

Multikernel 多内核架构简介

王聪

Multikernel Technologies, Inc. 创始人兼 CEO

什么是多内核？

一句话： 不靠虚拟化，在同一台机器上同时运行多个 Linux 内核

- **隔离而非复制：** 每个内核彼此独立，不是彼此的副本
- **资源独占：** 每个内核独享一块硬件分区
- **按应用定制内核：** 为每类负载量身打造 Linux 内核
- **完全兼容 Linux：** 应用无需任何修改

Web Server
Linux Kernel (Web-tuned)
CPU + NIC

Database
Linux Kernel (I/O-optimized)
CPU + NVMe

ML Training
Linux Kernel (GPU-optimized)
CPU + GPU

为什么需要多内核？

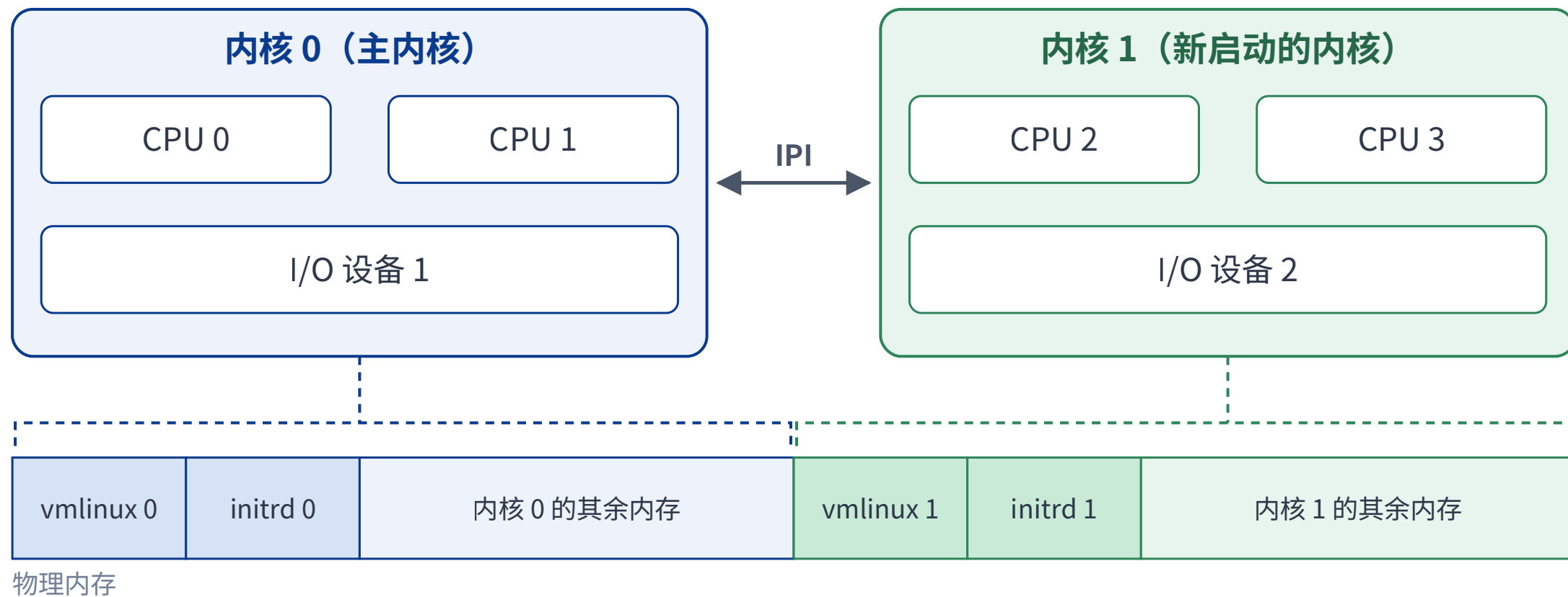
初衷：解决 Linux 内核的扩展性瓶颈

- 单个内核本质上是一个共享的数据结构，核数越多，争用越重
- 有些锁是架构性的，调参也绕不开：目录的 `i_rwsem`、每个超级块一把的 `s_vfs_rename_mutex`
- 48 核上跑 `unlink1`，48 个进程的总吞吐还不到 1 个进程的一半

多内核的思路：把一台机器拆成多个内核

- 同样 24 核，两个内核对比一个内核： `unlink1` 2.6 倍， `rename1` 2.1 倍， `stat2` 2.1 倍
- 按 NUMA socket 拆分： `unlink1` 提升 4 倍

整体架构



同一台物理服务器，没有 hypervisor，每个内核独占自己的 CPU、设备和内存

用 `kexec` 启动新内核

- **新的 kimage 类型**: 新增 `KEXEC_TYPE_MULTIKERNEL`，启动新内核的同时保留当前内核
- **基于 `kexec_file_load()`**: 复用现有 kexec 机制，天然支持内核签名
- **64 位直达 64 位**: 直接加载 vmlinux，无需解压，不走实模式，完全不占用低端 1M 内存

静态还是动态？

我们的观点：

- 静态分区只是动态分配的一个初始状态
- 反过来则不成立

对云计算和 HPC 而言：

- 动态资源管理不是可选项，而是刚需
- 难点在于：既要动态，又要隔离
- 灵活性终将胜出

资源管理

为什么选择 Device Tree?

- 描述硬件的成熟标准
- Overlay 机制天然支持运行时动态变更
- 内核中已有完善的 Device Tree 基础设施

如何使用 Device Tree?

- 基线：描述所有可用硬件资源
- Overlay：描述各实例的动态变更
- 事务：支持失败回滚

内核文件系统接口

```
/sys/fs/multikernel/  
├── device_tree           # 基线资源（可写）  
├── overlays/            # 动态变更  
│   ├── new              # 写入 overlay  
│   └── tx_101/          # 已应用的事务  
│       ├── status       # applied | failed  
│       └── dtbo          # Overlay 数据  
└── instances/           # 运行中的实例  
    ├── web-server/  
    │   ├── device_tree  # 实例资源（只读）  
    │   ├── id  
    │   └── status  
    └── database/
```

动态资源分配

分配流程：

1. 用户态生成描述变更的 overlay
2. 主内核将其转换为资源更新消息
3. 通过 IPI 发送给目标内核
4. 目标内核通过热插拔完成变更

全部复用 Linux 现有能力：CPU / 内存 / PCI 热插拔

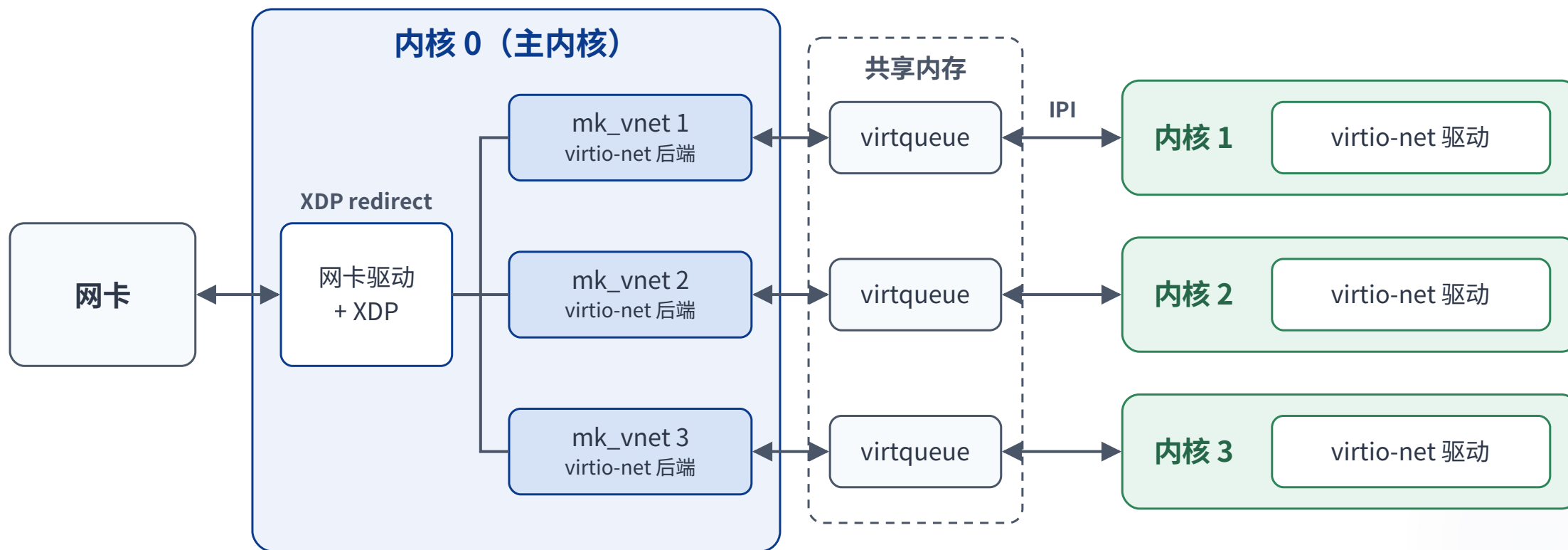
设备共享： virtio

问题： 设备数量远少于 CPU 核数

方案： 复用 virtio 共享设备

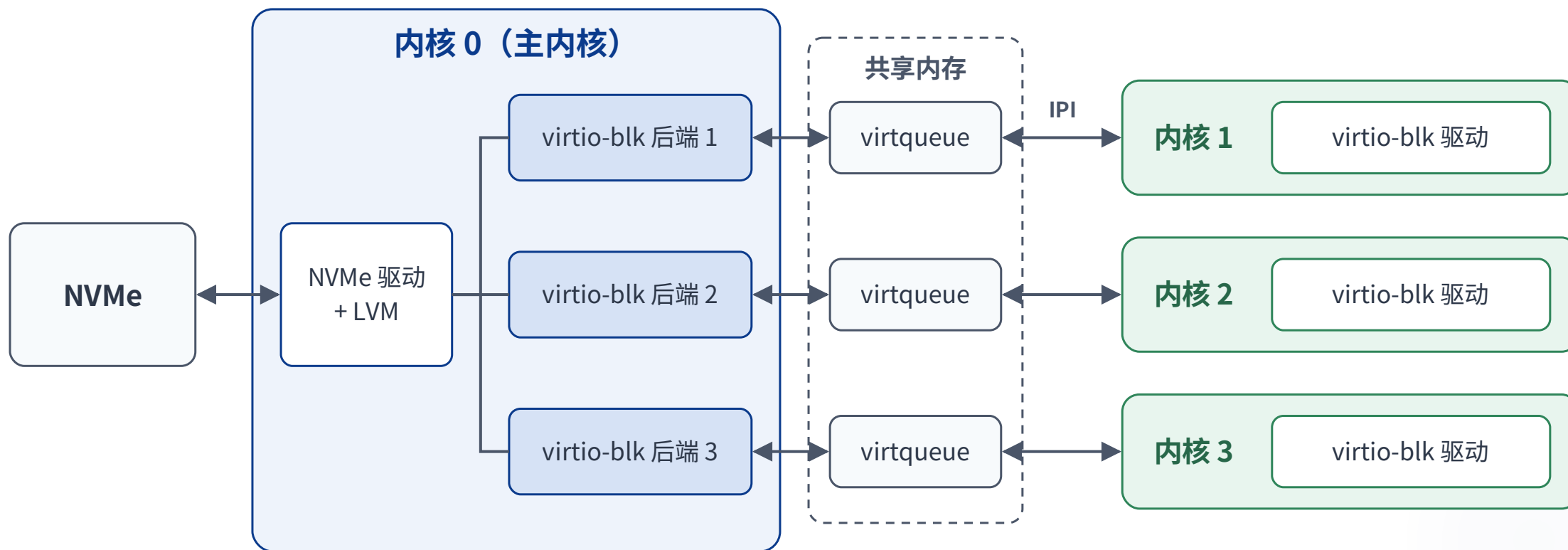
- 主内核持有物理设备，并运行 virtio 后端
- 其他内核使用标准 virtio 驱动，无需开发新驱动
- virtqueue 放在共享内存中，通知通过 IPI 发送
- 性能敏感的负载仍可独占物理设备

网络：virtio-net



网卡与 mk_vnet 之间双向 XDP redirect, virtqueue 在共享内存中, 通知走 IPI

存储：virtio-blk



数据经共享内存中的 virtqueue 传递，通知通过 IPI 发送

文件系统共享： DAXFS

问题： 新启动的内核需要访问主机上的文件系统

方案：

- **DAXFS**：构建在 DAX 内存上的共享文件系统，所有内核可同时挂载
- 读操作直接 load/store 零拷贝，写操作基于 CAS 无锁，无需中心协调者
- 共享基础镜像 + 每实例 COW 层：可直接使用 Docker 容器镜像
- 协同页缓存：内存中只存一份，所有内核共用

内核间通信：vsock

问题： 运行在不同内核上的应用之间需要通信

方案：

- 复用 vsock 作为内核间通信接口
- 新增基于 IPI + 共享内存的本地 vsock 传输层
- 服务之间通信对应用透明

应用场景：私有云

虚拟机的局限：

- 现代虚拟化开销虽小但不为零（约 5%）
- 云主机本身就是虚拟机，再加一层隔离就得嵌套虚拟化

多内核的优势：

- **原生性能**：没有 hypervisor，也没有嵌套虚拟化
- **轻量**：定制内核，不背完整操作系统的包袱
- **栈更简单**：直接访问硬件，层次更少

应用场景：云上容器

容器的局限：

- 所有容器共享同一个内核，只做到资源级隔离
- "吵闹邻居"问题始终存在
- 一个内核漏洞影响所有容器

多内核的优势：

- **内核级隔离**：每个租户拥有自己的内核
- **性能隔离**：彻底消除吵闹邻居
- **定制内核**：按负载优化的轻量 Linux
- **安全**：内核漏洞只影响单个租户

应用场景： AI Agent 沙箱

MicroVM 的局限：

- 仍然依赖虚拟化层
- 受嵌套虚拟化限制，在云主机里难以使用
- 内存难以共享
- GPU 支持受限，通常需要直通或 vGPU

多内核的优势：

- **直接访问 GPU**：无需 vGPU，内核直接驱动 GPU，性能无损
- **无需嵌套虚拟化**：内核之间直接隔离，不需要 hypervisor
- **更快的启动**：内核启动时间约 50ms

应用场景： 内核热升级

传统方式： 必须重启，业务中断

多内核的做法：

1. 在旧内核旁边启动新内核
2. 逐步迁移进程
3. 迁移完成后下线旧内核
4. 任何阶段都可以回滚

效果： 应用零中断

由 Kmorph 项目实施

应用场景：内核自愈

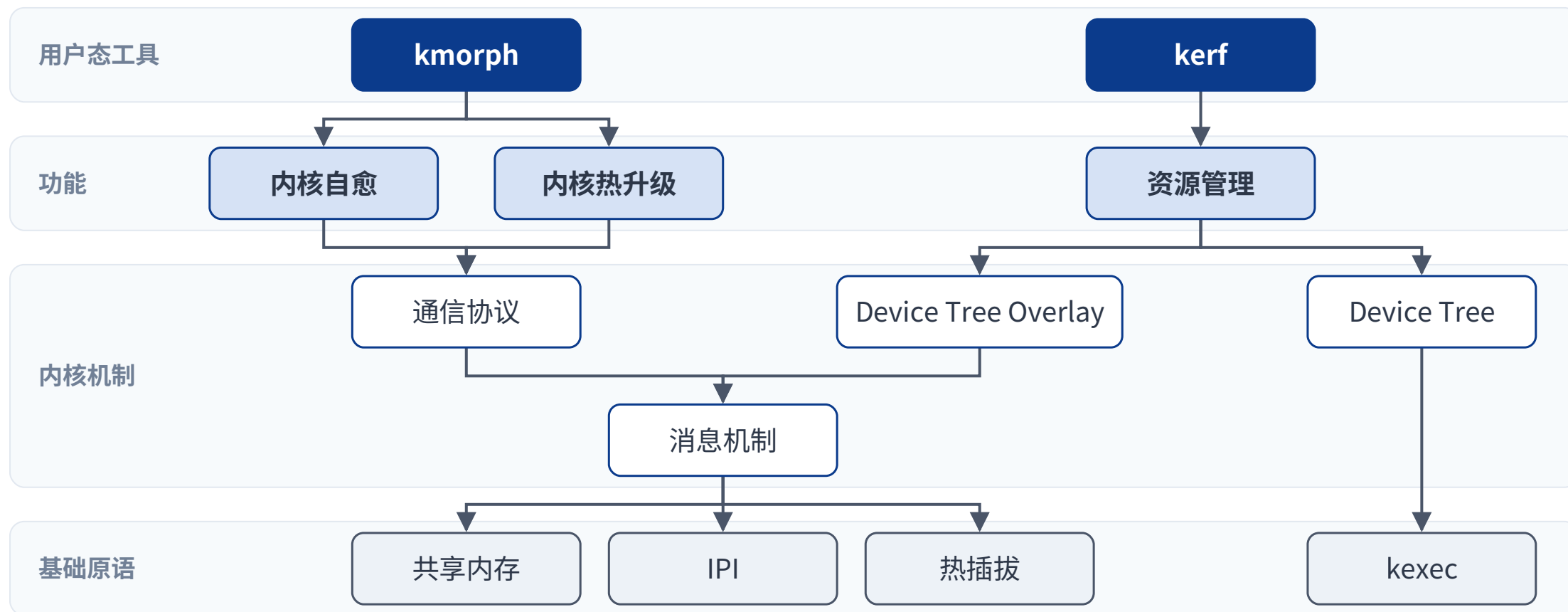
传统方式： Kdump，再加两次重启

多内核的做法：

1. 备用内核通过心跳持续监控
2. 发现故障后接管设备
3. 从保存的内存中恢复进程状态
4. **亚秒级**恢复服务

由 Kmorph 项目实施

全景图



安全模型

前提：

- 信任每个内核都会遵守自己的资源边界
- 攻击面更小

适用场景：

- 同一组织内部的多租户系统
- 访问受控的云环境
- 内核由厂商严格掌控的专用设备

取舍： 以性能优先，而非追求绝对的安全隔离

多内核与其他方案对比

	裸机 Linux	容器	虚拟机	多内核
应用兼容性	完全	完全	完全	完全
隔离级别	无	进程级，共享内核	硬件虚拟化	内核级
性能	原生	原生	接近原生（约 5% 开销）	原生
需要 hypervisor	否	否	是	否
每个负载独立内核	否	否	是	是
动态调整资源	不适用	是	是	是
内核零停机升级	否	否	是（需热迁移）	是

后续计划

- **GPU 支持**: 借助 PCI 热插拔, 可能已经可用
- **ARM / RISC-V 移植**: 面向 ARM 和 RISC-V 服务器
- **快照与恢复**: AI Agent 状态持久化和快速恢复的关键
- **Kubernetes 支持**: 实现 `libkerf`, 对接 K8S CRI
- **内核切换**: 通过内核分时复用支持 CPU 超分

研究方向

我们正在探索的开放问题：

- 内核自动裁剪
- 基于 profile 的内核自动优化
- 基于机器学习的内核优化
- 零停机内核升级
- 多内核上的机密计算

快速上手 (1/2)

1. 编译多内核版 Linux:

```
git clone https://github.com/multikernel/linux.git
cd linux
cp /boot/config-$(uname -r) .config
make menuconfig # 打开 CONFIG_MULTIKERNEL 和 CONFIG_MKTTY
make -j$(nproc)
sudo make modules_install && sudo make install
```

重启进入新内核后即可使用，无需额外的启动参数

快速上手 (2/2)

2. 安装管理工具 kerf:

```
git clone https://github.com/multikernel/kerf.git  
cd kerf && pip install -e .
```

3. 启动第一个实例:

```
kerf init --cpus=4-31 --memory=8GB  
kerf create web-server --cpus=4-7 --memory=2GB  
kerf load web-server --kernel=/boot/vmlinuz \  
    --image=nginx:latest --console=mktty0  
kerf start web-server --console
```

`--image` 通过 DAXFS 直接用 Docker 镜像启动（需先加载 `daxfs.ko`）

欢迎发行版合作

希望与发行版一起做的事：

- 打包 kerf、DAXFS 等用户态工具和模块
- 提供为多内核实例裁剪的轻量内核配置与镜像
- 在更多硬件和发行版内核上测试，反馈问题
- 共同推动内核补丁进入上游主线

我们提供：

- 直接参与设计决策
- 集成与部署期间的技术支持
- 及时响应发行版提出的问题和需求

问答与交流

联系方式: cwang@multikernel.io
开源项目: github.com/multikernel
视频演示: youtube.com/@multikernel-tech

欢迎订阅邮件列表: multikernel@lists.linux.dev



扫码加入微信群