bus Controller 配置支持的最大端口数量，每个 UB 设备需占用一个 port，因 UBus Driver 目前 port index 为 8bit，所以目前 Bus Controller 最大支持 256 个 port，该参数支持自动配置，不配 ports 标签的时候，默认为 128。

   businstance guid 是 UBFM 为虚拟机创建的 dynamic 类型的 BusInstance 的 guid，可以通过 lsub 工具查询

   ```
   # lsub -b
   BusInstance show format: guid type eid upi
   00e0fc-0-0-a000-0000-0000000001 Static_Server       00002 7fff
   00e0fc-0-0-a000-0000-0000000012 Dynamic_Server   00007 7fff
   ```

2. 虚拟机配置 UMMU

   需配置 iommufds 和 ummu，配置如下。

   ```
   <iommufds>1</iommufds>
    <devices>
        <iommu model='ummu'>
        </iommu>
    </devices>
   ```

3. 虚拟机配置 UB 直通设备

   UB 直通设备通过 hostdev 标签进行定义，最简配置如下。

   ```
   <hostdev mode='subsystem' type='ub' iommufd='1'>
   ```

```
    <driver name='vfio'/>
    <source>
        <address guid="00-e0fc-2-0-a003-0200-000000-0000000001"/>
    </source>
</hostdev>
```

其中 source 信息指定了在 Host 上的物理 UB 设备信息，guid 为设备厂商信息，可以在 Host 上具体要直通的对应的 UB 设备路径下查询，例如：

```
# cat /sys/bus/ub/devices/00001/guid
 00-e0fc-8-0-a000-0000-000000-0000000001
# cat /sys/bus/ub/devices/00002/guid
00-e0fc-2-0-a001-0200-000000-0000000001
# cat /sys/bus/ub/devices/00003/guid
00-e0fc-2-0-a003-0200-000000-0000000001
# cat /sys/bus/ub/devices/00004/guid
 00-e0fc-2-0-a005-0a00-000000-0000000001
```

因为 UMMU 权限表的内存连续性要求，虚拟机还必须配置使用大页，大页内存配置 XML 如下

```
  <memoryBacking>
     <hugepages>
       <page size='2048' unit='KiB'/>
     </hugepages>
   </memoryBacking>
```

配置完成后，后续虚拟机启动流程和传统虚拟机无异。

6.4 维测

qemu 中新增了一系列 hmp 调试命令用来分析虚拟机上的 UB 总线以及 UB 设备信息。可以通过 libvirt 调用 qemu hmp 命令，具体方法如下：

```
  # virsh qemu-monitor-command --help
NAME
   qemu-monitor-command - qemu Monitor Command
SYNOPSIS
   qemu-monitor-command <domain> [--hmp] [--pretty] [--return-value] {[--cmd] <string>}...
DESCRIPTION
   qemu Monitor Command
```

```
OPTIONS
    [--domain] <string>    domain name, id or uuid
    --hmp                  command is in human monitor protocol
    --pretty               pretty-print any qemu monitor protocol output
    --return-value     extract the value of the 'return' key from the returned string
    [--cmd] <string>    command
```

使用示例：virsh qemu-monitor-command *my_vm* --hmp *my_hmp_cmd*

1. 查询 UB 总线信息

   命令 ： info ub

   hmp 命令行 info ub 可以列出当前虚拟机的 UB 总线中的所有 UB 设备的简略信息，示例：

```
# virsh qemu-monitor-command vm_xxx --hmp info ub
ID          Name    Bus  Eid  CNA  UeIdx  Guid                                    Type  PortNum
myvfioub1 vfio-ub   0    2    0    0      00-e0fc-2-0-a001-0008-000000-1234567890  2     1
myubc       ubc     0    1    0    0      00-e0fc-8-0-1111-0000-000000-0000000123  8     3
```

2. 查询 UB 设备详细信息

   命令： info ub [id] -- show UB info

   该命令用来查询指定 ID 的 UB 设备详细信息，其中 id 为要查询的 UB 设备的 id 信息，可以通过 info ub 查询到所有的 UB 设备列表中获得。示例：

```
# virsh qemu-monitor-command vm_xxx --hmp info ub myubc
┌──────────────────────────┬──────────────────────────────────────────┐
│ id                       │ myubc                                    │
│ dev_type                 │ 8              type_controller           │
│ cluster_mode             │ 0                                        │
│ fm_deployment            │ 0                                        │
│ config                   │ 0xfffc275c7010                           │
│ config_size              │ 2147483648                               │
│ name                     │ ubc                                      │
│ eid                      │ 1                                        │
│ config0:total ports      │ 3                                        │
│ config0:total UEs        │ 1                                        │
│ config0:cap_bitmap       │ 0x00000004                               │
 ...   省略 ...
│ MR msgq_reg_mem name     │ ub-bus-controller                        │
│ MR io_mmio name          │ UB_MMIO                                  │
```

```
┃hi_msgq_info sq addr          ┃gpa 0x0          hva 0x0                          ┃

┃hi_msgq_info cq addr          ┃gpa 0x0          hva 0x0                          ┃
┃hi_msgq_info rq addr          ┃gpa 0x0          hva 0x0                          ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

3. 查询 UB 设备配置空间信息

   命令：info ub-config id offset len -- show UB config space information

   用来查询指定 UB 设备对应偏移位置的配置空间信息，其中 ubdev_id 为要查询的 UB 设备的 id，offset 为要查询的配置空间偏移地址，len 为要查询的配置空间的长度。示例：

```
# virsh qemu-monitor-command vm_xxx --hmp info ub-config myvfioub1 0 1924

       0x0        0x4        0x8        0xC
     ┏━━━━━━━━━━┯━━━━━━━━━━┯━━━━━━━━━━┯━━━━━━━━━━┓
 0x0 ┃ 00307830 ┆ 00010001 ┆ 00000000 ┆ 00000000 ┃
0x10 ┃ 00000000 ┆ 00000000 ┆ 00000000 ┆ 00000000 ┃
 ... ┃          ┆          ┆          ┆          ┃
0x30 ┃ 00000000 ┆ 00000000 ┆ 34567890 ┆ 00000012 ┃
0x40 ┃ a0010008 ┆ 00e0fc20 ┆ 00000000 ┆ 00000000 ┃
0x50 ┃ 00000000 ┆ 00000000 ┆ 00000000 ┆ 00000000 ┃
 ... ┃          ┆          ┆          ┆          ┃
0x780┃ 00000000 ┆          ┆          ┆          ┃
     ┗━━━━━━━━━━┷━━━━━━━━━━┷━━━━━━━━━━┷━━━━━━━━━━┛
```

7 可靠性（RAS）

7.1 概述

RAS 指的是可靠性（Reliability）、可用性（Availability）和可服务性（Serviceability）。该章节主要描述 OS 新增的可靠性能力，包含基础的 UB 故障感知、隔离、恢复能力以及使用者（用户、UBFM、UBPRM、UBMM 等组件）的处理建议。

表 7-1 RAS 特性列表

RAS 特性	说明
UB 设备故障隔离与恢复	对 UB Entity、Bus Controller 等故障进行检测与恢复。
远端内存故障隔离与恢复	对借用或共享的远端内存访问故障进行检测和恢复。
通信故障隔离与恢复	对 URMA 通信故障进行检测和恢复。
OOM 预防与恢复	利用 UB 内存借用能力预防和缓解 OOM。
节点故障隔离与恢复	节点 Panic 故障时通过内存迁移等策略保证池化内存的业务不受故障扩散影响。

注 1：对于 UBPU 硬件模块的 Local RAS 故障，同传统单机处理方式，由 BIOS 上报 BMC 和 OS，此文档不展开描述。

注 2：在《灵衢基础规范 2.0》中第 10.6.2 章节中，将故障分为三类：

A 类：此类错误的特征是 UB Fabric 的物理拓扑是完好的，错误发生之后能对应到单个事务，错误发生之后不影响其它事务的执行。

B 类：此类错误的特征是 UB Fabric 的物理拓扑是完好的，但错误发生之后无法对应到单个事务，仅能对应到单个 UB Entity、TP Channel 或 Jetty 等（UB Entity 级错误、TP Channel 级错误、Jetty 级错误等），需要对错误单元的全部资源进行处理。

C 类：此类错误的特征是设备级或端口级的公共资源发生错误。

7.2 功能架构

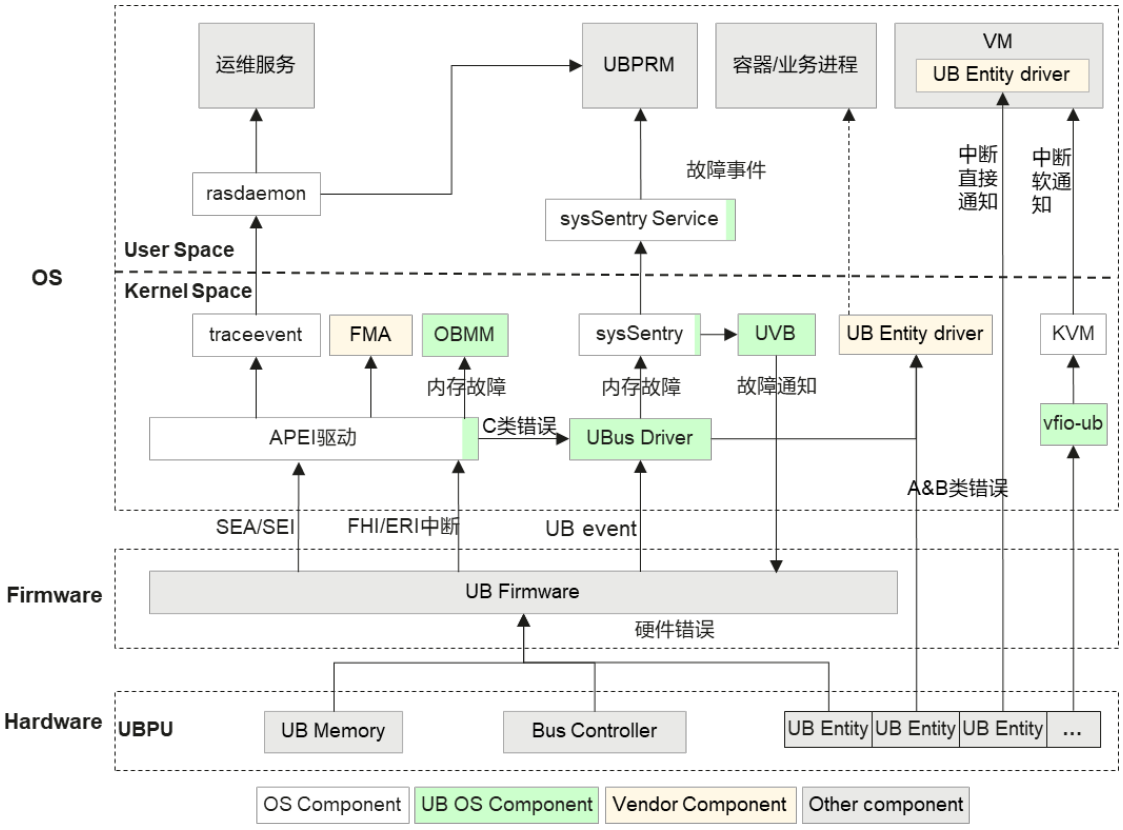

图 7-1 UB 可靠性处理框架

各个组件的分类及作用如下：

1. 硬件&固件层：
   (1) UB Memory、Bus Controller、UB Entity 等组件：硬件在运行中可能会发生故障，此时硬件会上报对应的错误至 UB Firmware。
   (2) UB Entity：发生业务类故障（A&B 类）则直接上报至对应的 UB Entity driver 驱动。
   (3) UB Firmware：接收硬件上报的故障事件，将消息通知至 OS。
2. OS 已有组件：
   (1) APEI 驱动：硬件故障按照 APEI 规范要求上报，并被 APEI 驱动处理。
   (2) traceevent：内核记录各种事件的机制，此处主要针对硬件错误发生时，硬件会通过 APEI 驱动上报对应的 non_standard_event 事件。
   (3) rasdaemon：rasdaemon 会监听 EDAC/traceevent 上报的事件，并记录相关的日志，供上层服务监听并进行对应的处理。
3. UB 新增组件：
   (1) UBus Driver：负责接收 UB Firmware 通过 APEI 上报的 C 类错误，并通知对应的模块进行处理。

(2) sysSentry/sysSentry Service：当紧急事件（OOM、Panic、reboot 等）发生时将相关的紧急事件阻塞，并上报事件到 UBPRM 中，防止发生数据丢失或业务中断，同时也负责统一管理通过 UB event 上报的事件。

(3) OBMM：处理内存相关错误，Home 侧内存故障时， OBMM 对相应内存进行标记，防止被再次使用访问。

(4) UVB：在 Home 侧节点发生故障时，通过该模块将故障信息广播至其他节点进行处理。

(5) vfio-ub：当使用中断软通知机制，设备发生故障后上报至 vfio-ub，然后通过 KVM 通知到虚拟机。

4. 厂商自定义组件：

(1) FMA：内存故障隔离通知链，业务可通过该通知链，根据需要跳过 OS 处理内存故障隔离逻辑，自行进行内存故障处理。

(2) UB Entity driver：处理 UB 各种故障的驱动，可根据业务逻辑需要自定义处理逻辑。

7.3 故障检测

7.3.1 UB 设备故障检测

UB 设备故障是指 UB Entity、Bus Controller 等，不同类型错误处理方式不同。

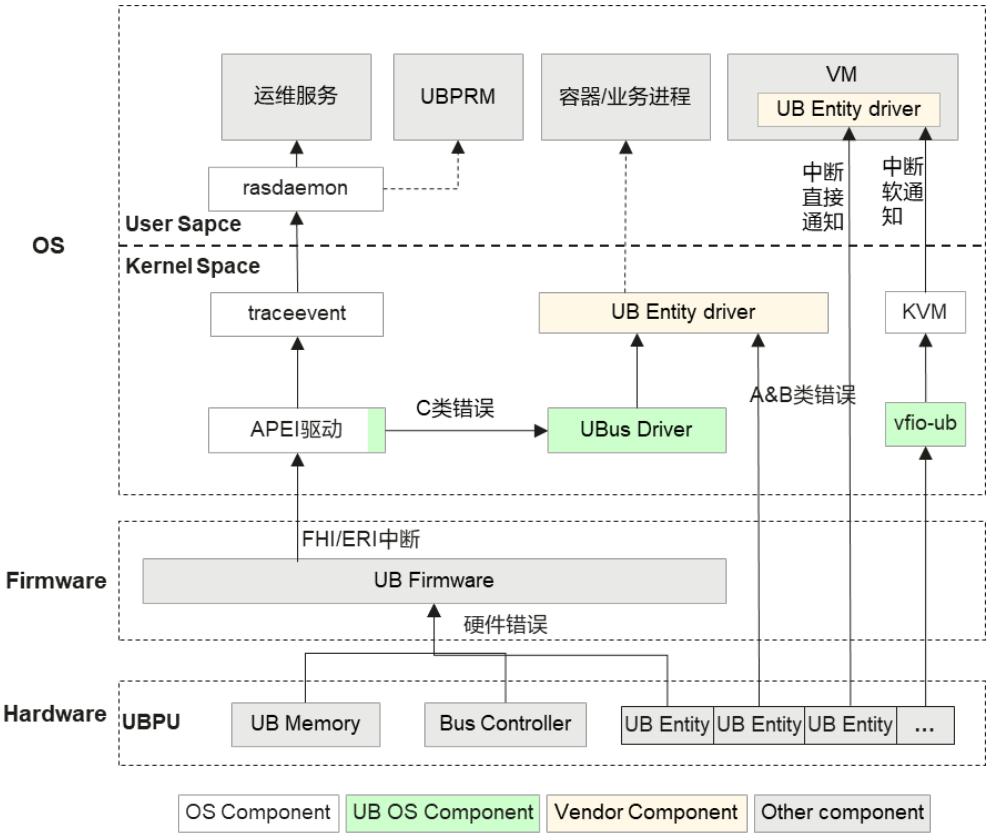

图 7-2 UB 设备故障检测框架

UB 设备故障通知分为两类，具体为：

1. 对于 A&B 类故障：
   (1) 对于 UB Entity 业务故障，UB Entity driver 通过查询 UB Entity 配置空间中的状态寄存器获取，并由对应使用 UB Entity 的组件传递给业务。
   (2) 虚拟化场景下，当 UB Entity 中断直通虚拟机时，则由虚拟机内部的 UB Entity driver 接收。当 UB Entity 中断需要 Host 参与软通知时，则由 Host 上的 vfio-ub 接收中断，并通过 KVM 通知虚拟机。
2. 对于 C 类故障：
   (1) UB Firmware 触发 FHI/ERI 中断。
   (2) OS 内核通过 APEI 驱动上报错误事件 traceevent，同时通知 UBus Driver。
   (3) OS 内核通过 traceevent 将故障上报给 rasdaemon。

7.3.2 远端内存故障检测

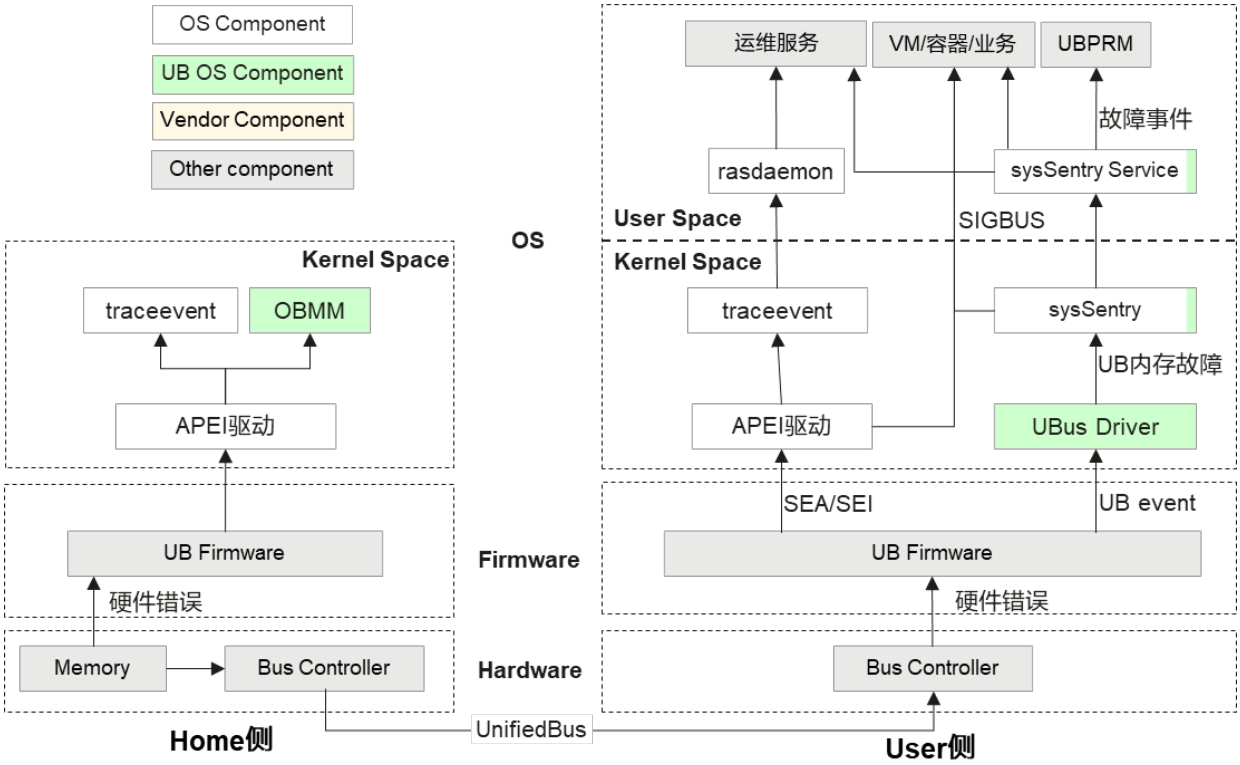

图 7-3 远端内存故障检测框架

借用/共享内存场景下，内存出现故障时，需要 User 侧和 Home 侧同时检测和上报。具体分以下场景：

1. Home 侧内存故障上报：
   (1) Home 侧内存控制器上报内存故障至 UB Firmware。
   (2) UB Firmware 将故障上报至 OS。
   (3) OS 内核中的 APEI 驱动通知 traceevent 和 OBMM。

2. User 侧读远端内存故障：
   (1) UB Firmware 触发 SEA/SEI 中断。
   (2) OS 内核 APEI 驱动上报故障给 traceevent。
3. User 侧写远端内存故障：
   (1) UB Firmware 上报 UB event 给 UBus Driver。
   (2) UBus Driver 将故障信息上报给 sysSentry。
   (3) sysSentry 上报消息给 sysSentry Service。
   (4) sysSentry Service 上报故障信息给 UBPRM 和运维服务。

7.3.3 通信故障检测

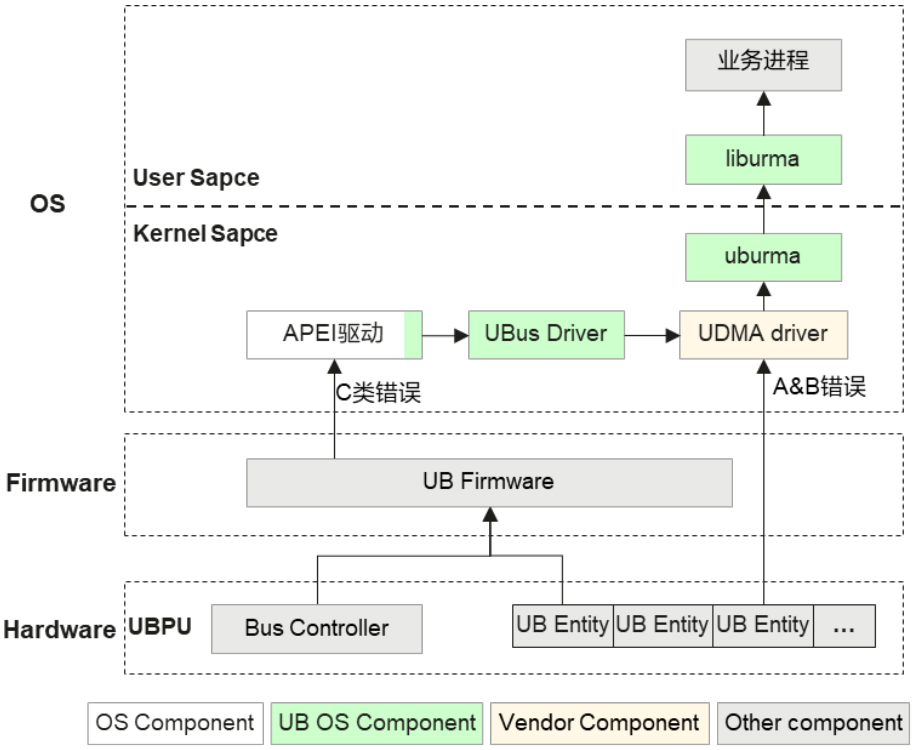

图 7-4 通信故障上报流程

1. 通信设备（UDMA）故障，见第 7.3.1 章节。
2. 通信管理面故障（协议 B 类故障），当硬件遇到无法处理的 WR，例如访问超出本端或远端内存的权限、Jetty 或者 JFC 溢出、驱动卸载、端口状态异常等情况下，UDMA driver 通过查询 UDMA 配置空间中的状态寄存器获取异常事件，UDMA driver 向上通过 uburma&liburma 最终通知业务进程。
3. 通信数据面故障（协议 A 类故障），当数据面发包出现异常，例如 ACK 超时、操作模式不支持等，故障通知流程同通信管理面故障通知流程。

7.3.4 OOM 检测

在内存借用场景下，UBPRM 会按照固定的时间周期检测内存占用水位情况，该水位可配置，根据实际的使用情况采用内存归还/内存借用的策略。当短时间内发生大量的内存占用，导致内存水位迅速上涨时，UBPRM 检测机制可能无法及时发现该现象，导致业务节点发生进程被杀死或节点重启，发生业务中断。此时可通过 OOM 预防机制将 OOM 事件上报给 UBPRM，触发 OOM 的紧急借用策略。总体处理策略如下图所示：

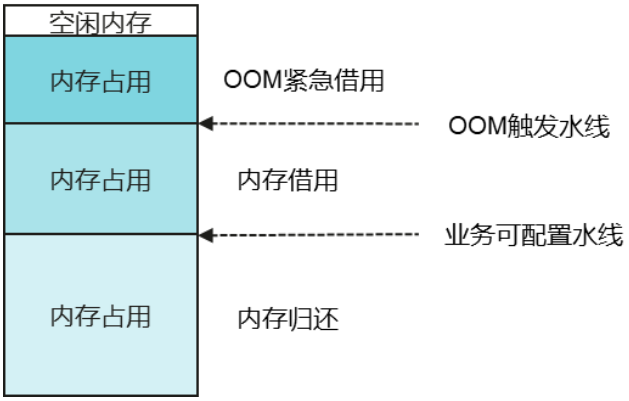

图 7-5 OOM 预防处理策略

当触发 OOM 紧急借用时，通知流程如下：

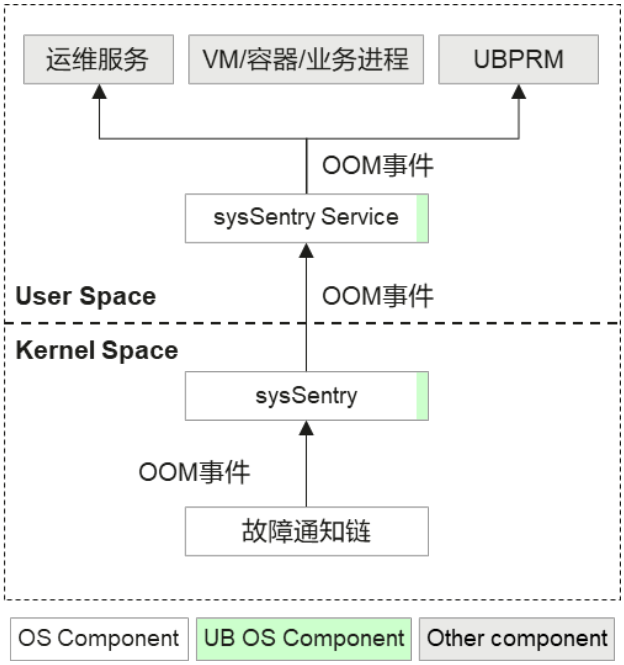

图 7-6 OOM 紧急借用通知流程

1. 发生 OOM 的节点通过 OOM 故障通知链上报 OOM 事件至 sysSentry。
2. sysSentry 将 OOM 事件上报至 sysSentry Service。
3. sysSentry Service 将 OOM 事件上报至 UBPRM 或运维服务。

7.3.5 节点故障检测

当 Home 侧节点发生 Panic/reboot 时，该节点借出的内存将无法访问，会影响 User 侧的业务进程，在此情况下需要将保存在 Home 侧内存中的数据迁移出来。

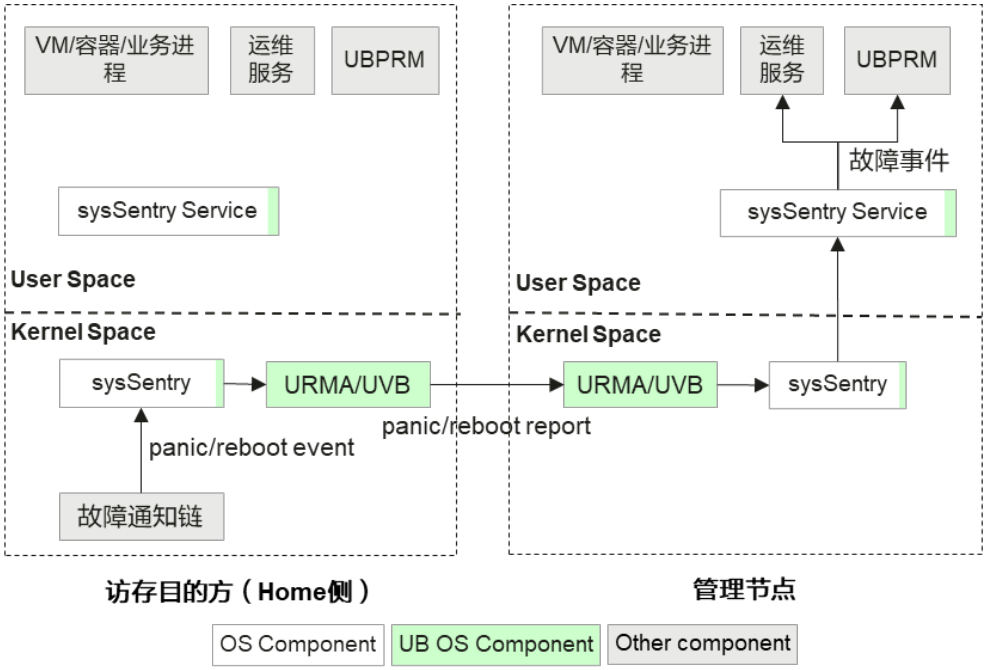

图 7-7 节点系统故障检测框架

详细通知流程如下：

1. Home 侧节点内核故障流程中（例如 panic、reboot、shutdown 流程等）通知 sysSentry 即将复位重启。
2. sysSentry 通过 URMA/UVB 通道进行故障事件广播。
3. 故障事件从 Home 侧节点的 URMA/UVB 通道发送到管理节点的 URMA/UVB 通道。
4. 管理节点的 sysSentry 收到故障事件，通过 sysSentry Service 通知到管理节点 UBPRM。

7.4 故障隔离与恢复

7.4.1 UB 设备故障隔离与恢复

1. 对于 A&B 类故障，则直接由对应的驱动处理或业务处理，例如释放对应的资源、重试等。
2. 对于 C 类故障，如果为可纠正错误可自行恢复，则 UBus Driver 记录日志，不做其他处理。如果为不可纠正错误，则通知对应的 UB Entity 驱动并做相应的处理，例如端口故障，则复位端口，并记录日志。
3. UBPRM 通过系统 traceevent 订阅 non_standard_event 类型的事件，通过对故障事件进行处理。
4. 运维平台通过监控 rasdaemon 相关事件的日志来监测并处理故障。

7.4.2 远端内存故障隔离与恢复

业务或 UBPRM 可通过订阅 sysSentry 远端内存故障访问失败类型的事件，接收对应的故障信息，并根据收到的故障信息对故障内存隔离或者业务进行恢复操作。

1. Home 侧内存故障隔离与恢复：
   (1) APEI 驱动对故障内存进行隔离。
   (2) OBMM 将故障内存标记为不可借出。
2. User 侧读远端内存故障隔离与恢复：
   (1) 当远端内存上线到 NUMA node 时，故障内存会被 APEI 驱动隔离，并发送 SIGBUG 给使用内存的业务进程。
   (2) 当远端内存上线到 char device 时，sysSentry Service 通过事件的方式通知 UBPRM、运维服务、业务等订阅者，订阅者可做相应的故障处理，例如隔离内存或重新申请内存。
   (3) 运维服务/UBPRM 监控 rasdaemon 日志，并进行相应故障处理。
3. User 侧写远端内存故障隔离与恢复：
   (1) sysSentry 发送 SIGBUG 给使用内存的业务进程。
   (2) sysSentry Service 通过事件的方式通知 UBPRM、运维服务、业务等订阅者，订阅者可做相应的故障处理，例如隔离内存或重新申请内存。

7.4.3 通信故障隔离与恢复

1. 通信管理面故障恢复流程：
   (1) 用户调用 urma_get_async_event 接口获取异常事件。
   (2) 用户根据异常事件类型（详见 urma_async_event_type），进行分类处理，例如打印 log 信息等。
   (3) 用户调用 urma_ack_async_event 接口，通知 URMA 已经处理完异常。
2. 通信数据面故障恢复流程：
   (1) 用户调用 urma_poll_jfc 接口获取异常状态。
   (2) 用户根据异常事件类型（详见 urma_cr_status），进行分类处理，例如打印 log 信息等。

7.4.4 OOM 预防隔离与恢复

UBPRM 可订阅 sysSentry 的 OOM 类型事件，当节点发生 OOM 事件后，UBPRM 会收到 sysSentry 服务上报的事件通知，并做相应的处理，建议处理流程：

1. UBPRM 借用其他节点内存，并上线到业务节点的 OS。
2. OS 在 OOM 中尝试申请内存，如果申请到，则返回给对应的应用，否则杀死对应的应用。如果是内核态申请内存出现 OOM，则走入 Panic 流程整机复位。

7.4.5 节点故障隔离与恢复

UBPRM 可通过 sentryctl 相关命令配置 sysSentry 开启对 Panic/reboot 事件劫持、配置跨节点通信必备参数配置，通过订阅 OS Panic 以及 OS reboot 类型事件来监听 Panic/reboot 事件。

当 UBPRM 收到 sysSentry Service 上报的故障通知后，可采取相应的故障隔离和恢复措施，例如将 Home 侧节点内存迁移至其他节点并解除和故障节点的借用关系，确保使用池化内存的业务不受影响。

处理完成后，需返回故障处理结果，系统按照如下流程进行恢复：

1. UBPRM 通知 sysSentry Service 故障处理结果。
2. 故障处理结果经过 sysSentry 和 URMA/UVB 通道发送到 Home 侧节点 URMA/UVB 通道。
3. Home 侧节点 sysSentry 从 URMA/UVB 通道拿到故障处理结果。
4. Home 侧节点返回内核故障流程，继续复位重启。
5. Home 侧节点如果在一定时间内未收到 UBPRM 的通知，则同样会复位重启。

附录 A 缩略语

缩略语	缩略语全称
Bus Controller	UB 总线控制器（Bus Controller）
OBMM	基于所有权的内存管理（Ownership-Based Memory Management）
NUMA node	NUMA 节点（Non-Uniform Memory Access Node）
UB	灵衢（UnifiedBus）
UBComm	灵衢通信（UnifiedBus Communication）
UB Fabric	UB 互联结构（UB Fabric）
UB Entity	功能实体（UB Entity）
UB Firmware	灵衢固件（UnifiedBus Firmware）
UBFM	灵衢互联结构管理器（UnifiedBus Fabric Manager）
UBMM	灵衢内存管理（UnifiedBus Memory Management）
UBPRM	灵衢池化资源管理（UB Pooled Resource Manager）
UBPU	UB 处理单元（UB Processing Unit）
UBRT	UB 设备根表（UB Root Table）
UBS Engine	灵衢系统高阶集群引擎服务（UB Service Core Engine）
UDMA	灵衢 DMA （UnifiedBus Direct Memory Access）
UMDK	灵衢内存语义开发包（UnifiedBus Memory Development Kit）
UMMU	UB 内存管理单元（UB Memory Management Unit）
UMS	灵衢内存语义 socket（UB Memory based Socket）
URMA	统一远程内存访问（Unified Remote Memory Access）
URPC	统一远程过程调用（Unified Remote Procedure Call）
UVB	统一虚拟总线（Unified Virtual Bus）

注：其它术语，参见 1.3 章节引用文档。